LiPS: A Cost-Efficient Data and Task Co-Scheduler for MapReduce

Moussa Ehsan, Yao Chen, Hui Kang, Radu Sion, Jennifer Wong
Computer Science, Stony Brook University
{mehsan, yaochen, hkang, sion, jwong}@cs.stonybrook.edu

***Abstract*—We introduce LiPS, a new cost-efficient data and task co-scheduler for MapReduce in a cloud environment. By using linear programming to *simultaneously* co-schedule data and tasks, LiPS helps to achieve minimized dollar cost globally.**

We evaluated LiPS both analytically and on Amazon EC2. Results are significant. LiPS saved up to 81% of the actual dollar costs when compared with both the Hadoop default and the more performant delay scheduler, while also allowing users to fine-tune the cost-performance tradeoff. LiPS presents today's most cost-efficient scheduler on and should be deployed when constraints on overall makespan are flexible.

I. Introduction

The advent of global computing infrastructures endowed "large scale" with an order of magnitude upgrade over its meaning just a few years ago. Efficient large-scale data processing has become more important than ever. Facebook's data sets will soon exceed 20PB (!) and increase by 60TB daily [10]. As early as 2008, Google was handling 28PB also on a daily basis. Google's MapReduce [17] and its Apache open-source implementation Hadoop [7] constitute today's state of the art in distributed data processing. Hadoop has been widely embraced by both the open-source and commercial communities including companies such as Yahoo!, Amazon, Facebook [11] among others [2]. Just three years ago, Google's infrastructure was processing daily over 100,000 large MapReduce jobs spread across its massive computing clusters [17].

Further, the cloud computing outsourcing model has enabled everyone with a credit card to deploy data processing jobs of arbitrary complexity within minutes on hundreds to thousands of computation nodes, at very low price-points [13, 14]. An essential part of achieving the low-cost proposition is increasing and leveling utilization through multi-tenancy, currently achieved using mostly virtualization techniques. CPUs, RAM and network bandwidth are shared among cloud customers' workloads.

However, scheduling various workloads in a heterogeneous computing environment is challenging. Traditional scheduling problems that aim to minimize job makespans are known to be NP-complete. This has led researchers to largely abandon attempts to achieve optimality, and instead a number of heuristic-based scheduling mechanisms have been proposed.

The overall dollar cost-optimal schedule problem however, has been explored less. Yet, with today's massive jobs spanning thousands of tasks each, this has become more important than ever, as modern distributed data processing paradigms such as MapReduce, can be significantly more sensitive to cost than makespan, especially for long jobs and when deployed in commercial clouds. Further, as we will show here, dollar-cost aware scheduling is *not* NP-complete and maps naturally to linear systems that can be solved in polynomial time.

In clouds, increasingly data-intensive workloads require scheduler to optimize for *data locality*. Yet, common wisdom has been that computation is much cheaper to move than data and a vast majority of proposed large-scale scheduling mechanisms are task-centric and don't move data too much, beyond occasional replication.

In practice however, data has become increasingly mobile in data centers. With the advent of high-speed interconnection fabrics, intra-cluster bottlenecks have shifted back to storage media. In Amazon EC2 for example, mounted remote networked storage (EBS volumes) within hosted customer VMs have been shown to be faster than accessing the local (inter-VM shared and virtualized) disks: "the latency and throughput of Amazon EBS volumes is designed to be significantly better than the Amazon EC2 instance stores in nearly all cases" [1]. Indeed, others have proposed "flat storage" for datacenters [27].

This results in a fundamental shift: *data placement has to become a first class citizen* and should be performed jointly with job scheduling to achieve optimality. Especially, while technically we can move data around in real-time, we cannot do so for free (yet).

In heterogeneous environments, CPU costs vary wildly between different nodes and times. Consider a Job j with its data on Node A, requiring c CPU seconds per MB data. Assume the dollar costs for a CPU second on nodes A and B are a and b respectively, and data transfers between A and B cost d per MB. Then moving the data from A to B makes sense only when $c * a > c * b + d$.

This break-even point is illustrated in Figure 1. The job specifications can be found in Table I. (i) Grep searches for a pattern that only matches less than 0.01% of the input data, thus it is I/O bound. (ii) Stress1 and Stress2 are two sequential

	Grep	Stress1	Stress2	WordCount	Pi
Property	I/O	I/O	Mixed	CPU	CPU
CPU sec per 64MB	20	37	75	90	∞

TABLE I
CPU INTENSIVENESS FOR DIFFERENT JOBS. THE CPU SECOND IS THE EQUIVALENT OF A CPU SECOND ON ONE AMAZON EC2 COMPUTE UNIT. ONE EC2 COMPUTE UNIT PROVIDES THE EQUIVALENT CPU CAPACITY OF A 1.0-1.2 GHZ 2007 OPTERON OR 2007 XEON PROCESSOR.

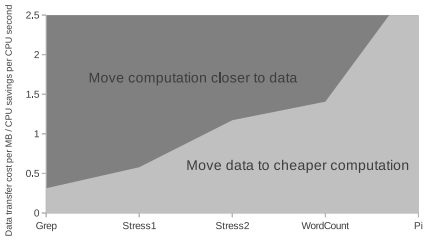


Fig. 1. Understanding when it makes sense to move the data: can the data transfer costs be made up by the CPU savings (due to different CPU costs on the source and destination node)? This figure plots the answer to this question as a function of their ratio. CPU intensive applications (Pi) are suitable to moving the data to cheaper cycles; data intensive applications (Grep) should move computation near data.

data readers with a customized tunable CPU intensive operation for every input data byte. (iii) WordCount counts the occurrence of each word in the input data. It spends significant CPU time on sorting the words on each mapper. (iv) The Pi estimator generates 1 billion samples per task and since it does not require any input it is the most CPU intensive (CPU second / data size = ∞) task.

While sharing can increase resource utilization and lower the cost, it also has the potential to raise significant *resource contention and interference* which may degrade performance. For example, scheduling multiple network-I/O intensive tasks on the same hardware may result in network saturation. This is why it is important to take into account the heterogeneity of both the computing environment and the workloads during scheduling. In a multi-tenant sharing cloud, it is also important to distribute the resource fairly among users.

Workload resource constrains, data placement, job interference and fair sharing – any one of these dimensions alone is a significant scheduling challenge. It is very tempting to try to optimize each of them in turn, however, individual optimizations for one dimension often ends up sacrificing the others and results in sub-optimality. In this paper we propose to overcome this by a co-scheduling approach in which all the above dimensions are considered jointly.

We validate the model by building the scheduler in Hadoop and experimenting in a real cluster on Amazon EC2. The experiments and simulation show that LiPS can save up to 79% of the costs when compared to the standard Hadoop scheduler. LiPS' users can also easily balance the cost-performance tradeoff. Finally, the results also demonstrate its significant fairness and utilization improvements.

