Priority IO Scheduling in the Cloud

Filip Blagojević, Cyril Guyot, Qingbo Wang, Timothy Tsai,
Robert Mateescu and Zvonimir Bandić

HGST Research
3403 Yerba Buena Road
San Jose, CA 95135

Abstract

Current state of the art runtime systems, built for managing cloud environments, almost always assume resource sharing among multiple users and applications. In large part, these runtime systems rely on functionalities of the node-local operating systems to divide the local resources among the applications that share a node. While OSes usually achieve good resource sharing by creating distinct application-level domains across CPUs and DRAM, managing the IO bandwidth is a complex task due to lack of communication between the host and IO device. In our work we focus on controlling the hard disk drive (HDD) IO bandwidth available to user-level applications in a cloud environment. We introduce priority-based (PBS) IO scheduling, where the ordering of IO commands is decided cooperatively by the host and IO device. We implemented our scheduling policies in the Linux storage stack and Hadoop Distributed File System. Initial results show that in a cloud environment, the real-time commands managed by PBS outperform the real-time IO scheduling of the Linux kernel by up to a factor of ~ 5 for the worst case latency, and by more than 2x for average latency.

1 Introduction

In recent years, cloud platforms emerged as an indispensable resource for development of novel large-scale applications in industry and academia. Cost-effectiveness and availability of computational and storage resources at a large scale, provide opportunities for complex algorithms that are computationally expensive and can generate large amounts of data.

The potential of cloud computing has been convincingly demonstrated in numerous studies and real-world examples [13, 2]. In terms of computational models, cloud environments are not restrictive, but a common computational model used in these environments is embarrassingly parallel (map-reduce). Widely accepted due to the easiness of use, map-reduce is also convenient for performing large-scale data analysis – a workload frequently present in large scale storage systems.

Cloud systems allow sharing and oversubscription of distributed resources. Two main reasons motivate resource sharing: (i) to further improve the cost effectiveness and reduce power consumption, cloud providers frequently enforce resource sharing among the users; (ii) background system jobs can interfere with user initiated jobs. Furthermore, it is very common that the jobs sharing the resources in a cloud have different latency requirements. For example, while serving real-time queries from a distributed database, certain nodes can also host background, low-priority jobs. Some examples of the low-priority jobs are data analytics map-reduce jobs, load-balancing jobs, and data recovery jobs that are triggered by the system in case of a node failure. Computing environments, where multiple jobs with different latency requirements run simultaneously, require careful scheduling of computational and IO tasks.

To achieve fair resource distribution among users/applications, system architects developed various runtime systems that provide resource virtualization and to each user/application give the impression of running in isolation. Sharing techniques in modern operating systems have been studied for resources such as CPUs and DRAM [11, 3], i.e. individual processes can be limited on the number of CPU cores and the amount of DRAM they are allowed to use. However, achieving efficient multiplexing of the IO requests is a challenging problem, mostly due to the independent operational structure of the IO device.

We explore the opportunities for host-HDD co--operative IO scheduling and examine the effects that IO *misscheduling* has in a shared resource environment. In current computing/storage systems, IO request scheduling is a two-level process: (i) the initial request reordering/merging is performed by the local filesystem, and (ii)

the second-level request reordering is performed within the IO device. The existing host-side IO request schedulers do not interact with the IO device, and any prioritized request issued from the host-side will almost certainly be re-ordered once it reaches the drive (we refer to this re-ordering as *missscheduling*). As a result, in a cloud where multiple jobs might exist on the same node, high--priority real-time data accesses cannot gain significant advantage over low-priority IO requests.

To address IO multiplexing in a cloud, we design a priority-based scheduler (PBS) that exploits HDD priority commands and instructs the drive to process certain (usually real-time) requests at high priority. To the best of our knowledge, this is the first IO scheduler that allows passing priority information between a distributed filesystem and the storage device, aiming at improving the response latency for high-priority requests. PBS is not limited to cloud environments. However, priority scheduling is most effective in a cloud due to the intensive sharing of storage devices among applications with different latency requirements. We integrate PBS with Hadoop distributed filesystem (HDFS) and HBase – a NoSql database frequently used to manage large amounts of cloud data. In our prototype implementation, we assign priorities to HDFS and HBase IO requests, and allow these priorities to be passed to the priority-based scheduler and further to the storage device.

