# Coflow: An Application Layer Abstraction for Cluster Networking

*Mosharaf Chowdhury*
*Ion Stoica*

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2012-184
http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-184.html

August 7, 2012

# Coflow

## An Application Layer Abstraction for Cluster Networking

Mosharaf Chowdhury, Ion Stoica
EECS, UC Berkeley
{mosharaf, istoica}@cs.berkeley.edu

*Two elements are needed to form a truth –*
*a fact and an abstraction.*

–Remy de Gourmont

## ABSTRACT

Cluster computing applications, whether frameworks like MapReduce and Dryad, or customized applications like search platforms and social networks, have application-level requirements and higher-level abstractions to express them. Networking, however, still remains at the level of forwarding packets and balancing flows, and there exists no networking abstraction that can take advantage of the rich semantics readily available from these data parallel applications. The result is a plethora of seemingly disjoint, yet somehow connected, pieces of work to address networking challenges in these applications.

We propose an application layer, data plane abstraction, *coflow*, that can express the requirements of (data) parallel programming models used in clusters today and makes it easier to express, reason about, and act upon these requirements.

## 1 Introduction

With the proliferation of public and private clouds, cluster computing is becoming the norm. An increasing number of organizations are developing a mixed variety of cluster computing applications to run user-facing online services, long-running data analytics, and to support interactive short queries for exploratory purposes.

Cluster computing applications serve diverse computing requirements, and to do that, they expect a broad spectrum of services from the network. On the one hand, some applications are throughput-sensitive; they must finish as fast as possible and must process every piece of input (e.g., MapReduce [14], Dryad [19]). On the other hand, some are latency-sensitive with strict deadlines, but they do not require the exact answer (e.g., search results from Google or Bing, home feed in Facebook). A large body of point solutions has emerged to address the communication requirements of cluster computing applications [6, 7, 12, 18, 27, 32, 33].

Unfortunately, the networking literature does not provide any construct to encapsulate the communication requirements of datacenter-scale applications. For example, the abstraction of flows cannot capture the semantics of communication between two groups of machines in a cluster application, where multiple flows are created between the machines in different groups. Since developers can neither precisely express their requirements nor can they understand the similarities or differences between their problems and the problems already solved in some existing application, purpose-built solutions thrive.

Lack of an abstraction that can capture the communication semantics of cluster applications has several consequences. First, it promotes non-modular solutions, making code reuse harder and promoting buggy, unoptimized code. Second, it limits the flexibility and usability of a solution, because it is harder to understand what other problems can be solved with that solution, and with how much modifications. Finally, without an abstraction it is hard to reason about the underlying principles and to anticipate problems that might arise in the future.

Abstractions generalize by extracting the essential parts of a problem through distillation of many of its instantiations; hence, finding an abstraction is often a bottom-up process. In this paper, we study multiples classes of parallelization models (e.g., Dataflows with and without barriers, bulk synchronous parallel, partition-aggregate etc.) used in cluster computing and their communication requirements, along with the solutions that have been developed to address them. We observe that most of these applications are organized into multiple stages or have machines grouped by functionalities, and communication happens at the level of machine collections, often dictated by some application-specific semantics.

Based on our observations, in this paper, we propose *coflow*, an application layer networking abstraction that captures diverse communication patterns observed in cluster computing applications. Each coflow is a collection of concurrent flows between two groups of machines with associated semantics and an aggregate objective. The semantics allow one to take different actions on the collec-