II. Related Work

Scheduling. The massive amount of work on scheduling includes the seminal 1969 paper of Lampson on multiprocessing [24], and ranges from low level process scheduling [26, 28, 34] all the way to high-level task scheduling in compute clusters [22, 35, 36]. Despite differences, all of these works share a set of inherent affinities, and techniques for tightly-coupled multiprocessor compute resources have found their way in more loosely coupled networked cluster systems. Since conceptually, differences between processors and nodes are often less important than their similarities, queue- and graph-based scheduling originally targeted for multiprocessors has found immediate applicability in clusters, as early as the mid-seventies [34].

Data Placement. The insight of moving data synchronized with job scheduling in both tight and loosely coupled infrastructures has been around for several decades in various flavors.

In 1999, Alhusaini et al. [6] introduce "max-min" and "min-min" heuristics for per-level scheduling of DAGS expressing inter-task dependencies.

In [8] the authors outline a certain "convergence of computational and data grids" and discuss a number of mechanisms that couple scheduling in a compute grid (NetSolve) with a distributed storage infrastructure (IBP) control. Further, [31] proposes to either closely couple data transfers with their associated job schedules or, alternately, monitor data access patterns and then place data accordingly asynchronously. Their conclusion, somewhat surprisingly, is that tight coupling is not always needed.

In 2006, Agarwal et al. [4] proposed a "revenue"-centric model in which providers accrue revenue upon servicing a job within a certain time-frame, and penalties are incurred upon failing to do so. Other researchers [32] have started to look at various optimizations related to the Hadoop MapReduce (see below) per-task data segment allocation mechanisms, with the ultimate aim to increase data locality.

The MapReduce Processing Paradigm. MapReduce is a distributed data processing framework first introduced by Google around 2004 for highly parallelizable jobs over massive data in large infrastructures.

MapReduce loosely draws from the functional programming *map* and *reduce* high order functions. In a "map" stage, master processors partition the processing problem (a "job") into a set of sub-problems ("tasks") which can then be sent to workers. Workers in turn may act as masters, recursively. Upon completion of the sub-problem, a worker returns the result back to its master. Upon receipt of sub-problem results, the master combines them to compute the problem's solution (the "reduce" stage).

Data flow is handled efficiently. Map operations effectively cluster the input data set into (key,value) pairs. For each unique key, the MapReduce framework itself then constructs the set of all values emitted by any and all map workers and runs a reducer on it.

A MapReduce framework requires an application developer to define only a few of its key points to function: an input parser, a mapper, a partition function, a comparator, a reducer, and an output writer.

While deceptively simple, and only applicable to certain types of jobs (e.g., in the ideal case, all running reducers are independent of each other), MapReduce has significant benefits including the potential for close to optimal resource utilization, high execution speed, and ability to gracefully handle failure of infrastructure nodes and benefit from already-performed work.

A number of straightforward optimizations can be deployed in the scheduling and data placement process. For

example, to reduce bandwidth consumption, in most current implementations, reduce operations are scheduled preferably close to their target data. We discuss more such optimizations below.

Hadoop: A MapReduce Implementation. Hadoop [7] is a Java-centric open-source MapReduce implementation. A standard Hadoop setup includes a master node as well as multiple workers. Workers run the following functionalities: DataNode (stores data, can talk to other DataNodes) and/or TaskTracker (runs tasks – map, reduce, shuffle). Masters can run the same functionalities as workers as well as two additional ones: JobTracker (assigns map, reduce and shuffle tasks to TaskTrackers, "ideally [to] the [ones] that have the data, or at least are in the same rack" with the DataNode that has the data) and NameNode ("directory namespace manager" and "inode table" for the Hadoop FS).

Conceptually, Hadoop is structured in two layers. The MapReduce computation layer is composed of JobTrackers and TaskTrackers. The Hadoop data layer is composed of (often one) NameNode(s) and multiple DataNodes which abstract away any underlying storage mechanisms used.

The Hadoop FS (HDFS) is the choice implementation of the data layer. Hadoop can also work with several other filesystems within reach of the underlying OS, at the price of locality loss. However, a set of Hadoop-specific FS bridges (for Amazon S3 and CloudStore) have been developed that can aid in propagating data locality information to JobTrackers.

An exhaustive discussion on Hadoop is out of scope. A number of excellent MapReduce books are available and excellent Hadoop manuals can be found online [7].

Locality-aware MapReduce task scheduling. By default, Hadoop schedules jobs in FIFO order, with 5 priorities. When a TaskTracker becomes idle, the JobTracker assigns it the oldest highest priority task in the incoming queue. For increased data locality, the JobTracker greedily picks the task with data closest to the TaskTracker: on the same node if possible, otherwise on the same rack, and finally on a remote rack if nothing closer is available [7].

Developed at Facebook, FairScheduler defines job pools such that every pool gets a fair share of the cluster capacity over time. Pools are allocated a guaranteed minimum number of Map and Reduce slots. Jobs can be assigned to pools based on some of their properties such as user, job class, etc.. FairScheduler ensures that pools (and therefore jobs within the pools) receive equal amount of resources and therefore, short jobs can finish faster while longer jobs do not starve. [18]

Further, to also increase fairness and locality, Zaharia et al. [35] propose "delay scheduling" which improves a certain metric of max-min fairness: when the job that should be scheduled next according to fairness cannot launch a data-local task, it yields shortly to other jobs launching their corresponding tasks instead. The strategy works well if jobs are short enough and servers become idle quickly enough, making it worth waiting for a data-local task. It has been shown that delay scheduling can lead to almost 100% data locality. This can significantly increase the system's throughput. Delay scheduling is the best example of "move computation" schedulers, therefore, we compare it with LiPS in our experiments.

Interference-aware MapReduce task scheduling. Bu et al. [12] propose an interference and locality-aware task scheduler (ILA) for MapReduce in virtual clusters. Using an interference-aware task performance prediction system, ILA adaptively delay jobs to enhance data locality. ILA significantly improves data locality (up to 65%) and increases the system thoughput.

TRACON [15] is another interference-aware scheduling for data-intensive applications in virtual environments. Using an interference prediction model and based on the applications resource consumption, TRACON predicts how applications perform on different VMs. TRACON's task and resource monitor collects application characteristics in runtime for model adaption. TRACON's scheduler schedules the tasks based on the predicitons and monitor outputs. Their simulation results show up to 50% improvement in application runtime and up to 80% improvement in I/O throughput for data-intensive applications.

MapReduce task scheduling in heterogeneous environments. One issue in MapReduce frameworks in general and Hadoop in particular is the ability of a particularly slow server to significantly impact overall execution. But, since tasks run independently from each other and are thus unaware about their data source specifics, when idle cycles become available, Hadoop can schedule multiple additional copies of remaining tasks with the goal of speeding up the pipeline. The decision of whether to run such a "speculative" task copy is made by default based on a progress score for the considered task. To improve on this, Zaharia et al. [36] proposed a new mechanism (LATE) which predicts the remaining running time of a task before deciding whether to schedule an additional speculative copy.