We conduct a set of experiments where real-time HBase queries compete with background map-reduce jobs and find that co-operative priority-based scheduling improves the worst-case response time for real-time queries by up to a factor of ~ 5 and the average response time by more than 2x.

2 Hadoop / HBase

Although our work is not limited to any specific cloud environment, we choose to integrate our scheduler with Hadoop/HBase, as these are runtimes widely used in current cloud systems. Hadoop/HBase, or a similar variation, is currently installed in a large number of academic and industrial cloud systems used by companies such as Facebook, Google and Amazon [2, 7].

Hadoop is a scalable runtime designed for managing large-scale storage systems. The Hadoop runtime contains Hadoop Filesystem (HDFS) and a map-reduce framework, suitable for algorithms that target processing of large amounts of data. HDFS was initially developed as an open source version of the Google filesystem (GFS) and contains some of the key features initially designed for GFS.

HDFS also serves as an underlying distributed filesystem for HBase – a NoSQL database that allows real-time fine-grained accesses to a large amount of data. HBase is designed based on Bigtable [4], a NoSQL database initially developed by Google. HBase can serve many users simultaneously, and at the same time the HBase data can be used in map-reduce style data analytics.

3 IO Request Scheduling in OS / HDD

On a single node, scheduling of the IO requests is performed at two levels: (i) at the top level, the IO request reordering and merging is performed by the filesystem scheduler, (ii) at the lower level, the IO requests are reordered by the IO device. The lack of communication between the two scheduling layers can result in poor scheduling decisions for the real-time IOs.

For example, the default Linux scheduler, Completely Fair Queueing (CFQ), maintains multiple scheduling classes: real-time (RT), best-effort (BE), and idle. RT requests have priority over the other two classes. However, the HDD does not recognize the IO priorities assigned at the OS level and allows request reordering, aiming at achieving the highest possible IOPS rate. Consequently, if reordered within the HDD, the RT requests can experience high latencies or even starvation.

4 Priority-Based Scheduler

Priority-Based Scheduler (PBS) enables prioritization of read IO requests by issuing SATA priority commands directly to the HDD, therefore forcing the drive to process certain IO requests with high priority. SATA priority commands are part of the SATA specification and are passed to the drive by setting a specific bit in the Command Descriptor Block (CDB). PBS is middleware that resides between the distributed filesystem and the HDD, as presented in Figure 1, and is composed of kernel and user modules. Spanning kernel- and user-space is required: the scheduler intercepts *read* IO commands issued by the distributed filesystem (user-level) and further passes the intercepted commands to the HDD (from within the kernel). Depending on the application issuing the IO requests, PBS decides if the requests will be prioritized or not. Note that setting high priority for a request does not guarantee immediate processing when the HDD contains multiple high-priority requests.

It is worth mentioning that the strict ordering of IO requests can be achieved if the device queue is reduced to a single request, i.e. the host never issues two or more IO requests concurrently. In that case, the ordering of the commands established by the host scheduler will be enforced. However, reducing the device queue depth to one can reduce the random IOPS rate by up to 50% (even more for sequential IOs), if the storage device is hard disk drive. PBS enforces command ordering (based on

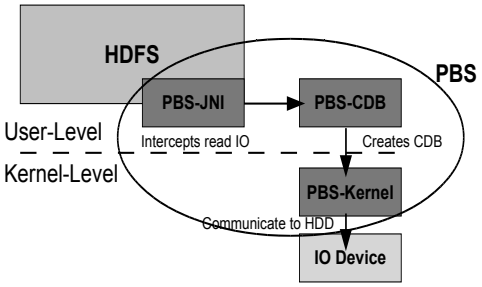

Figure 1: PBS spans both kernel and user space. PBS communicates with the IO device through the modified SGIO interface, and with the HDFS through the PBS-JNI module.

IO priorities), but PBS also allows multiple concurrent IO requests to be issued to the storage device, therefore maximizing the IOPS rate.

4.1 User-Level PBS

The user-level PBS module is composed of two entities: (i) the PBS-JNI library that is attached to HDFS and intercepts the *read* IO requests issued by the HDFS-Server, and (ii) the PBS-CDB library that creates the priority CDB and passes the CDB to the kernel PBS module.

