Container Based Test Bed for Characterizing Power and Performance of Big Data Workloads

Bhanuprasad Patibandla

Department of Electrical Engineerng

Univeristy of North Carolina, Charlotte NC 28262

bpatiban@uncc.edu

Abstract

Containerization is an emerging cloud technology which exploits new features in the Linux kernel to deliver operating system level virtualization. This project introduces a methodology to characterize the power and performance of big data workloads on a container based cluster. The platform used is a Penguin Computing Niveus 5200 workstation equipped with PowerInsight. PowerInsight is a BeagleBone based embedded system for subsystem power measurement. Metrics at the level of containers are tracked using the accounting functionality of cgroups whereas power consumption at the subsystem level is tracked by requesting data from a server on the Beaglebone. We demonstrate the utility of this method by characterizing the sort workload from Intel's HiBench benchmark for Hadoop MapReduce.

1 Introduction

Recent times have witnessed an explosion in the amount of data available online. Due to the unstructured nature of the data, distributed batch processing systems like Google's MapReduce or it's open source implementation Hadoop are used to process this data instead of relational databases. A large part of the expense of operating data centers which house the compute infrastructure for these services comes from energy costs. The energy consumption of data centers is increasing by 15% annually. It is estimated that in 2010, energy consumed by global

data centers accounted for between 1.1% to 1.5% of the total energy. As more and more data becomes available and is exploited by businesses, this figure is bound to rise. Therefore, it is critical to improve the energy efficiency of frameworks like Hadoop. This kind of systems research requires the measurement of performance and power consumption which would be difficult to do on an actual cluster, especially in academic settings.

Linux Containers are a lightweight alternative to hypervisor based virtual machines. They deliver operating-system level virtualization by relying on new kernel features of cgroups and namespaces. Instead of emulating hardware and booting a full OS, containers provide an isolated environment with a restricted view of the file system, separate process and network spaces along with isolation of resources like CPU, memory, disk I/O, etc.

This project aims to develop a methodology to track performance and power consumption at the sub-system level for any Big Data Framework on a container based cluster. This would help application developers determine performance bottlenecks and make system level optimizations, enable developing statistical/machine learning models that could be used in a system simulator and serve as a platform for performing multi node scalability experiments on a single node.

Although, for running applications, containers can be used just like a physical machine, when it comes to performance monitoring, it is better to collect performance metrics from outside the container. The container does not report accurate statistics from within. Running *top* inside the container for instance will just return the statistics of the host. This can be done by using the accounting functionality of cgroups. Whereas performance monitoring tools like Ganglia and Cactii exist for Hadoop clusters, they are not supported on Linux Containers. For collecting power metrics at the subsystem level a BeagleBone based embedded system instrumented in the server has been used.

2 Background

2.1 Platform

The platform used is a Niveus workstation form Penguin Computing with dual Intel Xeon E5-v2 IvyBridge-EP processor, 1 TB SATA, 32 GB RAM and equipped with PowerInsight - a Beagle Bone based embedded system for subsystem power measurement.

PowerInsight enables the collection of power measurements at the level of components like

CPU, memory, disk,etc. It comes in the form of an embedded system based on the Beaglebone which collects data from four ADC's - one on the Beaglebone and three on a custom carrier board. Harnesses which carry the sensor module are integrated in-line between the power supply and the motherboard connectors. The sensor module consists of a hall effect current sensor and a voltage divider. These sensor modules are in turn attached to the ADC connectors on the custom carrier board. The Beaglebone affords the advantage of a full fledged Linux OS and also network connectivity to collect the results. It contains the firmware to collect the results from the ADC's through an SPI interface and scale the results.

2.2 Introduction to containers

Linux Containers are similar to a chroot environment. Chroot on Unix operating systems allows a running process and its children to be attached to a new root directory. It amounts to changing the apparent root of the process. A program executed inside a chroot 'jail' cannot see any files outside the designated root directory. Thus, using chroot with a utility like debootstrap it is possible to simulate an entire different distribution on your Linux machine.

Containers can be thought as a more sophisticated version of a chroot jail. Whereas a chroot jail only provides isolation for the file system, linux containers go further and provide isolation and control for resources like CPU, memory, block I/O, etc. Containers are fundamentally different from hypervisor based virtualization in that they do not emulate any hardware and do not boot a new OS instance. The host kernel manages the isolation of instances and therefore virtualization is limited to Linux guests. Since all instances are using the same kernel and with the use of copy-on-write, bringing up a new instance is a matter of seconds and not minutes. It is also very cheap in terms of memory overhead.

LXC is essentially a container management tool which uses the kernel features of namespaces for isolation and cgroups for resource control and accounting. Cgroups are hierarchical groups of processes which can be controlled by cgroups Subsystems or Resource controllers. Subsystems include controllers for CPU, memory, block i/o, devices. With these the particular cgroup can be restricted to certain CPU cores, memory usage can be limited, disk usage can be limited and access to devices can be restricted. The cgroups functionality is exposed through a pseudo filesystem which is found at /sys/fs/cgroup. Cgroups also expose metrics like CPU usage, Block I/O, memory usage through this filesystem. By default, networking is virtualized by creating two virtual ethernet devices, one on the container and the other on the

host connected through a bridge. This configuration allows connectivity between the host and all containers on the host.

2.3 HDFS and Hadoop MapReduce

Apache hadoop is a highly scalable and fault tolerant distributed system for storing and processing large data sets on clusters of commodity hardware. It uses a distributed file system with a unix like interface called HDFS and an implementation of Googles MapReduce programming model. The MapReduce implementation allows for a simplified abstraction of map and reduce tasks which are suitable for processing unstructured data while managing the distributed processing, fault tolerance and redundancy.