Hybrid cloud is another main stream in the current cloud computing research. Hybrid clouds are heterogeneous by nature. Scheduling in such environments tend to be interesting for researchers [9, 25, 33] due to the imbalance between the in-house and public cloud infrastructure. As an instance, Tekin Bicer et al. [9] studied data-intensive computing in hybrid clouds when time and cost constraints should be met. Their resource allocation framework adaptively acquires cloud resources to meet time or cost constraints for data analysis tasks. Their evaluation results show that their system is capable of meeting execution deadlines with reasonable error.

Graph-based MapReduce task scheduling.

Quincy [22] is a graph-based scheduling model targeting fairness and data locality. Its main idea is to map the scheduling problem onto a min-cost network flow model [5, 21]. The competing demands of data locality, fairness and delay penalty are encoded in the network flow model's edge weights and capacities, and its solution is a schedule that minimizes global cost. Fischer et al. [19] introduce an idealized model they call the "Hadoop Task Assignment problem", its objective being finding the minimal job makespan assignment given a placement of input blocks over the set of servers. Flow-based heuristics (called "MaxCover-BalAssign") are deployed.

Recently, Palanisamy et al. [29] proposed a scheduling framework called Purlieus to couple data and VM placement. They argue that having a separate storage infrastructure is inefficient and suggest to have the data persistently stored together

on the computation nodes. Purlieus places the data on the computation nodes that will likely have enough computation capacity to host jobs that will process the data in the future. Unfortunately these assumptions are at odds with cloud models where computation instances and associated data cannot be permanently reserved without sacrificing overall efficiency.

III. Model and Definitions

The MapReduce model and Hadoop in particular (Section II) is designed to handle highly parallelizable jobs of varying resource requirements. Hadoop jobs are usually split into a set of virtually identical, independent tasks that are scheduled to run in parallel on assigned target data subsets.

Workloads with inter-task dependencies (often expressed as a DAG) can be reduced to the independent task setting through leveling techniques, in which sets of mutually independent tasks of the DAG are organized into "levels" within which independent task set scheduling is then applied [6].

Workloads with strong dependencies preventing any reasonable parallelism can also benefit from advanced scheduling mechanisms, beyond the straightforward dependency-driven constraints (i.e., scheduling tasks close to their "predecessors" since the "successors" target data is more likely to have been stored nearby etc). This becomes obvious if temporality is considered – e.g., when different, loosely-dependent tasks access overlapping data segments at different points in time.

Jobs and Data. Let $\mathcal{J}$ be a set of independent jobs (e.g., "rebuild keyword search indexes"). A job $J \in \mathcal{J}$, may be divided into a number of tasks t, $\{t_i^j$: ith task of jth job$\}$ to be run in parallel. Let $\mathcal{D}$ be the set of data objects. A data object $D \in \mathcal{D}$, can be divided into segments stored on a distributed file system such as the Hadoop FS. $Size(D)$ denotes the size of D.

Computation nodes. Let $\mathcal{M}$ be the set of physical computation nodes. In Hadoop, $M \in \mathcal{M}$ is in fact a TaskTracker. $TP(M)$ denotes a machine's CPU cycles/sec throughput, and $uptime(M)$ its uptime.

Data vs. Computation driven tasks. A majority of Hadoop jobs' compute profiles are strongly data-driven, and the number of required CPU cycles / ingested data unit remains often relatively uniform across the entire data set for a given job. This is certainly the case with the traditional Hadoop suspects such as searching, indexing, log parsing, crawling. To also capture this case, we will define a job or task's computation throughput as $TCP(x)$ in CPU time/MB (for a given agreed-upon CPU).

Data stores. Let $\mathcal{S}$ be the set of data stores in the cloud. The capacity of each data store $S \in \mathcal{S}$ is denoted by $Cap(S)$. A data store may be co-located with a computation node or it can also be a remote storage node such as in the Amazon S3 environment. In Hadoop, the most commonly used data stores are the DataNodes in the HDFS.

Job-Data relationship. Let JD be a $m \times n$ matrix denoting the data access pattern of the jobs in $\mathcal{J}$ on the data objects in $\mathcal{D}$, defined by $JD_{ij} = 1$ if J_i accesses D_j and 0 otherwise (where $m = |\mathcal{J}|$, $n = |\mathcal{D}|$). If $JD_{ij} = 1$, J_i is considered to access D_j entirely (or in some fashion that does not allow partial transfer only). This is true for many MapReduce workloads. In later stages we will also consider the case of partial data accesses with fractional values in JD_{ij} representing the ratio of the expected data traffic between J_i and D_j to the total size of D_j. In practice this matrix is populated by the scheduler itself aided by the application developer or by a monitoring framework otherwise.

Symbol	Meaning
$\mathcal{J}$	Set of Jobs
$\mathcal{D}$	Set of Data
$\mathcal{M}$	Set of Machines
$\mathcal{S}$	Set of Data stores
$CPU(J)$	CPU cycles needed for job J
$CPU\_Cost(M)$	Unit CPU cycle cost on machine M
JD	Data access matrix
JM	Job execution cost matrix
MS	M - S data transfer cost
SS	S - S data transfer cost
B	B_{ij} network bandwidth between S_i and S_j or M_j
O_i	The original location of D_i
$Cap(S)$	Storage capacity of S
$TP(M)$	Computation throughput of M
$TCP(x)$	CPU cycles/MB required by task/job x
$uptime(M)$	Uptime of M
x_{ij}^d	Portion of D_i on S_j
x_{klm}^t	Portion of J_k on M_l, accessing S_m

TABLE II
NOTATION SUMMARY. NOTE THAT DETERMINING MATRICES SUCH AS M, S, AND MS IS A PURELY INFRASTRUCTURE ISSUE AND IT IS POPULATED ONCE WHEN THE SCHEDULER IS SETUP.

Data store - data store relationship. Let SS denote the $n \times n$ matrix containing the unit data transfer cost between any two data stores. O_i is used to denote the data store where data object D_i is originally stored. We use B_{ij} to denote the network bandwidth between S_i and S_j or M_j.

Computation nodes - data stores relationship. Let MS be a $k \times l$ matrix denoting the data transfer cost between computation nodes and data stores, where $k = |\mathcal{J}|$ and $l = |\mathcal{D}|$. MS_{ij} represents the unit data transfer cost from M_i to S_j. Since download and upload cost are sometimes asymmetric, another matrix can be easily added denoting the data transfer cost from $\mathcal{S}$ to $\mathcal{M}$. For illustration, in the following we assume the costs are the same. Naturally, if a data store S_j is local to a computation node M_i, MS_{ij} is very small.

