Graviton: Twisting Space and Time to Speed Up CoFlows

Akshay Jajoo Rohan Gandhi Y. Charlie Hu
Purdue University

1 Introduction

Improving network performance [5, 11] is crucial in improving the performance of applications running in datacenters, especially data-intensive applications such as MapReduce. It is shown that traditional approaches that improve flow completion time (FCT) [10] may not improve application performance [4], because they often fail to capture the application requirements. The CoFlow abstraction [4] was proposed to enable applications to specify their network demands. It is shown that improving the completion time of CoFlows leads to better application performance [4, 6].

Prior online CoFlow schedulers [4, 6, 8, 5, 12] have focused on improving CoFlow Completion Time (CCT). A rule of thumb to improve the overall CCT is to schedule Short CoFlows First (SCF) [6], *i.e.,* shorter CoFlow is scheduled before longer CoFlow. However, SCF requires apriori knowledge about the CoFlow sizes. To make the scheduler practical, Aalo [5] approximates SCF without apriori knowledge of CoFlow sizes, using priority queues and weighted fair sharing across the queues. When a CoFlow first arrives, it starts in the highest priority queue, and moves to the lower priority queue as it sends more data yet does not finish (hinting that it is likely a long CoFlow). Effectively, short CoFlows get prioritized over longer CoFlows, improving the overall CCT. Additionally, Aalo uses FIFO to schedule the CoFlows from the same queue to avoid starvation.

In essence, in scheduling CoFlows in datacenter, Aalo did a straightforward parody of the classic Shortest-Job-First (SJF) policy [13], by simply representing the progress of the CoFlow as the total bytes sent at all ports, *i.e.,* a normalized notion of progress in the *time dimension*. In doing so, it throws away a potentially very useful, *spatial dimension* of the problem domain, *i.e.,* different CoFlows can have flows running on many ports (wide CoFlows) or only a few ports (thin CoFlows).

Aalo's scheduler has two implications to its behavior by being oblivious to CoFlow width. (1) Wide CoFlows move up the queues fast, but always wait for equal-sized thin CoFlows to catch up. This is because the queue thresholds for determining when to move CoFlows to lower priority queues are based on the *total bytes sent*. (2) At each queue, using FIFO misses the opportunity that scheduling CoFlows likely to finish in this queue first reduces CCT for this CoFlow and the overall CCT.

In this paper, we make a key observation that using multiple priority queues and weighted fair sharing at each port, Aalo does a good job in approximating SJF, but it does so only at *the queue-granularity*, as using FIFO to schedule CoFlows in each queue is rather simplistic, and has no reminiscence of SJF.

Instead, we discuss three insights into Aalo's scheduler where exploiting the spatial dimension of the problem domain, *i.e.,* the width (number of ports) of the CoFlows, can lead to better scheduling policies *within each priority queue*, improving the overall CCT.

In particular, we show different queues should use different scheduling policies based on the CoFlow width. First, we use Thin-CoFlow-First in high priority queues to help small CoFlows complete quickly, reducing their CCT without affecting the CCT of larger CoFlows. Second, we use Wide-CoFlow-First in the medium-sized priority queues to avoid the long (and thin) CoFlows blocking wide (but short) CoFlows. Third, we again use FIFO in the lowest priority queues to lessen the degree of starvation of early arrival CoFlows.

Secondly, we expose a unique problem in Aalo that flows in individual CoFlows finish out-of-sync at different ports, again due to lack of coordination among the ports, and Aalo's choice of *total bytes* to move the CoFlows to lower queues. We observe that CoFlows can get demoted to lower priority queues even if some of its ports have sent almost no data (observation-4 in §3)! This translates into worsened overall CCT. A thorough treatment to this important problem is quite involved, which we leave for future work.

We have designed and prototyped GRAVITON that in-

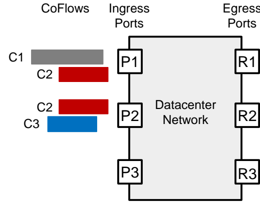


Figure 1: CoFlow example. The arrival time for CoFlows C2 < C1 < C3.

herits the basic architecture of Aalo for approximating SJF at the queue-granularity, but replaces the per-queue FIFO scheduling policy with the above list of refined policies for different priority queues. Our design further handles practical issues including CoFlow starvation.