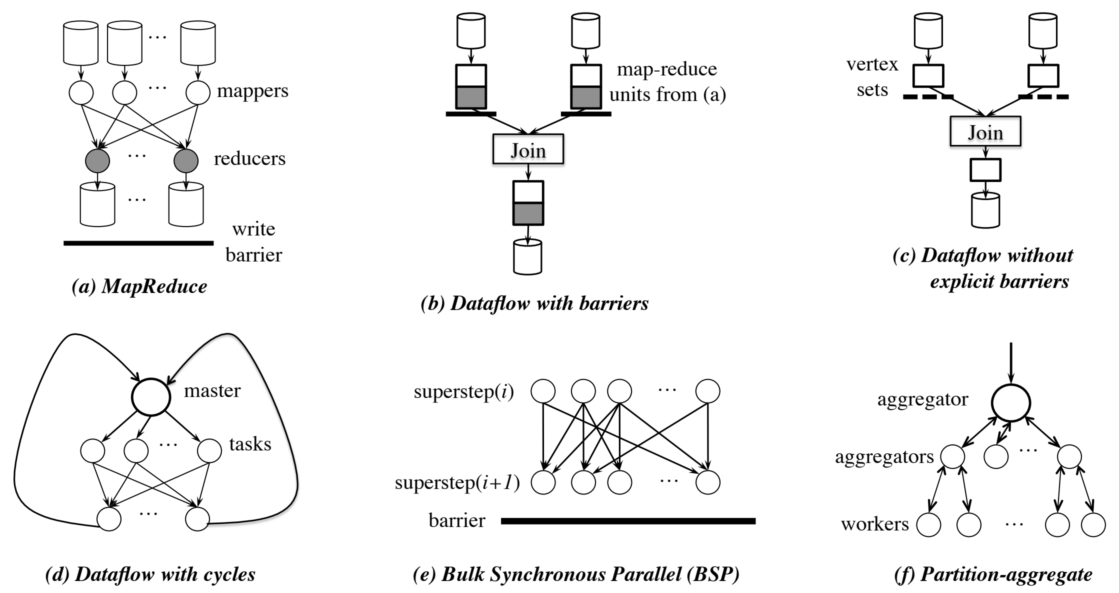


Figure 1: Communication patterns in cluster computing: (a) Shuffle and DFS replication in the perennial MapReduce [14] model; (b) Shuffle across multiple MapReduce jobs in dataflow pipelines that use MapReduce as the building block (e.g., Hive [4]); (c) Dataflow pipelines with streaming (e.g., Dryad [19]); (d) Dataflow with cycles and broadcast support (e.g., Spark [31]); (e) Bulk Synchronous Parallel or BSP model (e.g., Pregel [21]); (f) Partition-Aggregate communication in online services (e.g., user-facing portion of search engines, social networks etc.).

tion to achieve the collective end goal. We must note that our proposal is a culmination of existing work, where researchers have used the notion of such collections or hinted at it, albeit in limited scopes [12, 18, 27]. We generalize the abstraction and take the next step by using it to explain a large body of existing research, use it to express complex cluster applications, and identify new problems to set a possible agenda of pragmatic research in cluster networking.

## 2 Communication in Cluster Applications

Most cluster computing applications are actually frameworks (e.g., MapReduce [14]), where analysts submit jobs that follow particular workflows enabled by corresponding programming models. Some others are user-facing pipelines, where user requests go through a multistage architecture to eventually send back the requested outcome (e.g., search results from Google or Bing, home feed in Facebook). In this section, we summarize and compare the communication requirements of popular cluster computing applications along with the existing work that try to address these requirements.

### 2.1 MapReduce

MapReduce [14], specially its open-source implementation Hadoop [2], is the most well-known and widely used cluster computing framework. In this model, mappers read input from disks, perform computations, shuffle them to reducers based on some partitioning function, and reducers receive multiple partitions, merges them, and writes the output to the distributed file system (DFS) [5, 15], which then replicates it to at least two racks.

Given $m$ mappers and $r$ reducers, a MapReduce job will create $m \times r$ flows for the shuffle and at least $r$ flows for output replication. The primary characteristic of communication in the MapReduce model is that the job will not finish until the last reducer has finished [12]. Consequently, there is an explicit barrier at the end of the job, which researchers have exploited for optimizing communication in this model [12]. Similar optimization can also be performed for DFS replication.