Job execution costs on Computation nodes. The heterogeneity of the considered scenarios requires considering the different costs of running certain (task,machine) combinations. Let JM be the matrix corresponding to the cost of running the entire job J_i on machine M_j, $JM_{ij} = CPU(J_i) * CPU\_Cost(M_j)$.

For both data transfer and execution, in practice, different types of costs can and will be considered: in cloud infrastructures, from a customer's point of view, the actual network costs are specified by the cloud provider. From a data center operator perspective, these costs are mainly related to operating the infrastructure [14]. Also, in heterogeneous environments where configurations vary greatly CPU cycle costs differ with computation nodes and markets.

Job resource requirements. Job-specific resource requirements can be specified by the user pre-submission, e.g. CPU and memory.

Minimize:

$$\sum_{k,l,m} (JM_{kl} + MS_{lm} * Size(D_k))x^t_{klm} \quad (1)$$

Subject to:

$$\forall k \in \mathcal{J}, \sum_{l,m} x^t_{klm} \geq 1 \quad (2)$$

$$\forall k \in \mathcal{J}, \forall m \in \mathcal{S}, \sum_{l} x^t_{klm} \leq x^d_{im} \text{ if } JD_{ki} = 1 \quad (3)$$

$$\forall l \in \mathcal{M}, \sum_{k,m,i} x^t_{klm} * D_i * JD_{ki} * TCP(k) \leq TP(M_l) * uptime(M_l) \quad (4)$$

$$0 < x^t_{klm} \leq 1 \quad (5)$$

Fig. 2. Offline Simple Task Scheduling Model

The Schedule. A *task schedule* is a mapping from $\mathcal{J}$ to $\mathcal{M}$. Since a job is usually divided into a set of tasks to be run on multiple machines, this is a 1:M mapping. Similarly, *data placement* is a 1:S mapping from $\mathcal{D}$ to $\mathcal{S}$. Let x^d_{ij} denote the portion of D_i stored on S_j, $\{x^d_{ij}|0 < i \leq |\mathcal{D}|, 0 < j \leq |\mathcal{S}|\}$. Let x^t_{klm} be the portion of J_k scheduled to run M_l while accessing data from S_m, $\{x^t_{klm}|0 < k \leq |\mathcal{T}|, 0 < l \leq |\mathcal{M}|\ 0 < m \leq |\mathcal{S}|\}$. Further, for MapReduce, a job's tasks are usually identical in data processing behavior and only target different data segments. Therefore, often, task relative running times are proportional to their target data segment sizes.

The size of the solution space. For task scheduling and data placement, there are $\binom{|\mathcal{M}|+|\mathcal{T}|-1}{|\mathcal{T}|}$ and $\binom{|\mathcal{S}|+|\mathcal{D}|-1}{|\mathcal{D}|}$ possible assignments, respectively. Efficient exhaustive search for optimality is thus highly impractical.

Table II summarizes the above notations.

IV. Offline Simple Task Scheduling

To understand the intuitions behind complex co-scheduling mechanisms, we begin with a simple task scheduling problem with no data placement, in an offline setting – in which all data has been pre-populated in data stores (data placement $\{x^d_{ij}\}$ is known) and all jobs arrive at time 0.

If the goal is to find a task schedule X^t that minimizes the combined cost of job execution and runtime data transfer, then this can be naturally formulated as a linear programming (LP) problem (Figure 2).

Constraint (2) ensures every job gets scheduled. Constraint (3) makes sure that if J_k accesses the portion of data object D_i hosted on data store S_m, its data size cannot be larger than the actual portion hosted there. Constraint (4) prevents scheduling more tasks on a node than it can handle.

If the CPU capacity of every nodes in the cluster exceeds the total CPU requirement of the entire job set, a simple greedy algorithm would also give the optimal solution: for each job J_k and its data portion on S_m, the greedy algorithm chooses M_l with lowest $JM_{kl} + MS_{lm}$ $m \in \{k_1, \ldots k_a\}$.

This is exactly what the default Hadoop scheduler achieves in the presence of sufficient resources. Despite not considering job execution costs, Hadoop's scheduler tries to maximize locality by greedily assigning tasks to nodes with minimum data access cost MS_{lm}. In reality however, since the number of tasks that can be scheduled on the same node is limited, the simple greedy algorithm does not guarantee optimum and in fact may result in quite inefficient assignments. Modeling this as a linear programming problem guarantees optimality. Various LP algorithms can be deployed, e.g., the simplex method is widely used and relatively efficient with poly-time average-case complexity [16].

Integrality of the solution. Job scheduling is often an NP-hard problem because of the Integrality requirement (the solution requires integer values). In MapReduce, since jobs can be divided into tasks and data can be split up into arbitrary chunks, scheduling does not require an integral solution. However, since starting a thread requires a small fixed amount of CPU time, no matter how small the task, a minimum viable task size exists, beyond which thread creation overheads dominate. If the LiPS solution yields tasks smaller than that, we will round them to that minimum size. Note that in the presence of Integrality requirements, rounding the solution does not guarantee the optimality in the original problem. However, the optimal solution to the problem with integral requirements cannot be better than the solution to the same problem without the requirements. This is so because the integral solution space is a subset of the fractional solution space. Therefore, we can always determine the upper bound of the distance from the optimal solution.

V. Co-Scheduling Tasks and Data

A. Offline cost-efficient co-scheduling

We will now present initial results that suggest overall cost-centric scheduling is not NP-complete and in fact can be formulated as a linear system solvable in polynomial time. The scheduling problem can then be formalized as illustrated (Figure 3).

The input. As in offline simple task scheduling, J, D, M, S, JD, JM, MS, $TP(M)$ are known. Since we no longer assume data is pre-placed in data stores, $\{x^d_{ij}\}$ (the portion of D_i stored on S_j) become unknown variables and should be included in the model. The original data locations S_{O_i} and the inter-store unit data transfer costs SS are known. Data placement considerations should include not only transfer costs but also data store capacities $Cap(S)$.

The objective. For illustration we consider here the goal to be the minimization of the combined cost of: data transfer from original locations to data stores (Equation (6)), job execution (Equation (7)), and data transfer between stores and machines during execution (Equation (8)).

The constraints. (9) and (10) ensure all data and tasks get scheduled. Constraint (11) limits the amount of data that will be scheduled on a data store. Constraint (12) ensures all nodes can handle their task load. (13) guarantees that the data accessed by a job truly exists on the accessed store. Note that this co-scheduling model is still a linear programming (LP) problem with a few more variables and constraints than the simple task scheduling model.