Our preliminary evaluation using real traces from Facebook MapReduce jobs [1] shows GRAVITON achieves 1.25x (P50) and 8x (P90) speed-up over Aalo.

2 Background

In this section, we briefly detail on the CoFlow abstraction and summarize prior art in CoFlow scheduling.

2.1 CoFlow

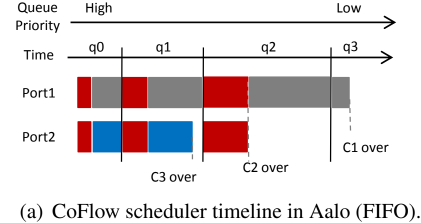
The CoFlow abstraction [4] enables datacenter applications (*e.g.,* MapReduce, web-servers) to easily specify their network demands such as faster CoFlow completion time, or meeting latency deadlines [4].

In datacenters, many applications and many instances of the same application (*e.g.,* jobs in MapReduce) run concurrently. As a result, many CoFlows exist simultaneously that compete for the network resource (depicted in Figure 1). The goal of CoFlow scheduling algorithms is to schedule the CoFlows so as to improve the application performance. Particularly in data-intensive applications (*e.g.,* MapReduce) [7, 14], completing a CoFlow consisting of many parallel flows between distributed endpoints is more important than finishing individual flows to improve job completion time [8, 6, 5].

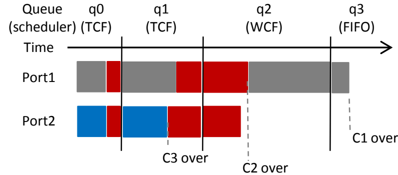
2.2 Online CoFlow Scheduling

The basic idea behind prior CoFlow scheduling algorithms which all aim to minimize the average completion time of the CoFlows, is a variation of the classic Shortest Job First algorithm, called Shortest CoFlow First (SCF). They differ in how they implement the online approximation of the SCF, without prior knowledge about CoFlows.

De-centralized scheduler. Baraat [8] uses a pure de-centralized scheduler, where individual end-points (called *ports*) schedule CoFlows independently. Individual ports schedule the shorter CoFlows first, based on the CoFlow size observed *locally*. Such de-centralized scheduling policy is sub-optimal as different flows of a CoFlow may compete with different other CoFlows on different ports, easily causing them to end up with different local priorities and progress out of sync and finish at different times.



(a) CoFlow scheduler timeline in Aalo (FIFO).



(b) CoFlow scheduler timeline in GRAVITON.

Figure 2: CoFlow scheduling for CoFlows in Figure 1. Solid vertical lines denote when both CoFlows cross queue thresholds.

Hybrid scheduler. Aalo [5] addresses the limitation of Baraat using a hybrid of centralized and decentralized scheduler. To improve the odds different flows of the same CoFlow at different ports make similar progress, it uses a centralized controller to summarize the *total progress*, *i.e.,* the total number of bytes sent by the CoFlow on all its ports. Locally at each port, a local scheduler approximates SCF by using multiple priority queues, moving flows across different queues based on the total progress (bytes) of its CoFlow, and using weighted fair queuing across queues. Within each priority queue, the local scheduler uses FIFO when deciding the order of the flows to send to avoid starvation.

2.3 The Essence

In essence, in scheduling CoFlows in datacenter networks, Aalo did a straightforward parody of the classic SJF [13, 9], by simply representing the progress of the CoFlow as the total bytes sent at all ports by the CoFlow.

In doing so, it throws away a potentially very useful, spatial dimension of the problem, *i.e.,* different CoFlows have flows running on different numbers of ports.

Figure 2 illustrates the outcome and the essence of Aalo's scheduling algorithm. Effectively the algorithm only joggles the time dimension, just like a generic online SJF algorithm in a single machine. Upon arrival, flows of a CoFlow start from the first queue, which has the highest priority, and gradually move to lower priority queues as its CoFlow's total bytes crosses queue thresholds. This design choice has two implications.

