On the Impact of Virtualization on the I/O Performance of Analytic Workloads

Son-Hai Ha
Orange Labs/EURECOM
sonhai.ha@orange.com

Daniele Venzano
EURECOM
daniele.venzano@eurecom.fr

Patrick Brown
Orange Labs
patrick.brown@orange.com

Pietro Michiardi
EURECOM
pietro.michiardi@eurecom.fr

***Abstract*—In this work we study the I/O performance of long, sequential workloads that mimic those of Big Data applications, to understand the implications of system virtualization on data-intensive frameworks such as Apache Hadoop and Spark, which are frequently run in clusters of Virtual Machines (VMs). We do so through an experimental measurement campaign that collects low-level traces and metrics, to show the role played by important parameters such as the I/O schedulers and caching mechanisms involved in the I/O path, and the VM configuration in terms of dedicated resources. Our findings are important, especially for determining appropriate deployment strategies for today's emerging Analytics Services hosted both on public and private clouds.**

***Index Terms*—Virtualization, I/O performance, Analytic Workloads, Big Data, Cloud Computing**

I. Introduction

As modern large-scale analytics applications – that we call Big Data *workloads* – have grown mainstream in the last years, more and more companies [1]–[3] are offering Analytics-as-a-Service to their users. Such services offload users from the burdens of setting up and configuring clusters of computers, and installing and tuning the parallel processing frameworks that execute their data analysis. Today, most of such analytics services are hosted by cloud computing providers, such as Amazon Web Services, and run their frameworks – what we call Big Data *applications* – in clusters of VMs.

This is in sharp contrast with the common best practice of running such systems on bare-metal clusters, for performance reasons. Indeed, "data intensive" scalable computing systems such as Apache Hadoop [4] and Spark [5], have been originally designed for I/O bound workloads, where both network and disk performance are crucial for fast executions [6]–[8].

It is thus natural to investigate what the potential performance loss is – if any – in executing applications and workloads in virtualized environments. However, understanding the low-level implications of system virtualization is currently missing in the literature, that has focused predominantly on application-level views of system performance [9]–[12].

In this work we focus on I/O performance of the disk subsystem, when I/O bound Big Data applications run in VMs. We do so using a thorough measurement methodology: we collect low-level traces from measurements and perform our experiments on a single physical server, which runs several "flavors" of VM configurations. In this work we *emulate* the I/O pressure produced by Big Data applications to avoid overheads and gain control on low-level operations. We study the impacts on I/O performance of the disk subsystem of parameters such as the concurrency level of the I/O request pattern, the kernel modules that implement I/O scheduling, and the kernel-level caching mechanisms that are present on the I/O path of a disk request.

We use as a baseline the performance of a bare-metal system supporting the same kind of workloads ingested by the VMs, which is characterized by a *sequential access pattern*. In addition, we vary also the access mechanisms used to perform I/O requests, to understand the performance implications of the current trend in modern Big Data applications of shifting from synchronous to asynchronous mechanisms.

Our results allow us to draw some important observations on *i)* the role played by the various OS-level caching layers, *ii)* the importance of the host-level I/O scheduler and *iii)* the lack of a model to optimize VM configuration, that is to determine automatically their "flavor", to achieve the best I/O performance, given a physical system configuration.

The remainder of the paper is organized as follows. In Section II, we review the related work on the topic. In section III, we provide some background information required to interpret our results. In Section IV we describe the details of our measurement methodology and the system under measurement. Finally, we present our results in section V and draw the conclusion in Section VI.

II. Related work

The literature is rich of studies on the performance and optimization of Big Data applications [5], [8], [13]–[21]. However, most of such works focus on bare-metal deployments of applications such as Apache Hadoop and Spark. Only lately, some works [9], [11], [12] tackled the problem of understanding the performance of such large-scale systems when deployed in an environment using virtualization.

The rapid evolution and adoption of virtualization in both academia and industry contribute to the optimization of virtual machine execution [22]–[24]. There have been many significant works around optimizing processing [25]–[28], memory [29]–[31], and I/O operation [7], [32]–[35]. Although much of the prior works focus on improving network I/O performance [36]–[39], relatively little work has been done for disk I/O [40]. Studying the storage subsystem is important, as it is

well known that disk I/O can be a severe bottleneck for data intensive applications [6], [7].

In [41], Boutcher *et al.*, demonstrated that choosing an appropriate I/O scheduling algorithm *at the guest OS level* is important to achieve performance gains, while scheduling I/O at the host has no measurable advantage. In contrast, since we focus on sequential rather than random I/O workloads, we observe a fair difference in performance when switching between I/O schedulers *at the host* and we explain this behavior later in the paper.

In [9], the authors conduct an experimental measurement campaign on three different hypervisors, with standard MapReduce benchmarks workloads. They found that differences in the workload type (CPU or I/O intensive), workload size and VM placement yield significant performance differences among the hypervisors. In particular, they observed significant performance variations for I/O-bound benchmarks. Our focus in this work is on a single hypervisor, and dedicated to disk I/O performance only: our metrics use lower level information to compute performance, and our parameters are system instead of application level ones. This allows us to establish an "upper bound", that the system underlying the application can deliver.

