COMPARISON OF PERFORMANCE OF BIG DATA APPLICATIONS IN DIFFERENT ENVIRONMENTS

by

Devang Motwani
A Thesis
Submitted to the Graduate Faculty of George Mason University in Partial Fulfillment of The Requirements for the Degree of Master of Science in Computer Engineering

Committee:

Dr. Houman Homayoun, Thesis Director
Dr. Avesta Sasan, Committee Member
Dr. Setareh Rafatirad, Committee Member
Dr. Monson Hayes, Department Chair
Dr. Kenneth S. Ball, Dean, Volgenau School of Engineering

Date: 06/22/18

Summer Semester 2018
George Mason University Fairfax, VA
Comparison of Performance of Big Data Applications in Different Environments

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at George Mason University

by

Devang Motwani
Bachelor of Engineering
Gujarat Technological University, 2014

Director: Dr. Houman Homayoun, Professor
Department of Computer Engineering

Summer Semester 2018
George Mason University
Fairfax, VA
DEDICATION

I dedicate this work to my parents, Mukesh Motwani and Kanta Motwani, to my sister Ketki Motwani and my love Khushi Dudani. I would also like to dedicate this to my elders in Extended Family and thank them for all their support and also to my friends.
ACKNOWLEDGEMENTS

I would like to express my gratitude to my advisor Dr. Houman Homayoun for his guidance and encouragement throughout the course of my research. His feedback and direction have been helpful to great extent. I would also like to thank Hosein Mohammadi Makrani for his immense help and direction and it was great to have worked with you. In addition to this, I would like to thank Dr. Sai Manoj P D for his help and support and all the members of the GOAL Lab for their time and help.
TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>ix</td>
</tr>
<tr>
<td>Abstract</td>
<td>xi</td>
</tr>
<tr>
<td>INTRODUCTION</td>
<td>13</td>
</tr>
<tr>
<td>VIRTUALIZATION</td>
<td>14</td>
</tr>
<tr>
<td>Benefits of Virtualization</td>
<td>15</td>
</tr>
<tr>
<td>Virtual Machines</td>
<td>16</td>
</tr>
<tr>
<td>Type 0 Hypervisor</td>
<td>18</td>
</tr>
<tr>
<td>Type 1 Hypervisor</td>
<td>19</td>
</tr>
<tr>
<td>Type 2 Hypervisor</td>
<td>20</td>
</tr>
<tr>
<td>Paravirtualization</td>
<td>21</td>
</tr>
<tr>
<td>Programming environment virtualization</td>
<td>21</td>
</tr>
<tr>
<td>Emulators</td>
<td>21</td>
</tr>
<tr>
<td>Application containment</td>
<td>22</td>
</tr>
<tr>
<td>Containers</td>
<td>24</td>
</tr>
<tr>
<td>Namespaces – Limits what you can see</td>
<td>24</td>
</tr>
<tr>
<td>Cgroups – Limit how much you can use</td>
<td>31</td>
</tr>
<tr>
<td>Copy-on-write</td>
<td>34</td>
</tr>
<tr>
<td>What is a container?</td>
<td>34</td>
</tr>
<tr>
<td>Docker</td>
<td>35</td>
</tr>
<tr>
<td>INTRODUCTION TO BIGDATA FRAMEWORKS</td>
<td>37</td>
</tr>
<tr>
<td>Apache Hadoop</td>
<td>40</td>
</tr>
<tr>
<td>HDFS (Hadoop Distributed File System)</td>
<td>41</td>
</tr>
<tr>
<td>MapReduce</td>
<td>43</td>
</tr>
<tr>
<td>YARN</td>
<td>46</td>
</tr>
<tr>
<td>Apache Spark</td>
<td>47</td>
</tr>
<tr>
<td>RDD – Resilient Distributed Data</td>
<td>50</td>
</tr>
<tr>
<td>Cluster Mode Overview</td>
<td>50</td>
</tr>
<tr>
<td>Apache Flink</td>
<td>51</td>
</tr>
<tr>
<td>System Architecture</td>
<td>51</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>--------------</td>
<td>------</td>
</tr>
<tr>
<td>Wordcount</td>
<td>90</td>
</tr>
<tr>
<td>IPC</td>
<td>90</td>
</tr>
<tr>
<td>L3 Hit Rate</td>
<td>92</td>
</tr>
<tr>
<td>L2 Hit Rate</td>
<td>93</td>
</tr>
<tr>
<td>C0 Residency</td>
<td>95</td>
</tr>
<tr>
<td>CPU Energy</td>
<td>96</td>
</tr>
<tr>
<td>CPU Watt</td>
<td>97</td>
</tr>
<tr>
<td>DRAM Energy</td>
<td>99</td>
</tr>
<tr>
<td>DRAM Watt</td>
<td>99</td>
</tr>
<tr>
<td>BW</td>
<td>100</td>
</tr>
<tr>
<td>BFS</td>
<td>102</td>
</tr>
<tr>
<td>IPC</td>
<td>102</td>
</tr>
<tr>
<td>L3 Hit Rate</td>
<td>103</td>
</tr>
<tr>
<td>L2 Hit Rate</td>
<td>104</td>
</tr>
<tr>
<td>C0 Residency</td>
<td>105</td>
</tr>
<tr>
<td>CPU Watt</td>
<td>106</td>
</tr>
<tr>
<td>DRAM Watt</td>
<td>107</td>
</tr>
<tr>
<td>BW</td>
<td>107</td>
</tr>
<tr>
<td>Grep</td>
<td>109</td>
</tr>
<tr>
<td>IPC</td>
<td>109</td>
</tr>
<tr>
<td>L3 Hit Rate</td>
<td>110</td>
</tr>
<tr>
<td>L2 Hit Rate</td>
<td>111</td>
</tr>
<tr>
<td>C0 Residency</td>
<td>112</td>
</tr>
<tr>
<td>CPU Watt</td>
<td>113</td>
</tr>
<tr>
<td>DRAM Watt</td>
<td>114</td>
</tr>
<tr>
<td>BW</td>
<td>115</td>
</tr>
<tr>
<td>CC</td>
<td>117</td>
</tr>
<tr>
<td>Kmeans</td>
<td>118</td>
</tr>
<tr>
<td>PageRank</td>
<td>120</td>
</tr>
<tr>
<td>FUTURE WORK</td>
<td>123</td>
</tr>
<tr>
<td>APPENDIX</td>
<td>124</td>
</tr>
<tr>
<td>References</td>
<td>125</td>
</tr>
</tbody>
</table>
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1: Non-Virtual system vs Virtual System [3]</td>
<td>17</td>
</tr>
<tr>
<td>Figure 2: Type 0 Hypervisor [3]</td>
<td>18</td>
</tr>
<tr>
<td>Figure 3: Type 1 Hypervisor</td>
<td>19</td>
</tr>
<tr>
<td>Figure 4: Type 2 Hypervisor</td>
<td>20</td>
</tr>
<tr>
<td>Figure 5: Process Isolation [4]</td>
<td>25</td>
</tr>
<tr>
<td>Figure 6: Visual Representation of Namespace [8]</td>
<td>27</td>
</tr>
<tr>
<td>Figure 7: PID Namespace</td>
<td>29</td>
</tr>
<tr>
<td>Figure 8: IBM Big Data and Analytics Hub [12]</td>
<td>38</td>
</tr>
<tr>
<td>Figure 9: Information Storage Capacity [14]</td>
<td>39</td>
</tr>
<tr>
<td>Figure 10: HDFS Architecture [15]</td>
<td>42</td>
</tr>
<tr>
<td>Figure 11: Apache Hadoop YARN [15]</td>
<td>46</td>
</tr>
<tr>
<td>Figure 12: Apache Spark [17]</td>
<td>48</td>
</tr>
<tr>
<td>Figure 13: Apache Spark with HDFS [17]</td>
<td>49</td>
</tr>
<tr>
<td>Figure 14: Cluster Mode Architecture [16]</td>
<td>51</td>
</tr>
<tr>
<td>Figure 15: Apache Flink Software Stack [18]</td>
<td>52</td>
</tr>
<tr>
<td>Figure 16: Apache Flink Process Model [18]</td>
<td>55</td>
</tr>
<tr>
<td>Figure 17: Governors</td>
<td>64</td>
</tr>
<tr>
<td>Figure 18: Server info</td>
<td>65</td>
</tr>
<tr>
<td>Figure 19: Abstract View of Code</td>
<td>81</td>
</tr>
<tr>
<td>Figure 20: Wordcount - IPC</td>
<td>91</td>
</tr>
<tr>
<td>Figure 21: Wordcount - L3 Hit Rate</td>
<td>92</td>
</tr>
<tr>
<td>Figure 22: Wordcount - L2 Hit Rate</td>
<td>94</td>
</tr>
<tr>
<td>Figure 23: Wordcount - C0 Residency</td>
<td>95</td>
</tr>
<tr>
<td>Figure 24: Wordcount - CPU Energy</td>
<td>97</td>
</tr>
<tr>
<td>Figure 25: Wordcount - CPU Watt</td>
<td>98</td>
</tr>
<tr>
<td>Figure 26: Wordcount - DRAM Energy</td>
<td>99</td>
</tr>
<tr>
<td>Figure 27: Wordcount - DRAM Wattage</td>
<td>100</td>
</tr>
<tr>
<td>Figure 28: Wordcount - BW</td>
<td>101</td>
</tr>
<tr>
<td>Figure 29: BFS - IPC</td>
<td>102</td>
</tr>
<tr>
<td>Figure 30: BFS - L3 Hit Rate</td>
<td>103</td>
</tr>
<tr>
<td>Figure 31: BFS - L2 Hit Rate</td>
<td>104</td>
</tr>
<tr>
<td>Figure 32: BFS - C0 Residency</td>
<td>105</td>
</tr>
<tr>
<td>Figure 33: BFS - CPU Watt</td>
<td>106</td>
</tr>
<tr>
<td>Figure 34: BFS - DRAM Wattage</td>
<td>107</td>
</tr>
<tr>
<td>Figure 35: Grep - IPC</td>
<td>109</td>
</tr>
<tr>
<td>Figure 36: Grep - L3 Hit Rate</td>
<td>110</td>
</tr>
<tr>
<td>Figure 37: Grep - L2 Hit Rate</td>
<td>111</td>
</tr>
<tr>
<td>Figure 38: Grep - C0 Residency</td>
<td>112</td>
</tr>
<tr>
<td>Figure 39: Grep - CPU Wattage</td>
<td>114</td>
</tr>
</tbody>
</table>
ABSTRACT