Implication 1: Wide CoFlows move up the queue stack fast, but always wait for equal-sized thin flows to catch up. This is because the queue thresholds are based on the *total bytes sent* and individual queue thresholds are same for all the CoFlows. When a wide CoFlow

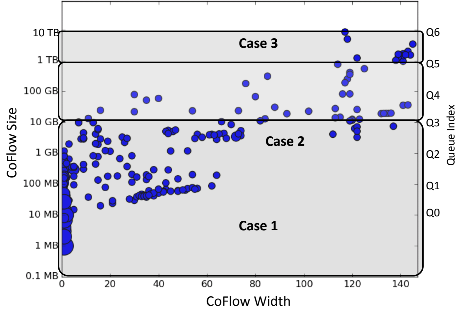


Figure 3: Width (#sender ports) and total size of all the CoFlows from Facebook trace [1]. The CoFlows follow heavy tailed distribution where 66% of the CoFlows have width <10 ports, whereas 8.3% CoFlows are >100 ports in width. Y2-axis shows the queue thresholds.

is scheduled it quickly reaches the threshold as it can send data on multiple ports than a thin CoFlow. However, once it reaches the threshold, it moves to the lower priority queue, where it needs to *wait* till all other CoFlows from the prior high priority queues are scheduled.

Implication 2: At a given queue, scheduling CoFlows that are likely to finish while in this queue first reduces CCT. This follows from implication-1. Scheduling CoFlow that does not finish in that queue first does not improve CCT of that CoFlow, as it anyway needs to wait in the next queue for all other CoFlows from the same queue to be scheduled (implication-1). Instead it delays and worsens CCT of a CoFlow that would finish in that queue. This is illustrated in Figure 2(a). In queue q1, Port2 schedules CoFlow C2 first as per FIFO. As C2 sends more data it reaches queue threshold and moves to queue q2[1]. However, C2 needs to wait for C3 to be scheduled as C2 now has lesser priority than C3. Instead, if C3 is scheduled prior to C2, the waiting time for C3 is reduced without any change to the waiting time of C2, thus improving overall CCT.

Naturally, no online scheduler can know which CoFlows finish in individual queues as CoFlow sizes are not known apriori. However, *spatial dimension (number of ports) provides useful hints about the CoFlow sizes.*

3 Key Insights

In this section, we discuss three aspects of Aalo's scheduling algorithm where exploiting the spatial dimension of CoFlows leads to opportunities for significant improvement of the overall CoFlow completion time.

Figure 3 plots the width versus size of all the CoFlows in a MapReduce trace from Facebook datacenter [1]. The trace contains O(1000) CoFlows across O(100) ports.

We divide the CoFlows into three regions: small-sized, medium-sized, and large-sized, as shown in figure.

Observation 1 (Case 1). Our first observation is that most small CoFlows are also narrow, *i.e.,* they have only a few flows each. Of the CoFlows with size <1GB, 79% CoFlows have width <10 ports. This suggests for using a *Thin-CoFlow-First* (TCF) policy to schedule CoFlows in each queue, as many thin CoFlows are likely to finish without going to higher-up queues (Implication 2).

Figure 7 which plots the speed-up in color for individual CoFlows based on their width and size, shows that switching from FIFO to TCF for queues in this region achieves median speed-up of 25% over Aalo, when evaluated using the Facebook trace (more details in § 5).

Observation 2 (Case 2). However, scheduling using TCF in all queues has its limitations. In medium-sized queues, the queue thresholds are fairly large. As the queue thresholds are constant for wide and thin CoFlows, wide CoFlows quickly reach the queue threshold than thin CoFlows (Implication 1). But as the thin flows that have reached these medium-sized queue have similar total bytes (on smaller ports), they must have *longer individual flows compared to those of wide CoFlows.*

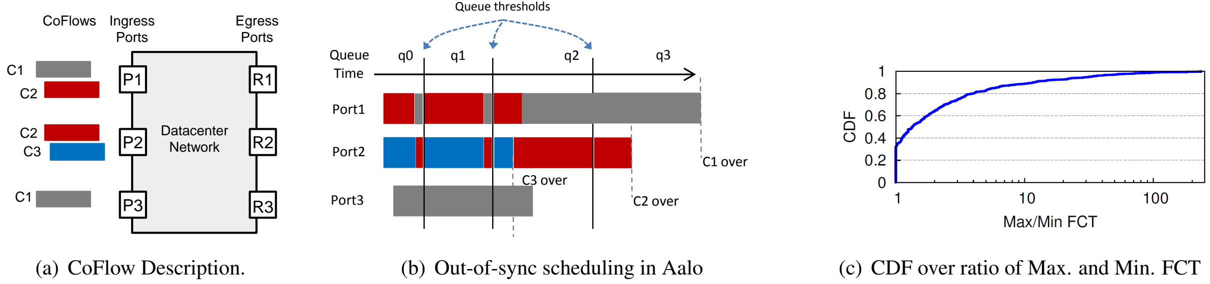
Assume both wide and thin CoFlows have equal probability of completion while in any of the medium-sized queues. In those queues, scheduling long flows of thin CoFlows contradicts the SJF spirit and worsens the CCT of short (and wide) CoFlows. However, if the wide CoFlows are scheduled first, the overall CCT can be improved. We call this policy *Wide-CoFlow-First (WCF).*