Prior studies showed that data locality affects the throughput of Hadoop jobs significantly [14]. Thanks to the flexibility of virtual infrastructure, the authors in [11] introduced a reconfiguration technique to "hot-adjust" VM's size during runtime, allowing more tasks to be scheduled on the node that has local data. This technique improves throughput of Hadoop jobs up to 41% on the cluster with constrained network connection. The study showed that data locality is important, and improving the I/O throughput for VMs is of paramount importance. Our work complements this study with a deeper understanding on what are the low level parameters affecting most I/O performance of virtualized systems.

Conley *et al.*, studied the performance of Map-Reduce applications with many different types of VMs on Amazon public cloud using a cost model [12]. They focused on I/O bound workloads, and showed that network attached storage backed by a very high-speed network fabric and flash storage devices can deliver performance levels comparable to that of a traditional local storage deployment. Now, local (ephemeral) flash storage can provide far higher levels of I/O performance but it is not (yet) used as popularly as magnetic hard drive. One avenue for our future work is to study I/O performance with modern storage drives.

III. Background

This section introduces fundamental concepts required to understand our work and to interpret our experimental results.

A. The path of a VM's I/O request

When a VM is created, QEMU (the userspace process that performs the device emulation to serve the guest) forks a vCPU thread for each virtual CPU and an "iothread" thread for serving I/O requests. These dedicated vCPU threads use

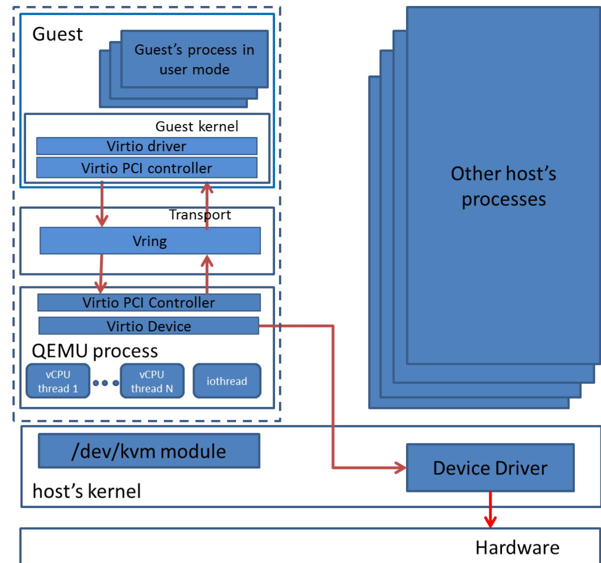


Fig. 1: *The path of the I/O request*

the kvm.ko module in the kernel to execute guest code. Application and guest kernel work similarly to bare metal. Host kernel treats guest I/O like any user space application. QEMU presents emulated storage interfaces to the guest.

We next describe how a guest I/O request is processed between the guest and the host. The guest issues I/O request to the virtual device; this device then fills in the request descriptors which are then written to the buffer in "vring". QEMU issues I/O request from "vring" to the block device on behalf of the guest like other applications. Later, when the requested data arrives at QEMU buffer, QEMU fills in the request footer and injects completion interrupt to the guest kernel. Guest receives the interrupt and executes handler. Application is then notified to read data from the guest buffer.

Inside the guest or the host, applications submit I/O requests using kernel system calls, that convert I/O requests into a data structure called block IO. They are transferred to either "libaio" for asynchronous IOs or directly to the block layer for synchronous IOs. Once an IO request is submitted, the corresponding block IO is buffered in the staging area, which is implemented as a request queue. The block layer then performs IO scheduling and adjusts accounting information (as described in Section III-B) before sending IO requests to the appropriate storage device driver. When the IO requests complete at the device driver, this driver calls up to the block layer generic completion function. The block layer then calls an IO completion function in the "libaio" library, or returns from the synchronous read or write system call, which provides the IO completion signal to the application.

B. Linux IO Schedulers

At the block layer, IO requests are scheduled to access the block device. Next, we describe the IO schedulers that we consider in our experiments.

CFQ scheduler. The CFQ I/O scheduler places synchronous requests submitted by processes into a number of per-process queues and then allocates time slices for each queue to access the disk. The length of time slices depends on the scheduling priority of the process. This helps dividing the available I/O bandwidth among the processes in fine-grained control. The scheduler maintains at most 64 per-process queues. Asynchronous requests for all processes are batched together in fewer queues, one per priority. After the CFQ I/O scheduler moves requests to the dispatch queue, it sorts the requests to minimize disk seeks and then services the requests accordingly.
Deadline scheduler. The Deadline I/O scheduler maintains the deadline queues by the expiration times, or deadlines, of the I/O requests and the sorted queues by the positions of the requests on the disks, or sector numbers. Each set of queues, deadline queues and sorted queues, includes read queue and write queue. Because processes often block on read operations, the Deadline I/O scheduler prioritizes read requests over write requests and assigns read requests shorter expiration times than write requests. Based on prioritization and expiration times, this scheduler determines which request from which queue to dispatch. It also prioritizes deadline queues over sorted queues. To improve disk efficiency, the Deadline I/O scheduler not only services one but a batch of requests taken at the top of the queue in each round.
NOOP scheduler. The NOOP scheduler inserts all incoming I/O requests into a simple FIFO queue without re-ordering requests. The scheduler assumes that the requests may be re-scheduled at the lower level, resequencing I/O request at the host level has no productive reason. Indeed, merging of requests does happens, however, as its simple form of coalescing the adjacent requests only.