COMPARISON OF PERFORMANCE OF BIG DATA APPLICATIONS IN DIFFERENT ENVIRONMENTS

Devang Motwani, M.S.
George Mason University, 2018
Thesis Director: Dr. Houman Homayoun

Virtualization is utilized by all the Tech companies in their software product development and deployment on a regular basis. Containers lead to a huge hype due to the success of Docker, a tool to use containers, that made software development and deployment easier as well modular. Prior to Containers, Virtual Machines were almost the only form of Virtualization that was used and was considered stable. Due to these new emerging technologies, there has been a lot of research published over comparison in performance of Host, Virtual Machine and Containers. But there has not been much work done in the area of Big Data applications such as Microbenchmarks, Graph applications, Search applications and Machine learning running in these different environments over frameworks such as Apache Hadoop, Spark, Flink, etc. so that we can understand how different components of computer architecture are affected. This thesis compares the performance of these standard big data applications on host, virtual machines and containers (specifically Docker) by running these applications and fetching hardware
counter values of C0 state, L3/L2 caches hits, CPU power etc. which would help create a pictorial representation of their comparison. This comparison has been able to justify that application running on Apache Flink, considering some trade-offs, executes Graph, Search and Machine Learning Applications execute faster and more efficiently than on Apache Spark and Hadoop MapReduce. However, Microbenchmark applications perform faster and efficiently on Apache Spark than on Hadoop MapReduce or Apache Flink. Previous research has stated that host and container environment would have almost same performance and there would be some overhead faced in running applications on Virtual Machines but looking at the results, it can be stated that nothing can specifically be drawn about the performance in different environments, different applications running in different frameworks have different performance and this has been discussed and detailed in this work.
INTRODUCTION

Virtualization has changed the way today applications are used. As a matter of fact, all the servers that are utilized to run various applications are running on one or the other virtualized platform. Hence, performance of applications on these different platforms is vital to companies as well as developers. There is some overhead that is added, and consideration of that overhead is important for certain type of applications which have to be soft real-time systems, while building or running these applications. There has been previous work published wherein metrics have to been gathered to compare various applications while running them on Host, Virtual and Docker (Container) environment. [1]. There has also been research on running applications on para-virtualized platforms. However, there is still scope for comparison of Bigdata applications while running them on Host, Virtual and Docker environment and this thesis document, using various metric tools and benchmarks, displays comparison of these applications and discusses the probable effects of virtualization on performance of each application while running on different platforms.
VIRTUALIZATION

Virtualization is a concept to abstract the physical computing resources from the end-user/s or application utilizing the underlying hardware. That being said, the application or user would only be able to determine what virtual computing resources have been assigned to it, which in physical terms can either be a part of the physical resources of a machine or can also be a cluster of physical machines virtualized to look as a single unit. So, the application will not be able to figure out any details about the hardware due a layer indirection being present above the hardware and below the assigned virtual hardware. [1]

Another loose definition of Virtualization as defined by Amit Singh, which is exactly accurate to define it, is:
“Virtualization is a framework or methodology of dividing the resources of a computer into multiple execution environments, by applying one or more concepts or technologies such as hardware and software partitioning, time-sharing, partial or complete machine simulation, emulation, quality of service, and many others.” [2]
This definition clearly presents all the possible scenarios by which Virtualization in systems is achieved.
Benefits of Virtualization

- Virtual machines can be used to consolidate the workloads of several under-utilized servers to fewer machines, perhaps a single machine (server consolidation). Related benefits (perceived or real, but often cited by vendors) are savings on hardware, environmental costs, management, and administration of the server infrastructure.

- Legacy applications can be supported which are not supported by the newer hardware or operating systems.

- Combine lighter workloads running on multiple servers that are under-utilizing the physical server to run on a single machine by migrating the virtual machines and increasing the efficiency of the server.

- Migration of Virtual Machines for server maintenance or repairs.

- Virtual Machines can be useful in running different types and versions of Operating Systems on a single Server, which is a common scenario required while testing a newly developed application to run successfully on different platforms.

- Virtual machines can help in isolated environments to not affect current host and can be helpful in testing applications in isolated environment or to run any application that is potentially infected by malware or is untrusted.

- Virtual Machine manager can also be useful in dynamically building a Virtual Machine as the need to prepare one is required by an application.
- Virtual machines can isolate what they run, so they provide fault and error containment. You can inject faults proactively into software to study its subsequent behavior.

- Virtual machines allow for powerful debugging and performance monitoring. You can put such tools in the virtual machine monitor, for example. Operating systems can be debugged without losing productivity or setting up more complicated debugging scenarios.

- Virtualization can enable existing operating systems to run on shared memory multiprocessors. [2]

**Virtual Machines**

The primary idea using which Virtual Machines are implemented is to abstract the underlying hardware resources in one or more execution environments such that each of the execution environment has an illusion of operating on its own physical hardware but in fact they are operating on a single or a cluster of physical machines without being aware of the actual hardware involved and are only aware of the virtual hardware assigned to that specific machine.

Consider it as a machine operating in a sandbox environment and completely isolated from its host and other environments unless it has been set to communicate with host and other environments by setting up some sought of networking among them.
To achieve this, a layer of abstraction is involved that helps in managing the virtual hardware for one or more execution environments. This managing component is called as a **Virtual Machine Manager or a Hypervisor** which creates and environment similar to the host. Based on where this layer of abstraction is placed in the stack, with support to out-of-the-box guest operating systems, we have following types of hypervisors:
**Type 0 Hypervisor**

These types of hypervisors are firmware based and provide management of virtual machines from the firmware. To achieve virtualization, the hardware components are partitioned to support different operating systems. Hence, these hypervisors are unable to take advantage of the hardware to the maximum efficiency and performance. Oracle LDOMs is an example of such a hypervisor. [3]

![Figure 2: Type 0 Hypervisor](image-url)
Type 1 Hypervisor

These hypervisors are specially built software similar to operating systems providing all the components of operating systems such as scheduling, memory management, etc. for the purpose of virtualization only. They can support numerous virtual machines at a time in such a way that they are able to get the best out of the hardware by supporting requests from different operating systems efficiently as well as consecutively. VMware ESX, Citrix XenServer, Microsoft Hyper-V, Linux distro with KVM etc. are examples of Type 1 Hypervisor. [3]
Type 2 Hypervisor

These hypervisors are normal application software that can operate on any operating system hence providing VMM similar functionalities to the guest operating system. However, the hardware on which this software can run should be able to support virtualization. VMware Workstation, VMware Fusion (for Mac), Oracle VirtualBox and many more are examples of Type 2 Hypervisor. [3]

![Figure 4: Type 2 Hypervisor](image)
There are other virtualization techniques utilized which specifically cannot be classified under the above classification as they include some minor modifications to the guest operating systems to have more efficient virtualizations. Also, there are applications in which following options can operate more efficiently than the ones discussed earlier: [3]

**Paravirtualization**

In this virtualization technique, guest operating systems are modified to work with the virtual machine manager so as to improve efficiency and performance and reduce the overhead involved. A version of Xen is available that supports virtualization.

**Programming environment virtualization**

Virtual machine manager does not virtualize real hardware in this kind of virtualization. Instead, they are built to extend support for applications of specific programming languages by providing an optimized virtual machine for the language. Java Virtual Machine (JVM) and Microsoft .NET are examples that utilize this technique.

**Emulators**

This is a piece of software that allows to run applications of a specific hardware on a completely different hardware. There are situations in which computer engineers have to test a particular application built for ARM Architecture on x86 Hardware. In such situations, emulators are used. QEMU, an open source emulator, is an example of this type.
**Application containment**

This is not a virtualization technique but instead it provides virtualization-like features by separating applications from operating systems. Examples are Solaris Zones, BSD Jails, Docker etc. contain applications by isolating them from the host Operating System and making them more manageable. This concept is discussed further in this chapter. [3]

Virtual Machines are needed when we have to use cross-platform Operating Systems, which means when a Windows VM needs to run on a Linux machine or vice-versa. But for scenarios where a Linux VM needs to run on a Linux Machine, or any Operating System that is running on VM requires the same kernel, there are other options that are performance efficient as well as provide environment similar to VM with less overhead. For that to be discussed lets first consider the overhead that we have to deal with while using Virtual Machines.

So, reviewing the concept of Virtual Machines, we can consider the following levels:

- **Host User level**: This level technically consists of the Guest user level and Guest kernel level. All the kernel level (privilege) calls of the guest are sent to the Host kernel level for it to natively run on the CPU.
  - **Guest user level**: All the user level applications of the guest operating system operate in this level.
  - **Guest kernel level**: All superuser privilege calls occur of guest operating systems occur in this level which are later directed to the Host kernel by
– trap-and-emulate
– binary translation
– (in the modern systems with virtualization support) the use of special virtualization instructions supported by the hardware (VT-x or AMD-v).

- Host Kernel level: This level deals with the privilege instructions of the Host. The instructions without privilege of the any of the above levels run natively on hardware.

Considering number of levels that any user-level applications have to go through to run on the hardware, there is some latency involved which certainly results in a performance hit for the application. Over and above that, the overhead of setting up a virtual machine to run an application and then delete it if the virtual machine corrupts or migrating a whole virtual machine to another server for maintenance will increase the overhead and responsiveness especially in production environment. To solve the issue, but keep the virtualization-like features, the concept of containers has served promising and there are various tools that support the use of containers as well as various huge communities have been involved in developing as well sustaining this new concept and other technologies have been developed to orchestrate the containers so as to achieve better results and make maintenance easy. So, next topic will deal with some background of containers and also discuss how docker containers work and what is the underlying concept using which they are implemented under the hood.
Containers

As discussed earlier, for certain application scenarios we use guest operating system same as the one used for Host to achieve isolation, multiple execution environment, etc. for the purpose of testing or production. But for such scenarios, containers are hugely employed in testing as well as production environment. Hence to understand how exactly containers are used, we need to discuss what are namespaces and cgroups in Linux kernel and why are they implemented. [4]

Namespaces – Limits what you can see

Inspired by namespace functionality of Plan 9 (a distributed operating system) developed at Bell Labs, namespaces were implemented in the Linux Kernel with primary initiative to partition kernel resources such that a set of processes can see one set of resources while another set sees a completely different set and does not see the first one. [5]

So, essentially that means a process, or a set of processes are isolated from the rest of the operating system as shown in the following figure:
P = Process to the Linux Operating System

CP = Container or namespace processor

As shown in the above figure, all the processes, titled as “P*” share address space, share same namespace as regular processes on an operating system. Suppose that we want to isolate a process from other processes and create a sandbox or isolated environment so that the process has its own virtual system (process namespace, pid namespace, etc) inside it. P1, P2 and P5 are regular processes however P3 and P4 look like processes to P1, P2 and P5 however they have a virtual system inside them that further has different processes running inside them. This virtual system is created by using namespaces. The “CP” processes running inside the “P” processes have no communication with “P” processes except if “P” process is the parent of “CP”. Hence, namespaces are utilized to build containers. But there is another feature of Linux kernel
that is being utilized for the purpose of governing resource management and control which will be discussed in a further topic. [3]

So, how are these namespaces created? Well, there are two ways of creating a process in Linux – fork() and clone(). With fork(), what happens is a copy of the parent is created and then using exec() call a new binary is replaced in the address space of the child. The thing to note is that the parent and the child run in different address space and nothing among them is shared. Unlike fork(), using clone() function call, when a child is created, it allows the child to share parts of its execution context with its parent (calling process) such as the table of file descriptors, the virtual address space and the table of signal handlers.