The user-level PBS library communicates directly to the kernel-level PBS module and bypasses the standard kernel IO interface. To allow PBS to read from the DRAM cache and avoid accessing the HDD when data is cached, we use the *mincore* system call, which determines if a file (or parts of a file) is present in the DRAM. In case data is cached, the PBS issues a *read* call through the standard filesystem interface and let the filesystem retrieve the data from the cache.

In the current implementation, data fetched through PBS does not get cached in the DRAM. Caching can be achieved either by introducing a PBS cache layer or by merging PBS with the existing filesystem infrastructure. The reader should note that the current lack of caching only hurts PBS, and the advantage of PBS over the Linux IO scheduling could only improve.

The PBS-JNI library accepts several parameters from the HDFS, such as the file descriptor, the offset in the file, the number of bytes to read, the buffer where the returned data is stored, and the IO priority level. Using the *fiemap* system call, the PBS-JNI library translates the file descriptor and the file offset into the LBA. In case of high priority IO, the PBS-JNI sends all the IO request information to PBS-CDB. Upon generating the Command Description Block, PBS-CDB sends the CDB to the drive via the PBS-kernel module.

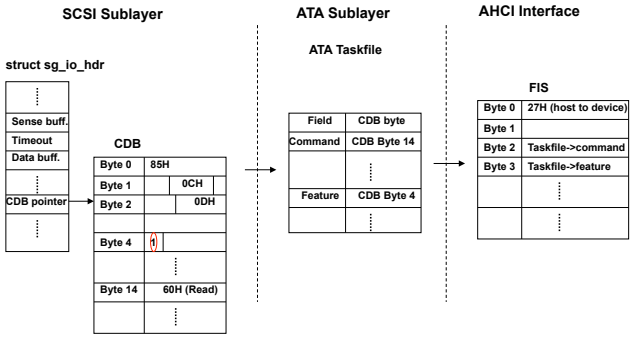

Figure 2: PBS-Kernel module

4.2 Kernel-Level PBS

Kernel-Level PBS implements two steps. First, PBS converts the CDB to an appropriate data format, such as FIS (Frame Information Structure), when using a SATA AHCI interface. The second step includes dispatching of the prioritized request to the HDD, where the HDD scheduler prioritizes the request. We further introduce implementation schemes for each of the steps.

LIBATA in the Linux kernel can conveniently be extended to accomplish the first step. When a SATA drive is used, a SCSI command with the priority bit set in the CDB structure can be sent via the SCSI generic driver interface (SG_IO) directly to the HDD. This process is explained as an example in Figure 2. We use a 16-byte CDB with operation code 0x85H (ATA Command Pass-Through). Byte 5, bit 7 of the CDB is set to 1 for a priority command and 0 for non-priority one. The SCSI ATA Translation Layer (SATL) transfers the content of the CDB into the Task File of the SATA protocol without losing priority setup. This data structure is further converted into a Frame Information Structure (FIS). During the second step a FIS is sent to the HDD using the AHCI interface. Note that the CDB is part of a data structure called sg_io_header when it is passed from user-level library to the kernel-level drivers. Certain other fields within sg_io_header are also essential for the success of the priority-based scheduling but will not be discussed in details here. Kernel-level patch (PBS-Kernel) and user-level library that creates priority CDB (PBS-CDB) are open-source and available for download[1]. Within the drive, the priority commands are placed ahead of all non-priority commands in the IO queue.

4.3 PBS Integration with HDFS/HBase

To allow interaction between HDFS and PBS, we introduce IO priorities in Hadoop. In our current work we assume that read requests from various applications run-

[1] http://sourceforge.net/projects/iccncq/

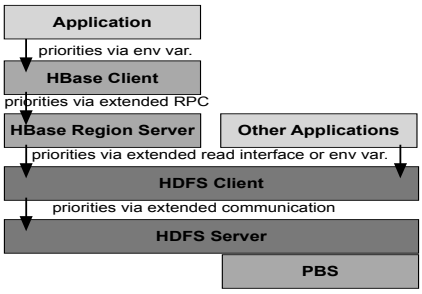

Figure 3: Hadoop software stack with extensions for passing IO priorities.

ning in the Hadoop environment are prioritized and the priorities have been determined ahead of time. One common prioritization scheme is to allow real-time operations (such as HBase queries) to be processed at a high priority, while long running map-reduce (analytics) jobs would be assigned low priority.

