Performance Inconsistency in Large Scale Data Processing Clusters

Mingyuan Xia and Nan Zhu
McGill University

Yuxiong He and Sameh Elnikety
Microsoft Research Redmond

Xue Liu
McGill University

Abstract

A large shared computing platform is usually divided into several virtual clusters of fixed sizes, and each virtual cluster is used by a team. A cluster scheduler dynamically allocates physical servers to the virtual clusters depending on their sizes and current job demands. In this paper, we show that current cluster schedulers, which optimize for instantaneous fairness, cause performance inconsistency among the virtual clusters: Virtual clusters with similar loads see very different performance characteristics.

We identify this problem by studying a production trace obtained from a large cluster and performing a simulation study. Our results demonstrate that when using an instantaneous-fairness scheduler, a large VC that contributes more resources during underload periods can not be properly rewarded during its overload periods. These results suggest that not using resource sharing history is the root cause for the performance inconsistency.

1 Introduction

Data-intensive computing is important for a large number of applications, including large-scale data mining, data analytics, and bioinformatics. At the same time, clusters of commodity servers are a major computing platform, powering these large-scale data-intensive applications. Driven by this trend, researchers and practitioners have been developing various cluster computing frameworks to simplify the programming of clusters and to use cluster resources efficiently. Prominent examples include MapReduce [7], Hadoop [4], Dryad [10], and Cosmos [6], among others [19, 15, 17].

A large cluster is normally shared among several teams within an organization, rather than being dedicated to a single team. The benefits of sharing are compelling: First, sharing allows teams to exploit a large number of servers that would be infeasible without sharing. Second, from the system point of view, sharing improves resource utilization by multiplexing the resources among several teams. For example, the web document ranking team in large commercial search engine runs its ranking algorithm (e.g., PageRank) daily for a massive number of crawled documents, running on thousands of servers and lasting for a few hours. Without sharing, the ranking team would need to provision a large dedicated cluster, which will be underutilized.

As a concrete example we consider Cosmos [6], which is a production system that executes jobs similar to those in MapReduce and Hadoop and is used extensively inside Microsoft. A Cosmos cluster can span over 100,000 servers. Organizational units within Microsoft pay for a portion of the cluster, and in return receive a "virtual cluster" (VC). For example, a cluster user (i.e., an organizational unit) pays the cost of 1,000 servers and in return receives a VC of 1,000 servers to run its jobs. Servers in a VC are not dedicated, but are allocated dynamically whenever the VC has jobs. Furthermore, additional idle servers (if available) can be allocated to a busy cluster temporarily. Therefore, although the size of a user VC is only 1,000 servers, its VC can use many more idle servers from other idle VCs.

Sharing brings a key challenge: long term fairness. We want to ensure fairness within a large enough time window among VCs when they compete for resources. Figure 1 shows the performance of 115 VCs in a large Cosmos cluster during one month. Each point represents one VC. The X-axis shows the load, which is equal to the total work (server hours) of the VC in the month divided by its total capacity (number of servers). In other words, load factor = 1 is equivalent to having 100% utilization for the VC during the month. The Y-axis shows the average stretch of jobs in the VC. *Stretch* is response time normalized to job size and VC capacity (as defined later). The figure shows both the merits and challenges of sharing. On the positive side, sharing allows some VCs to use more than their capacity. VCs with load factor

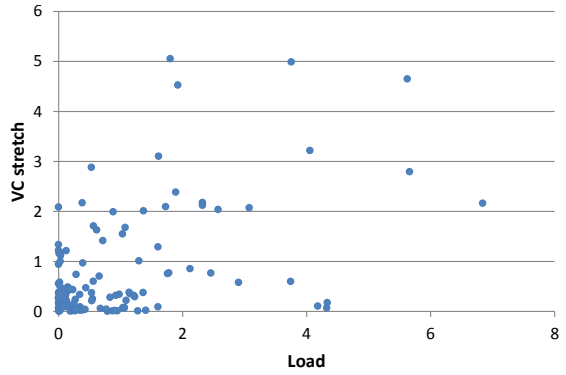


Figure 1: The performance of 115 virtual clusters (VCs) monitored from a large Cosmos system during one month (Sep. 2011).

>1 benefit from sharing: In particular 10 VCs have load factor >3, which is equivalent to 300% utilization. Furthermore, overall system utilization is increased; without sharing many clusters would be underutilized. On the negative side, the figure shows a major problem: Some VCs with less than 50% utilization have long stretches (with response times over a few hours), and in contrast some VCs with very high load have short stretches (with very short response times in minutes).