Hence, clone() is utilized to implement threads and there are flags implemented in Linux for the purpose of creating namespaces with clone() system call. [6][7]
To achieve isolation various types of namespaces are utilized. As of Linux Kernel 4.10, there are following namespaces. They are:

- Mount (mnt)

  This namespace controls mount point. While creating mounts, they are moved from the current mount namespace and are copied to the new namespace, however mount points created afterwards do not propagate between namespaces.

Figure 6: Visual Representation of Namespace [8]
The flag value that is used with the clone() system call for creating a new mount namespace is CLONE_NEWNS.

- Process ID (pid)

The PID namespace provides a completely new set of process IDs by creating a namespace isolated from the current namespace. The first process created inside the namespace is assigned process ID 1 and receives similar treatment as init() but it need not be init(). However, all the processes created in this namespace can be seen by the kernel and they are assigned PID in the primary namespace as well though the secondary namespace is unaware about those process IDs. This is explained clearly by the following example figure:
The flag value that is used with the clone() system call for creating a new pid namespace is CLONE_NEWPID.

Figure 7: PID Namespace
• **Network (net)**

Network namespaces are utilized to virtualize the network stack. When a network namespace is created it contains only a loopback interface. Each and every network interface can be present in exactly one namespace and it can be moved among namespaces. Each namespace, just like a network adapter, will have its own routing table, a set of IP addresses, firewall, socket listing, etc. network-related resources. This works with the same pattern as having a private router to setup a network. When a network namespace is destroyed, all virtual interfaces within that would be destroyed too and any physical interfaces are moved within it back to the initial namespace state.

The flag value that is used with the clone() system call for creating a new net namespace is CLONE_NEWNET.

• **Interprocess Communication (ipc)**

IPC namespaces are utilized to isolate certain IPC resources, such as System V IPC objects POSIX message queues. Each namespace has its own set of System V IPC identifiers. Each also own POSIX message queue filesystem. Objects created in an IPC namespace will be only visible to all other processes that are in that namespace but will not be visible to processes in other IPC namespaces.

The flag value that is used with the clone() system call for creating a new net namespace is CLONE_NEWIPC.

• **Unix Timesharing System (uts)**

This namespace allows a single system to have different domain names for different processes.
The flag value that is used with the clone() system call for creating a new net namespace is CLONE_NEWUTS.

- Used ID (user)

User namespaces help isolate security-related identifiers and attributes such as user IDs and group IDs, the root directory and capabilities. Any process's user ID and also group ID can be different inside the namespace and outside a user namespace. A process can have a normal unprivileged user ID outside a user namespace while at the same time have a user ID of 0 inside the namespace which means that the process has full privileges for operations inside the user namespace but is unprivileged for operations outside the namespace.

The flag value that is used with the clone() system call for creating a new net namespace is CLONE_NEWUSER. [6][7][9]

**Cgroups – Limit how much you can use**

- Memory cgroup

Memory cgroup is utilized to for the following purposes:

  o Accounting

    - We can do accounting by monitoring the number of pages that are used by a group that we are monitoring using memory cgroup.

    - Each page is charged to a group.

    - Also, same page can be shared among multiple groups. So, one of the group is charged for that and then if that group has stopped using the page then the charge will be moved to another group that is using the page.
- Limits
  - Limits are optional. Each group has individual limits – Hard limit and Soft limit.
    - Hard limit: If the group goes beyond hard memory limit, the process group gets killed.
    - Soft limit: This comes into picture when the kernel is going to be out of memory and it needs to free memory. It will check if the group is using more than the assigned soft limit and hence swap out pages that are beyond the limit so as to make space for important pages.

Memory cgroups do add some overhead for when the kernel swaps out pages it has to update various hardware resources for that purpose and also there is latency involve due to the swapping.
- Cpu cgroup
  - This cgroup helps us track the CPU usage for the group of process. It can allot to set weights, but it does not allow you to set limits.
- HugeTLB cgroup
  - As we are aware, there are multiple page sizes that a process can ask for from the kernel based on what size it requires for it running. Normally, the process can use all the pages it wants but to make sure that all the huge pages are not utilized by the same process, this cgroup allows to control the assignment of the Huge pages to the
process so that the huge pages are evenly allocated in the operating system and not all the pages are utilized by the same process.

- **Cpuset cgroup**

  It allows to set or pin a group of processes to a CPU or set of CPUs. We can the following features with this cgroup:
  
  - This allows us to reserve CPUs for specific apps.
  - Stop the bouncing of a process among the CPUs.
  - This is also true for NUMA (Non-Uniform Memory Architecture) Systems.

- **Blkio cgroup**

  This cgroup helps in the following:
  
  - To keep track of I/Os for each group.
  - Set throttle limits for each group.
  - Set relative weights.

- **Net_cls and net_prio cgroup**

  This cgroup lets you keep control of the network parameters and its functions are as follows:
  
  - Set priority for traffic generated by processes in the group.
  - Net_cls helps assign traffic to a class.
  - Net_prio helps assign traffic to a priority.

- **Device cgroup**

  This cgroup lets you control which container can read or write on devices.
• Freezer cgroup

  This allows to freeze the container or and then unfreeze the container.

[9][10]

**Copy-on-write**

  This is one of the other key features that is important to understand when looking at the building block of containers. This is the feature that helps you achieve the following:

  • Create a new container instantly which saves time and space on the machine by not building the whole filesystem.

  • It also keeps a track of changes in the system. This is one of the important features, as if nothing has changed, the previous image is pulled from the cache that saves a lot of time in building containers.

  • Different containers tools employ different methods such as AUFS, overlay, ZFS and many more.

  • Container running time and booting time is also reduced along with reduction in container image sizes by reuse of previously build images. [9][10]

**What is a container?**

  Container is an amalgamation of Linux features – namespaces, cgroups and copy-on-write which are put together. To put it as a concept, a notion of the container is a process or a set of processes running in an isolated environment without affecting or knowing anything outside the environment. There are different tools available for using
containers that under the hood use these concepts to achieve virtualization-like environment though each of them works in their own way in their management. The different tools are:

- LXC
- System-nspawn
- Docker Engine
- OpenVZ
- Rkt, runC

Each of them has some advantages over the other. Docker, among them, has a huge community as well support for containers. The implementation of Docker makes it light-weight as well as easy to use and for a fleet of these containers there are other platforms that support Docker and has become the industry standard for production as well as testing environments. For the purpose of the performance experiments, Docker has been chosen over other tools and will be further expanded on now. [4][9]

**Docker**

An open-source platform for developing, shipping, and running applications is basically what Docker is. By taking advantage of Docker’s methodologies for shipping, testing, and deploying code quickly, the delay between writing code and running it in production can significantly be reduced. [9][13]

The Docker Documentation can be found on the link - [https://docs.docker.com](https://docs.docker.com).
The concepts related to the implementation of Docker to use Containers, conceptual differences between an image and a container, Union file system using which Docker stores images have been mentioned in detail under the title “Docker Overview” section under “Getting Started” section of the documentation. [13] To understand the implementation section, it is necessary to read the updated documentation on the Docker as they are utilized in implementation. [13]
INTRODUCTION TO BIGDATA FRAMEWORKS

Bigdata is enormous amount of data, structured or unstructured, that is complex and difficult to handle using traditional data-processing applications. Big data includes searches on websites, data storage, social web feeds, video sharing online, data analysis, data source, information privacy.

There are 4 V’s of Big Data:

- Volume – Scale of Data
- Velocity – Analysis of Streaming Data
- Variety – Different Forms of Data
- Veracity – Uncertainty of Data.

Data is growing rapidly. As per Wikipedia, world's technological per-capita capacity to store information roughly doubles every 40 months, as of 2012, every day 2.5 exabytes (2.5×1018) of data are generated. Moreover, it is estimated that by the year 2020, about 1.7 megabytes of new information will be created every second for every human being on the planet. [11]
So, there is undoubtedly a lot of data in huge amounts that needs to be managed, but there is an interesting application that can be built using this data – Data Analysis to find correlations among different types of data which is profitable and advantageous to governments, businesses, scientists etc. for advancement in different domains.

Hence, to process such data for large organizations there have been various projects developed to solve the problem of Data Analysis and they have evolved over time. Google was the company to come up with a paper on MapReduce in 2004, which changed the face of Big Data Analysis and since then various projects have been
developed to process different types of data and different techniques are still being applied to enhance them to achieve better performance and reliability.

Later, The Apache Software Foundation launched a collection of open-source software utilities to solve problems for distributed processing of large datasets and
computations across a cluster of computers. It has been developed such that it can scale up from one server to thousands of machines, each offering computation and storage. These softwares have been designed for reliability, scalability, higher availability and distributed computing by not relying on hardware and providing enough features on the application layer in software libraries.

The project includes following modules:

- Hadoop Common
- HDFS
- Hadoop YARN
- Hadoop MapReduce
- Cassandra
- Spark
- Flink
- Tez, Etc. [15]

**Apache Hadoop**

The Core Hadoop Project consisted of a way of storing data (HDFS) and process data (MapReduce). However, many projects have been developed under the Hadoop Ecosystem and there have been other options for storing and processing data. Here we are comparing some of the frameworks to the performance offered by them and hence for the purpose of Apache Hadoop we will consider using HDFS and MapReduce. Typically, the compute nodes and storage nodes are the same, which means each node in the Hadoop Cluster is compute and storage node, like a distributed system. [15]
HDFS (Hadoop Distributed File System)

The Hadoop Distributed File System (HDFS) is a distributed file system that runs on a machine or a cluster of machines and this is the layer in the Hadoop Ecosystem that deals with storage of data.