C. The caching system

Figure 2 is the view of our cache stack. At guest and host, Page Cache is maintained by the OS to improve I/O performance. In the KVM environment, both the host and guest OSes can maintain their own page caches, resulting in two copies of data in memory. We disable the host's page cache so the guest's application can access directly the storage device when it is desirable.

Linux supports several flags to manage the caching behavior. *O_DIRECT* flag implies no caching at kernel space, DMA is used, and write request is supposed to return after data was written at storage device. *O_DSYNC* flag ensures each write request returns only after data has been written to the physical storage by issuing the flush() command after each issued request. However, the device layer may lie to the application after data was placed in its buffer. By tweaking *O_DIRECT* and *O_DSYNC* flag when loading the VM image, QEMU uses the host's page cache to offers these cache types:
None. VM image is opened with *O_DIRECT* flag so the host page cache is bypassed and I/O happens directly between the guest and the storage device. However the real storage device may immediately report a write as completed when placed in its buffer only. The guest's virtual storage adapter is informed that there is a write-back cache, so the guest application can send down flush commands to commit data to disk permanently.
Writethrough. This mode causes QEMU to interact with the disk image file or block device with O_DSYNC flag, where writes are only reported as completed when the data has been committed to the storage device. However, host's page caching is used so written data can be re-read more efficient if there is application requesting for it soon after.
Writeback. This mode causes QEMU to interact with the disk image file or block device with neither O_DSYNC nor O_DIRECT semantics, so writes are reported to the guest as completed when placed in the host's page cache.
Directsync. QEMU interacts with the disk image file or block device with both *O_DSYNC* and *O_DIRECT* semantics. Writes are reported as completed only when the data has been committed to the storage device, and host page cache is not used. The guest is informed there is write-through cache at the virtual device so applications do not need to send flush() for data integrity.

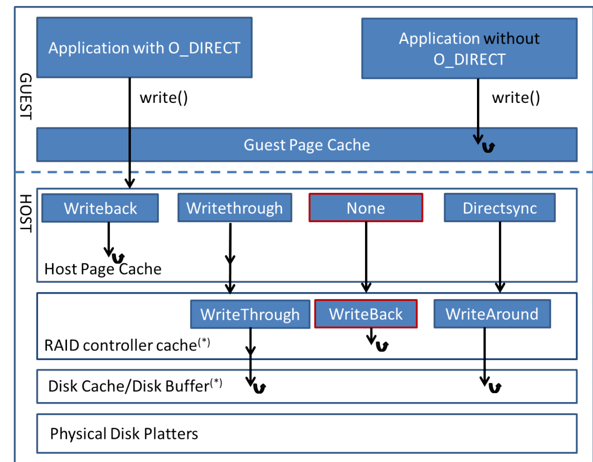


Fig. 2: *The cache system of the host with running VMs*

The RAID controller also maintains its own cache space. *Write-back* cache returns the system calls when I/O placed in the cache. *Write-through* places the I/O request in cache, but only returns the system call after the data has been committed to disk. *Write-around* writes I/O requests to the hard disk first. A copy is only promoted to cache if access is frequent.

IV. Analysis of Big Data workloads

We now present our approach to measure the impact of virtualization on the workloads that characterize many Big Data applications. As such, we focus on long, sequential and concurrent I/O operations, which capture well how applications like Hadoop and Spark interact with storage subsystems [21], [42].

A. The system under measurement

We perform our measurements on one server with a dual, octa-core Xeon E5-2650L CPU with hyper-threading enabled, clocked at 1.8GHz. The server has 128GB of RAM and 10 x 1TB SEAGATE ST91000640SS disks. The RAID controller is configured with RAID 0 for each disk with a 512MBs write-back cache enabled. To stress the system by high concurrency workloads, we only experiment on one disk.

Both the host and the VMs run the long term support Ubuntu 14.04 distribution, updated with the most recent patches and used with out-of-the-box settings. QEMU is used as the virtualizer to execute the guest code directly on the host CPU using the KVM kernel module in Linux. The host is configured to use LVM for VM storage. VMs are set up to use para-virtualization device drivers (`virtio` and `vhost_net`) to boost performance. In addition, the host's Page Cache is bypassed to prevent "double caching".

The guest operating system (OS) uses an EXT4 file-system and does not use LVM. It is configured to use the Noop I/O scheduler. The VM's Kernel Page Cache is only bypassed when we study I/O performance with no caching mechanism enabled, neither at the host, nor at the guest.

B. Methodology and Metrics

Tools, Scenarios and Workloads. In our work, we use FIO [43], which is a flexible tool that allows designing a variety of workloads and that provides detailed statistics to compute our metrics. FIO is widely used in academia and industry for standard benchmarking, stress testing and for I/O verification purposes.