2.4 Intel HiBench

HiBench is Hadoop benchmark suite offered by Intel containing typical Hadoop workloads. It includes micro benchmarks, DFS benchmarks, machine learning benchmarks, web search benchmarks and data analytics benchmarks. The input data for the workloads are automatically generated by the launch scripts of the benchmark. It has options for using input/output compression with zlib.

3 Characterizing Performance

As mentioned earlier, the cgroups functionality is exposed through a pseudo-filesystem which also includes accounting of system resource usage. This filesystem is found /sys/fs/cgroup. Under this subdirectory multiple subdirectories like cpu, memory,blkio, cpuacct,etc can be found each of which corresponds to a different cgroup hierarchy. For each container one cgroup should be created under each hierarchy inside another subdirectory called lxc. Thus if you want cpu usage you would have to read /sys/fs/cgroup/cpuacct/lxc/containername/cpu.usage, for number of bytes read or written to disk you have to process /sys/fs/cgroup/blkio/lxc/containername/ blkio.throttle.io_service_bytes,etc.

A python script collects container statistics on the host system as well power readings from the BeagleBone for each user defined interval which can be specified on the command line. The script connects with a data server on the BeagleBone to collect power readings. The output is in the form of .txt files for each container and separate .txt file for the power measurements.

The files are written inside the history directory under a dynamically created subdirectory. The name of this output directory can be also be specified on the command line while running the script, otherwise the default is datetime. The cluster configuration can be specified in a text file named cluster-config.txt with names of each container on a single line. It is a good idea to clear this file with /dev/null before writing a new configuration.

In addition to this, scripts have been developed to automatically deploy a hadoop and spark cluster on lxc containers. These scripts are written in python, bash and tcl-expect. All scripts are available on GitHub

Following is the procedure to create a hadoop cluster and run the scripts:

1. Create a container based Hadoop cluster using lxcdeploy:

   Clone the repository lxcdeploy from GitHub

   `$git clone https://github.com/pbprasad99/lxcdeploy`

   Inside the repository run the script to deploy the cluster

   `$./make-cluster.py -n <ClusterName> -w  <number of workers (N)>`

   This will create containers named Cluster-Name1, Cluster-Name2....Cluster-Name`<w+2>`

   The first container is the hadoop namenode and secondary namenode. The rest are hadoop slaves.

2. You may wish to configure your container to run on certain cores. This can be done with the lxc-cgroup command.

   `$lxc-cgroup -n <ContainerName> cpuset.cpus 0-15`

3. Clone the repository LXC_STATS from github

   `$git clone https://github.com/pbprasad99/LXC_STATS`

4. On the first container of the cluster download HiBench 2.2. After untarring it, edit bin/hibench-config.sh to point HADOOP_HOME to your hadoop directory which lxcdeploy installs in your home directory. Edit the file conf/benchmarks.lst to choose the benchmarks to be executed and HiBench using the script bin/run-all.sh

5. Start the data server on the Beaglebone (Pyserver.py)

6. Start the hadoop daemons by running start-all.sh on the first container

7. Run the script LXC_STATS/lxc_stats_ver3.2.1.py

   `$./lxc_stat_ver3.2.1.py -o <output folder> -t <time interval in seconds>`

   `-i <ip address of beaglebone>`

The script will connect to the data server on the BeagleBone and start capturing power and performance statistics.

8. When the hadoop job terminates stop the script by pressing q. The script will exit gracefully.

LXC_STATS generates an output perfromance data file for each container and a separate file for the power measurements. The power measurements are in mW and include power for the two CPUs, two memory banks and the power for the dual 12v and 5v supply of the hard disk. The container statistics and power measurements are synchronized to the same time stamp.

The data format for the container statistics is a list of comma seperated values: Date Time,Time Elapsed,Epoch TIme,CPU Usage for Container (ns),CPU Usage for host, Disk I/O Read for Container(bytes), Disk I/O Write for Container, Disk I/O Read for host, Disk I/O Write for host, Memory forContainer (bytes),Memory for host, Network Read (bytes), Network Write (bytes), CPU usage for CPU1, CPU usage for CPU2. The data format for the power readings is a list of space delimited values : CPU1, Memory1, CPU2, Memory2, Hard disk 12V, Hard disk 5V supply. These correspond to the two CPU's each with 8 cores, two memory banks and dual power supply for the hard disk.

TimeStamp	CPU util(%)	MemUse(GB)	DiskRd	DiskWr
11:47:04	12.66	15.15	57.72	151.72
11:47:14	11.70	14.78	51.14	4.91
11:47:24	12.58	15.12	51.13	0.78
11:47:34	12.20	15.12	67.38	0.42

Table 1: Sample snapshot of statistics for one of the containers executing the k-means workload. All data values unless indicated are in MBs.

Table 1 shows a sample snapshot of container statistics extracted from performance data for a k-means workload. It should be noted that CPU usage and Disk I/O measurements captured in the file are counters and as such they go on increasing. CPU utilization and Disk Read and Write are calculated over a time interval from the captured performance data.

Memory consumption readings in the performance data file are instantaneous values of memory at that point in time.