B. The Online Case

In the offline setting, the entire job set $\mathcal{J}$ is scheduled on $\mathcal{M}$ in one static schedule, all jobs are assumed to arrive at

Minimize:

$$\sum_{i,j} x^d_{ij} * SS_{O(i)j} \quad (6)$$

$$+ \sum_{k,l,m} x^t_{klm} * JM_{kl} \quad (7)$$

$$+ \sum_{i,k,l,m} x^t_{klm} * MS_{lm} * Size(D_i) * JD_{ki} \quad (8)$$

Subject to:

$$\forall i \in \mathcal{D}, \sum_j x^d_{ij} \geq 1 \quad (9)$$

$$\forall k \in \mathcal{J}, \sum_{l,m} x^t_{klm} \geq 1 \quad (10)$$

$$\forall j \in \mathcal{S}, \sum_i x^d_{ij} * Size(D_i) \leq Cap(S_j) \quad (11)$$

$$\forall l \in \mathcal{M}, \sum_{i,k,m} x^t_{klm} * TCP(k) * Size(D_i) * JD_{ki} \leq TP(M_l) * uptime(M_l) \quad (12)$$

$$\forall k \in \mathcal{J}, \forall m \in \mathcal{S}, \sum_l x^t_{klm} \leq x^d_{im} \text{ if } JD_{ki} = 1 \quad (13)$$

$$0 < x^t_{klm} \leq 1 \quad (14)$$

$$0 < x^d_{ij} \leq 1 \quad (15)$$

Fig. 3. Offline Cost-efficient Co-scheduling

time 0, and can be scheduled on nodes for the entire uptime.

Obviously this is not the case in the "online" reality. Numerous ideas have been proposed on dealing with the online case [30]. In practice however, greedy mechanisms (e.g. default Hadoop scheduler) that immediately schedule on available resources have mostly dominated. This is due to the perception that any idling is usually counter-productive. However, when dynamic data placement and co-scheduling multiple jobs are considered, there are often significant benefits to non-greedy patience (e.g., by waiting for a critical mass of tasks to reach the scheduling queue). In [35], authors also found that waiting for a small period before scheduling tasks improves data locality. In addition, assuming that the entire uptime is available during scheduling may result in poor performance. Intuitively, to minimize total cost, the scheduler may choose to schedule many tasks on the cheapest node.

In the following we will discuss an epoch-based model. The scheduler first waits for a small period of time to collect more jobs from the queue. And instead of assuming the nodes' entire uptime is available to these jobs, we introduce "time epochs" (of duration denoted by e). Only the total CPU time available in the next epoch will be considered when LiPS computes the schedule for the jobs in the current queue.

Of course, it may happen that the amount of CPU time required by the jobs exceeds the available CPU time in the epoch. In this case, the linear programming model will not have a feasible solution. To address this, we introduce a fake node F, of "unlimited" CPU capacity and an extremely high CPU cycle cost. This will ensure the problem always has a feasible solution, and since the cost of running on this fake node is so high, the model will not choose to use this node unless the real CPU capacity in the epoch is lower than

Minimize:

$$\sum_{i,j} x^d_{ij} * SS_{O(i)j} \quad (16)$$

$$+ \sum_{k,l,m} x^t_{klm} * JM_{kl} \quad (17)$$

$$+ \sum_{i,k,l,m} x^t_{klm} * MS_{lm} * Size(D_i) * JD_{ki} \quad (18)$$

Subject to:

$$\forall i \in \mathcal{D}, \sum_j x^d_{ij} \geq 1 \quad (19)$$

$$\forall k \in \mathcal{J}, \sum_{l,m} x^t_{klm} \geq 1 \quad (20)$$

$$\forall k \in \mathcal{J}, \forall l \in \mathcal{M}, \sum_{m,i} \frac{x^t_{klm} * Size(D_i) * JD_{ki}}{B_{lm}} \leq e \quad (21)$$

$$\forall j \in \mathcal{S}, \sum_i x^d_{ij} * Size(D_i) \leq Cap^e(S_j) \quad (22)$$

$$\forall l \in \mathcal{M}, \sum_{i,k,m} x^t_{klm} * TCP(k) * Size(D_i) * JD_{ki} \leq TP(M_l) * e \quad (23)$$

$$\forall k \in \mathcal{J}, \forall m \in \mathcal{S}, \sum_l x^t_{klm} \leq x^d_{im} \text{ if } JD_{ki} = 1 \quad (24)$$

$$0 < x^t_{klm} \leq 1 \quad (25)$$

$$0 < x^d_{ij} \leq 1 \quad (26)$$

Fig. 4. Online Cost-efficient Co-scheduling Model

the actual CPU requirement of jobs. Any tasks the solution schedules on F will in effect not be scheduled in this epoch and be put back in the waiting queue to be considered in the next epoch.

In the online setting, this scheduling model is invoked for every epoch e. The objective function and constraints are mostly the same as in the offline case – with the exception of $uptime(m)$ being replaced with e. Constraint (21) ensures that the data transfer time does not exceed the epoch length – this is important especially for data intensive jobs.

Epoch. Epoch-based mechanisms with short epoch durations naturally converge to greedy scheduling. Indeed, we can define an epoch size of zero by convention to mean the greedy scheduling of the lowest cost task from the queue. The choice of epoch length also turns out to be related to the tradeoff between performance and cost. Longer epochs allow the scheduler to more greedily schedule jobs across the entire time spectrum, to lower costs. This may result in longer job execution times. A more detailed discussion and evaluation of epoch lengths can be found in Section VI-B

In practice the epoch length can be either fixed in advance, or adaptively changed as the performance and cost preferences are changed by users.

Instance Price	CPU / EC2 Compute Units	Mem. (GB)	Storage (GB)	Price ($ per hr.)
m1.small	1 / 1	1.7	160	0.08-0.12
m1.medium	1 / 2	3.75	410	0.13-0.23
c1.medium	2 / 5	1.7	350	0.17-0.23

TABLE III
AMAZON EC2 INSTANCE TYPES. ONE EC2 COMPUTE UNIT PROVIDES THE EQUIVALENT CPU CAPACITY OF A 1.0-1.2 GHZ 2007 OPTERON OR 2007 XEON PROCESSOR. ALTHOUGH C1.MEDIUM COSTS ABOUT THE SAME COST AS M1.MEDIUM, IT HAS A 2.5 TIMES HIGHER CPU CAPACITY THAN M1.MEDIUM. AS A RESULT, IN TERMS OF COST PER EC2 COMPUTE UNIT CPU SECOND, C1.MEDIUM IS 4-5 TIMES CHEAPER THAN M1.MEDIUM (C1.MEDIUM: 0.92-1.28 MILLICENT; M1.MEDIUM: 4.44-6.39 MILLICENT)[1].

VI. EVALUATION

A. Experimental Environment

The LiPS scheduler is an instance of the Hadoop TaskScheduler interface and plugs into JobTracker. It also includes a new ReplicationTargetChooser for data placement in the NameNode. LiPS uses the GNU Linear Programming Kit (GLPK [23]) to solve its linear programming problems.