We run FIO on the physical host with different numbers of concurrent threads. Then with the same configuration, we run FIO on 5 different VM flavors as shown in Table I. Finally, we also study the case of multiple active VMs being instantiated on the same physical host, each with one FIO thread performing I/O operations. We use identical OS and settings for consistency across different measurement scenarios. The same fixed amount of data, namely 4 GBs, was used and distributed evenly across all the threads in our experiments. Our reported performance figures are the result of 10 runs with error bars to verify results variability. [1]

Resource	SVM1	SVM2	SVM4	SVM8	SVM16
vCPU	1	2	4	8	16
Mem.(GB)	1	4	8	16	16

TABLE I: VM specs

In summary, we consider the following scenarios:

- Single VM with multiple threads **(SVM)**: in this case, a single VM is active on the host system, running several concurrent I/O tasks. Our experiments are performed on 5 different VM "flavors", as shown in Table I. For clarity of presentation, we only show results for SVM2, SVM4, and SVM16. Note that this scenario is particular stressful for the guest OS scheduler since it has to switch between active threads that perform IO operations.
- Multiple VMs with a single thread **(MVM)**: in this case, we consider that many "slim" VMs are active on the host system, each with only one thread dedicated to I/O tasks. In this case, due to the underlying system configuration, each VM is of flavor "SVM1", which allows accommodating 16 VM on the same host without overcommitting CPU and memory. Note that this scenario stresses the hypervisor scheduler, since the host has to switch very frequently between active VMs.
- Bare-metal **(BM)**: in this case, multiple FIO threads concurrently run I/O operations on directory placed on the same disk used by the VMs. The bare-metal system is used as the baseline to understand the impact and overheads of virtualization.

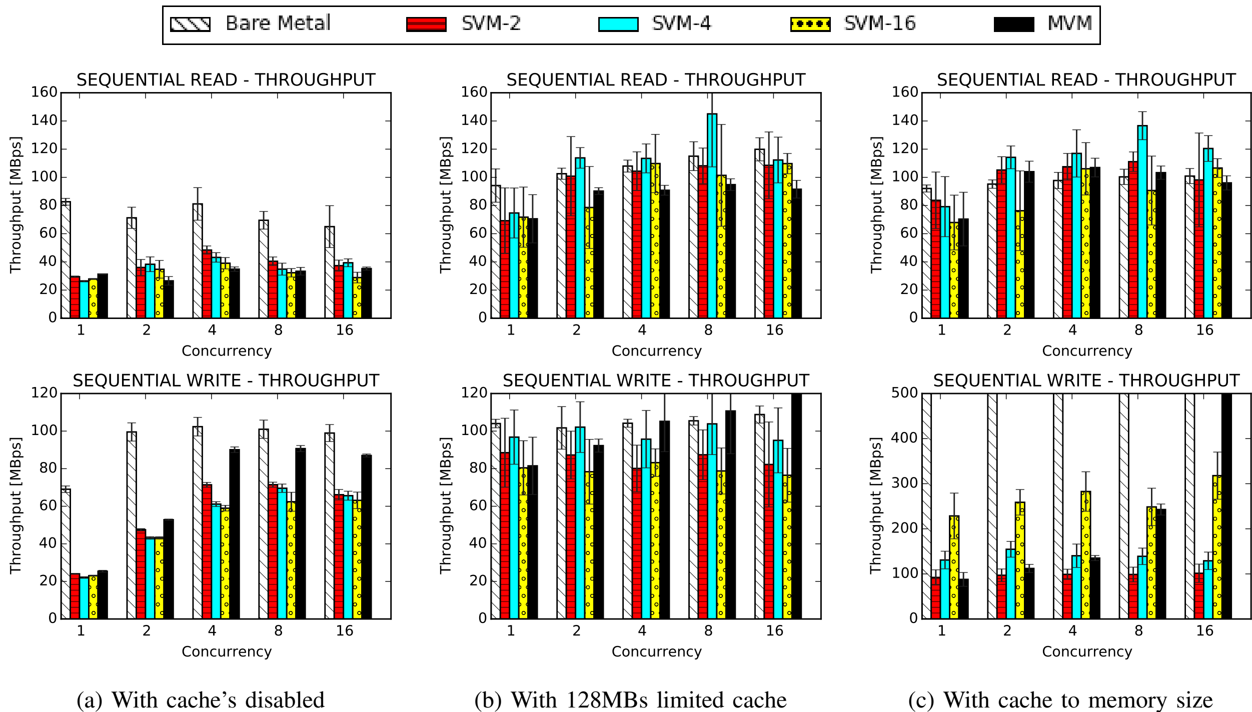
Workloads. To mimic the typical workload of a Big Data application, we configure FIO as follows:

- **I/O mode:** we focus on long, sequential I/O operations. This stems from the application scenario we consider in this work: typical analytics tasks executed by scalable processing frameworks involve batch processing for data transformation, and large-scale machine learning algorithms, for example. These applications operate on data stored in distributed file systems, which is split in large blocks, representing the access "unit" of a parallel task. In this context, Big Data applications perform full scans of the data, which is accessed sequentially [21], [42], to favor throughput over access latency.
- **size:** FIO operates on a 4 GBs data file, both for read and write access. This data is split evenly across the active FIO threads: so, for example, with 16 active threads, each has to read or write 256 MBs of data.[2]
- **direct I/O:** I/O operations performed on files opened with O_DIRECT bypass the kernel's page cache, writing directly to the storage. We examined both situations when direct I/O is set and unset.
- **iodepth:** accounts for how many IO requests to keep in flight. This parameter is only available in asynchronous experiments. We set it to 256 since this is the default size of the I/O scheduler submit queue's size, which ensures we can fill the whole queue with only one thread.
- **ioengine:** indicates the type of I/O library to call. We ran our experiments with Synchronous IO (sync) and Native Asynchronous I/O (libaio). Synchronous IO is a blocking mechanism whereas Native Asynchronous IO is non-blocking. Our results revolve around synchronous and asynchronous modes: the former is representative of the current Hadoop distributed file system, which uses the synchronous Java IO [21], [42]; the latter access method is gradually being adopted by modern parallel frameworks such as Spark [44].