TimeStamp	CPU1	MemBank1	CPU2	MemBank2	HardDisk12V	HardDisk5V
11:47:04	46436	7973	46695	6670	3454	3416
11:47:14	46758	6348	47016	6348	4766	1747
11:47:24	45792	6025	44753	6348	5089	2304
11:47:34	47401	6025	46052	6993	4103	1886

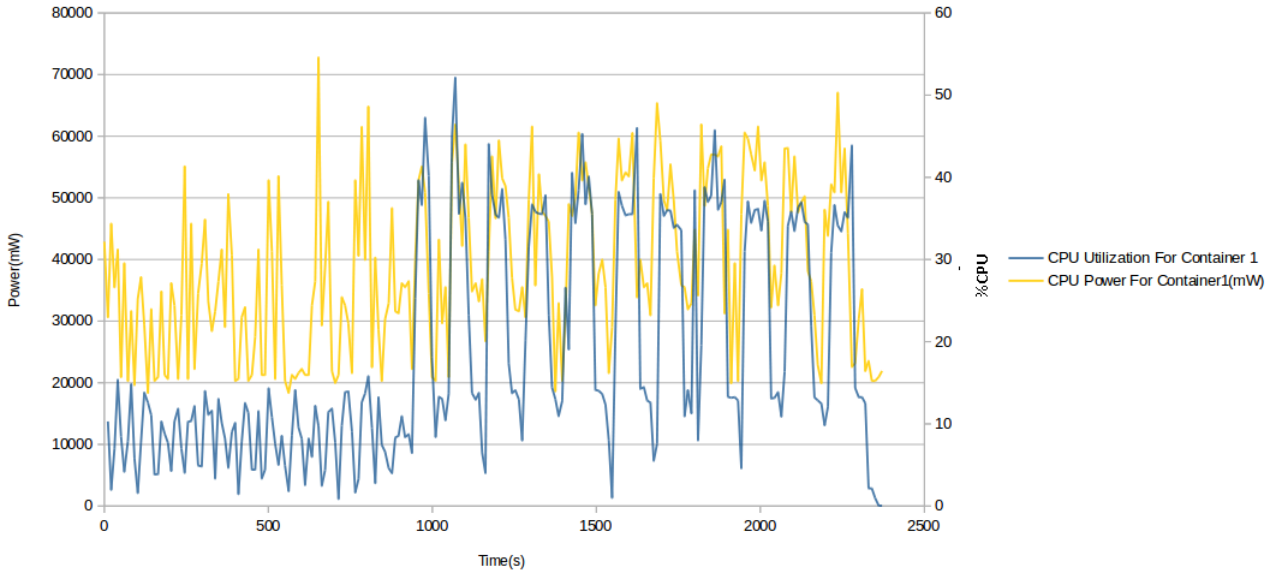
Table 2: Sample snapshot of subsytem power measurements for the k-means workload. All power is measured in mW. Memory is organized as two physical banks. The hard disk has a dual power supply.

Table 2 shows a sample of the power measurements captured for a k-means workload. All readings in the power measurement file are synchronized to the same timestamp as the container statistics.