In HDFS, we introduce priorities on both the client and the server side. The HDFS client receives the IO priority from the application either through an environment variable, or through an extended *read()* call that accepts the priority level as an additional parameter. Upon obtaining the application's IO priority, the HDFS client passes the priority to the HDFS server through an extended client-server communication protocol.

We also introduce priorities in HBase. HBase also has a client-server architecture. In the software stack HBase is one level above HDFS – HBase runs on top of HDFS, see Figure 3. To enable priority-based reads in HBase, we extend the HBase client side to accept the IO priorities from the application and pass them to the HBase server side. On the server side of HBase, we replace the HDFS *read()* calls with the extended calls that accept the priority value.

5 PBS Performance

In this section we focus on comparing PBS with CFQ, the default Linux IO scheduler. For performance comparison we run real-time HBase queries generated by Yahoo! Cloud Service Benchmark (YCSB) [6], as well as low-priority map-reduce applications. Our software stack comprises PBS at the bottom, HDFS above the PBS and HBase above HDFS.

5.1 Experimental Platform

The experimental platform we use is composed of four servers connected via a 1GbE network. Each server contains 2 Intel Xeon E5620 CPUs running at 2.4Ghz, 24GB of DRAM and four 3TB Hitachi UltraStar 7K3000

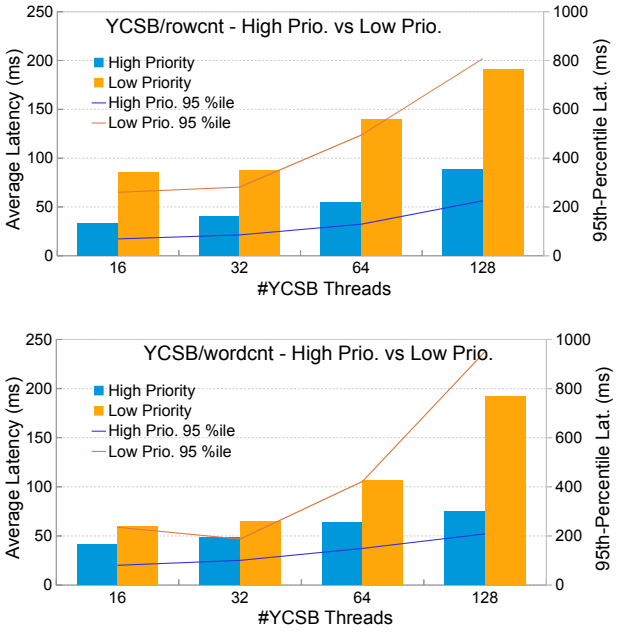

Figure 4: YCSB with high and low priority IOs, when rowcounter and wordcount applications run in the background. Bars represent average latency and are associated with the left y-axis. Lines represent 95th-percentile latency and are associated with the right y-axis.

HDDs with a customized firmware that supports request prioritization. The servers run Linux kernel 2.6.35, extended with the PBS scheduling module. We use HDFS 0.20.2 and HBase 0.20.6, with our extensions that allow IO prioritization as well as priority passing to lower levels of the system stack.

When running HDFS across the cluster, one node is exclusively used as a metadata server, while the other 3 nodes are data nodes. In HDFS, the data replication factor is 3. Since each node contains 4 HDDs, we run 4 Hadoop datanode processes on each node, and we let each datanode process manage a single HDD. Therefore, our setup contains 12 HDFS datanodes. When running HBase, we found that two region servers per physical node provide the best performance in terms of memory consumption and read latency.

5.2 HBase with MapReduce Applications

We capture the effects of PBS in a multi-application environment by simultaneously running the YCSB-HBase--HDFS benchmark and map-reduce jobs. Figure 4 presents the average and 95th-percentile HBase read latency for high/low priority IO requests and map-reduce applications running in the background. High priority HBase IOs are issued through the PBS, whereas low-priority HBase queries are completed by the Linux

scheduler. We set the YCSB benchmark to complete 2000 queries, randomly distributed across the dataset. As background jobs, we use two map-reduce applications: *rowcounter* and *wordcount*. *rowcounter* counts the total number of rows in the HBase database, and *wordcount* returns the number of occurrences for each word in a given set of files. Both mapreduce applications come with an HDFS/HBase distribution, and represent a typical workload used in a cloud environment. *rowcounter* runs on our 73GB HBase set, while for *wordcount* we create a set of files with a total size of 50GB.