Some of the features of HDFS:

- It is highly fault-tolerant and can be deployed on any inexpensive hardware. This fault-tolerance is achieved by Replication of the data on various nodes of the cluster.
- It is designed for basically batch processing and not for interactive use with users.
- It provides large bandwidth for large data sets.
- It can handle large amounts of data by arranging them in different block sizes and many more. [15]

System Architecture

HDFS has a master/slave architecture built using Java language. A HDFS cluster consists of a single NameNode which performs the role of a master server that manages the file system namespace and is responsible in regulating access to files by clients and there are one or more DataNodes which manage storage attached to the nodes that they run on. Internally, each file in DataNode will be split into one or more blocks and these blocks are stored in a distributed manner. HDFS also provides various block sizes that can be used in storing data, default being 64 MB. The NameNode is responsible for executing file system namespace operations like opening files, closing files, and renaming them. It is also responsible to determine the mapping of blocks to DataNodes.
and hence is called the meta-data of DataNodes. The DataNodes serve read and write requests from the file system’s clients. The DataNodes also are responsible for performing block creation, deletion, and replication as instructed by the NameNode. [15]

Figure 10: HDFS Architecture [15]

Namenode – Stores Metadata of the Datanodes
Datanode – Stores Data by replicating them and splitting them in block sizes

Data Replication – Replicates data for Fault-tolerant File System (Default Replication: 3)

There is an important point to note when involving multiple locations of Namenodes and Datanodes. Whenever multiple locations for Datanodes are specified and Hadoop is started, it will start the replication process so that the minimum replication of data is achieved and will spread out the data among the clusters managing optimum and equal utilization of the server or locations. So, all the Datanodes will not necessarily store same data. Whereas in the case of Namenode, whenever multiple locations are specified, and Hadoop is restarted, the Namenode will be exactly copied at the new location. However, it will take some time in replication based on the size of the Namenode. So, it kind of adds redundancy which would be useful if the first location fails, data can be still be accessed using second Namenode. It is also important to note that if the all Namenodes are unavailable, even though the Datanodes are available, the data cannot be accessed as the Namenodes store which blocks of what file are stored at which location in the cluster or machine.

MapReduce

Let’s take a simple example. Let’s say we want to total up sales of thousands of stores worldwide. Now, if there is only a single computer going through each of them serially, it will be a long time. To solve, this issue, MapReduce is used. Hadoop MapReduce is a software framework for easily writing applications which process vast
amounts of data of various petabytes in-parallel on large clusters (thousands of nodes) of commodity inexpensive hardware in a reliable, fault-tolerant manner. So, basically, this task of going through files would be split among Mappers and Reducers.

A MapReduce job is designed to split the input data-set into multiple independent chunks so as to be processed in a completely parallel manner by map functions. The framework is designed to sort the outputs of these map functions, which then act as input to the reduce tasks. The input of the jobs and output files generated after running the job are stored in a file-system such as local, cluster or HDFS. The framework schedules tasks, monitors them and if they fail re-executes them.

These Map and Reduce functions have to be implemented based on the type of data and can be written in supported languages such as Java, Python, etc. Once they are implemented, the framework can use those functions to run the MapReduce job and solve the problem. When a job is submitted, the JobTracker splits it to Mappers and Reducers which has a single instance for the whole cluster. Next, the actual Map and Reduce tasks are handled by the TaskTrackers that run on each node.

So, mappers run in parallel and deal with small amounts of data. Number of mappers involved depend on the total number of blocks of a file. This means that the mappers will be more if the block size is small leading to more parallelism. So, for instance, an input file of 1 TB data and stored in block size of 128 MB can have 8192 maps. The output by the mappers is termed as intermediate records. Reducers will reduce these intermediate records having same key and will reduce them one value. Reducer has three phases: Shuffle, sort and reduce. Shuffle and sort is the phase to which the
intermediate records are passed, then they are sorted and passed to the reducer to reduce it. To get the correct number of reduces 0.95 or 1.75 needs to be multiplied by (number of nodes) * (number of maximum containers per node).

The MapReduce framework consists of a

- one master `ResourceManager`
- one slave `NodeManager` per cluster-node
- one `MRAppMaster` per application

The client then submits job to `ResourceManager` which distributes the job along with configuration to slaves by scheduling and monitoring the tasks and pushes info to the job-client. [15]
YARN

YARN – Yet Another Resource Negotiator as the name suggests is the one responsible for resource management and job scheduling into different daemons. The *ResourceManager* is global for the cluster, *ApplicationMaster* is per-application and *NodeManager* runs on every node and is responsible for containers and their monitoring.
Now, the *ResourceManager* has two components: Scheduler – responsible for allocation of resources to various running applications and ApplicationsManager – responsible for accepting submission of jobs negotiating the first container for executing the application specific ApplicationMaster and provides the service for restarting the ApplicationMaster container on failure. [15]

**Apache Spark**

To overcome some drawbacks in Hadoop MapReduce such as writing lengthy codes for MapReduce job and data processing delays happening the way scheduling was happening, Apache Spark was developed with features to overcome these as well provide real-time analytics and stream analytics. [16] Apache Spark is an open source framework utilized as processing engine which is built around speed, ease of use, in-memory
analytics and has APIs supported in popular languages such as Java, Python, Scala, etc.

Spark runs a number of times faster than Hadoop MapReduce in memory and is around 10x faster on disk. It supports Spark SQL for SQL and structured data processing, GraphX for graph processing, MLlib for machine learning and Spark Streaming for streaming data. Spark can run on Hadoop, Mesos, standalone and also in cloud. It also can support various data sources such as Apache Cassandra, Apache HBase, HDFS, etc. [16]
The Spark Configuration that has been used for running experiments is the one in the following figure which uses HDFS and the version of Spark is 1.6 with Hadoop 2.6:

Figure 13: Apache Spark with HDFS [17]
RDD – Resilient Distributed Data

Apache Spark at its core is the notion of data abstraction as distributed collection of objects which is known as RDD – Resilient Distributed Data which allow you to write programs that transform these distributed datasets. RDDs are immutable distributed collection of elements of data that are stored in (RAM) memory or (Hard drive) disk on a machine or across a cluster. The data in RDDs is split across the cluster that can be operated in parallel with a low-level API which is utilized for offers transformations and actions. RDDs are also designed to be fault tolerant as they are responsible for tracking data lineage information to rebuild lost data automatically on failure. [16]

Cluster Mode Overview

Spark Applications are managed by SparkContext object in main or driver program and these applications run as independent set of processes. SparkContext can connect to various cluster managers such as Standalone, Mesos or YARN (Resource Manager of Hadoop) which will allocate resources among applications. Once the SparkContext has established the connection, Spark can acquire executors on the nodes of the cluster which do both run computations and store data for the application. In the next step, the application code is sent to the executors and finally tasks are sent to these executors to run. [16]
Each driver program has a web UI on port 4040 which can be changed in spark configuration files, that displays information about running tasks, executors, and storage usage. To access Web UI of Spark just use this address - http://<driver-node>:4040 in a web browser. [16]

**Apache Flink**

**System Architecture**

This section lays out details of the software stack used in the architecture of Apache Flink. [18]
Software Stack

Flink, since its introduction, continues to grow the stack but primarily can be distinguished into the following 4 layers namely – Deployment, Core, APIs and Libraries.

Figure 15: Apache Flink Software Stack [18]
Deployment modes

Apache Flink can be deployed on local machine, on a cluster which can be standalone pseudo cluster or on Hadoop YARN and can also be deployed on Cloud such as Amazon Web Services EC2. [18]

Core

Flink’s core is a distributed streaming dataflow engine, meaning that data is processed an event-at-a-time rather than as a series of batches unlike other data processing systems, and this is what enables many of Flink’s resilience and performance features and make its standout in comparison to other batch data processing systems. [18]

APIs

Flink supports mainly two types of APIs:

- **Flink’s DataStream API** is for programs that implement transformations such as filtering, defining windows, aggregating, etc. on data streams.

- The Flink’s DataSet API is for programs that implement transformations such as grouping, mapping, joining, etc. on data sets. To adapt the bounded datasets to process in streaming core, Flink treats the bounded data as a special case of finite stream. This is how DataSet API is made to operate. [18]

Libraries

Flink also has support for other domain specific libraries that generate DataSet and DataStream API programs. The following are currently bundled with Flink:
• The Table API of Flink is a SQL-like expression language for stream and batch processing that can be easily embedded in Flink’s DataSet and DataStream APIs.

• Streaming SQL enables SQL queries to be executed on streaming and batch tables.

• FlinkML is for Machine Learning applications.

• Gelly is for graph processing applications which has been utilized in experiments for BigDataBench. [18]

**Process model**

Any Flink cluster comprises of three main processes:

• Flink Client

• Job Manager

• TaskManager (atleast one)

Flink client takes in the program code, transforming it to a dataflow graph and submits it to Job Manager. The JobManager is responsible for coordinating the distributed execution of the dataflow. It keeps the tracks of the state and progress of each operator and stream, schedules new operators, and coordinates checkpoints and recovery. In a setup with high availability, the JobManager persists a minimal set of metadata at each checkpoint to a fault-tolerant storage, so that a JobManager which is in standby mode can reconstruct the checkpoint, if anything fails and recover the dataflow execution from there. This is how Flink gets to stay fault-tolerant.

The actual processing of data will take place in the TaskManagers. A TaskManager is responsible for executing operators that produce streams then report the
status of the streams to the JobManager. The TaskManagers also maintain the buffer pools. These pool help buffer the streams, and the network connections to exchange the data streams between operators and hence there should optimum space kept on the memory for that. [18]

![Apache Flink Process Model](image)

**Figure 16: Apache Flink Process Model [18]**

**Other Frameworks**

There are various projects that are supported in Apache Hadoop Ecosystem and they are applications for which they would be efficient and powerful and not for others. Based on the type of the workloads or applications, a developer or user can
choose from these frameworks and built the system to solve problems. Some of others are:

- Pig
- HBase
- Oozie
- Sqoop
- Avro
- Cassandra
- Kafka
- Storm
- Tez
- Zookeeper

Etc. [15]
INTEL PERFORMANCE COUNTER MONITOR

The Intel® Performance Counter Monitor provides a set of C++ routines and these help a user estimate the resource utilization of various components of the Intel® Xeon® and Core™ processors. Embedding these in an application helps a user gain a significant performance boost by understanding how the Architecture is affected. [19]

Why it is needed?