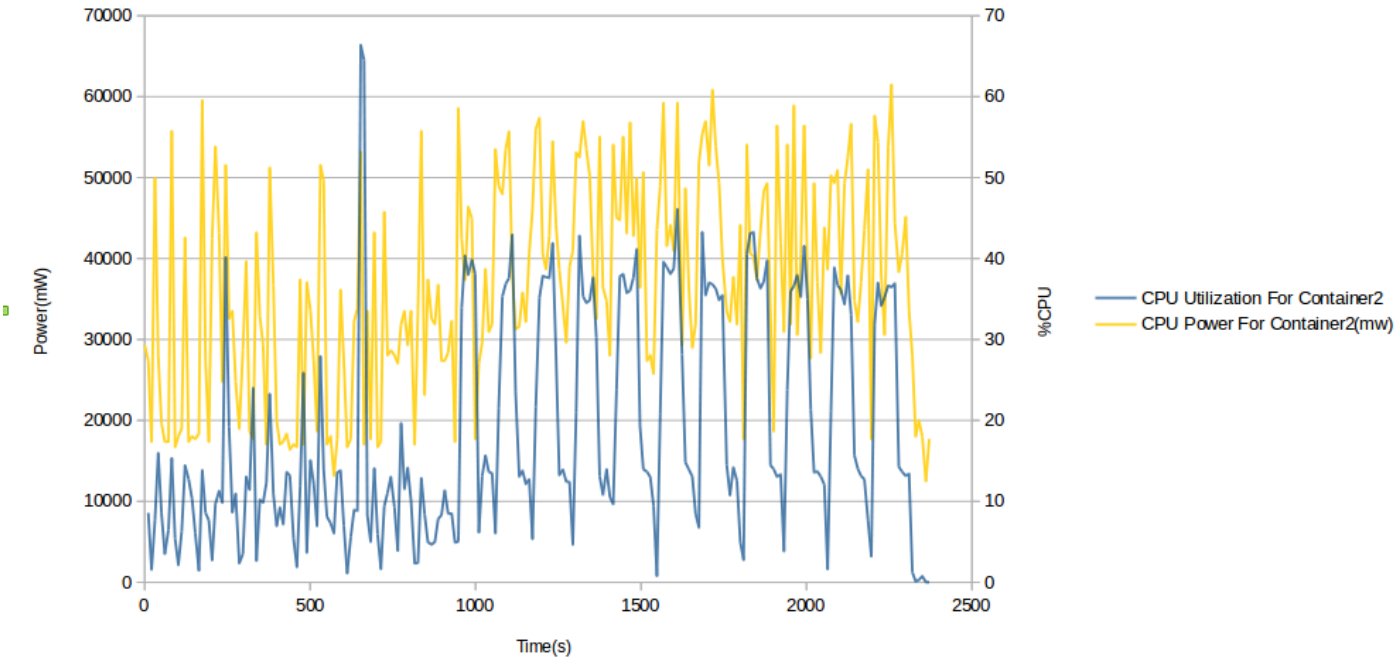
4 Results

The results obtained for the sort workload of Hibench have been presented here. The experimental setup consists of two containers running on one physical CPU each.The workload operates on an input datasize of 24GB which is genereated by the launch scripts of the workload. The hadoop job consists of 16 maps and 48 reduces. Figure 1 shows the CPU utilization and power consumption of Container1 and Container2. From the web UI of the MapReduce JobTracker which is at port 50030 of the first container, we can find the duration of map and reduce tasks running on each container. The map phase lasts for the first 850 seconds. During this time two map tasks of a short duration of 12 to 15 seconds run in parallel on each container along with long running reduce tasks (15 mins) which process local data generated by the map tasks. The reduce phase starts after all map tasks have been executed. Reduce tasks with an average duration of 2mins are executed during this phase.

Figure 2 shows CPU Utilization and Power against the reduce tasks being executed on Container2. The pattern is similar for Container1, except that during the reduce phase only one reduce task is running on Container1. This explains the higher CPU Utilization on Container2



(a) CPU Utilization and Power for Container1



(b) CPU Utlization and Power for Container2

Figure 1: CPU Utilization and Power Consumption. Container1 and Container2 are running on CPU1 and CPU2 respectively.

during the map phase. It can also be seen that the average CPU utilization is lower during the map phase.

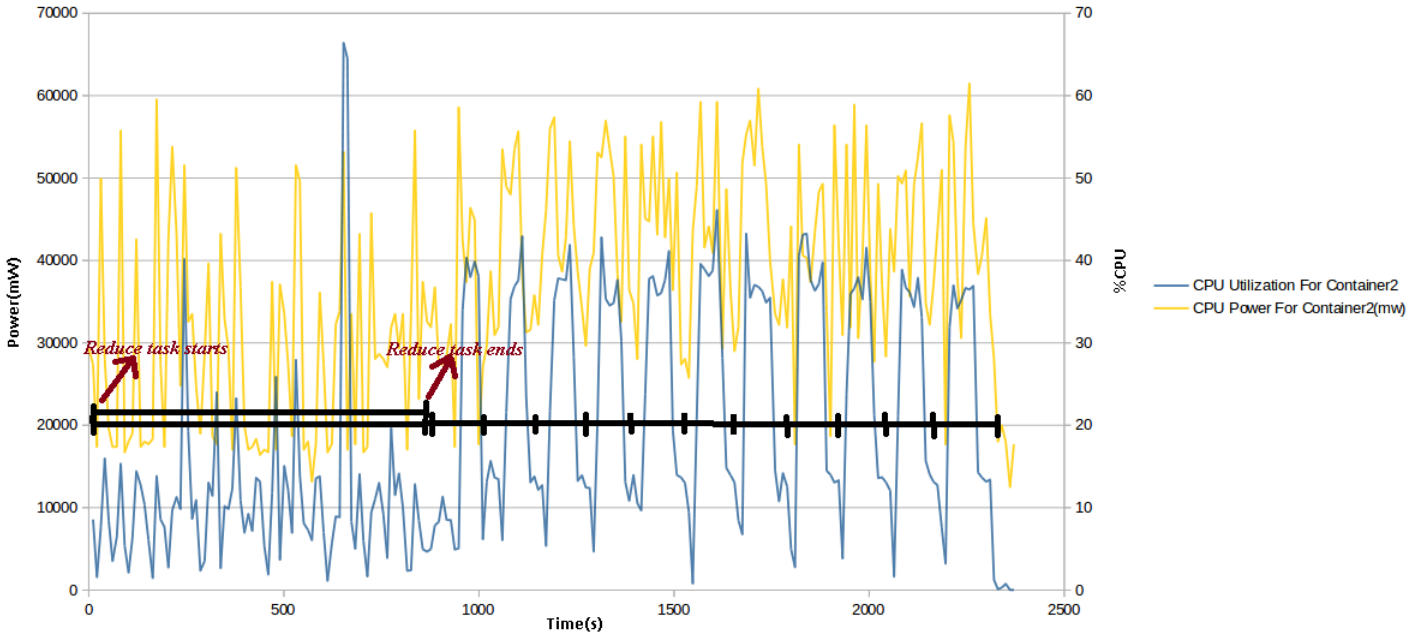
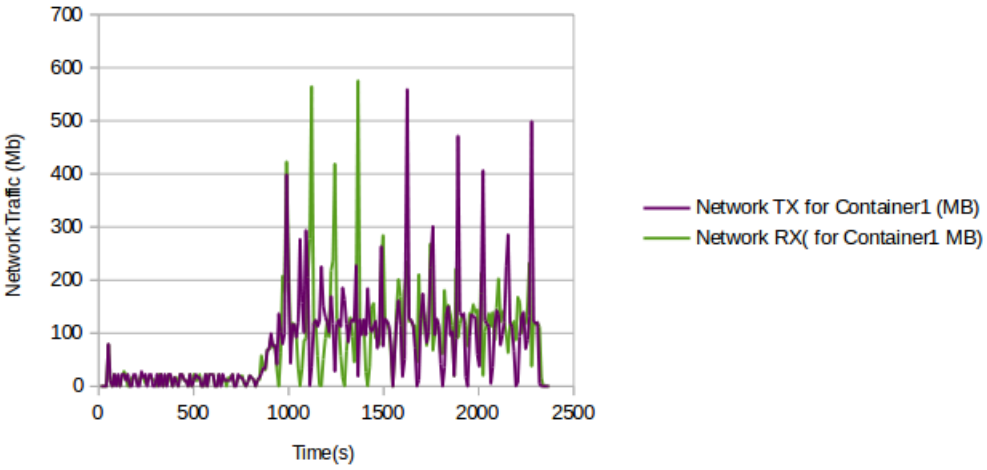
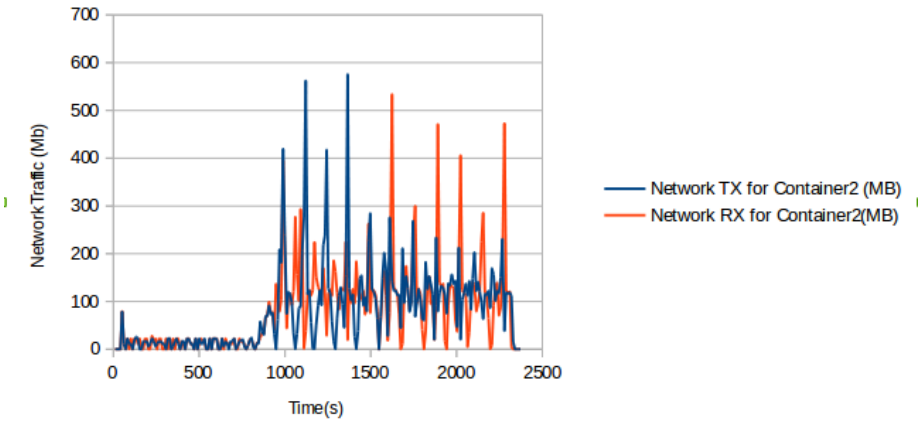