Observation 3 (Case 3). In large-sized (lowest priority) queues, most CoFlows are wide and have no significant distinction in terms of width, using FIFO respects their arrival time, and lessens the degree of starvation.

In summary, the above observations suggest that different queues should use different scheduling policies to improve the overall CCT. First, we use TCF in high priority queues to help small CoFlows complete quickly, reducing their CCT without affecting the CCT of larger flows (Case 1). Second, we use WCF in the medium-sized priority queues to avoid the long (and thin) CoFlows blocking wide (but short) CoFlows (Case 2). Third, we again use FIFO in the lowest priority queues to lessen the degree of starvation of early arrival CoFlows (Case 3). This is illustrated in Figure 2(b).

Observation 4 (out-of-sync CoFlow completion). As ports schedule the CoFlows independently, individual ports are not aware about how the same CoFlow is scheduled on other port, which could result in CoFlows being scheduled *out-of-sync*. The individual ports schedule CoFlows in the same priority queue until the *total number of bytes calculated* across all ports crosses the queue threshold. Without coordination, one port can easily send enough data to cross the threshold, and demote the CoFlow to the next queue while other ports have sent little to no data! (illustrated in Figure 4). This clearly

[1] C2, C3 have similar behavior in q0.



(a) CoFlow Description. (b) Out-of-sync scheduling in Aalo (c) CDF over ratio of Max. and Min. FCT

Figure 4: Out of sync scheduling in Aalo. CoFlow Arrival time is C3 $<$ C2 $<$ C1. (b) shows CoFlow C1 crosses queue threshold for q0 even if Port1 has sent very less data, as Port3 has sent enough data to cross threshold. Similar for C2. (c) shows CDF of ratio of Max. and Min. Flow Completion Time (FCT) for individual CoFlows.

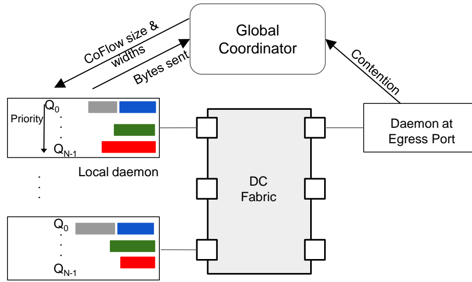


Figure 5: GRAVITON scheduler architecture.

inflates the max. flow completion time (FCT), which governs the CCT. Figure 4(c) shows that the problem is severe in practice – the max/min FCT of individual CoFlows varies between 1-233x (median 1.33x). The root cause of the problem is again lack of coordination in the spatial dimension, *i.e.,* across ports. A thorough treatment to this important problem is quite involved, which we leave for future work.

4 Design

In this section, we discuss details of GRAVITON architecture and scheduler as well as key policies to handle challenges including CoFlow starvation.

GRAVITON **Architecture:** Figure 5 shows the GRAVITON architecture. GRAVITON consists of a global coordinator and local daemons running on all individual ports. Similar to Aalo, individual ports schedule the CoFlows using multiple priority queues and weighted sharing across priority queues.

The global coordinator periodically collects the number of bytes sent by individual CoFlows from all the ports, and computes and pushes the total bytes sent per CoFlow to all the sender ports. The global coordinator also collects and pushes the width of the CoFlows.

Scheduling policy: Local daemons on individual ports schedule the CoFlows based on the total bytes and width sent by the coordinator. When the CoFlow arrives it is assigned to the highest priority queue, and as it sends more data it is progressively assigned to the lower priority queues. Note that all the ports have the same queue threshold, and assign the same CoFlow to the same queue. Similar to Aalo, we use priority queues with exponentially increasing thresholds in the power of 10, *i.e.,* $Q_{i+1}^{threshold} = 10 \cdot Q_i^{threshold}$ to balance between the number of queues and overall CCT [5]. Currently, the highest (lowest) priority queue has a threshold of 10MB (10TB). We also keep same weights to the priority queues as Aalo.