The advances in computer architecture made the prediction of CPU utilization on Operating Systems an unreliable metric because of introduction of multi core and multi CPU systems, Hierarchy of caches, simultaneous multithreading (SMT), pipelining, out-of-order execution of Superscalar Architecture, NUMA (Non-Uniform Memory Architecture), etc. The current implementation of this metric shows the portion of time slots that the CPU scheduler in the OS could assign to execution of running programs or the OS itself; the rest of the time is idle. For compute-bound workloads, the CPU utilization predicted this way for almost perfect for older architectures but as not as good for modern architecture systems.

An important example is the non-linear CPU utilization on processors that support Intel® Hyper-Threading Technology. It is a great performance feature that can boost performance as the context switching on threads happen almost instantly with least overhead. However, the users unaware about the HT technology can get confused by the CPU Utilization. For example, let’s consider an application that runs a single thread on
each physical core. Then, CPU utilization reported is 60% even though the application can use up to 80%-100% of the execution units. [19]

**Solution**

So, to solve this issue, some hardware level resource is required that is able to give an idea of the performance directly from the hardware and using which some prediction about the CPU Utilization can be made. For that purpose, Intel provides The Intel® Performance Counter Monitor tool which utilizes Performance Monitoring Units to have a more precise picture of CPU resource utilization, to access values of the counter that combined together can give information of the current Performance of the Hardware, implemented in Intel processors such as - current Intel® Xeon® 5500, 5600, 7500, E5, E7 and Core i7 processor series.

Following metrics are supported:

- **Core - Core related instructions**: elapsed core clock cycles, core frequency with Intel® Turbo boost technology, L2 cache rates, L3 cache rates
- **Uncore - Not core related instructions**: read bytes from memory controller(s), bytes written to memory controller(s), data traffic, etc.

The Intel PCM can be utilized by the user in their programs so as to know the resource utilization and hence act accordingly and there have methods stated on their website. Also, if user wants to just display the counters can be displayed on Terminal which update instantaneously. [19]
Overview of the files

The following list can be found on GitHub of Intel PCM [20] which is at the following link:

https://github.com/opcm/pcm

However, for easy reference, the file descriptions have been pasted in this section.

PCM provides a number of command-line utilities for real-time monitoring:

- pcm: basic processor monitoring utility (instructions per cycle, core frequency (including Intel(r) Turbo Boost Technology), memory and Intel(r) Quick Path Interconnect bandwidth, local and remote memory bandwidth, cache misses, core and CPU package sleep C-state residency, core and CPU package thermal headroom, cache utilization, CPU and memory energy consumption)

- pcm-memory: monitor memory bandwidth (per-channel and per-DRAM DIMM rank)

- pcm-pcie: monitor PCIe bandwidth per-socket

- pcm-iio: monitor PCIe bandwidth per PCIe device

- pcm-numa: monitor local and remote memory accesses

- pcm-power: monitor sleep and energy states of processor, Intel(r) Quick Path Interconnect, DRAM memory, reasons of CPU frequency throttling and other energy-related metrics

- pcm-tsx: monitor performance metrics for Intel(r) Transactional Synchronization Extensions

- pcm-core and pmu-query: query and monitor arbitrary processor core events
Graphical front ends:

- pcm-sensor: front-end for KDE KSysGuard
- pcm-service: front-end for Windows perfmon

There are various stats that can be obtained using Intel PCM and some discussion of Power Management is needed to understand the different stats and how they are impacting the performance and the energy of the CPU. [19][20]

**Power Management at CPU Level**

**C-States (Processor Operating States)**

There are various power saving modes called as C-states. These states reflect how an idle a processor is and will save power by turning off unused parts. The most powered up state is C0 state when the processor is executing instructions and it is the least power saving state. So, the higher the C-state, more is the power saved which means the processor is in idle state. So, deeper sleep states will save a lot of energy but on the other hand also slow down the processor introducing latency. Along with this some states also have sub-states wherein each of them has different latency level involved. It depends on the respective processor which one it supports.

Following are the most common C-states:

- C0 mode – CPU fully turned on, executing instructions.
- C1 mode – CPU internal clocks are stopped via software. However, bus interface unit and APIC keep running at regular speed.
C2 mode – CPU internal clocks are stopped via hardware. Longer wait for interrupts. Maintain all software-visible states.

C3 mode – All CPU internal clocks are stopped. Cache not coherent in this state.

All the features of C2 mode. [21]

**P-States (Processor Performance States)**

When the processor is in C0 state, it can be in any of the P-state. For all the other C-states, P-states are operational states relating to the voltage and frequency. Similar to C-state concept, the higher the P-state lower is the frequency and voltage at which the processor runs. So, P0 state is the highest performance state. P-states and C-states vary independently of one another.

**Turbo Features**

These features allow to dynamically overtick the CPU cores when other cores are in deep sleep hence increasing the performance of active threads and also complying with Thermal Design Power limits.

**In-Kernel Governors**

Governors are a part of the Linux Kernel. They are in CPUfreq infrastructure and are designed to dynamically scale the CPU frequencies at runtime. P-states are utilized by CPUfreq governors to change the frequencies and hence the power consumption. These dynamic governors help manipulate the frequencies, saving power consumption keeping the performance unchanged.
**Performance Governor**

As the name suggests, this governor keeps the performance maximum by setting the CPU frequency to the maximum value.

**Powersave Governor**

Again, as the name suggests, this governor sets the lowest possible frequency hence impacting the performance of the processor. The system will maximum run at this frequency even though the processors are busy. However, “intel_pstate” which defaults as powersave mode is an exception.

**On-demand Governor**

This is a way of optimizing the power and performance. That being said, it monitors the usage of the processor and if the usage exceeds certain threshold, the frequency is bumped up to the highest frequency similarly if the processor is underemployed or is not been utilized much then the frequency is lowered to the next lower frequency till it reached the lowest possible frequency.

**Userspace Governor**

Userspace Governor is utilized by user to control the frequency of the processor. So, user can keep it to any of the possible frequency steps and utilize the processor. Hence, there will be no performance or powersave guarantee as even though the processor is not being utilized it will still run at the same speed hence using the same power. Similar for the powersave scenario.
**Conservative Governor**

This Governor runs just like the on-demand except for the fact that it scales the frequency more gradually than on-demand.

**Scheduler Driven Governor**

This governor helps select Scheduler-driven CPU frequency selection. [21]

**cpupower Tools**

`cpupower` tools are designed to view or access all power related parameters of the CPU and also modify settings of CPUfreq which is kernel-related. Hence, it is useful for purpose of performance benchmarks which is the reason we utilize these tools to obtain information about the frequency of the processor. To install cpupower tools, follow the following steps:

```bash
sudo apt-get install linux-tools-common
sudo apt-get install linux-tools-$(uname -r)
sudo apt-get install linux-cloud-tools-$(uname -r)
```

To find out which the types of governors supported by the Kernel, using the following command would state the possible options:

`cpupower frequency-info`
Manual page of cpupower has detailed information about the command and what are the metrics that can obtained or set. [22]
IMPLEMENTATION DETAILS

The experiments were conducted on Intel Xeon CPU E5-2683 v4 @ 2.10 GHz.

Further details of the machine are listed as follows:

![Server Info](image)

Figure 18: Server info

CPU min frequency: 1200 MHz

CPU max frequency: 2100 MHz
Environment Setup for Host Machine

Enable frequency scaling on the machine

CPU frequency scaling has been implemented feature in Linux Kernel and the necessary modules are loaded and powersave governor using intel_pstate driver is enabled by default in Sandy Bridge and newer Intel CPUs.

By default, the intel_pstate in enabled and powersave governor is selected by default. The issue with using powersave governor is that it will keep on slowing down the processor for saving power but as we need to get the performance, the CPU needs to run at the highest throttle possible and hence for that we need to change the governor to userspace so as to gain control of the frequency of the CPU.

So, to switch the governor, disabling intel_pstate is necessary at boot-time. For that reason, the following needs to be done:

- Open /etc/default/grub
- Add this line to the file:

  \texttt{GRUB\_CMDLINE\_LINUX\_DEFAULT=\textasciitilde intel\_pstate=disable} \textendash

- Reboot the machine to disable intel_pstate.

This is will disable the intel_pstate frequency driver but the governor will not be changed. So, to change the governor:

\texttt{echo governor\_n > /sys/devices/system/cpu/cpu*/cputfreq/scaling\_governor}

\texttt{governor\_n} can be any of the governors discussed earlier.

So, for the userspace governor,
\[ echo \textit{userspace} > /sys/devices/system/cpu/cpu*/cpufreq/scaling_governor \]

This will change all the cores to userspace. [22]

Next, a list of commands that need to be executed before running Intel PCM or it will not run successfully, and we use these commands every time we run the experiments although some of them might be redundant to make sure we set the correct parameters.

\[ echo \textit{0} > /proc/sys/kernel/nmi_watchdog \]  \#Disable NMI

\[ modprobe msr \]  \#Read/write Model Specific Registers

\[ \textit{cpupower frequency-set -g userspace} \]  \#Set userspace governor

\[ \textit{cpupower frequency-set -f 2100000} \]  \#Set the frequency

The above commands can be saved as a bash script, so that the script comes handy while running experiments. [22]

**Hadoop Installation and Configuration**

For the purpose of experimentation, Hadoop is installed in pseudo standalone mode. So, it accesses all the file over localhost. All the Web UI hence use localhost to access Hadoop.

**Installation**

1) Install OpenJDK

\[ sudo \textit{apt install openjdk-8-jdk} \]

Add the path to the .bashrc. To open .bashrc for a particular user sign in as a user and there will be file named .bashrc in the home location of the user.

\[ vi ~/.bashrc \]
Add the following line for JAVA_HOME to the bashrc and save it:

```
export JAVA_HOME=$(readlink -f /usr/bin/java | sed "s:jre/bin/java::")
```

2) Change to Superuser

```
sudo su
```

3) Change the ownership of the project folder to “root” with group “root”

```
sudo chown -R root:root /home/hosein/project/
```

**Explanation:**

```
sudo chown -R(recursive) #user:#groupname #path
```

4) Configuring your machine to disable Ipv6

Open the file `/etc/sysctl.conf` using your preferable editor (I have used `vim`):

```
vi /etc/sysctl.conf
```

Add these following lines to the end of the file:

```
net.ipv6.conf.all.disable_ipv6 = 1
net.ipv6.conf.default.disable_ipv6 = 1
net.ipv6.conf.lo.disable_ipv6 = 1
```

5) Configuring localhost without password for Hadoop to run

```
ssh-keygen -t rsa
```

Generating public/private rsa key pair.