Figure 2: CPU Utilization and Power for Container2 shown against the reducs tasks running on the container.The start and end of reduce tasks are indicated by bars. Initially two reduce tasks are running on the container. Figure 1 shows the CPU utilization of each container against the power consumption of each CPU.

Figure 3 shows the network traffic on each container for the duration of the job. It can be seen that during the map phase the network traffic is low as the map and reduce tasks are operating on local data on each node. The network traffic spikes when the reduce phase begins. Figure 4 shows the netwrork traffic on one of the containers against the reduce tasks being executed on the container.



(a) Network RX and TX in MB for Container1



(b) Network RX and TX in MB for Container2

Figure 3: Network Transmitted and Received data for Container1 and Container2(MB). Network traffic is low during the map phase as tasks are operating on local data. It spikes during the reduce phase

Figure 5 shows the Disk I/O activity on one of the containers and disk power consumption for the duration of the job. The disk access pattern for the map and reduce phase are different. During the map phase a number of short map tasks write data to the disk, whereas during the reduce phase longer running reduce tasks write bigger volumes of data to the disk.

Figure 6 shows the memory consumption of both containers against the power consumption of both memory banks. The shorter map tasks consume less memory on average than the reduce tasks.

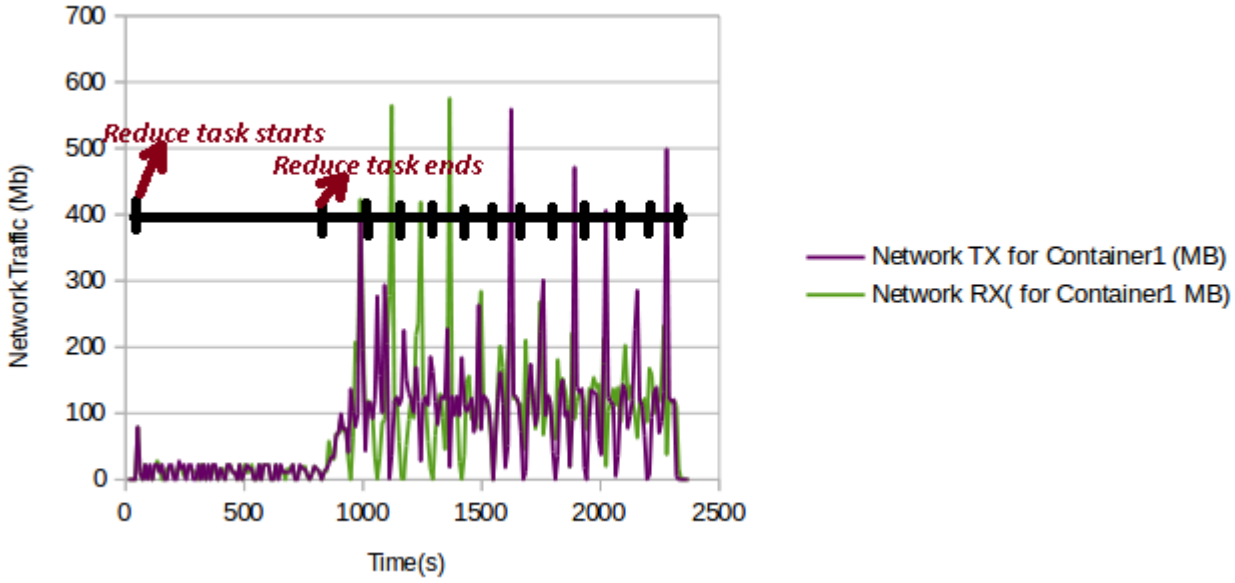


Figure 4: Network Traffic on Container1 shown against Reduce tasks being executed on the container.The first 850 seconds correspond to the map phase.A long running Reduce task executes locally on the data during this phase. Network Traffic spikes during the subsequent reduce phase.