The metrics. In this work we mainly focus on *aggregate throughput*, corresponding to the rate that data is transferred

[1]We also performed experiments with larger data chunks, however, the results are not quatitatively different

[2]This is a typical block size for the Hadoop distributed file system.



(a) With cache's disabled (b) With 128MBs limited cache (c) With cache to memory size

Fig. 3: Aggregate throughput with different cache sizes for **synchronous access**.

across all threads or all VMs being evaluated on the system.

We also evaluate system fairness using the *Jain Fairness Index.* [45]. Studying fairness across concurrent tasks is especially important when considering Big Data workloads. Indeed, an unfair distribution of system resources to I/O threads could lead to the well-known straggler problem [13]. Since analytics jobs are composed of tasks that concur in advancing the work they need to do, a slow task – because of a mistreatment in I/O allocation – can lead to bad overall job performance. Due to lack of space, we do not report details of our results: overall, we observe that fairness indexes are close to optimal value, for all our experiments. This indicates the systems achieve high fairness.

V. Experimental Results

We present the I/O performance of our system using two different access mechanisms. First, we focus on Synchronous I/O, which is the default disk access mechanism used mostly by current Java-based applications. Then, we study Asynchronous I/O, which is often used for system benchmarking and that modern "Big Data" frameworks are gradually adopting.

In particular, we study the impact of the following parameters: the concurrency level (*i.e.,* the number of concurrent threads performing I/O requests), the use of OS-level caching, and the kernel I/O schedulers.

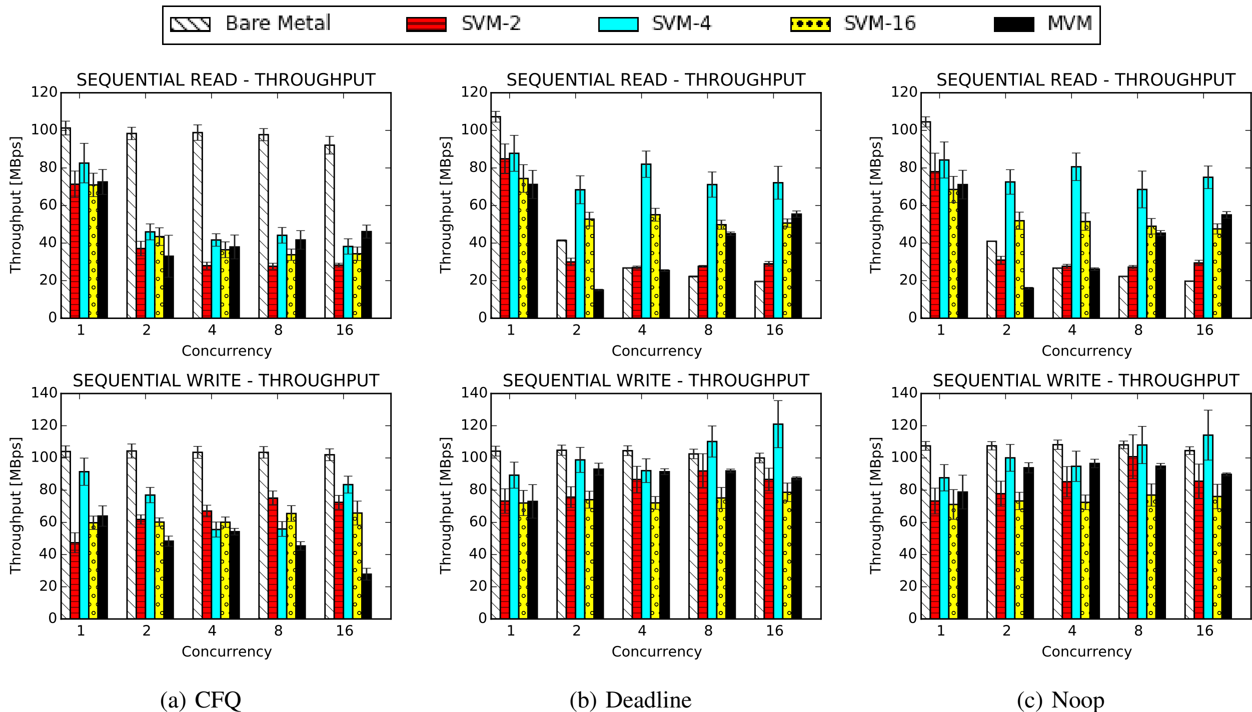
A. Synchronous I/O

With Synchronous I/O, each reader/writer is blocked between I/O requests. We focus on the role played by **caching at the guest OS-level**, use the Deadline I/O scheduler in the host, and the Noop I/O scheduler in the guest OSs, respectively. When caching is disabled (O_DIRECT is set), the application (FIO) running in the guest OS bypasses the intermediate kernel buffer and accesses the backing store directly (the RAID 0 controller, in our setup). When caching is enabled, applications can use the Kernel Page Cache of the guest OS.

Figure 3 shows the aggregate throughput of sequential I/O for 3 scenarios: BM, SVM, and MVM. In general, we observe that caching plays an important role in I/O performance, even with very simple workloads. In addition, we observe that high concurrency levels – which are expected for I/O requests stemming from data intensive frameworks like Hadoop and Spark – also yield substantial benefits for the I/O performance.