Table 1: Mapping between queue and scheduling policy. Q_0 denotes the highest priority queue.

Queue	Policy
Q_0 to Q_{n-4}	TCF
Q_{n-3} to Q_{n-2}	WCF
Q_{n-1}	FIFO

As mentioned in previous section, we exploit the opportunities to improve the CCT by using different scheduling policies (TCF, WCF and FIFO) in different queues (§3). Table 1 summarize the mapping between the queues and scheduling policy. In current design, the mapping between the queue and scheduling policy is fixed and derived using the Facebook trace. We leave generating this mapping at run-time as future work.

Computing width: Recall that TCF and WCF schedule the CoFlows in the same queue using their width information. We tried different ways to compute the width. For a CoFlow with m sender ports and r receiver ports, the width can be computed in many ways including (m), (r), $(m+r)$, $(m \cdot r)$, $\max(m,r)$. In our evaluation, we did not find substantial difference in overall CCT using these combinations. We used r and update it as flows complete.

Avoiding starvation: TCF and WCF potentially starve certain CoFlows. TCF starves wider CoFlows if thin CoFlows constantly arrive. Similarly WCF can starve thin CoFlows. We avoid starvation by limiting the waiting time. The individual ports keep track of the waiting time for individual CoFlows based on their arrival time, and immediately schedules the CoFlows whose waiting time exceeds the threshold. We call this design as GRAVITON-SF. As the waiting time for individual flows

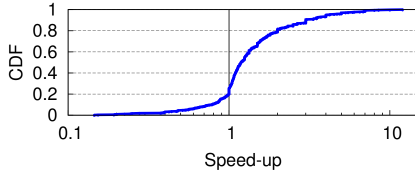


Figure 6: Speed-up distribution for all CoFlows.

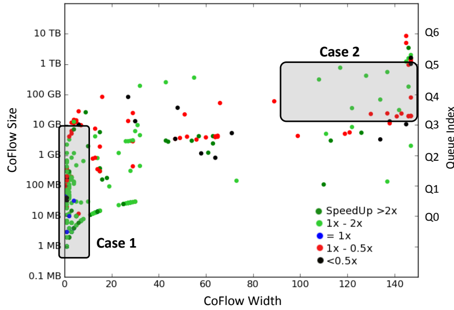


Figure 7: Speed-up in GRAVITON over Aalo for CoFlows of different width and total size.

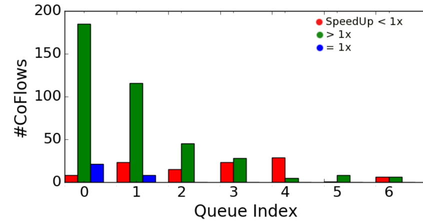


Figure 8: Queue-wise distribution of speed-ups.

of a CoFlows are same on all the ports, the flows are scheduled in sync on all their ports even when all the ports are heavily loaded.

Work conservation: Work conservation helps assign unused bandwidth especially when the receivers experience contention [5]. In GRAVITON, the receiver notifies the global coordinator of the contention, and the coordinator explicitly shuts of *entire* CoFlow instead of shutting off some of the flows as in Aalo. This is because of the *all-or-none* property of CoFlows [3, 16]. GRAVITON notifies the sender ports and delays scheduling a CoFlow if the receivers are not ready. This design choice is synergistic to the delay-scheduling principle [15]. However, this design choice could potentially lead to starvation if all its receiver ports are not ready at any point. We address the starvation problem by using the GRAVITON-SF, which we believe can be improved in future.

5 Evaluation

We implemented GRAVITON scheduler, and in this section we evaluate the speed-up using GRAVITON over Aalo when we simulate GRAVITON and open-sourced implementation of Aalo [2] using MapReduce trace from Facebook. The trace contains O(1000) CoFlows details for 1 hour across across O(100) ports. The queue thresholds and weights are same for Aalo and GRAVITON.