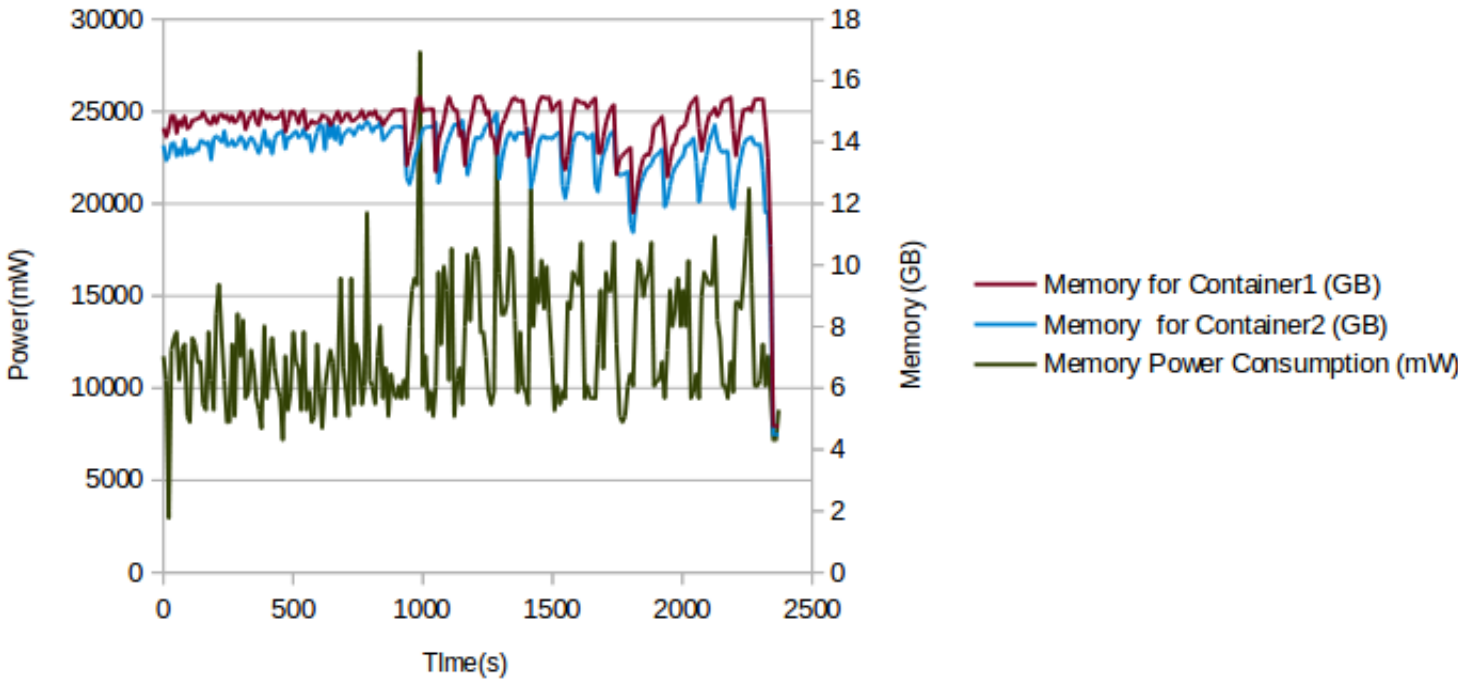


Figure 6: Memory Consumption (GB) and Power(mW) for Container1. The power readings are the sum of power consumed by both memory banks.