Enter file in which to save the key (`/root/.ssh/id_rsa`): [Press Enter]

Created directory '/root/.ssh'.

Enter passphrase (empty for no passphrase): [Press Enter]
Enter same passphrase again: [Press Enter]

Your identification has been saved in /root/.ssh/id_rsa.

Your public key has been saved in /root/.ssh/id_rsa.pub.

Switch to the following path to add the localhost key to the authorized_keys file:

```
cd /root/.ssh
```

Add the public key to the authorized_keys:

```
cat id_rsa.pub >> authorized_keys
```

These above steps will enable ssh to localhost without any password. If you face any error, try generating another key and trying it again as sometimes you might get an error. If you get any further errors or have to enter password while performing ssh to localhost, check the configuration in the following file.

6) This is to enable the root login access from remote pc or localhost from root to root

Configure the following in the file (/etc/ssh/sshd_config):

```
sudo vi /etc/ssh/sshd_config [To open the file]
```

Check these values in the file:

**PermitRootLogin yes**

(This enables the root login to the machine. This is required when you are configuring your Hadoop in root as you will be login as root to access localhost)

```
sudo service ssh restart
```

The above command is to restart the service after changing the configuration file of ssh.
7) Change root user password

There is a default password set by ubuntu which is not same as the one set by the user while installing the system. It is easier to reset that password ans set a new password for convenience.

Remain in the home directory of the user and enter the following command:

```
sudo passwd root
```

Enter new UNIX password: [Enter the new password]

Retype new UNIX password: [Enter the new password]

```
passwd: password updated successfully
```

8) Installing Hadoop:

```
wget http://ftp.wayne.edu/apache/hadoop/common/hadoop-2.7.4/hadoop-2.7.4.tar.gz
```

**Explanation:**

```
wget #link_of_hadoop
```

Unpack the file using the following command:

```
tar -xvf hadoop-2.7.4.tar.gz
```

To get superuser permissions:

```
sudo su
```

```
mkdir usr/hadoop
```

```
mv hadoop-2.7.4 /usr/hadoop/
```

To change the ownership of the Hadoop files to root:

```
sudo chown -R root:root /usr/hadoop/hadoop-2.7.1
```
Open root’s .bashrc to add the following lines:

```bash
export HADOOP_HOME=/usr/hadoop/hadoop-2.7.4
export PATH=$PATH:$HADOOP_HOME/bin
export PATH=$PATH:$HADOOP_HOME/sbin
export HADOOP_MAPRED_HOME=$HADOOP_HOME
export HADOOP_COMMON_HOME=$HADOOP_HOME
export HADOOP_HDFS_HOME=$HADOOP_HOME
export HADOOP_YARN_HOME=$HADOOP_HOME
export HADOOP_COMMON_LIB_NATIVE_DIR=$HADOOP_HOME/lib/native
export HADOOP_OPTS="-Djava.library.path=$HADOOP_HOME/lib"
```

**Configuration**

The Hadoop configuration file are in $HADOOP_HOME/etc/hadoop location of the hadoop folder. There are various configuration files that support various parameters that can help control and configure hadoop for various different situations and configurations.

- Edit the **core-site.xml** to add the following lines:

```xml
<configuration>
  <property>
    <name>hadoop.tmp.dir</name>
    <value>/usr/hadoop/tmp</value>
  </property>
  <property>
    <name>fs.default.name</name>
</configuration>
```
• Edit the **hdfs-site.xml** and add the following lines:

```xml
<configuration>
    <property>
        <name>dfs.replication</name>
        <value>1</value>
    </property>
    <property>
        <name>dfs.namenode.name.dir</name>
        <value>file:/usr/hadoop/hadoop_data/namenode</value>
    </property>
    <property>
        <name>dfs.datanode.data.dir</name>
        <value>file:/usr/hadoop/hadoop_data/datanode</value>
    </property>
</configuration>
```

dfs.namenode.name.dir → Path to the file where you want to store the namenode

dfs.datanode.data.dir → Path to the file where you want to store the datanode

Also make sure to create folders at the necessary places and also, they should have the permissions same as the hadoop folders (in this example “root” permissions):
sudo su

cd /usr/hadoop/

mkdir hadoop_data

cd hadoop_data

mkdir namenode

mkdir datanode

Change the permission if not already root:

sudo chown -R hduser:hadoop /usr/hadoop/hadoop_data

- Edit the mapred-site.xml and add the following lines:

  cp /usr/hadoop/etc/hadoop/mapred-site.xml.template

  /usr/hadoop/etc/hadoop/mapred-site.xml

  Open the file and add the following lines:

  <configuration>
  <property>
    <name>mapred.job.tracker</name>
    <value>localhost:9001</value>
  </property>
  </configuration>

  <configuration>
  <property>
    <name>mapreduce.framework.name</name>
    <value>yarn</value>
  </property>
  </configuration>
The value of mapreduce.framework.name can be yarn, local or classic.

Yarn – enables the YARN Resource allocator

local – acts as if it is a local machine

- Edit the file **yarn-site.xml** to add the following lines:

```xml
<configuration>
  <property>
    <name>yarn.nodemanager.aux-services</name>
    <value>mapreduce_shuffle</value>
  </property>
  <property>
    <name>yarn.nodemanager.aux-services.mapreduce.shuffle.class</name>
    <value>org.apache.hadoop.mapred.ShuffleHandler</value>
  </property>
</configuration>
```

- Edit the **hadoop-env.sh** file add the path for **JAVA_HOME**:

```bash
export JAVA_HOME=$(readlink -f /usr/bin/java | sed "s:jre/bin/java::")
```

Format Hadoop before starting:

```
hdfs namenode -format
```

Starting hadoop:

Be in root user mode to start Hadoop:

```
/usr/hadoop/hadoop-2.7.4/bin/start-all.sh
```
If prompted for any connection enter “yes”. Once the task is completed, enter the following command to check the daemons running:

```
jps
```

5938 DataNode
5732 NameNode
7062 Jps
6683 NodeManager
6221 SecondaryNameNode
6494 ResourceManager

All of the above daemons should be running else there can be unpredictable results while running experiments.

To check the web UI for Hadoop:

```
localhost:50070
```

To check the web UI for YARN:

```
localhost:8088
```

[15]
Important note:

Make sure to keep the ownership of hadoop and files related to hadoop same. Also, Hadoop will only allow the user who is the owner of the Hadoop files to view, delete or modify files in Hadoop or else access will be denied. This is a common problem faced if you try and install Hadoop as root user.

Spark Installation and Configuration

Installation

Download sbt and spark from their respective websites and provide the paths for both of them in bashrc of the user:

```bash
export SPARK_HOME=/home/hduser/project/spark-1.6.0-bin-hadoop2.6
export PATH=$PATH:$SPARK_HOME/bin:$SPARK_HOME/sbin
export SPARK_MASTER_HOST=127.0.1.1
export SBT_HOME=/home/hduser/project/sbt
export PATH=$PATH:$SBT_HOME/bin
```

Add the following lines to the file spark-env.sh file stored in the conf folder of Spark:

```bash
SPARK_MASTER_IP=127.0.1.1
JAVA_HOME=$(readlink -f /usr/bin/java | sed "s:jre/bin/java::")
SPARK_WORKER_MEMORY=4G
```

[16]
Install SBT:

```bash
echo "deb https://dl.bintray.com/sbt/debian /" | sudo tee -a /etc/apt/sources.list.d/sbt.list

sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 2EE0EA64E40A89B84B2DF73499E82A75642AC823

sudo apt-get update

sudo apt-get install sbt
```

To check the web UI for Spark:

`localhost:8088`

[16]

**Flink Installation and Configuration**

**Installation**

Install Java as mentioned earlier in Hadoop and Spark.

Download the version of Flink from website:

```bash
tar xzf flink-*.tgz  # Unpack the downloaded archive

cd flink-1.5.0
```

Start a Local Flink Cluster:

`.bin/start-cluster.sh  # Start Flink`
Stop a Local Flink Cluster:

./bin/stop-cluster.sh  # Start Flink

Configuration

Update the .bashrc file to have the following environmental variables:

`export JAVA_HOME=$(readlink -f /usr/bin/java | sed "s:jre/bin/java::")`

`export PATH=$PATH:$JAVA_HOME`

`export FLINK_HOME=/home/hosein/Flink/flink-0.10.0`

`export PATH=$PATH:$FLINK_HOME/bin`

[18]

Configuration of Benchmarks

The Benchmarks used for the purpose of experimentation is BigDataBench[23][24][25].

The configuration for these benchmarks has been explained in a document provided on the website. There are minute configuration details that would be important to notice in order to make sure that the experiments run for that a link to the Github repository has been posted in the Appendix of this document.

Environmental Setup

`export JAR_FILE_FLINK=
/home/hosein/Flink/BigDataBench_V3.2_Flink/JAR_FILE/bigdatabench-flink-0.10.0.jar`
export JAR_GRAPH_FLINK=/home/hosein/Flink/BigDataBench-Graph/Flink-Gelly/graph-flink.jar

export JAR_FILE_SPARK=/home/hosein/Flink/BigDataBench_V3.2.5_Spark/JAR_FILE/bigdatabench-spark_1.3.0-hadoop_1.0.4.jar

export JAR_GRAPH_SPARK=/home/hosein/Flink/BigDataBench-Graph/Spark-Graphx/graph-spark.jar

Environment Setup for Virtual Machine

VirtualBox 5.2 was used for the purpose of experiments which a Type-2 Hypervisor. Ubuntu 16.04 needs to be installed by setting up a New Virtual Machine. All the configuration states in the Host has to be copied to the Virtual Machine and their respective environmental variables have to be defined in user’s .bashrc.

Copying the files of Hadoop, Spark and Flink from the Host machine and configuring similarly would also serve the purpose.

Setting network configuration

Setting up the Virtual network in the Virtual Machine is important. Why we need to setup everything and run the code from the Host Machine has been explained in the further section. So, to setup the network configuration, enable the host only adapter that connects the Host Machine with Virtual Machine over a private network. After that the Virtual Machine needs to be power on and we need to install the necessary modules.
These modules are also listed in Dockerfile and can be used as reference. Make sure all the necessary packages are installed. Next, run the command `ifconfig` and find the IP address of the Host-only adapter as this address will be utilized in the code. The IP address for the VM in the experiment is 192.168.56.101. Also, port forwarding can be setup for the ports that need to extend to the Host that are utilized by the Frameworks to view the Web UI.