The figure shows long-term unfairness: Two VCs with similar load can have dramatically different responsiveness, and a much more heavily loaded VC may even have much better performance (smaller stretch) than a lightly-loaded VC. This causes several problems to the system operator. Performance inconsistency is the most prominent problem. Teams may even double the size of their VCs with little or no performance improvement. This has direct financial implications since teams are charged for owning servers, and teams paying the same amount of money may have very different performance experiences.

To address these challenges, we perform a trace-driven study based on a production trace of a large Cosmos cluster to reveal the causes of performance inconsistency in real systems. A traditional cluster scheduler [6, 18] uses mainly the current demand and capacity to make scheduling decisions, which we call an instantaneous-fairness scheduler. The well-known MaxMin fairness scheduler [13] and Hadoop fair scheduler [3] are examples of an instantaneous-fairness scheduler. We find that such schedulers do not exploit VC usage and sharing history, and therefore, they do not provide performance consistency among VCs over time (or "long-term fairness").

The contributions of this work are two-fold: (1) We identify the performance inconsistency problem in shared computing clusters. (2) We build a simulator and use a production trace from a large cluster to show how instantaneous-fairness schedulers cause performance inconsistency.

The remainder of the paper is organized as follows: Section 2 elaborates the scheduling model. Section 3 describes our evaluation methodology including workload, simulation design and performance metric. Section 4 uses the simulation results to illustrate performance inconsistency of instantaneous-fairness schedulers and its cause. Section 5 discusses related work and Section 6 discusses the design challenge of the solution. Finally, Section 7 presents our conclusions.

2 Scheduling Model

We explain how users interact with Cosmos and model the system in terms of resource distribution and allocation.

Each Cosmos user (a user here is a team) owns a virtual cluster (VC) that has a *capacity* in terms of the number of servers purchased by the team. A user submits jobs to its VC, and the demand of all jobs in a VC constitutes the VC's *demand*. Instead of allocating a fixed number of servers to VCs, most systems [3, 18] allow VC server *allocation* to fluctuate dynamically: When a VC demands less than its capacity, the VC gets what it demands and the idle servers are allocated to other VCs as additional servers. In return, the VC may receive, during overload, additional servers from under loaded VCs. In our model, we use a_i, c_i and d_i to denote the allocation, capacity and demand of VC_i at time t. We focus on the system scheduler that allocates servers to VCs instead of the VC-level scheduler that schedules jobs within a VC. We assume that the jobs are malleable: the number of servers allocated to a job can be adjusted during the job's execution; Cosmos and MapReduce jobs belong to this category.

3 Experimental Methodology

We evaluate performance consistency using trace-driven simulations with workload from a commercial data center. We use the well-known MaxMin scheduler [13] for instantaneous fairness. Notice that the widely adopted Hadoop Fair scheduler [3] is also an example of a MaxMin scheduler.

3.1 Workload

Cosmos [6] is a large production data-intensive computation platform system similar to MapReduce systems. Cosmos clusters contain tens of thousands of servers for hundreds of users (VCs). We collect a one-month trace (Sep 2011) of a commercial cluster containing about 50,000 servers shared by 115 users. We observe that job

distribution does not follow the usual diurnal cycle, as the system serves teams from all over the world. This fact brings more challenges because the workload behavior and size of jobs are more diverse.

To reproduce the diversity of workload behavior, we choose six VCs (two under-utilized, three fully-utilized and one over-utilized) with different load characteristics to assess the performance inconsistency. Figure 2 depicts the daily aggregated load of the six VCs as well as the daily load of each VCs. The figure shows that it is common that one VC is over-demanding while another is under-demanding. Under such circumstances, sharing is a major factor that affects VC performance. Scheduling resources to achieve performance consistency is critical in such a sharing system.

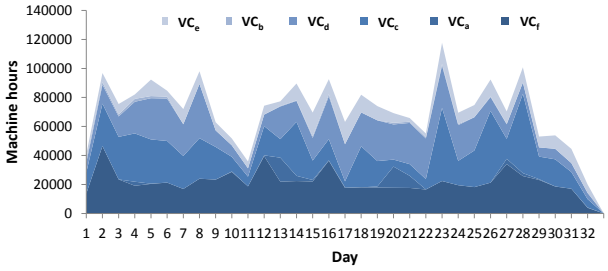


Figure 2: The load fluctuation of six VCs in the one-month trace. The Y-axis is the amount of work of the day. One machine hour denotes the work done by one machine in one hour.