Here we experimentally evaluate LiPS (Hadoop 0.20.203.0) in terms of *reducing cost* . We built two different Amazon EC2 testbeds with 20 and 100 EC2 nodes, spread across three physical availability zones. For illustration purposes, we used three types of EC2 instances (see Table III) –the medium instances (m1.small and m1.medium) and the high CPU medium instance (c1.medium). Note that the results hold across the entire spectrum of instances (e.g. including m1.large). Budget limitations prevented us from deploying larger clusters. We further validated the scalability and performance of LiPS through simulations at larger scale.

Network. To bring diversity at the network level –since Amazon does not allow network control– we were first using netem to modulate the network bandwidth between nodes –nodes that were in the same zone were allocated 500Mbps bandwidth and nodes in different zones 250 Mbps. But later, we discovered that the RTT latency across availability zones is about three times the RTT latency within each zone. Also, the RTT latency is not the same within (or across) different availability zones. This gives us enough diversity which eliminates the use of netem [3]. Amazon charges $0.01 per GB (62.5 Millicent per 64MB block) for data transfer between different zones.

Jobs. For the first set of experiments on 20 EC2 nodes, we used four different benchmarks – grep (I/O intensive), PiEstimator (CPU intensive), WordCount (mixed), and StressTest (a custom sequential data reader with a tunable CPU intensive operation performed on each word). In the experiments we created 9 of these jobs of different sizes (see Table IV).

For the scaled-up experiments on 100 EC2 nodes, we created a 400-job workload using SWIM [20]. The workload is a customized version of Facebook's workload jobs trace available on SWIM's website.

Speculative Tasks. As discussed in section II in order to help jobs finish faster, Hadoop may run a part of the job in speculative execution mode. In this mode, several copies of one task will run simultaneously on different nodes and whichever finishes first becomes the final copy and the rest are killed. Speculative execution is enabled by default; however it interferes with the LiPS scheduler as LiPS scheduler predetermines where each task should run. Moreover, LiPS aims on reducing the cost, not job execution time and such extra copies will only result in additional unnecessary cost. Hence, we disable this feature. Note that keeping this feature enabled may lead to better performance for both delay and default schedulers but it will also increase their dollar cost.

Timeouts. The schedule may result in tasks being run far from their data sources. In these cases, it is important to allow for data transfers without triggering any default timeout mechanisms. Hadoop has a default 10 minutes timeout after which it kills any job that is not reporting progress. However, since in LiPS data transfers can take longer, especially in the case of slow or saturated networks, we increased the timeout setting to an empirically validated 20 minutes.

The LiPS Scheduler Overhead. LiPS relies on the LP module which is implemented using GLPK [23]. Since LiPS needs to solve the LP problem in each epoch, we expect that the scheduling overhead is higher than that in the Hadoop default scheduler. However in our experiments for problems involving thousands of tasks, its execution time was almost negligible (10s of ms) especialy when compared to job durations (10s of mins).

B. Cost Reduction

Before we experimentally evaluate LiPS on actual amazon instances with actual dollar costs, to understand the cost reduction of LiPS, we first run a simulation with clusters ranging up to 100 nodes and jobs up to 1000 tasks.

We then deploy lips in Amazon, and validate these numbers on actual incurred dollar charges. When compared with the default scheduler, we observe that in general LiPS saves more when there are more opportunities – including high node diversity, larger clusters, and longer epochs.

Size of the cluster and jobs. The size of the cluster and the size of jobs are two key factors which affect the potential

	J1-2	J3-4	J5-7	J8-9
	Pi	WordCount	Grep	Stress2
Job size	4	160	320	160
Input	–	10GB	20GB	10GB

TABLE IV
JOB DETAILS. THE CPU REQUIREMENT PER BLOCK INPUT IS EXPRESSED IN AMAZON EC2 COMPUTE UNIT SECONDS. THE PI ESTIMATOR GENERATES 1 BILLION SAMPLES PER TASK AND IT DOES NOT REQUIRE ANY INPUT DATA. GREP SEARCHES FOR A PATTERN THAT ONLY MATCHES LESS THAN 0.01% OF INPUT DATA, SO IT IS I/O BOUND. THE JOBS CONTAIN TOTAL INPUT SIZE OF 100GB AND MORE THAN 1608 MAPS TASKS.

[1]Please note that our price-model is slightly different than that in Amazon. Amazon charges users based on instances per hour; however, for demonstration purposes and in order to use actual prices we break down the charges to EC2 CPU unit per second.

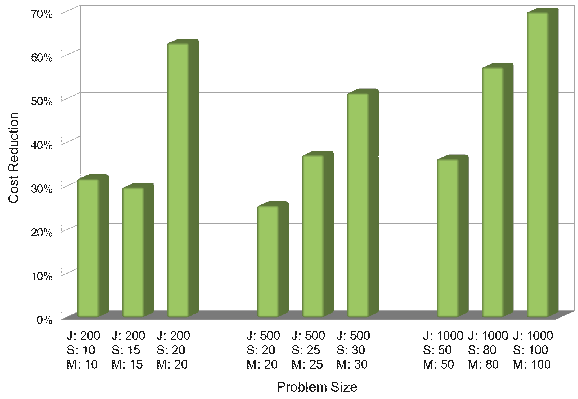


Fig. 5. Average cost reduction of LiPS compared to default scheduler in simulated environments. CPU second cost range: 0 - 5 Millicent; Data input size range: 0 - 6GB; Range of data transfer cost between two nodes: 0 - 60 per 64MB; Job CPU requirement range: 0 - 1000 CPU second. X-axis represents the size of the problem – J: total number of tasks, S: number of data stores, M: number of computation nodes. LiPS results in higher cost reduction with increasing cluster size, since it has more freedom placing data and tasks in the cluster.

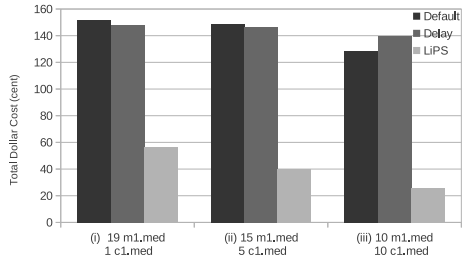


Fig. 6. Cost Reduction of LiPS executing J1-J9 (1608 maps in total) in a 20 node cluster with different settings. LiPS can save 62-81% of the costs when compared with the delay scheduler –almost the same with the Hadoop default scheduler.

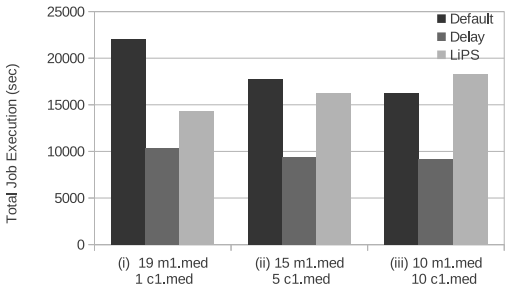


Fig. 7. The total job execution time in Figure 6. Scheduling with LiPS will result in longer job execution times in comparison with the delay scheduler (between 40%-100% longer). LiPS gives priority to the cheaper –and at the same time slower– instances; therefore, adding more powerful instances results in longer job execution time.

cost reduction. Consider the extreme case in which there are many available nodes but only one job with one single map task to be scheduled. In this case, there is not much room left for cost reduction. On the other hand, if there are many jobs but only one node, the only possible schedule will be having the jobs wait in the queue to run on that node. In this case, any scheduler will generate the same schedule and have the same cost.

