Netco: Cache and I/O Management for Analytics over Disaggregated Stores

Virajith Jalaparti
Microsoft

Chris Douglas
Microsoft

Mainak Ghosh*
UIUC

Ashvin Agrawal
Microsoft

Avrilia Floratou
Microsoft

Srikanth Kandula
Microsoft

Ishai Menache
Microsoft

Joseph (Seffi) Naor
Microsoft

Sriram Rao†
Facebook Inc.

ABSTRACT

We consider a common setting where storage is disaggregated from the compute in data-parallel systems. Colocating caching tiers with the compute machines can reduce load on the interconnect but doing so leads to new resource management challenges. We design a system Netco, which prefetches data into the cache (based on workload predictability), and appropriately divides the cache space and network bandwidth between the prefetches and serving ongoing jobs. Netco makes various decisions (what content to cache, when to cache and how to apportion bandwidth) to support end-to-end optimization goals such as maximizing the number of jobs that meet their service-level objectives (e.g., deadlines). Our implementation of these ideas is available within the open-source Apache HDFS project. Experiments on a public cloud, with production-trace inspired workloads, show that Netco uses up to 5× less remote I/O compared to existing techniques and increases the number of jobs that meet their deadlines up to 80%.

CCS CONCEPTS

• **Theory of computation → Caching and paging algorithms**; • **Computer systems organization → Cloud computing**;

KEYWORDS

Disaggregated architectures; data analytics; cloud computing; caching

ACM Reference Format:
Virajith Jalaparti, Chris Douglas, Mainak Ghosh, Ashvin Agrawal, Avrilia Floratou, Srikanth Kandula, Ishai Menache, Joseph (Seffi) Naor, and Sriram Rao. 2018. Netco: Cache and I/O Management for Analytics over Disaggregated Stores. In *Proceedings of SoCC '18: ACM Symposium on Cloud Computing, Carlsbad, CA, USA, October 11–13, 2018 (SoCC '18),* 13 pages. https://doi.org/10.1145/3267809.3267827

*Work done during an internship at Microsoft.
†Work done while at Microsoft.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SoCC '18, October 11–13, 2018, Carlsbad, CA, USA

© 2018 Association for Computing Machinery.

ACM ISBN 978-1-4503-6011-1/18/10…$15.00
https://doi.org/10.1145/3267809.3267827

1 INTRODUCTION

Public clouds physically separate compute resources from the storage tiers [4, 6, 9, 19, 23]. A typical Spark deployment on Amazon web services uses VMs from EC2 (the compute tier) but stores the data in S3 (the storage layer) [11]. Since data-parallel analytics frameworks have been built with the assumption that the storage is colocated with the compute, the compute-storage disaggregation in public clouds creates a bottleneck in the interconnection between the store and compute. This bottleneck delays jobs and adds to their performance variability.

Existing techniques to mitigate the storage–compute bottleneck are reactive and do not consider job-level objectives. For example, Alluxio [3] and Databricks IO Cache [13] maintain a cache on compute-local memory, SSD or disks with cache replacement policies such as LRU-k [64]. These techniques do not optimize job-level objectives such as meeting deadlines because they ignore the job structure. For example, if a job joins two inputs, caching just one of the inputs may not speed up the job. Furthermore, these techniques have a poor cache hit rate because they ignore predictability in workloads. Many prior works show that a large fraction of the workload is recurring and predictable [24, 47, 53, 54]; prefetching the inputs for these queries into the cache can substantially increase the cache hit rate.

Motivated by these observations, we design Netco which manages one or more caching tiers that are colocated with compute resources in the cloud. To this end, Netco uses as input information from previous job executions, the limits on the cache size and the limits on usable bandwidth on the compute–storage interconnect. Netco generates a schedule that determines when and which datasets to prefetch into or evict from the cache, the bandwidth allocated for each prefetch and the bandwidth allocated for jobs to load data that is not prefetched or cached. The key idea behind Netco is to jointly optimize allocations of the various resources in order to meet end-to-end objectives such as maximizing the number of jobs that meet deadlines. Example outcomes of this optimization include (i) to preferentially prefetch datasets that are used by multiple contemporaneous jobs, (ii) to preferentially cache smaller datasets, (iii) to preferentially cache datasets for jobs with tight deadlines and (iv) to coordinate the cached fraction of datasets that are used in joins.

Determining such a caching schedule requires Netco to solve a novel *joint* bandwidth and cache allocation problem. We make careful design choices when formulating this problem to obtain a

scalable solution (§3). In particular, we adopt a *hierarchical* optimization approach, in which we perform the high-level planning at the granularity of files, and use a separate (lower-level) algorithm to assign resources (network bandwidth between storage and compute tiers, cache capacity) to blocks within each file[1]. The higher-level optimization problem uses a *unified* Linear-Programming (LP) formulation which allows operators to flexibly choose between different end-to-end objectives for different operational regimes: (i) maximize the number of deadline SLOs satisfied when all SLOs cannot be met and otherwise (ii) minimize bandwidth used on the interconnect to the primary store. While (ii) follows from the LP-formulation, (i) is NP-hard and hard-to-approximate (§4.2). Accordingly, we develop efficient heuristics using LP-rounding techniques. The lower-level algorithm then translates the solution of the LP to an actual resource assignment (§4.3), which can be implemented in practice. Such decoupling helps us significantly reduce the complexity of the underlying optimization problems.