3.2 Simulation design

Trace-driven simulator. We build a trace-driven simulator using desmoj [1], which is a discrete-event library. Our simulator replays a trace from a trace file containing job information, including the submission time, job size, and parallelism. The output of the simulator includes detailed execution information for each job as well as general statistical information such as mean response time.

The total number of machines simulated is 4,000, which is a sum of the capacity of the six VCs plus additional 1,250 machines owned by Cosmos system. Cosmos has additional machines for providing fault tolerance and for running system maintenance jobs; these machines are available to the VCs when they are not running system jobs.

3.3 Performance metric

We measure the performance of each job using the *stretch* metric, which indicates normalized responsiveness of the job. Stretch is defined as the execution time divided by the ideal execution time of the job. To compute the ideal execution time, we divide the job size (in terms of server hours) by its ideal resource allocation — when the job takes the entire VC capacity. We choose ideal resource allocation as the capacity of its VC based on two considerations. On one hand, if idea resource allocation is smaller than the VC capacity, then the VC may require at least two concurrently running jobs to fully use its resources. On the other hand, if it is higher than the VC capacity, the VC will always over-demand its capacity (even with only one running job), which obviates sharing opportunities. Therefore, the definition of job stretch is as follows:

$$\text{job stretch} = \frac{\text{real execution time}}{\text{ideal execution time}} = \frac{\text{real execution time}}{\text{job size}/\text{VC capacity}}.$$

Notice that job stretch is a normalized performance metric and thus overcomes the shortcomings of real-valued metrics such as response time. As jobs have diverse sizes, comparing the response time of jobs from two VCs may not be meaningful. Stretch eliminates this drawback. A larger stretch of a job indicates the job performs worse. In particular, a job with stretch of 1 means that the job is performing the same as the case that the job owns the entire VC by itself. The VC stretch is the mean stretch of all its containing jobs. In later experiments, we distinguish between VC stretch and job stretch.

Stretch can be computed once the job has finished, as job size can be obtained only after job completion.

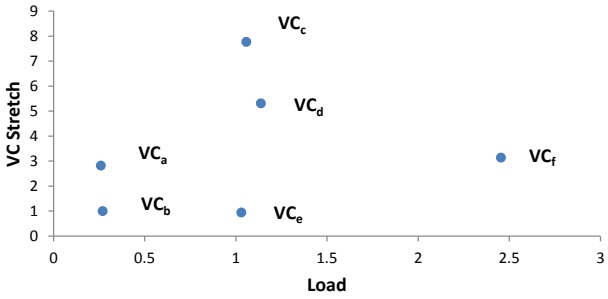


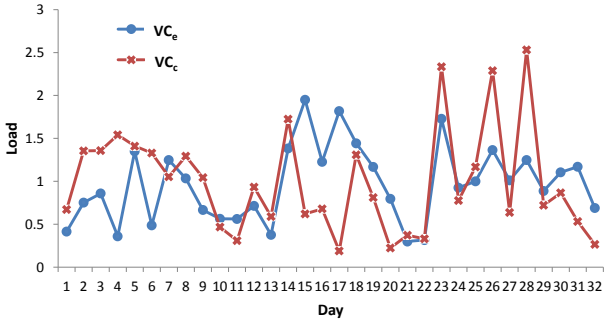
Figure 3: VC stretch with different loads under production workload.

4 Experimental Results

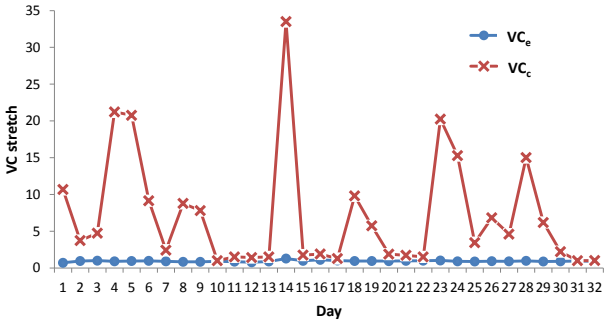
This section illustrates performance inconsistency and its cause based on simulation results driven by Cosmos trace of a commercial data center.

Figure 3 shows the results when simulated with the MaxMin scheduler. We want VCs with similar load to have similar performance, and we call this property performance consistency. However, as shown in the results, three fully-utilized VCs (VC_c, VC_d and VC_e) with different burstiness patterns observe different performance. In the meantime, the over-utilized VC (VC_f) has a better

performance than two fully-utilized VCs. These results show that instantaneous-fairness schedulers do not maintain performance consistency among VCs with similar load; They fail to provide long-term fairness for practical workloads. Furthermore, to reveal the cause of this performance inconsistency, we choose two fully-utilized VCs with different performance and examine their load and performance over time.