Accordingly, LiPS achieves best results for cases when there are enough jobs and a sufficiently provisioned cluster. To understand this behavior (within our limited Amazon budget), we simulated larger clusters up to 100 nodes and multiple jobs with sizes up to 1000 tasks. The jobs were completely random as well as the size of the cluster and its topology. The simulator creates and solves the LP problem, and therefore, computes the dollar cost of the optimal scheduling result. With the same setting, it then shuffles the data blocks randomly within the cluster and then schedules ALL tasks local to the data blocks. This is the best possible task scheduling with 100% data locality. The result of such a default scheduling is the same as the ideal delay scheduler.

Figure 5 shows the average cost reduction of LiPS compared to the default scheduler. It can be seen that with increasing cluster size, increasing cost reduction can be achieved, ranging from 30% for 200 tasks in a 10 node to nearly 70% for 1000 jobs on a 100 node cluster.

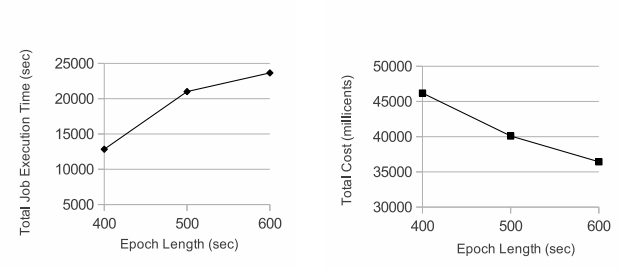
Node diversity. When there are more nodes with different CPU costs in the cluster, LiPS can optimally choose the best nodes to minimize cost within the defined constraints. In a 20-node cluster across different zones, we ran a set of jobs listed in Table IV. The experiments start with nodes of similar type (m1.medium) in which LiPS saves the least – 62% of the cost when compared with both the default and delay schedulers (Figure 6). This is so due to lack of diversity which results in less saving opportunities to save. Note that the majority of the savings comes from scheduling optimizations related to network transfers and not so much from computation cost, because all slave nodes run at similar costs.

We gradually add a different type of node (c1.medium) to the cluster. Since c1.medium features a lower cost per CPU second, all schedulers would benefit from this; however, since LiPS seeks for the optimal solution, it reduces total cost by 79% to 81% (for 50% c1.medium nodes). This suggests that in order to keep utilization high, using larger instances is desirable from an end user's point of view, even though they are apparently more expensive than smaller instances.

The total job execution time for the same set of experiments are shown in Figure 7. We can see that adding more powerful instances results in longer job execution times in LiPS. This natually follow LiPS tends to schedule jobs on cheaper instances despite their limited computation capacity. On the other hand, the delay scheduler benefits from the more powerful instances that results in shorter makespans.

To validate at scale, we increased our cluster size to a 100 nodes Further to ensure having a diversity the cluster consisted of three types of instance types (m1.small, m1.medium and c1.medium), located in three different availability zones (us-east-a, us-east-b and us-east-c). We used a workload similar to Facebook's composed of 400 jobs created using SWIM.

These were derived from one of SWIM's Facebook work-



(a) Total job execution time (b) Total cost

Fig. 8. The relationship of total job execution time and cost using the experiment settings in Figure 6 (iii). As we increase the epoch length the cost decreases, at the expense of higher execution time, however.

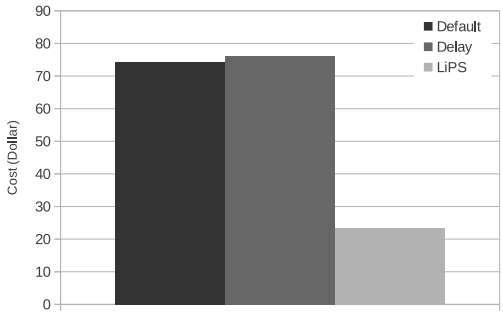


Fig. 9. The total dollar-cost in a 100-node EC2 cluster. Scheduling with LiPS results in 68% to 69% cost reduction in comparison with both schedulers, which shows the power of LiPS when applied to diverse clusters.

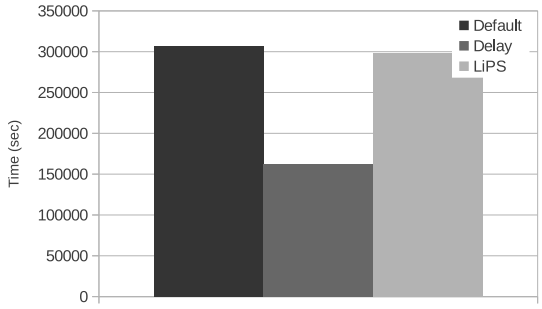


Fig. 10. The total job execution time in Figure 9. Scheduling with LiPS will result in 40% to 100% longer job execution times when compared with the delay scheduler. LiPS gives priority to the cheaper and often slower instances; therefore adding more powerful instances results in longer job execution time.

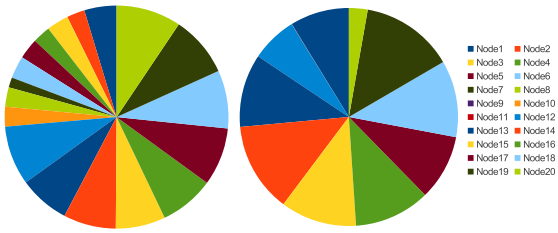


Fig. 11. Accumulated CPU time breakdown per node. Epoch length 400 second (left) and 600 second (right). Shorter epoch length results in higher parallelism and faster job executions (but also higher cost).

load traces (FB-2010_samples_24_times_1hr_0.tsv), one day in duration, containing 24 historical trace samples, each 1 hour long. The workload is composed of interactive (short), medium-size and long jobs.

LiPS saves 68% to 69% when compared with both the delay and default scheduler (Figure 9). On the other hand, as shown in Figure LiPS does not optimize for job execution time (Figure 10) which results in execution time similar to the default Hadoop scheduler.

Epoch length. The key idea of LiPS is to find the most cost-efficient schedule within the given constraints. One of the key constraints is the epoch length. Choosing the length of an epoch is a trade-off between performance and cost. In an extreme case when the epoch length is infinite, LiPS will generate a schedule that greedily places every job be executed on their most favorable node. The catch is that, in this case, it is likely that many jobs will be scheduled to run on the same node, resulting in slow job execution. One way for LiPS to balance performance and cost is by tuning the length of the scheduling epoch.