**Environment Setup for Docker Container**

The Dockerfile for setting up the configuration for these experiments is mentioned in the appendix of this document. It is self-explanatory along with the comments mentioned to elaborate on the aspects that need to be considered. Docker Documentation states all the commands that can be used to convert Dockerfile to Docker image and then run that image in a container. There is also a startup file that is called while starting the container and it necessarily sets up the rest of the environment for running the experiments. The startup.sh file can be found in the GitHub repository posted in the Appendix Section of this document.

**Understanding the code**

First of all, various reporting parameters from different Intel PCM files were put in one file pcm-memory.cpp and the code developed is modifying it to support all these parameters concurrently when the benchmark application is running.

The concept utilized is:
Figure 19: Abstract View of Code
So, using numerous fork() in the main file that basically split into parent – that obtains the performance counters and saves it to the file and child -that runs the application on either Host or VM or Docker container based on what the values of the preprocessor selected.

The following is the part of the code that displays FRAMEWORK_BENCHMARK pattern for all the variables and it easily enables/disables different parts of the code. The value 1 means that corresponding code to that benchmark is enabled and for 0 it means that it is disabled.

Control of the code for activating/deactivating different parts of the Code:

```c
#define HOST 1
#define VM 2
#define DOCKER 3

#define ENV HOST
/*
ENV HOST  ---> will select HOST while running the experiments
ENV VM   ---> will select VM while running the experiments
ENV DOCKER ---> will select DOCKER while running the experiments
*/ /*If to select the Environment: By default, it will set as HOST*/
```
/*Control variables to activate/deactivate different Benchmarks of BIGDATABENCH*/

#define HADOOP 1
#define SPARK 0
#define FLINK 0

#if HADOOP

#define HADOOP_WORDCOUNT 0
#define HADOOP_GREP 0
#define HADOOP_PAGERANK 1
#define HADOOP_NAIVEBAYES 0

/*HADOOP_NAVEBAYES has issue running so it cannot be automated. Need to run it manually*/
#endif
#if SPARK
#define SPARK_WORDCOUNT 1
#define SPARK_GREP 1
#define SPARK_PAGERANK 1
#define SPARK_KMEANS 1
#define SPARK_BFS 1
#define SPARK_CC 1
#define SPARK_NAIVEBAYES 0
#else
#define SPARK_WORDCOUNT 0
#define SPARK_GREP 0
#define SPARK_PAGERANK 0
#define SPARK_KMEANS 0
#define SPARK_BFS 0
#define SPARK_CC 0
#define SPARK_NAIVEBAYES 0
#endif

#if FLINK
#define FLINK_WORDCOUNT 1
#define FLINK_GREP 1
#define FLINK_PAGERANK 1
#define FLINK_KMEANS 1
#define FLINK_BFS 1
#define FLINK_CC 1
#define FLINK_NAIVEBAYES 0
#else
#define FLINK_WORDCOUNT 0
#define FLINK_GREP 0
#define FLINK_PAGERANK 0
#define FLINK_KMEANS 0
#define FLINK_BFS 0
#define FLINK_CC 0
#define FLINK_NAIVEBAYES 0
#endif
/*End of Control Variables for BIGDATABENCH*/

Generating the results

Steps for running the experiments:

1) Check the number of enabled cores is correct:
   This setting can be set in BIOS and the number of cores can be selected. Also check and disable hyperthreading if it is enabled.
2) Check the RAM size on hardware before booting up and also see if the frequency of the RAM needs to be changed for the experiment. This setting can also be set in BIOS.

3) Power on the system and follow ahead.

4) Check the number of cores and each core detail by running the following command:

\texttt{cpufreq-info} \hspace{1cm} \texttt{OR} \hspace{1cm} \texttt{cpupower frequency-info}

It would give information about all the cores and their current frequencies. You can change the frequency of the core if the governor is "userspace".

Check the RAM size and frequency on the system:

\texttt{sudo lshw -short -C memory}

5) To Start Hadoop

Run the following command to start hadoop:

\texttt{/usr/local/hadoop/sbin/start-all.sh}

config location:

\texttt{/usr/local/hadoop/etc/hadoop}

To check if hadoop is running:

http://localhost:50070/dfshealth.html#tab-overview

To check the YARN scheduler in Web UI:

http://localhost:8088/
6) To Start Spark

Change the Worker RAM to 80% of the total memory in the spark-env.sh file of conf folder:

./conf/spark-env.sh

Run the following command to start spark:

/home/hoseinmmm/project/spark/spark-1.6.0-bin-hadoop2.6/sbin/start-all.sh

Spark location:

/home/hoseinmmm/project/spark/spark-1.6.0-bin-hadoop2.6/sbin

To check if the Spark is running:

http://localhost:8080/#

7) Run the file named newsetup.sh before moving forward to the experiment:

/home/hoseinmmm/project/autobench/newsetup.sh

This file will set the CPU frequency to 2.6 GHz

Change the frequency to your required frequency in the file and then run the file.

Make sure to run the dstat command and ./pcm-memory.x command in the following directory;

/home/hosein/project/Fall_17_Experiments

8) Remove hadoop from safemode:

    hadoop dfsadmin -safemode leave

9) Reset this system before running the cpp compiled .x file from experiments:

    /home/hoseinmmm/project/IntelPCM/pcm.x -r
10) Before starting the experiment also generate log of the memory usage by using this command:

   dstat -m -c -d --output ssdout.csv

11) Check the hibench.csv is clean and empty. Copy the first line from the last hibench.csv.

12) Finally run the following file from the Location:

   /home/hosein/project/Fall_17_Experiments

   File name: ./pcm-memory.x

   Followed by parameters separated by space

   Memory size (in GB), Memory Frequency, Memory channel (1,2 or 4), 1 (CPU Frequency), 8 (Number of cores), 1 (Input size)

   CPU Frequency: 0 --> 2.1 MHz
   1 --> 1.9 MHz
   2 --> 1.2 MHz

13) Run the following command:

   ./pcm-memory.x 8 2133 1 2 16 1 >> output_8_2133_1_2_16.txt

   Replace the arguments based on configuration referring to the arguments mentioned in point 12. Also make sure to use similar values of configuration in saving the output file and dstat result file.

14) Command in point 10 should be started first and then command 13 should be run as soon as possible. Run them in two different terminals. Make sure to monitor and stop dstat once the experiment stops.
15) Once the experiments are done, there will be 3 files generated.

    output.txt

    ssdout.txt

    hibench.csv

Put all these files in a folder with the configuration name as done in the excel
sheet.

Then, cd in to the folder and run the following command:

    grep "Application_count" * > count.txt

This will generate the second counts for all the applications and add them to the
excel file.
PERFORMANCE COMPARISON

For understanding how different components of the Computer Architecture perform, Intel PCM helps in fetching the hardware counters such as IPC, L3 cache Hit rate, L2 cache Hit rate, C0 State of the processor, CPU and DRAM Energy and power utilization, etc. The code has been designed to fetch these counters periodically after 1000 milliseconds when the applications are running concurrently. Various types of benchmarking applications such as Microbenchmarks, Graph applications, Search applications and Machine learning have been used to obtain the hardware details to understand how the performance is affected while these applications are running in different environments. In this module, graphs that showcase different components have been displayed and discussed and some observations or inferences have been detailed further. We understand these graphs from two perspectives:

1) How different environments affect the performance

2) How different frameworks affect the performance

Wordcount

IPC

The X-axis displays time in milliseconds and Y-axis displays the IPC value of the processor.
Environment comparison

While running wordcount in Hadoop and Spark, Docker and Host environments have similar patterns for change in IPC. Virtual Machine comparatively more time in execution in comparison to others and also has less IPC over the course of application runtime.

While running wordcount in Flink, Docker comparatively has lower IPC than others over the course of application as seen in the graph.
Framework comparison

Spark IPC remains at the peak and almost consistent throughout the application whereas for Hadoop and Flink there quite change in the IPC value throughout the course of application runtime. Spark outperforms Hadoop and Flink in execution time for the case of Microbenchmark applications.

L3 Hit Rate

The X-axis displays time in milliseconds and Y-axis displays the L3 Cache Hit Rate of the processor.

Figure 21: Wordcount - L3 Hit Rate
Environment comparison

While running wordcount in Hadoop, Docker, VM and Host environments have similar patterns for change in L3 hit rate, but it fluctuates from 0.2 to 0.6 while the application is running.

While running wordcount in Spark, Docker, Host and VM environments perform almost similar and are consistent at L3 Hit Rate of 0.6 and above which is the best among all the scenarios.

While running wordcount in Flink, Docker comparatively has higher L3 Hit Rate ranging from 0.4 to 1 over the course of application runtime, which makes it a desirable condition. But for VM and Host, hit rate starts good at the beginning of the application but remains between 0.2 and 0.4 which is fairly low.

Framework comparison

Spark still outperforms by keeping the L3 hit rate constant and reasonably higher while the application is in execution in comparison to Hadoop and Flink.

L2 Hit Rate

The X-axis displays time in milliseconds and Y-axis displays the L2 Cache Hit Rate of the processor.
Environment comparison

While running wordcount in Hadoop, it is similar to L3 hit rate scenario.

While running wordcount in Spark, Docker, Host and VM environments perform almost similar and are consistent as in the case of L3 Hit Rate, except the hit rate is around 0.35.

While running wordcount in Flink, Docker comparatively has higher L2 Hit Rate. But for VM and Host, hit rate starts good at the beginning of the application but remains between 0.2 and 0.4 which is fairly low similar to L3.
Framework comparison

It is difficult to say which performs the best, but Flink and Hadoop seem to perform better but Spark seems to be consistent over the course of application runtime.

C0 Residency

The X-axis displays time in milliseconds and Y-axis displays the C0 Residency of the processor. The higher the value of C0 Residency higher is the utilization of the processor.

Figure 23: Wordcount - C0 Residency
Environment comparison

While running wordcount in Hadoop, the C0 residency value ranges from 60 to 100 over the course of application runtime for all environments.

While running wordcount in Spark, the C0 residency is set to a value to 80 throughout of runtime of the application for all environments.