(a) The load fluctuation.



(b) The VC stretch. Lower is better.

Figure 4: Daily load and performance of two VCs with similar overall load.

Figure 4 inspects the daily load fluctuation and performance for the two VCs (VC_c and VC_e). The two VCs have similar load but different capacities as well as load characteristics. VC_c is a large VC with a capacity of 900 servers while VC_e is smaller with a capacity of 350. Although both VCs are fully utilized ($load = 1$), their loads fluctuate daily, as shown in Figure 4(a). As for the performance, both VCs perform well ($stretch = 1$) during underload days ($load < 1$). But for overload days, VC_e can still perform well while the performance of VC_c is largely degraded, which demonstrates an unfair situation.

The load-performance behavior seen in Figure 4 is closely related to the scheduling algorithm. A traditional instantaneous scheduler, such as the MaxMin Scheduler [13] and the Hadoop Fair Scheduler [3, 18], typically maximizes the minimum allocation for all VCs at a given time point. More specifically, an instantaneous scheduler has the following four properties:

1) When a VC demands less than its capacity, the instantaneous scheduler always fully satisfies the VC demands. So in Figure 4, as long as the daily load of VC_c or VC_e is smaller than 1, its stretch is equal to 1. The free servers from these underloaded VCs will then be assigned to other over-demanding VCs if there are any.

2) When a VC is over-demanding, it competes with other over-demanding VCs for free servers contributed by underloaded VCs. So these VCs may not have a stretch of one. This explains why VC_c's performance degrades for overload days.

3) When competing for additional free servers, smaller VCs have a higher probability to be fully satisfied than larger ones with similar load. This is because when the load is the same, the exceeding demand is proportional to VC capacity. So to maximize the minimum allocation for all VCs, an instantaneous scheduler has to prioritize satisfying less demanding VCs, which makes it harder for large VCs to obtain extra allocation.

4) The scheduling decision is only made at a given time point. So even if a large VC contributes more resources during underload periods, it has to compete equally with other VCs during overload periods. As a result, a bursty large VC may fail to receive enough resource during busy hours regardless how many resources it has contributed earlier.

This case study demonstrates how an instantaneous scheduler casues long-term unfairness. A large VC that contributes more resources during underload periods cannot be properly rewarded during its overload periods. And this situation is caused by the nature of instantaneous fairness, where the sharing history is not considered for scheduling decision.

5 Related Work

The Hadoop Fair Scheduler is widely adopted in multi-user Hadoop clusters [18, 3]. It divides the Hadoop cluster into pools and assigns a pool to each user. The scheduler computes the fair share of each pool according to instantaneous information such as the weight, minimum share and demands of pools, without considering the resource usage history. Hadoop Fair Scheduler is an example of MaxMin Fair Scheduler used in datacenters; other schedulers in this category, including the Hadoop Capacity Scheduler [2], and Quincy [11], consider fairness at the moment of allocation rather than cluster usage history. The Dominant Resource Fair Scheduler [8, 9] schedules multiple types of resources to improve fairness and utilization. When there is only one type of resources, it performs exactly as a MaxMin fair scheduler. Variations of MaxMin scheduler are also used for scheduling shared-memory multiprocessor systems [14, 5]. All of the above prior work do not consider usage history, and

therefore, they cannot guarantee performance consistency, which is the focus of this work.

6 Discussion

We show in our experiment that not considering usage history serves as the root cause of performance inconsistency. So here we first present several existing history-based schedulers that can be potentially useful for our scenario. Then we explain the particular requirement of such a scheduler in large-scale data processing systems such as Cosmos. The discussion focuses on designing a practical history-based scheduler for similar systems.

Deficit Round-Robin (DRR) scheduler [16] proposes a technique that allows each flow passing through a network device to have a fair share of network resources. As packet size may differ, simple round robin algorithm may not be fair; DRR uses a deficit counter as a representation of usage history to revise the round robin algorithm to achieve long-term fairness. The Xen credit scheduler [12] applies similar mechanisms to allow multiple virtual machines to fairly share CPU resources.