Figure 6 shows the CDF of speed-up obtained using GRAVITON over Aalo for all the CoFlows. The speed-up was measured as $\frac{CCT \ in \ Aalo}{CCT \ in \ Graviton}$, and values >1 mean GRAVITON is faster. Figure 6 shows that GRAVITON improves the CCT between 0.16x (equivalent to -6x) to 12x (median=1.25x, average=1.65x). Furthermore, GRAVITON improves the CCT of over 80.2% CoFlows at an expense of worsened CCT of 19.8% CoFlows.

To further understand the benefits of GRAVITON, we zoom in the improvement in CCT for CoFlows of different widths and sizes (shown in Figure 7). Unsurprisingly, GRAVITON is highly effective in improving CCT of the thin (and small) CoFlows. It can be seen that the significant number of thin- CoFlows under 10GB size observed substantial improvements due to TCF. In fact 81% of the CoFlows with width <10 and size <10GB observed speed-up >1 (Case 1)! However, surprisingly, GRAVITON did not improve CCT for a small number of the thin-CoFlows (red- points for CoFlows with width <10). This is because FIFO favored these CoFlows based on their arrival time, whereas TCF pushed them back for other thin CoFlows. Similar to TCF, as expected WCF improved the CCT for the wide CoFlows in the lower priority queues (Case 2). FIFO (in the last queue) had a small improvement. This shows the effectiveness of different scheduling policies in different queues.

Figure 8 shows the number of CoFlows affected in individual queues. It shows that CCT was improved for 86.4% of the CoFlows in the first queue, and GRAVITON improves CoFlows across queues, except queue-4. We found this problem linked to the *out-out-sync* problem (§3) as the max/min FCT ratio for CoFlows in this queue was very high – 1.44x (P50), 94x (P90).

6 Conclusion

In this paper, we presented GRAVITON, a CoFlow scheduler to improve CoFlow Completion Time (CCT). We make two contributions: (1) we show that the prior art scheduler Aalo does not consider the *spatial* dimension, *i.e.,* the width of CoFlows when scheduling CoFlows, and misses out on opportunities to schedule the shortest CoFlows from same priority queues. In GRAVITON, we show how to use width of CoFlows differently in different queues as hints to the CoFlow sizes to facilitate scheduling of short CoFlows. Using a trace from Facebook datacenter, we show that GRAVITON speed-up CoFlows by 1.25x (P50) and 8x (P90) over Aalo. (2) We expose a unique problem in Aalo that flows in individual CoFlows finish out-of-sync because of the rootcauses embedded in Aalo's design of using FIFO and ignoring *spatial* dimension, which we leave for future work.

References

[1] Coflow trace from facebook datacenter. https://github.com/coflow/coflow-benchmark.

[2] Open-sourced aalo simulator. https://github.com/coflow/coflowsim.

[3] G. Ananthanarayanan et al. Reining in the outliers in map-reduce clusters using mantri. In *OSDI 2010*.

[4] M. Chowdhury and I. Stoica. Coflow: A networking abstraction for cluster applications. In *HotNets-XI*, 2012.

[5] M. Chowdhury and I. Stoica. Efficient coflow scheduling without prior knowledge. In *SIGCOMM*, 2015.

[6] M. Chowdhury, Y. Zhong, and I. Stoica. Efficient coflow scheduling with varys. In *SIGCOMM*, 2014.

[7] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI 2004*.

[8] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron. Decentralized task-aware scheduling for data center networks. In *SIGCOMM*, 2014.

[9] S. Eilon and I. Chowdhury. Minimising waiting time variance in the single machine problem. *Management Science*, 1977.

[10] C.-Y. Hong, M. Caesar, and P. Godfrey. Finishing flows quickly with preemptive scheduling. In *SIGCOMM 2012*.

[11] R. Mittal, R. Agarwal, S. Ratnasamy, and S. Shenker. Universal packet scheduling. In *HotNets 2015*.

[12] Z. Qiu, C. Stein, and Y. Zhong. Minimizing the total weighted completion time of coflows in datacenter networks. In *SPAA 2015*.

[13] A. Silberschatz, P. B. Galvin, G. Gagne, and A. Silberschatz. *Operating system concepts*, volume 4. Addison-Wesley Reading, 1998.

[14] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: a system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI 2008*.

[15] M. Zaharia et al. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys 2010*.

[16] M. Zaharia et al. Improving mapreduce performance in heterogeneous environments. In *OSDI 2008*.