For map-reduce jobs we allow up to 4 map and 4 reduce jobs per HDD. Since our server nodes contain 4 HDDs, maximum allowed number of jobs per node can be 32. Note that the number of jobs per node actually running at any time is smaller, since not all tasks run at the same time.

In both experiments average IO latency with PBS outperforms Linux scheduling by more than a factor of 2. It is interesting to note that the 95th-percentile latency gap significantly increases with the number of YCSB threads in use. This is explained by higher HDD contention which can result in starvation of low-priority requests. High-priority IO requests, routed through PBS, compete only with other high-priority requests, and in the presented experiments the HDD anti-starvation mechanism is not triggered for high-priority requests. The highest recorded 95th-percentile latency difference between PBS and standard scheduling is close to a factor of 5. PBS can significantly affect the performance of the background jobs, particularly when competing with a large number of high-priority requests. Our assumption is that the background jobs are long lasting analytics jobs that are not performance critical.

6 Discussion

In this section we discuss various properties of PBS, and address applicability of PBS in different environments.

While in the presented study we focus on a case where IOs are handled with either high or low priority, it is not difficult to imagine environments where multiple priority levels would bring significant benefits to the overall QoS. PBS relies on the SATA software stack in Linux kernel to deliver priority IO requests. Although the SATA command set natively supports only two priority levels, multiple priority levels can be achieved through the SATA isochronous commands, also supported by PBS.

Virtualized environments represent another area that could significantly benefit from priority-based IO scheduling. Prioritization in virtualized environment can be accomplished in two ways: (i) applications within a single guest domain could be assigned priorities, or (ii) priorities could be assigned per guest domain. Supporting either of the two approaches requires changes in the hypervisor storage virtualization driver. The storage virtualization driver is required to accept the CDB created by the user-level PBS and pass it down through the storage stack. Note that this CDB is of somewhat different format than what the driver is by default ready to accept (contains priority bits). Required changes in the hypervisor storage driver are similar to the implementation of the PBS kernel-level module, i.e. the CDB data structure needs to be updated with the priority bits and this information needs to be passed further down to the device. Enabling PBS in a virtualized environment will be addressed in future work.

With the introduction of priorities, possible IO request starvation becomes an important problem. If not handled with care, starvation significantly affects the performance of low-priority jobs. The starvation problem can be addressed in multiple ways, including both host and device-level approaches. Commonly, storage device firmware is designed to trigger the anti-starvation mechanism for IO requests that have spent a certain time in the processing queue without a successful completion. The anti-starvation mechanism enables progress of low-priority requests even in a highly contended environment. However, the latency of an IO request that was processed by the anti-starvation mechanism usually exceeds 5 seconds. Consequently, in a mixed environment of high and low-priority requests, no real-time queries can be processed at low-priorities, if only the device anti-starvation mechanism is used. To prevent starvation at the host level, PBS supports autonomous priority increase, where long lasting low-priority IO requests can be canceled and reissued as high-priority requests. This approach allows improving the worst case latency of low-priority commands, at the cost of increased latency for high-priority commands.

7 Related Work

IO scheduling has been the main topic of many studies in the past years. We briefly summarize published research in the area of scheduling in a multi-process environment. Closest to our work is the study performed by Young Jin et al. [15], which address the issue of host-IO device misscheduling. They acknowledge the misscheduling problem and design a dynamic scheduler that allows switching on/off the host and the device schedulers at runtime. They do not attempt passing the IO priorities to the device. A set of real-time scheduling algorithms has been developed [14, 5]. While they rely on SCAN to deliver highest throughput, we allow our priority commands to be processed completely independently of the standard drive request processing algorithm. Other traditional schedulers [10, 12] allow IO priorities, but do not

pass them to the IO device.

Multiple studies [1, 8, 9] improve IO scheduling in distributed systems by carefully placing IO- and computation-bound workload and focusing on the effects of multi-level IO scheduling in a virtualized/cloud environment. These studies are limited to the host side and do not include priority-aware IO devices.