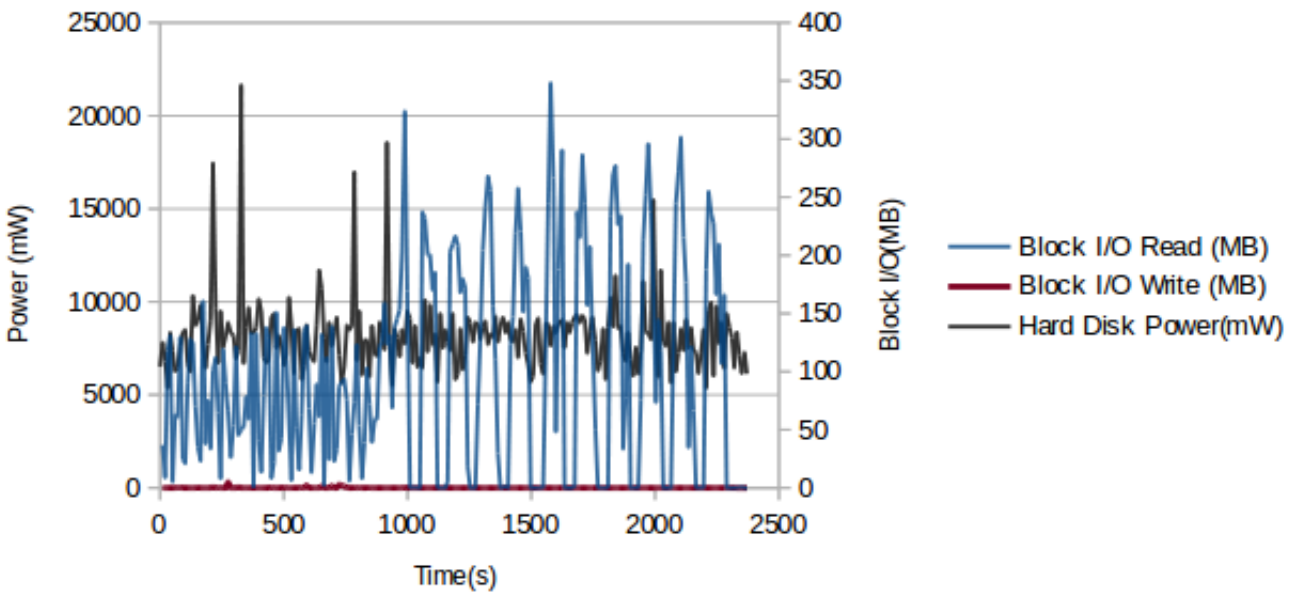


Figure 5: Disk I/O Read and Write (MB) and Disk Power Consumption(mW). This is the sum of readings on the dual power supply for the Hard Disk.

5 Conclusion

This paper describes a methodology to characterize performance and power at the subsystem level of big data workloads on linux containers.Container metrics are collected using the accounting functionality of cgroups and power measurements are collected using a beaglebone based embedded system. The system could be used by developers to determine performance bottlenecks and make system level optimizations, by researchers to collect data for developing statistical/machine learning models that could be used in a system simulator and serves as a platform for performing multi node scalability experiments on a single node.

Future work includes integrating the metrics collection with a real time graphing application like graphite. The system can be integrated with specific big data frameworks like Hadoop and Spark by collecting data about the duration of tasks on each node.

6 References

J. Koomey, *Growth in data center electricity use 2005 to 2010*, CA: Analytics Press. August, vol. 1, 2011

J. Hamilton, *Cooperative expendable micro-slice servers (cems): low cost, low power servers for internet-scale ser- vices*, Proc. of the Conf. on Innovative Data Systems Research, 2009

Shengsheng Huang, Jie Huang,et al, *HiBench: A Representative and Comprehensive Hadoop Benchmark Suite*,Intel Asia-Pacific Research and Development Ltd., Shanghai, P.R.China, 200241

James H. Laros III,et al, *PowerInsight - A Commodity Power Measurement Capability*, Sandia National Laboratories and & Penguin Computing

J. Dean and S. Ghemawat, *Mapreduce: Simplified data processing on large clusters*, Proc. of the 6th USENIX Symposium on Operating System Design and Implementation ,2004