### 2.2 Dataflow Pipelines

Once the usefulness of MapReduce was clear, researchers set out to extend this model to a more general dataflow programming model, which resulted in a collection of dataflow pipelines with diverse characteristics.

**Dataflow with Barriers** The most straightforward extension was to create dataflow pipelines with multiple stages by using MapReduce as the building block (e.g., Sawzall [24], Pig [23], Hive [4]). This introduced barriers at the end of each these building blocks; communication optimizations for the MapReduce model are still valid in this model.

**Streaming Dataflow** To avoid explicit barriers and to enable higher-level optimizations of the operators, streaming dataflow pipelines were introduced (e.g., Dryad [19], DryadLINQ [28], SCOPE [10], FlumeJava [11], MapReduce Online [13]). In this model, the next stage can start as soon as some input is available. Because there is no

Table 1: Summary of Communication Requirements in Cluster Computing Applications

| Model | Examples | Synchronous | Barrier | Loss Tolerance | Comm. Objective |
|---|---|---|---|---|---|
| **MapReduce** | [2,14] | Yes | Write to DFS | None | Minimize completion time |
| **Dataflow with Barriers** | [4,23,24] | Yes | Write to DFS | None | Minimize completion time |
| **Streaming Dataflow** | [10,11,13,19,28] | Yes | Input not ready | None | Minimize completion time |
| **Dataflow with Cycles** | [30,31] | Yes | End of iteration | None | Minimize completion time |
| **Bulk Synchronous Parallel** | [1,3,21] | Yes | End of iteration | None | Minimize completion time |
| **Asynchronous Models** | [20] | No | None | Partial | Either |
| **Partition-Aggregate** | Search/Social | No | End of deadline | Partial | Meet deadline |

explicit barrier, optimization techniques that depend on it are not useful. Instead, networking researchers focused on understanding the internals of the communication and optimizing them for specific scenarios [6,33].

**Dataflow with Cycles** Since traditional dataflow pipelines do not support cycles, they depend on unrolling loops to support iterative computation requirements. Spark and its variants [30,31] introduced cycles without unrolling by keeping states around in memory. In the process, they introduced communication primitives like broadcast and many-to-one aggregation. Implicit barriers at the end of each iteration allowed communication optimizations similar to that in MapReduce [12].

## 2.3 Bulk Synchronous Parallel (BSP)

Bulk Synchronous Parallel or BSP is another popular model in cluster computing with specific focus in graph processing, matrix computation, and network algorithms (e.g., Pregel [21], Giraph [1], Hama [3]). A BSP computation proceeds in a series of global supersteps, each containing three ordered stages: concurrent computation, communication between processes, and barrier synchronization. With explicit barriers at the end of each superstep, the communication stage can be globally optimized for the superstep.

## 2.4 Asynchronous Models

There are asynchronous cluster computing frameworks as well. Sometimes complete information is not needed for reasonably good results and iterations can proceed with partial results. GraphLab [20] is such a framework for machine learning and data mining on graphs. Unlike BSP supersteps, iterations can proceed with whatever information is available as long as it converging; missing information can asynchronously arrive later.

## 2.5 Partition-Aggregate

User-facing online services (e.g., search results in Google or Bing, home feed in Facebook) receive requests from users and send it downward to the workers using an aggregation tree. At each level of the tree, each request generates activities in different partitions. Ultimately, components generated by the workers are aggregated and sent back to the user within strict deadlines. Components that cannot make it within the deadline are either left behind [27] or sent later asynchronously (e.g., Facebook home feed). Research in this direction looked at dropping flows [27], preemption [18], and cutting the tails [32]; however, they do not exploit any application-level information.

## 2.6 Summary of Communication Requirements

Table 1 summarizes the key characteristics of the cluster computing applications discussed earlier with focus on the presence of synchronization barriers after communication stages, characteristics of such barriers, the ability of the application to withstand loss or delay, and the primary objective of communication. We could not possibly cover all the cluster computing frameworks or applications in this section, nor did we try to. However, we do believe that we have covered a large enough spectrum to raise the question: "what's common among them?"