Read performance. The top stripe in Figure 3a indicates read performance. When caching is disabled, virtualization overheads are prohibitive: we remark a performance drop of 50%-60% in aggregated throughput, due to the long I/O path taken for requests originating in the guest. Instead, when using the page cache, the "read-ahead" mechanism offers substantial benefits to I/O performance, especially for the kind of workloads we expect from "Big Data" applications, which are sequential in nature. Due to the simple nature of our workload, increasing the page cache size has no substantial impact on performance. Our workloads are read one time only, so caching and cache size would not have any effect.

Write performance. The bottom stripe in Figure 3a indicates write performance. The kernel page cache also plays a crucial role for I/O performance. When caching is disabled, the overhead of virtualization is severe. Instead, when caching is used, applications write data to the page cache and the



(a) CFQ (b) Deadline (c) Noop

Fig. 4: Aggregated throughput with the three I/O schedulers for **asynchronous access**.

page is marked dirty: disk I/O does not happen immediately, and I/O performance increases. Indeed, a synchronous *write()* call returns soon after the data is on the page cache without blocking the application from waiting for the data arrive at the disk. The memory-management subsystem tries to limit dirty pages to a maximum of 15% of the memory on the system. When this threshold is surpassed, or the number of dirty pages is large, or the dirty page reaches its timeout, the kernel starts flushing the dirty pages to disk [46]. So systems have more memory tend to get higher performance. Figure 3c also exposes a limitation in current systems: when the page cache memory is very large, individual VMs, even with a large number of cores, cannot achieve bare-metal throughput. Only multiple VMs (the MVM case) exhibit high performance when the concurrency level is high.

Summary. Our results indicate that caching is crucial. For **read operations**, the "read-ahead" technique and in general caching hot files drastically improves performance, because virtualization overheads due to "long data paths" are mitigated. However, current best practices for Big Data application deployments indicate to avoid OS caching, favoring application-level optimizations. We believe such practices can be challenged by our results. In addition, caching mechanisms for Big Data applications are still in their infancy, with some examples such as HDFS 2.0 and Tachyon [17], and more research is required. For example, memory backed systems (implemented in Java) often suffer from issues due to the garbage collector of the JVM, which is pushing research into off-heap memory management [47], [48]. For **write operations** additional considerations are in order. While our results indicate the benefits of caching in reducing virtualization overheads, failure tolerance should not be sacrificed. Data persistence on disk should be ensured by either direct calls to flush() primitives, or by similar methods at the application-level.

Finally, we observe that "fat" VMs perform poorly: we conjectures that this is the consequence of current NUMA architectures where we do not "pin" VMs to specific cores, and of bad scheduling decisions that allocates compute cores and data far from each other[3]. Thus, Big Data applications deployed in virtual machines should use many "slim" VMs instead of few "fat" ones.

B. Asynchronous I/O

Asynchronous I/O is believed to deliver superior performance by enabling higher level of IO concurrency [33]. In this series of experiments, we investigate the impact of I/O schedulers, since the role of caching is similar to the synchronous setting. In what follows, we disable caching (O_DIRECT is set). The guest OS I/O scheduler is set to noop, and we examine the role of **different I/O schedulers in the host OS**.

Figure 4 overviews the average aggregated throughput with three I/O schedulers: CFQ, Deadline, and Noop respectively. As a general remark, we observe that the host scheduler plays a crucial role in the measured performance: a bad choice can lead to degraded I/O performance. The Deadline and Noop schedulers are friendlier to virtualization: for both read and write operations, VMs outperform the bare-metal.

[3]This phenomenon is also observed at CERN's cluster [49]

Read performance. The top stripe in Figure 4 indicates the performance of asynchronous read operations. Clearly, the CFQ host I/O scheduler damages the performance of VMs. Instead, Deadline and Noop scheduler behave similarly: while this is an artifact due to our simple workload, they both share common observations. With the right choice of the host I/O scheduler, VMs can outperform bare-metal configurations: in our experiments, the VM flavor that reaches the best performance is that with 4 vCores. In addition, we see that for the MVM case, if I/O sharing happens between VMs (which are typical of Big Data applications), performance increases with concurrency levels. However, the performance is optimized when competing between VMs is not existed.

Write performance. The bottom stripe in Figure 4 indicates the performance of asynchronous write operations. In this case, the bare-metal configuration exhibit consistent performance across schedulers. Instead, VM performance drastically improves with Deadline and Noop schedulers, as compared to the CFQ scheduler, reaching roughly the same throughput of the bare-metal counterpart.[4] We also observe that write is superior to read performance: this can be due to the write-back cache mechanism we use in our RAID controller, as opposed to write-through operations, which acknowledges write operations when data resides on the controller memory. In addition, with write-back enabled, the controller can re-order write requests to achieve better throughput.

Summary. The three main conclusions we draw from the asynchronous set of experiments are as follows. First, as modern Big Data applications are gradually adopting the asynchronous model, it is important for cloud providers to configure appropriately the I/O scheduler at the host operating system: failure to do so, can severely impact read and write performance. In addition, we see that choosing an appropriate VM flavor is again crucial for performance. Our results indicate that it is preferable to opt for "not-too-slim" or "not-too-fat" VMs.Besides, if there are VMs sharing I/O bandwidth, either we prefer high concurrency or we improve sharing model of the hypervisor to achieve better performance. More research should be devoted to establish system models to map VM flavors to their expected I/O performance.