Figure 8 shows how job execution time and total cost change with increasing epoch length in the same testbed of Figure 6. Figure 11 shows the accumulated CPU time spent on each node in an epoch. It can be seen that shorter epochs lead to higher parallelism; jobs will be executed faster, but the cost may be higher. When the epoch is longer, the cost is lower, but jobs also take longer to execute.

VII. Conclusions

We proposed to make data placement a first class citizen in cloud scheduling. We introduced a new cost-efficient scheduling model. We implemented an instance of this model on the popular Hadoop platform. Results show significant cost savings of up to 79-81% when compared to the Hadoop default scheduler and existing optimized schedulers Finally, the model also allows for flexible control of the tradeoff between job makespans and costs. LiPS presents today's most cost-efficient scheduler on and should be deployed when constraints on overall makespan are flexible.

References

[1] Amazon Elastic Block Store (EBS). Online at http://aws.amazon.com/ebs/.

[2] Applications powered by hadoop. Online at http://wiki.apache.org/hadoop/PoweredBy.

[3] Network Latency Inside And Across Amazon EC2 Availability Zones. Online at www.orensol.com/2009/05/24/network-latency-inside-and_across-amazon-ec2-availability-zones/.

[4] Vikas Agarwal, Gargi Dasgupta, Koustuv Dasgupta, and Amit Purohit. Deco: Data replication and execution co-scheduling for utility grids. In *Proceedings of International Conference on Service Oriented Computing*, pages 4–7, 2006.

[5] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network flows: theory, algorithms, and applications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[6] Ammar H. Alhusaini, Viktor K. Prasanna, and C. S. Raghavendra. A unified resource scheduling framework for heterogeneous computing environments. In *Proceedings of the Eighth Heterogeneous Computing Workshop*, HCW '99, pages 156–, Washington, DC, USA, 1999. IEEE Computer Society.

[7] Apache. Online at http://hadoop.apache.org/.

[8] Dorian C. Arnold, Sathish S. Vadhiyar, and Jack Dongarra. On the convergence of computational and data grids. *Parallel Processing Letters*, pages 187–202, 2001.

[9] T. Bicer, D. Chiu, and G. Agrawal. Time and cost sensitive data-intensive computing on hybrid clouds. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 636–643, 2012.

[10] Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, Rodrigo Schmidt, and Amitanand Aiyer. Apache hadoop goes realtime at facebook. In *Proceedings of the 2011 international conference on Management of data*, SIGMOD '11, pages 1071–1080, New York, NY, USA, 2011. ACM.

[11] Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, Rodrigo Schmidt, and Amitanand Aiyer. Apache hadoop goes realtime at facebook. In *Proceedings of the 2011 international conference on Management of data*, SIGMOD '11, pages 1071–1080, New York, NY, USA, 2011. ACM.

[12] Xiangping Bu, Jia Rao, and Cheng-Zhong Xu. Interference and locality-aware task scheduling for mapreduce applications in virtual clusters. In *Proceedings of the 22st international symposium on High-Performance Parallel and Distributed Computing*, HPDC '13, 2013.

[13] Yao Chen and Radu Sion. On securing untrusted clouds with cryptography. In Ehab Al-Shaer and Keith B. Frikken, editors, *WPES*, pages 109–114. ACM, 2010.

[14] Yao Chen and Radu Sion. To cloud or not to cloud? musings on costs and viability. In *Proceedings of the 2nd ACM Symposium on Cloud Computing 2011*, 2011.

[15] Ron C. Chiang and H. Howie Huang. Tracon: interference-aware scheduling for data-intensive applications in virtualized environments. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 47:1–47:12, New York, NY, USA, 2011. ACM.

[16] George B. Dantzig. *Linear programming and extensions*. Rand Corporation Research Study. Princeton Univ. Press, Princeton, NJ, 1963.

[17] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[18] Apache FairScheduler. Online at http://tiny.cc/z2ewxw.

[19] Michael J. Fischer, Xueyuan Su, and Yitong Yin. Assigning tasks for efficiency in hadoop: extended abstract. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, SPAA '10, pages 30–39, New York, NY, USA, 2010. ACM.

[20] Statistical Workload Injector for MapReduce (SWIM). Online at https://github.com/SWIMProjectUCB/SWIM/wiki.

[21] George Heineman, Gary Pollice, and Stanley Selkow. *Algorithms in a Nutshell*. O'Reilly Media, Inc., 2008.

[22] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 261–276, New York, NY, USA, 2009. ACM.

[23] GNU Linear Programming Kit. Online at http://glpk-java.sourceforge.net/.

[24] Butler W. Lampson. A scheduling philosophy for multiprocessing systems. *Communications of the ACM*, 11:347–360, 1968.

[25] Kenli Li, Xiaoyong Tang, and Qifeng Yin. Energy-aware scheduling algorithm for task execution cycles with normal distribution on heterogeneous computing systems. In *Parallel Processing (ICPP), 2012 41st International Conference on*, pages 40–47, 2012.

[26] V. K. Naik, M. S. Squillante, and S. K. Setia. Performance analysis of job scheduling policies in parallel supercomputing environments. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, Supercomputing '93, pages 824–833, New York, NY, USA, 1993. ACM.

[27] Edmund B. Nightingale, Jeremy Elson, Jinliang Fan, Owen Hofmann, Jon Howell, and Yutaka Suzue. Flat datacenter storage. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 1–15, Berkeley, CA, USA, 2012. USENIX Association.

[28] J. K. Ousterhout. Scheduling techniques for concurrent systems. In *Proceedings of the Third International Conference on Distributed Computing Systems*, pages 22–30, 1982.

[29] Balaji Palanisamy, Aameek Singh, Ling Liu, and Bhushan Jain. Purlieus: Locality-aware Resource Allocation for MapReduce in a Cloud. 2011.

[30] Michael L. Pinedo. *Scheduling: Theory, Algorithms, and Systems (3rd Ed)*. Springer, 2008.

[31] Kavitha Ranganathan and Ian Foster. Decoupling computation and data scheduling in distributed data-intensive applications. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, HPDC '02, pages 352–, Washington, DC, USA, 2002. IEEE Computer Society.

[32] Saba Sehrish, Grant Mackey, and Jun Wang. Improving I/O Performance by Coscheduling of I/O and computation on Commodity based Clusters. Online at www.usenix.org/events/fast09/wips_posters/sehrish_wip.pdf.

[33] Bikash Sharma, Timothy Wood, and Chita R. Das. HybridMR: A Hierarchical MapReduce Scheduler for Hybrid Data Centers. In *Proceedings of the 33rd International Conference on Distributed Computing Systems*, ICDCS'13, 2013.

[34] H. S. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Trans. Softw. Eng.*, 3:85–93, January 1977.

[35] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 265–278, New York, NY, USA, 2010. ACM.

[36] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 29–42, Berkeley, CA, USA, 2008.