## 3 An Application Layer Abstraction

*It's one thing to have the tools, but you also need to have the methodology. Inevitably, the need to move up to a higher level of abstraction is going to be there.*

–Michael Sanie

With all the differences in programming models, levels of parallelism, communication patterns, and ad-hoc solutions for each of these patterns, one thing that all cluster computing applications have in common is their high-level architectural organization: they run on collections of machines (from tens to thousands) that are organized into multiple stages or grouped by functionalities [9], and each of these collections of machines communicate between themselves with diverse requirements. In this section, we leverage this similarity to abstract away various communication patterns in cluster computing applications using a single data plane construct.

### Coflow

The communication between two collections of machines actually consists of multiple flows between individual pairs of machines. Although, individual flows look the same at the transport layer, all the flows between two collections of machines often have application level semantics. For example, it does not matter what happens

Table 2: Coflow Representations of the Communication Requirements of Cluster Computing Applications

| Communication Pattern / Coflow | Mechanism to Achieve Coflow Objective |
|---|---|
| **Shuffle** | Set rates of individual shuffle flows such that the slowest one finishes as fast as possible [12]. |
| **DFS Replication** | Set rates of individual replication flows such that the slowest one finishes as fast as possible. |
| **Broadcast** | Allocate rates at broadcast participants so that the slowest receiver finishes as fast as possible [12]. |
| **Streaming Dataflow Comm.** | Set rates/prioritize so that later stages do not block on this coflow. |
| **BSP Communication** | Set rates at computation nodes so that the slowest sender finishes as fast as possible |
| **Asynchronous Communication** | Prioritize if the application requires more information from this coflow; else lower priority. |
| **Partition-Aggregate** | - Drop flows that are likely to miss the deadline [27].<br>- Set higher priorities for all the flows in a coflow that has the earliest deadline [18]. |

to the $(m \times r) - 1$ flows in an $m \times r$ shuffle as long as the very last flow has not finished [12]. Similarly, if one decides to delay flows that will miss their deadlines [27], it is better to restrict the delays/drops within as few requests as possible.

Any collection of flows will not do, however. Three conditions must be satisfied: first, there must be higher-level semantics – made available by the application layer – that apply to all the flows in a collection and allow a combined decision; second, flows have to be concurrent within a reasonable time window for them to be considered together; finally, all the flows should have a shared objective that allows joint optimizations on the aggregate. We refer to such collections of flows as *coflows* and define them as the following:

*A coflow is a semantically-bound collection of concurrent flows with a shared objective.*

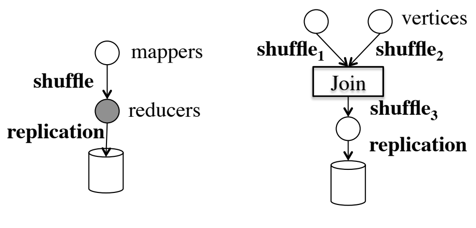
By appropriately defining the objective function and considering the semantics of corresponding applications, we can now concisely describe the communication requirements of different cluster applications using coflows. Table 2 summarizes the communication requirements discussed in Section 2.

## 4 Applications of the Abstraction

*Abstraction is a mental process we use when trying to discern what is essential or relevant to a problem;*

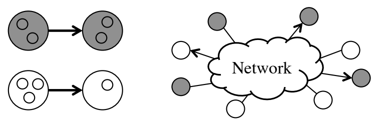
–Tom G. Palmer

As cluster computing is maturing, its users are starting to run diverse applications on shared infrastructures – each purpose-built for specific activity. Researchers have responded by developing platforms to enable such coexistence [17]. However, they do not manage the network in such multi-resource environments [16, 17]. Part of it is because the network is a distributed resource unlike any other. We believe an even bigger reason is that it is harder to discern the communication requirements of diverse cluster computing applications without an abstraction to reason about the sharing, prioritization, ordering, or preemption of coexisting network activities from one or more applications. In this section, we discuss how



***(a) MapReduce*** ***(b) Dryad***

Figure 2: Graphical representations of cluster computing applications using coflows.