VI. Conclusion

Understanding the implications of system virtualization has been a long-standing goal for many years. Such questions are even more relevant today, as many companies and services are migrating their assets to Public and Private Cloud, especially for large-scale data analytics applications.

The goals of our work were to understand the implications and overheads of virtualization on I/O performance, focusing on the storage subsystem. Indeed, many Big Data applications are I/O bound in nature, and a proper assessment and understanding of I/O performance in Cloud environments is essential. To answer our questions, we used an in-depth, low-level measurement study and analyzed the behavior of several configurations supporting the specific workloads that characterize analytics applications, that is, *long sequential operations*. Our findings are instrumental for defining how to configure cloud computing environments to meet high I/O performance demands by modern Big Data applications, and to indicate areas requiring further research efforts.

We showed that current best practices for Big Data application deployments are not reaping the benefits of decades of research and engineering done at the OS level. The caching layers and mechanisms embedded in most modern OSs substantially contribute to high-performance I/O, both for read and write operations. However, critically, it is important to understand the impact of write caching and tolerance to failures. An important avenue for future research is the design of application-level caching mechanisms, that can better exploit the application semantics to improve both read and write performance, without sacrificing failure tolerance.

We also showed that the current trend of modern Big Data applications of using the asynchronous I/O model could face the drawbacks of a poor choice of the I/O scheduler of host operating systems. It is important, for cloud providers, to know that I/O schedulers dramatically affect VM performance: in the asynchronous case, our results indicate that the CFQ scheduler might "interfere" with the asynchronous calls coming from applications running in virtual machines, resulting in performance degradation.

Finally, our results indicate that, depending on the access method, the choice of VM flavor to run Big Data applications also plays a crucial role on performance. In both synchronous and asynchronous case, size of virtual machines demonstrates big impact on I/O performance. This indicates that there is the need to develop models to map VM flavor performance to the underlying physical system, to inform Big Data application deployments.

In the future, we plan to conduct a new measurement study to assess the performance of caching mechanisms such as HDFSv2 and Tachyon, paying particular attention to understand the benefits and overheads of application-level caching with respect to OS-level caching mechanisms.

[4] This is also observed in [41], albeit for different kinds of workloads.

References

[1] Amazon, "Amazon Web Services," https://aws.amazon.com/.

[2] Cloudera, "Cloudera Hadoop on AWS," http://www.cloudera.com/.

[3] DataBricks, "DataBricks Cloud," https://databricks.com/product/databricks.

[4] Apache, "Apache Hadoop," http://hadoop.apache.org/, 2015, [Online; accessed 01-June-2015].

[5] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, vol. 10, p. 10, 2010.

[6] A. Rasmussen, M. Conley, G. Porter, R. Kapoor, A. Vahdat *et al.*, "Themis: an i/o-efficient mapreduce," in *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM, 2012, p. 13.

[7] J. Shafer, "I/o virtualization bottlenecks in cloud computing today," in *Proceedings of the 2nd conference on I/O virtualization*. USENIX Association, 2010, pp. 5–5.

[8] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, B.-G. Chun, and V. ICSI, "Making sense of performance in data analytics frameworks," *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)(Oakland, CA*, pp. 293–307, 2015.