Both schedulers regulate user's future resource allocation using a counting scheme that measures the previous usage. The counting scheme ensures that a user that overuses its fair share in previous time slot will be charged evenly (or even more) in the future. This guarantees long-term fairness, i.e., over-demanding users will not hurt other users. However, from the view of system operators, promoting overall system utilization is as important as maintaining fairness among users. We argue that using existing strict history-based schedulers will harm the overall utilization to a certain extent. For example, in order to promote overall utilization, the system operators should encourage users to use the system when it is under-loaded. However, by using existing schedulers, over-demanding users will always be penalized in the future regardless of how under-loaded the overall system is. As a result, these schedulers fail to provide incentive for users to use the system during idle periods, which in turn reduces the overall utilization. Thus balancing the system utilization and fairness is an important design challenge for long-term fair schedulers in large-scale data processing systems.

7 Conclusion

A large computing cluster is normally shared among users within an organization to have high system utilization and to offer more computing resources. However, sharing comes with an important fairness problem, resulting in performance inconsistency among users. We identify this problem by conducting a simulation study using a production trace from a large cluster. The results show that traditional cluster schedulers that optimize for instantaneous fairness cannot guarantee performance consistency in the long term. The main reason for this is that instantaneous-fairness schedulers do not consider the sharing history of users. As a result, users with large and bursty workloads do not gain credits for contributing to the system during idle periods. In contrast, they may observe bad performance during busy periods. Our study demonstrates that instantaneous-fairness schedulers may incur performance inconsistency in long run so sharing history should be utilized to provide a better scheduling decision.

Acknowledgement

The authors thank Omer Reingold and Moshe Babaioff (from Microsoft Research) for valuable discussions and feedback.

References

[1] A framework for discrete-event modelling and simulation. `http://desmoj.sourceforge.net/home.html`.

[2] Hadoop capacity scheduler. `http://hadoop.apache.org/docs/stable/capacity_scheduler.html`.

[3] Hadoop fair scheduler. `http://hadoop.apache.org/docs/stable/fair_scheduler.html`.

[4] The hadoop project. `http://hadoop.apache.org/`.

[5] AGRAWAL, K., HE, Y., HSU, W. J., AND LEISERSON, C. E. Adaptive scheduling with parallelism feedback. In *IPDPS* (2007).

[6] CHAIKEN, R., JENKINS, B., LARSON, P.-A., RAMSEY, B., SHAKIB, D., WEAVER, S., AND ZHOU, J. Scope: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow. 1*, 2 (2008), 1265–1276.

[7] DEAN, J., AND GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. *Commun. ACM 51*, 1 (jan 2008), 107–113.

[8] GHODSI, A., ZAHARIA, M., HINDMAN, B., KONWINSKI, A., SHENKER, S., AND STOICA, I. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI* (2011).

[9] HINDMAN, B., KONWINSKI, A., ZAHARIA, M., GHODSI, A., JOSEPH, A. D., KATZ, R., SHENKER, S., AND STOICA, I. Mesos: a platform for fine-grained resource sharing in the data center. In *NSDI* (2011).

[10] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: distributed data-parallel programs from sequential building blocks. In *Eurosys* (2007).

[11] ISARD, M., PRABHAKARAN, V., CURREY, J., WIEDER, U., TALWAR, K., AND GOLDBERG, A. Quincy: fair scheduling for distributed computing clusters. In *SOSP 2009* (2009).

[12] KANG, H., CHEN, Y., WONG, J. L., SION, R., AND WU, J. Enhancement of xen s scheduler for mapreduce workloads. In *HPDC* (2011).

[13] KESHAV, S. *An engineering approach to computer networking: ATM networks, the Internet, and the telephone network*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[14] MCCANN, C., VASWANI, R., AND ZAHORJAN, J. A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors. *ACM Trans. Comput. Syst. 11*, 2 (1993), 146–178.

[15] POWER, R., AND LI, J. Piccolo: Building fast, distributed programs with partitioned tables. In *OSDI* (2010).

[16] SHREEDHAR, M., AND VARGHESE, G. Efficient fair queueing using deficit round-robin. *IEEE/ACM Transaction on Networking 4*, 3 (1996), 375–385.

[17] YU, Y., ISARD, M., FETTERLY, D., BUDIU, M., LFAR ERLINGSSON, GUNDA, P. K., AND CURREY, J. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI* (2008).

[18] ZAHARIA, M., BORTHAKUR, D., SEN SARMA, J., ELMELEEGY, K., SHENKER, S., AND STOICA, I. Job scheduling for multi-user mapreduce clusters. Tech. rep., Berkeley, 2009.

[19] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Spark: Cluster computing with working sets. In *HotCloud* (2009).