***(a) Two coflows*** ***(b) Coflows in the network***

Figure 3: Network modeled as a black box with two ongoing coflows.

coflows make it simpler to reason about cluster applications and sharing the network between them, as well as the minimal mechanisms that must be there to take advantage of the abstraction.

### 4.1 Preliminaries

We start by defining the application and network models we will be using throughout the section.

**Application Model** If we represent a distributed cluster computing application using a graph $G^C = (V^C, E^C)$ and $V^C$ is the set of its stages or functional groups, then $E^C$ is the set of coflows that connect elements of $V^C$. Each coflow $c(u, v) \in E^C$ is a collection of flows between machines in $u$ and $v$, with an associated objective function, $O(c)$. We denote the cardinality or the number of elements in a coflow $c$ by $|c|$. Figure 2 represents MapReduce and dataflow applications in Figure 1(a) and Figure 1(c) expressed using this application model.

**Network Model** If we denote the physical topology of the cluster network using a graph $G^P = (V^P, E^P)$, where $V^P$ and $E^P$ are the sets of physical machines and links, a coflow $c(u, v)$ is a subset of paths in $E^P$, and $u$ and $v$ are subsets of $V^P$ that might be distributed all over the network based on some machine allocation or task scheduling mechanism. For simplicity of exposition, we represent the network as a black box ($\mathcal{N}$) that connects all the machines in the cluster (as shown in Figure 3). Using this abstraction of the network, we can consider coflows from one or more applications to be competing for a shared resource and focus on the interactions between multiple concurrent coflows.

## 4.2 Interactions Between Coflows

Many applications run simultaneously in a shared cluster, and multiple coflows can be active in parallel. We can express the interactions between multiple coflows using the following concepts.

**Sharing** Sharing the cluster network among multiple tenants is an active research problem [8, 25, 26]. Given two concurrent coflows $c_1$ and $c_2$ from two different applications, we consider how to express their allocations $A(.)$, of the shared network $\mathcal{N}$, given their demands $D(.)$.

Reservation schemes [8] are the easiest to articulate: each coflow gets whatever fraction of $\mathcal{N}$ they asked/paid for. One can also ensure max-min fairness between $c_1$ and $c_2$ over $\mathcal{N}$ by progressively filling their demands, $D(c_1)$ and $D(c_2)$. To provide network proportionality [25], one just has to maintain $A(c_1) : A(c_2) = |c_1| : |c_2|$. Finally, using coflows, one will be able to include the network as a resource in an DRF offer [16, 17].

**Prioritization** Not all applications have the same priority. While frameworks allow assigning priorities to individual jobs, they cannot ensure these priorities in the network. By using priorities as weights, one can provide larger allocations to coflows from applications with higher priorities, i.e., $A(c_1) : A(c_2) = P(c_1)D(c_1) : P(c_2)D(c_2)$, where $P(.)$ is priority function of coflows $c_1$ and $c_2$.

**Preemption** When two applications have the same priority, preemption might be of use [18, 27]. Consider a shared cluster with two coflows $c_1$ and $c_2$, where $c_1$ has no deadline but the application cannot tolerate loss (e.g., shuffle in MapReduce) and $c_2$ belongs to an application that can trade accuracy off for lower latency (e.g., D-Streams [30]). Simply weighing by their priorities will not do in this case; one must consider the objectives $O(c_1)$ and $O(c_2)$ as well as the deadlines to determine which one to preempt and to what extent.

**Ordering** All coflows from the same application have the same priority. However, we often observe an ordering of coflows within the same application. Consider the example Dryad application in Figure 2(b), and assume that shuffle$_i$ is denoted by $c_i$.