8 Conslusions and Future Work

Communication between distributed filesystems and IO devices is a powerful tool that enables low IO access latencies for real-time workloads in a multi-user cloud environment. We proposed and implemented PBS, a scheduling scheme that allows passing of IO priorities directly to the storage device and therefore prevents reordering of high-priority IO requests once they reach the HDD. Our prototype implementation of PBS is employed with the Hadoop/HBase software stack, and the focus of our study is on the impact that misscheduling and IO priorities might have on the cloud system user. We find that misscheduling can have significant impact on real-time IO requests when applications with different priorities compete for the same storage resources. Our experiments show that priority scheduling outperforms default Hadoop/Linux scheduling by up to a factor of ~ 5 for the worst case latency and by more than 2x for average latency. Prioritizing certain IO requests comes at a significant cost for low-priority requests, and frequently these low-priority requests can be completed only after triggering the anti-starvation mechanism.

The focus of our future work is multi-faced. We will further investigate anti-starvation schemes that would allow latency trade-off between high and low-priority IO requests. Also, we believe the applicability of prioritization will become important in virtualized environments where users have different IO latency expectations. Finally, using multi-level priorities, either through SATA deadline commands or as a part of a host-side scheduler, is another important aspect of our future work.

References

[1] D. Aragiorgis, A. Nanos, and N. Koziris. Coexisting scheduling policies boosting I/O virtual machines. In *Proceedings of the 2011 International Conference on Parallel Processing - Volume 2*, Euro-Par'11, pages 407–415. Springer-Verlag, 2012.

[2] D. Borthakur, J. Gray, J. S. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, R. Schmidt, and A. Aiyer. Apache Hadoop goes realtime at Facebook. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 1071–1080, 2011.

[3] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: an operating system for many cores. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08. USENIX Association, 2008.

[4] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th USENIX Conference on Operating Systems Design and Implementation*, OSDI '06. USENIX Association, 2006.

[5] H.-P. Chang, R.-I. Chang, W.-K. Shih, and R.-C. Chang. Reschedulable-Group-SCAN scheme for mixed real-time/non-real-time disk scheduling in a multimedia system. *J. Syst. Softw.*, 59(2):143–152, Nov. 2001.

[6] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154. ACM, 2010.

[7] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proceedings of the nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[8] Y. Hu, X. Long, J. Zhang, J. He, and L. Xia. I/O scheduling model of virtual machine based on multi-core dynamic partitioning. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 142–154, New York, NY, USA, 2010. ACM.

[9] S. Ibrahim, H. Jin, L. Lu, B. He, and S. Wu. Adaptive Disk I/O Scheduling for MapReduce in Virtualized Environment. In *Proceedings of the 2011 International Conference on Parallel Processing*, ICPP '11, pages 335–344, Washington, DC, USA, 2011. IEEE Computer Society.

[10] S. Iyer and P. Druschel. Anticipatory scheduling: a disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *Proceedings of the eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 117–130, New York, NY, USA, 2001. ACM.

[11] M. K. Jeong, D. H. Yoon, D. Sunwoo, M. Sullivan, I. Lee, and M. Erez. Balancing dram locality and parallelism in shared memory cmp systems. In *HPCA*, pages 53–64, 2012.

[12] C. R. Lumb, J. Schindler, G. R. Ganger, D. F. Nagle, and E. Riedel. Towards higher disk head utilization: extracting free bandwidth from busy disk drives. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, OSDI'00, pages 7–7, Berkeley, CA, USA, 2000. USENIX Association.

[13] A. Matsunaga, M. Tsugawa, and J. Fortes. CloudBLAST: Combining MapReduce and Virtualization on Distributed Resources for Bioinformatics Applications. In *Proceedings of the 2008 Fourth IEEE International Conference on eScience*, ESCIENCE '08, pages 222–229, Washington, DC, USA, 2008. IEEE Computer Society.

[14] A. L. N. Reddy, J. Wyllie, and K. B. R. Wijayaratne. Disk scheduling in a multimedia I/O system. *ACM Trans. Multimedia Comput. Commun. Appl.*, 1(1):37–59, Feb. 2005.

[15] Y. J. Yu, D. I. Shin, H. Eom, and H. Y. Yeom. NCQ vs. I/O scheduler: Preventing unexpected misbehaviors. *ACM Transactions on Storage*, 6(1):2:1–2:37, Apr. 2010.