[9] J. Li, Q. Wang, D. Jayasinghe, J. Park, T. Zhu, and C. Pu, "Performance overhead among three hypervisors: An experimental study using hadoop benchmarks," in *IEEE International Congress on Big Data (BigData Congress)*. IEEE, 2013, pp. 9–16.
[10] K. Ye, X. Jiang, Y. He, X. Li, H. Yan, and P. Huang, "vhadoop: a scalable hadoop virtual cluster platform for mapreduce-based parallel machine learning with performance consideration," in *IEEE International Conference on Cluster Computing Workshops (CLUSTER WORKSHOPS)*. IEEE, 2012, pp. 152–160.
[11] J. Park, D. Lee, B. Kim, J. Huh, and S. Maeng, "Locality-aware dynamic vm reconfiguration on mapreduce clouds," in *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*. ACM, 2012, pp. 27–36.
[12] M. Conley, A. Vahdat, and G. Porter, "Achieving cost-efficient, data-intensive computing in the cloud," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*. ACM, 2015, pp. 302–314.
[13] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving MapReduce performance in heterogeneous environments," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 29–42.
[14] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *Proceedings of the 5th European conference on Computer systems*. ACM, 2010, pp. 265–278.
[15] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2.
[16] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: distributed, low latency scheduling," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 69–84.
[17] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Tachyon: Reliable, memory speed storage for cluster computing frameworks," in *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 2014, pp. 1–15.
[18] A. A. Bhattacharya, D. Culler, E. Friedman, A. Ghodsi, S. Shenker, and I. Stoica, "Hierarchical scheduling for diverse datacenter workloads," in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 4.
[19] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 423–438.
[20] A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Choosy: Max-min fair sharing for datacenter jobs with constraints," in *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 2013, pp. 365–378.
[21] J. Shafer, S. Rixner, and A. L. Cox, "The hadoop distributed filesystem: Balancing portability and performance," in *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*.
[22] U. Drepper, "The cost of virtualization," *Queue*, vol. 6, no. 1, pp. 28–35, Jan. 2008.
[23] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Antfarm: Tracking processes in a virtual machine environment," in *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference*, ser. ATEC '06, 2006, pp. 1–1.
[24] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel, "Diagnosing performance overheads in the xen virtual machine environment," in *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, ser. VEE '05, 2005, pp. 13–23.
[25] L. Cherkasova and R. Gardner, "Measuring cpu overhead for i/o processing in the xen virtual machine monitor," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05, 2005, pp. 24–24.
[26] J. LeVasseur, V. Uhlig, M. Chapman, P. Chubb, B. Leslie, and G. Heiser, "Pre-virtualization: Slashing the cost of virtualization," Fakultät für Informatik, Universität Karlsruhe (TH), Technical Report 2005-30, Nov. 2005.
[27] A. Sundararaj, A. Gupta, P. Dinda *et al.*, "Increasing application performance in virtual environments through run-time inference and adaptation," in *Proceedings. 14th IEEE International Symposium on High Performance Distributed Computing (HPDC-14)*. IEEE, 2005, pp. 47–58.
[28] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski, "Towards scalable multiprocessor virtual machines." in *Virtual Machine Research and Technology Symposium*. Citeseer, 2004, pp. 43–56.
[29] J. F. Kloster, J. Kristensen, and A. Mejlholm, "On the feasibility of memory sharing," Ph.D. dissertation, Aalborg University. Department of Computer Science, 2006.
[30] M. Schwidefsky, H. Franke, R. Mansell, H. Raj, D. Osisek, and J. Choi, "Collaborative memory management in hosted linux environments," in *Proceedings of the Linux Symposium*, vol. 2, 2006.
[31] C. A. Waldspurger, "Memory resource management in vmware esx server," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 181–194, 2002.
[32] B. H. Lim *et al.*, "Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor," 2001.
[33] K. Huynh and A. Theurer, "KVM Virtualized I/O Performance: Achieving Unprecedented I/O Performance Using Virtio-Blk-Data-Plane Technology Preview in SUSE Linux Enterprise Server 11 Service Pack 3 (SP3)," IBM Linux Technology Center, Tech. Rep., 06 2013.
[34] R. Russell, "virtio: towards a de-facto standard for virtual i/o devices," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 95–103, 2008.
[35] M. Kesavan, A. Gavrilovska, and K. Schwan, "Differential virtual time (dvt): rethinking i/o service differentiation for virtual machines," in *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 2010, pp. 27–38.
[36] S.-H. Ha, D. Lopez-Pacheco, and G. Urvoy-Keller, "Networking in a virtualized environment: The tcp case," in *IEEE 2nd International Conference on Cloud Networking (CloudNet)*. IEEE, 2013, pp. 50–57.
[37] V. Chadha, R. Illiikkal, R. Iyer, J. Moses, D. Newell, and R. J. Figueiredo, "I/o processing in a virtualized platform: a simulation-driven approach," in *Proceedings of the 3rd international conference on Virtual execution environments*. ACM, 2007, pp. 116–125.
[38] S. Govindan, A. R. Nath, A. Das, B. Urgaonkar, and A. Sivasubramaniam, "Xen and co.: communication-aware cpu scheduling for consolidated xen-based hosting platforms," in *Proceedings of the 3rd international conference on Virtual execution environments*. ACM, 2007, pp. 126–136.
[39] D. Ongaro, A. L. Cox, and S. Rixner, "Scheduling i/o in virtual machine monitors," in *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. ACM, 2008, pp. 1–10.
[40] S. R. Seelam and P. J. Teller, "Virtual i/o scheduler: a scheduler of schedulers for performance virtualization," in *Proceedings of the 3rd international conference on Virtual execution environments*. ACM, 2007, pp. 105–115.
[41] D. Boutcher and A. Chandra, "Does virtualization make disk scheduling passé?" *SIGOPS Oper. Syst. Rev.*, vol. 44, pp. 20–24, Mar. 2010. [Online]. Available: http://dx.doi.org/10.1145/1740390.1740396
[42] E. Sammer, *Hadoop Operations*, 1st ed. O'Reilly Media, Inc., 2012, ch. 2.
[43] J. Axboe, "Flexible I/O Tester," https://github.com/axboe/fio, 2015, [Online; accessed 01-June-2015].
[44] Apache, "Spark 0.6.0 Release, Engine changes," https://spark.apache.org/releases/spark-release-0-6-0.html, 2012, [Online; accessed 01-June-2015].
[45] R. Jain, D. Chiu, and W. Hawe, "A Quantitative Measure Of Fairness And Discrimination For Resource Allocation In Shared Computer Systems," Sep. 1998. [Online]. Available: http://arxiv.org/abs/cs/9809099
[46] D. Bovet and M. Cesati, *Understanding the Linux Kernel*.
[47] A. Flink, "Off-heap Memory in Apache Flink and the curious JIT compiler," http://flink.apache.org/news/2015/09/16/off-heap-memory.html.
[48] I. Gog, J. Giceva, M. Schwarzkopf, K. Vaswani, D. Vytiniotis, G. Ramalingam, M. Costa, D. G. Murray, S. Hand, and M. Isard, "Broom: sweeping out garbage collection from big data systems," in *Proc. of the Usenix HotOS Workshop*, 2015.
[49] S. Crosby, A. Wiebalck, and U. Schwickerath, "NUMA and CPU Pinning in High Throughput Computing," http://openstack-in-production.blogspot.fr/2015/08/numa-and-cpu-pinning-in-high-throughput.html, 2015, [Online; accessed 01-Feb-2016].