Because Dryad does not introduce explicit barriers between its stages, $c_3$ can start while $c_1$ and $c_2$ are still active. However, the progress of $c_3$ depends on the progress of its predecessors, $c_1$ and $c_2$. This suggests a partial ordering between the three coflows, which we denote by $c_1, c_2 \geq c_3$. The relationship '$\geq$' allows concurrent execution of two coflows, whereas '$>$' denotes a strict ordering between them.

## 4.3 Developing Cluster Applications with Coflows

Using the coflow abstraction, writing a brand new cluster computing application or extending an existing one to accommodate new communication requirements boils down to specifying the end points of coflows with respective priorities, ordering, and objectives. How the functionality of a coflow will be made available is part of the underlying mechanism, which an application developer should not have to worry about – they seldom ponder how a flow moves data from one process to another.

## 4.4 Immediate Research Agenda

While defining coflow, we have identified several mechanism challenges that require immediate attention. We consider this to be a small victory for the abstraction.

**Decide on and Develop a Coordination Mechanism** We have to decide upon the architecture that will make coflows available to cluster applications and act upon the priorities, ordering, and objectives attached to individual coflows. It can be centralized (as seen in many cluster computing systems), distributed (as idealized in the networking literature), or a hybrid of both. Whatever the choice – even if it is not a one-size-fits-all – it must exist. We should also investigate the nature of interactions between various coflows, impacts of such interactions, and associated tradeoffs.

**Optimize Common Coflows** Table 2 shows that a handful of coflows can satisfy most communication requirements across a wide spectrum of cluster applications. We must optimize them and make them available for reuse. This will ensure less bugs and more optimizations. An example candidate would be the Weighted Shuffle Scheduling (WSS) algorithm in [12], which claims to be an optimal implementation of shuffle in certain scenario.

**Define Interfaces** Finally, we must define how an application will be able to access the coflows exposed by the coordination mechanism, e.g., whether through the host OS, or a cluster OS [29], or using some REST-based API.

We should expose interfaces to add new coflows, if nothing suitable exists, as well as to compose new coflows using the existing ones as building blocks. Developers should be able to create distributed cluster applications without having to reinvent communication solutions.

As we analyze further using coflows and understand more about the fundamentals of communication requirements in cluster computing, we expect to uncover additional practical and intellectual challenges.

### 4.5 Evaluating Potentials

The fundamental goal of any abstraction, including coflow, is to help understand disjoint solutions, to discern essential problems, and finally, to provide intellectual underpinnings to our understanding. Still, one might wonder about tangible gains from co-optimizing collections of concurrent flows. We refer to three instances, where thinking at the level coflows instead of individual flows have had substantial practical gains and motivated us.

- WSS [12] reported $1.5\times$ improvement in MapReduce-style shuffles by allowing the slowest flow progress faster at the expense of faster flows.
- $D^3$ [27] doubled the peak load that a datacenter network can support by dropping flows with impending deadlines in partition-aggregate coflows.
- Orchestra [12] ensured $1.7\times$ improvement of high-priority coflows through cross-coflow prioritization.

Currently, we are working on developing coflows for streaming dataflows with support for coflow ordering.

## 5 Related Networking Abstractions

**Control Plane Abstractions** Software Defined Networking or SDN [22] is gaining wide adoption in both academia and industry. The primary objective of SDN is to abstract away all the control plane mechanisms we use and replace them with abstractions that allow systematic decomposition of the problems our protocols try to solve into composable modules for reuse and conceptual separation of concerns. Techniques developed in the context of SDN might be useful for optimizing coflow implementations.

**Data Plane Abstractions** Unlike the control plane, the data plane has long had abstractions at different layers. The notion of bits in the physical layer forms frames in the link layer, which in turn are combined into packets in the network layer. On top of that, one has the abstraction of flows between two processes. Not only are these abstractions intellectually appealing, but they are practical as well – making it easier to express, solve, and optimize many of the problems we face. Coflows aim to do the same for cluster computing applications.