While running wordcount in Flink, VM and Host have a range of value from 90 to 100 while the application is running. However, for Docker the utilization is not maximum as the value is neither constant nor consistent which means the CPU is idle state for a longer time.

Framework comparison

The maximum utilization is hit by Flink as the value for Host and VM are almost beyond 90 for the whole application which is good as we want the maximum utilization and do not want idle state to we can utilize the best out of the CPU. For Spark the value stays at 80 and for Hadoop, it varies a lot for all of the environments.

CPU Energy

The X-axis displays time in milliseconds and Y-axis displays the CPU Energy of the processor.
CPU Watt

The X-axis displays time in milliseconds and Y-axis displays the CPU Wattage of the processor.
Environment comparison

All of them in each of the framework consume almost same power except running wordcount on Docker using Flink.

Framework comparison

Point to notice is that as Spark is able to run this faster the amount of power consumption will be significantly lower. Spark outperforms others as they take a longer time to run even though consuming same power at each instant.
DRAM Energy

The X-axis displays time in milliseconds and Y-axis displays the DRAM Energy of the processor.

![Graph](image)

Figure 26: Wordcount - DRAM Energy

DRAM Watt

The X-axis displays time in milliseconds and Y-axis displays the DRAM Wattage of the processor.
The X-axis displays time in milliseconds and Y-axis displays the Bandwidth of the memory.
Overall Analysis

Spark performs the best for Wordcount as it is able to run the application many times faster than it can in Hadoop and Flink. Also, unlike others it consumes less wattage of energy and has reasonable L3 Hit Rate and L2 Hit Rate.
BFS

IPC

The X-axis displays time in milliseconds and Y-axis displays the IPC value of the processor.

![Figure 29: BFS - IPC](image)

Environment comparison

BFS seems to run similarly in all the environments as the patterns are almost similar. It can however be noticed that BFS executes faster in Host than VM and VM is faster than Docker when running in Flink. For Spark, all of them seem to be almost together.
Framework comparison

While running BFS, the average IPC while running on Flink is comparatively higher than Spark. Also, execution time while running on Flink is less than running on Spark.

L3 Hit Rate

The X-axis displays *time in milliseconds* and Y-axis displays the *L3 Cache Hit Rate* of the processor.

![Figure 30: BFS - L3 Hit Rate](image)

Environment comparison

There is no certain pattern that be displayed but they all seem to follow similar pattern for L3 hit rate while the application is running.
Framework comparison

Looking at the graphs above, L3 hit rate is comparatively better when running BFS on Flink than on Spark.

L2 Hit Rate

The X-axis displays time in milliseconds and Y-axis displays the L2 Cache Hit Rate of the processor.

Environment comparison

Similar to L3 hit rate, L2 hit rate seems to be having similar patterns while running in different environments.
Framework comparison

While running on Flink, the L2 hit rate remains consistent in the range of 0.4 to 0.5 whereas while running on Spark it fluctuates between 0.2 to 0.8.

C0 Residency

The X-axis displays time in milliseconds and Y-axis displays the C0 Residency of the processor. The higher the value of C0 Residency higher is the utilization of the processor.

Environment comparison

While running BFS on Flink, Docker seems to have lower value of C0 residency in comparison to others.

While running BFS on Spark, all of them seem to follow similar patterns.
Framework comparison

While running on Flink, the C0 value almost stays at 100 when the application is running whereas for Spark it averagely stays around 80. So, Flink keeps the core busy while running the application more than Spark.

CPU Watt

The X-axis displays time in milliseconds and Y-axis displays the CPU Wattage of the processor.

![Figure 33: BFS - CPU Watt](image)

Framework comparison

While running in Spark seems to consume average 50 Watts of power while running whereas while running in Flink the consumption averagely is higher at 60 Watts.


**DRAM Watt**

The X-axis displays *time in milliseconds* and Y-axis displays the *DRAM Wattage* of the processor.

![Graph of BFS DRAM Wattage](image)

**Figure 34: BFS - DRAM Wattage**

**Framework comparison**

Based on the graphs above, when running BFS in Spark versus Flink, it seems to consume more DRAM wattage while running in Spark than in Flink.

**BW**

The X-axis displays *time in milliseconds* and Y-axis displays the *Bandwidth* of the memory.
Overall analysis

BFS run faster on Flink than on Spark and consumes less DRAM power too. Though there is slightly more power consumed for CPU in comparison to running on Spark, but it still is able to use the maximum capacity of the CPU. So, for applications similar to BFS, they perform better when we run them on Flink.
**Grep**

**IPC**

The X-axis displays *time in milliseconds* and Y-axis displays the *IPC value* of the processor.

![Spark - Grep - IPC](image1.png)

![Hadoop - Grep - IPC](image2.png)

![Flink - Grep - IPC](image3.png)

**Figure 35: Grep - IPC**

**Environment comparison**

As seen in the figures above, for all the frameworks, running Grep in Host is faster than VM which is faster than Docker. This seems to oppose the concept that Docker and Host nearly perform the same. The reason for this is not clear but we can see
it is evident. Moreover, all of them seem to follow similar pattern for IPC change with Host mostly the lead.

**Framework comparison**

Again, running Grep in Spark is faster than the other. All of them averagely hold IPC of approximately 2 over the course of duration when the application is running. Along with that in Spark and Flink it is more consistent than in Hadoop.

**L3 Hit Rate**

The X-axis displays *time in milliseconds* and Y-axis displays the *L3 Cache Hit Rate* of the processor.

![Figure 36: Grep - L3 Hit Rate](image)

---

110
Environment comparison

While running Grep in Spark, Docker and Host seem to have same L3 hit rate around 0.7 to 0.8. whereas Virtual Machine sticks to a lower value of 0.6. While running Grep in Hadoop, they all similar and stick to a value of 0.8 approximately averagely. While running Grep in Flink, Docker leads by staying between 0.7 and 0.8 whereas for Host and VM it stays lower than 0.6.

Framework comparison

While running in Spark and Hadoop they stay the best at 0.8 approximately.

L2 Hit Rate

The X-axis displays time in milliseconds and Y-axis displays the L2 Cache Hit Rate of the processor.
For L2 Hit rate, they all seem to perform the same by having a 60% hit rate for all the frameworks and all environments.

**C0 Residency**

The X-axis displays *time in milliseconds* and Y-axis displays the *C0 Residency* of the processor.

![Grep - C0 Residency](image1)

![Grep - C0 Residency](image2)

*Figure 38: Grep - C0 Residency*
Environment comparison

While running in Spark, the value for Host stays at 80, for VM it stays around at 50 and for Docker it fluctuates a lot over the course of application runtime.

While running in Hadoop, it keeps the processor almost completely busy by averagely staying at 90.

While running in Flink, Host and VM seem to be stay around 80 to 90 and for Docker the value is less and fluctuates.

Framework comparison

There is not a certain quantification that can be made but running in Spark is better than Flink which is better than Hadoop.

CPU Watt

The X-axis displays time in milliseconds and Y-axis displays the CPU Wattage of the processor.
For all frameworks in all environments, they seem to consume same amount of CPU Power.

**DRAM Watt**

The X-axis displays *time in milliseconds* and Y-axis displays the *DRAM Wattage* of the processor.
For all frameworks in all environments, they seem to consume same amount of DRAM Power.

**BW**

The X-axis displays *time in milliseconds* and Y-axis displays the *Bandwidth* of the memory.
Overall Analysis

Running Grep on Host performs the best as the execution is the faster. Grep is yet another application which is categorized as Microbenchmark and Spark performs the best in comparison to other when considered various components.
Figure 42: CC
Kmeans

Figure 43: Kmeans
Overall Analysis

As seen in the graphs displaying comparison of different components while running Kmeans and CC application in different environments, it can be seen that more or less they vary in a similar manner over the course of application runtime. The execution time in different environments is also almost same. These results are a 5 GB of data for Kmeans. It can be possible that the performance is varied for large amounts of data which needs to be considered in future work. But for now, for these kinds of applications they have pretty much have same performance for all the environments. It is also important to note that Flink outperforms in execution time in comparison to Spark and Hadoop for all other applications except Microbenchmarks.
PageRank

Figure 44: PageRank
**Overall Analysis**

As seen in the graphs displaying comparison of different components while running PageRank application in different environments, it can be seen that more or less they vary in a similar manner over the course of application runtime. However, it can also be noticed that the execution time of the application in Host and Docker environment are almost same whereas for the Virtual Machine environment the execution time is significantly higher. That being said, Virtual Machine overhead can be seen in this and it could be one possible reason of more time taken in the execution of the application.
This work has entanglement with other relevant work where all the data generated is utilized or is going to be utilized. For that reason, research that are relevant to this work or from which this work is inspired have been mentioned in the references for the sufficiency of the document. [26][27][28][29][30][31] Moreover, there are numerous ways hardware counters are used for various other domains such as malware detection, scheduling, etc. These hardware counters give good insights about the current physical state of the hardware and there are many applications where these can be employed and also can be embedded into light weight applications. [32][33][34]
FUTURE WORK

Server systems are utilized by the companies for various purposes, managing their data, applications, etc. and infrastructure as a service is provided by companies like Amazon. Amazon’s EC2 service provides any consumer to use their servers for various purposes. Virtualization plays an important part in this and a chunk of big hardware is assigned to the consumer as a Virtual Machine and the consumer is billed by the number of requests made to the hardware. These Virtual Machines are utilized by the consumer to run further virtual machines over it or even utilized to run containers for the purpose of deployment of the application. Such a scenario of running one virtual machine or container into another or mix match of both is defined as Nested Virtualization. For the future scope of this work, the performance of various other applications can be tested by running them in different nested environments to understand what the effect on the hardware in such scenarios is.
APPENDIX

The related updated files to the setup can be found at the following links:

Intel PCM Modified:
https://github.com/devangmotwani/Modified-Intel-PCM

Automation scripts for generating results:
https://github.com/devangmotwani/Automate-generating-results

Dockerfile and Docker image on Dockerhub:
https://github.com/devangmotwani/Docker-image-Bigdatabench
https://hub.docker.com/r/devangmotwani/bigdatabench/

Benchmark and setup files:
https://app.box.com/s/re81thive1s1o5limvei4wwnrnw43lua

Note: The files are zipped while uploading. Make sure to unzip them after downloading and keep them under the same folder. Spark can have issues, so to solve those issues, try downloading a fresh copy from Apache Spark website.
REFERENCES


Devang Motwani received his Bachelor of Engineering from Gujarat Technological University in 2014. He received his Master of Science in Computer Engineering from George Mason University in 2018.