7 Appendix

Metrics collection script:

```
#!/usr/bin/python
from __future__ import division
import time
import re
import os,sys
import socket
import datetime
import tty
import getopt
from select import select
from datetime import datetime,timedelta

#This class helps the script exit gracefully
#http://code.activestate.com/recipes/203830-checking-for-a-keypress-without-stop-the-executio

class NotTTYException(Exception): pass



class TerminalFile:
    def __init__(self,infile):
        if not infile.isatty():
            raise NotTTYException()
        self.file=infile

        #prepare for getch
        self.save_attr=tty.tcgetattr(self.file)
        newattr=self.save_attr[:]
        newattr[3] &= ~tty.ECHO & ~tty.ICANON
 tty.tcsetattr(self.file, tty.TCSANOW, newattr)
```

```
    def __del__(self):
        #restoring stdin
        import tty  #required this import here
        tty.tcsetattr(self.file, tty.TCSADRAIN, self.save_attr)

    def getch(self):
        if select([self.file],[],[],0)[0]:
            c=self.file.read(1)
        else:
            c=''
        return c


##########################################################################

# Connection info
TCP_PORT = 5005
BUFFER_SIZE = 50
MESSAGE = "s"
#CPU LOCATIONS
cpuacct_location='/sys/fs/cgroup/cpuacct'
tot_cpu_location = '/sys/fs/cgroup/cpuacct/cpuacct.usage'
tot_percpu_location = '/sys/fs/cgroup/cpuacct/cpuacct.usage_percpu'
blkio_location = '/sys/fs/cgroup/blkio/lxc'
#BLKIO locations
hdp11_blkio_location='/sys/fs/cgroup/blkio/lxc/hadoop11/blkio.throttle.io_service_bytes'
hdp_tot_blkio_location='/sys/fs/cgroup/blkio/blkio.throttle.io_service_bytes'

#MEMORY locations
memory_location = '/sys/fs/cgroup/memory/lxc/'
tot_memory_location = '/sys/fs/cgroup/memory/memory.usage_in_bytes'
##################################################
```

```
#Show Usage
def usage():
 print "./lxc_stats_ver3.2.py -i <IP Address of beaglebone> -o <Name of output folder -a> -t

#########################################################################
#Check if container exists/is started

def container_exists(container_name):
  return os.path.exists(cpuacct_location + '/lxc/'+ container_name + '/cpuacct.usage')




#########################################################################
# Get virtual inerface for container

def get_virtual_interface(container_name):
 s = os.popen('lxc-info -n ' + container_name)
 link = re.findall("Link:.*\$",s.read().strip(),re.MULTILINE)
 return link[0].split()[1]

 ########################################################################
def get_percpu_usage():
 percpu_usage = []
 f = open(tot_percpu_location)
 list1 = f.read().strip().split()
 map(int,list1)
 list_cpu1 = list1[:16]
 list_cpu2 = list1[16:]
 sum_cpu1 = reduce(lambda x,y : x+y, map(int,list_cpu1))
 sum_cpu2 = reduce(lambda x,y : x+y, map(int,list_cpu2))
 percpu_usage.append(sum_cpu1)
 percpu_usage.append(sum_cpu2)
 return percpu_usage
```

```
#########################################################################
#Collect Container Statisics

def getstats(container_name):
 list = []
 list.append(str(time.time()))
 #CPU Usage for container in microseconds
 f = open(cpuacct_location + '/lxc/'+ container_name + '/cpuacct.usage')
 list.append(f.read().strip())
 f.close()
 # CPU Usage total in microseconds
 f = open(tot_cpu_location)
 list.append(f.read().strip())
f.close()
 # BLKIO for container - READ BYTES
 f = open(blkio_location + '/' + container_name + '/blkio.throttle.io_service_bytes')
 blkio_read =f.readlines()[0].strip().split()[2]
 list.append(blkio_read)
 f.seek(0)
 # BLKIO for container -WRITTEN BYTES
 blkio_write =f.readlines()[1].strip().split()[2]
 list.append(blkio_write)
 f.close()
 #blkio_hdp11_total = int(blkio_hdp11_read) + int(blkio_hdp11_write)
 # BLKIO TOTAL
 #f= open(hdp_tot_blkio_location)
 #total = re.findall("Total.*$", f.read().strip(),re.MULTILINE)
 #total_blkio = total[len(total) -1].split()[1]
 #list.append(total_blkio)

 #BLKIO for SYSTEM - READ BYTES
```

```
f= open(hdp_tot_blkio_location)
blkio_sys_read =f.readlines()[0].strip().split()[2]
list.append(blkio_read)
f.seek(0)
# BLKIO for SYSTEM -WRITTEN BYTES
blkio_sys_write =f.readlines()[1].strip().split()[2]
list.append(blkio_write)
f.close()
#Memory for container
f = open(memory_location + container_name + '/memory.usage_in_bytes')
list.append(f.read().strip())
f.close()

 #Memory Total
f = open(tot_memory_location)
list.append(f.read().strip())
f.close()

virtual_interface = get_virtual_interface(container_name)
#NETSTATS for container - RX
f = open('/sys/class/net/' + virtual_interface + '/statistics/rx_bytes' )
list.append(f.read().strip())
f.close()

#NETSTATS for container - TX
f = open('/sys/class/net/' + virtual_interface + '/statistics/tx_bytes')
list.append(f.read().strip())
f.close()

#PERCPU_USAGE
percpu_usage = get_percpu_usage()
list.extend(percpu_usage)
return list
```

```
#############################################################################
def main():
 #Check if root
 if os.geteuid() !=0:
   print 'Login as root to run this script..'
   sys.exit(0)
 #Local variables
 fldr_name = ' '
 TCP_IP = '10.16.30.9' # Default IP address
 INTERVAL = 15
 # Process flags
 try:
        opts, args = getopt.getopt(sys.argv[1:], "ho:i:at: ")
 except getopt.GetoptError as err:
        # print help information and exit:
        print str(err) # will print something like "option -a not recognized"
        usage()
        sys.exit(2)
 ALL = False
 for o,a in opts:
  if o == "-i":
            TCP_IP = a
  elif o in ("-h"):
            usage()
            sys.exit()
  elif o in ("-o"):
            fldr_name = a
  elif o in ("-a"):
            ALL = True
  elif o in ("-t"):
 INTERVAL = int(a)           #Handle exception
  else:
            assert False, "unhandled option"
```

```
#Check if all containers exist and started
try:

 for container in open('cluster_config.txt'):
  if not container_exists(container.strip()):
    print container.strip() + ' does not exist or is stopped \n'
    sys.exit(1)
except IOError:
    print 'Could not find cluster_config.txt in current directory.'
    sys.exit(1)
#Connect to server
print 'Connecting to server..'

#Check format of IP address
try:
    socket.inet_aton(TCP_IP)
except socket.error:
    print "Check IP address\n"
    sys.exit(1)

try :
    so = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    so.connect((TCP_IP, TCP_PORT))
except socket.error as err :
print "Could not connect to server"
    print str(err)
    sys.exit(1)
print 'Connection established'
stats_print =[]


#Read cluster containers from file
```

```
f = open('cluster_config.txt')
str_tmp = f.read()
str_tmp.strip()
container_list = str_tmp.split()

#create folder
if fldr_name == ' ':
 fldr_name = datetime.now().strftime("%y-%m-%d-%H-%M-%S")

os.makedirs('./history/'+ fldr_name)
# Make file to write power measurements
f_pwr = open(os.path.join('./history/',fldr_name,'pwr.txt'),"a")

#Get stats for each container
s=TerminalFile(sys.stdin)
print "Press q to quit..."
#Get start time
datestart = datetime.now()

while s.getch()!="q":

so.send(MESSAGE)
  data = so.recv(BUFFER_SIZE)
  f_pwr.write(data)
  f_pwr.write('\n')
  datenow = datetime.now()
  time_elapsed = datenow - datestart

  for container in open('cluster_config.txt'):
   stats_print =[]
   with open(os.path.join('./history/',fldr_name,container.strip()+'.txt'),"a") as containe
     stats_print.append(str(datenow))
     stats_print.append(str(time_elapsed.seconds))
```

```
      raw_stats = getstats(container.strip())

      stats_print.extend(raw_stats)
      stats_str = ','.join(map(str,stats_print))
      container_file.write(stats_str)
      container_file.write('\n')
   time.sleep(INTERVAL)

 print '--END--'



if __name__ == "__main__":main()
```