## 6 Conclusion

While there are a large number of cluster computing applications and a handful of paradigms to express them, networking is still handled in an ad-hoc manner. This results in repetition, under-optimization, and a general state of confusion about the principles underneath. In this paper, we looked at the communication requirements of diverse cluster computing applications and distilled resultant networking solutions developed for them to identify an application-level abstraction, *coflow*, that can succinctly represent such requirements. Not only do coflows provide a cleaner way to design, develop, and optimize the communication requirements in cluster applications, but using this abstraction and accompanying application and network models, we also exposed a set of research problems that must be addressed by the networking community – be it using the proposed abstraction or without.

## 7 References

[1] Apache Giraph. `http://incubator.apache.org/giraph`.
[2] Apache Hadoop. `http://hadoop.apache.org`.
[3] Apache Hama. `http://hama.apache.org`.
[4] Apache Hive. `http://hadoop.apache.org/hive`.
[5] Hadoop Distributed File System. `http://hadoop.apache.org/hdfs/`.
[6] S. Agarwal et al. Reoptimizing data parallel computing. In *NSDI*, 2012.
[7] M. Al-Fares et al. Hedera: Dynamic flow scheduling for data center networks. In *NSDI*, 2010.
[8] H. Ballani et al. Towards predictable datacenter networks. In *SIGCOMM*, 2011.
[9] P. Bodik et al. Surviving failures in bandwidth-constrained datacenters. In *SIGCOMM*, 2012.
[10] R. Chaiken et al. SCOPE: Easy and efficient parallel processing of massive datasets. In *VLDB*, 2008.
[11] C. Chambers et al. FlumeJava: Easy, efficient data-parallel pipelines. In *PLDI*, pages 363–375, 2010.
[12] M. Chowdhury et al. Managing data transfers in computer clusters with Orchestra. In *SIGCOMM*, 2011.
[13] T. Condie et al. MapReduce Online. In *NSDI*, 2010.
[14] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.
[15] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP*, 2003.
[16] A. Ghodsi et al. Dominant Resource Fairness: Fair allocation of multiple resource types. In *NSDI*, 2011.
[17] B. Hindman et al. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI*, 2011.
[18] C.-Y. Hong, M. Caesar, and P. B. Godfrey. Finishing flows quickly with preemptive scheduling. In *SIGCOMM*, 2012.
[19] M. Isard et al. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72, 2007.
[20] Y. Low et al. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. In *PVLDB*, 2012.
[21] G. Malewicz et al. Pregel: A system for large-scale graph processing. In *SIGMOD*, 2010.
[22] N. McKeown et al. OpenFlow: Enabling innovation in campus networks. *SIGCOMM CCR*, 38(2):69–74, 2008.
[23] C. Olston et al. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, 2008.
[24] R. Pike et al. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, 13(4), 2005.
[25] L. Popa et al. FairCloud: Sharing the network is cloud computing. In *SIGCOMM*, 2012.
[26] A. Shieh et al. Seawall: Performance Isolation for Cloud Datacenter Networks. In *HotCloud*, 2010.
[27] C. Wilson et al. Better never than late: Meeting deadlines in datacenter networks. In *SIGCOMM*, 2011.
[28] Y. Yu et al. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, 2008.
[29] M. Zaharia et al. The datacenter needs an operating system. In *HotCloud*, 2011.
[30] M. Zaharia et al. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. In *HotCloud*, 2012.
[31] M. Zaharia et al. Resilient distributed datasets: A fault-tolerant

abstraction for in-memory cluster computing. In *NSDI*, 2012.

[32] D. Zats et al. DeTail: Reducing the flow completion time tail in datacenter networks. In *SIGCOMM*, 2012.

[33] J. Zhang et al. Optimizing data shuffling in data-parallel computation by understanding user-defined functions. In *NSDI*, 2012.