We have implemented Netco on top of Apache Hadoop/HDFS [16], a widely-used store for data analytics. We added to HDFS the capability to *mount* and cache data present in other remote filesystems. The caching plan determined by Netco is enforced by a separate standalone component, where custom caching policies can be implemented. Such separation helped limit the amount of changes to HDFS. Our changes to HDFS are released as part of Apache Hadoop 3.1.0 [2]. While our implementation of Netco is aimed at data-analytics clusters that use HDFS on public clouds, the core ideas apply in other, similar, disaggregated scenarios including on-premise clusters.

We evaluate our implementation of Netco on a 50-node cluster on Microsoft Azure and on a 280-node in-house cluster that disaggregates compute and storage. Using workloads derived from production traces, we show that Netco improves SLO attainment by up to 80% while using up to 5× less remote I/O bandwidth, compared to workload-agnostic caching schemes such as LRU and PACMAN-LIFE [28], and simple prefetching techniques. These savings translate to Netco reducing the I/O cost per SLO attained by 1.5×–7× on Azure.

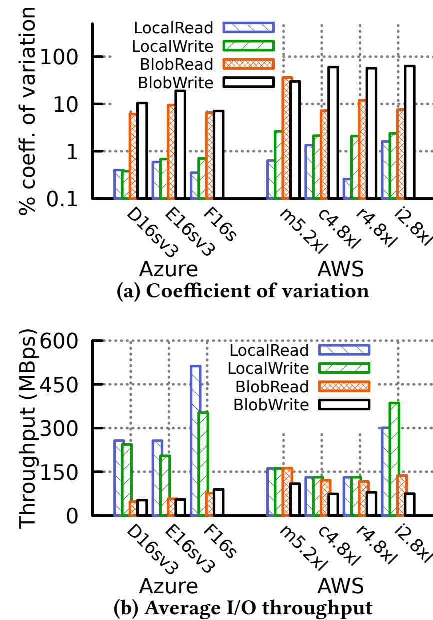
While Netco offers direct benefit to jobs that are known in advance, ad hoc jobs also benefit because (a) more resources are available to them due to Netco's efficient execution of the predictable jobs and (b) Netco offers reactive caching policies (e.g., PACMan [28]). In fact, our experiments show that the median runtime of ad hoc jobs reduces by up to 68%.

Ideally, if the interconnect bandwidth is sufficiently large, then caching tiers are not essential. However, public clouds do not allow independent control of the interconnect bandwidth. The primary method to increase the interconnect bandwidth today, is to pay for an even larger compute tier and/or storage tier [1, 10]. In this context, Netco can be seen as a cost-saving measure; Netco's caching tiers increase utilization on the compute tier and finish more jobs faster on fewer VMs.

In summary, our contributions are:

- A new architecture that computes and enforces a cache schedule that is aware of limits on both store–compute I/O bandwidth and cache size (§3). We implement this architecture on top of Apache Hadoop/HDFS (§5).
- A novel problem formulation which jointly optimizes I/O bandwidth and local storage capacity in disaggregated architectures to meet job SLOs, and practical algorithms to solve it (§4).
- An evaluation of Netco using real deployments and production workloads, demonstrating that Netco improves SLO attainment while significantly reducing bandwidth used on the storage–compute interconnect (§6).

[1] Distributed file systems such as Apache HDFS [16] partition files into one or more *blocks*, each representing a contiguous portion of the file.



(a) Coefficient of variation

(b) Average I/O throughput

Figure 1: Benchmark of local and remote stores on Azure and AWS. The graphs show (a) the coefficient of variation of time to read or write 512MB to local and remote stores, and (b) the average I/O throughput achieved, over 100 trials. The x-axis shows different VM types on Azure and AWS. Coefficient of variation is the ratio of standard deviation to mean and is a widely-used measure of variability.

2 MOTIVATION

In this section, we provide empirical evidence that motivates and guides the design of Netco (§2.1–§2.2). We also illustrate through a simple example (§2.3) the merits of jointly scheduling network and caching resources.

2.1 Storage in public clouds

Cloud storage offerings can be categorized into two types: (i) *primary*, remote storage such as Amazon S3 [6] and Azure Blob Storage [23], which can hold petabytes of data "at rest" in a globally addressable namespace, and (ii) *local* storage volumes which are only addressable by individual compute instances and can hold at most few tens of terabytes of data; examples include Amazon EBS [5], Azure Premium Storage [18] and local VM storage.

We measure the I/O throughput as well as the variability of both local and remote storage for two major cloud providers: Azure and AWS. We consider Azure Blob Store and S3 as remote stores,

(a) Histograms of files and total size, binned by the number of jobs accessing them. **(b) Scatter plot of avg. processing rate of a file vs. the number of jobs that read it.** **(c) CDF of *prefetch slackness* of files.**

Figure 2: Characteristics of workloads from production data analytics cluster at Microsoft.

and SSD volumes which are attached to VMs as local storage. We repeatedly write and read 512MB of data from different types of VMs (compute-optimized, memory-optimized, storage-optimized).[2] Figure 1a shows that local reads/writes have low variance (although the variance depends on the type of VM used). Reads/writes to primary remote stores have a much larger variance (5×–30× larger). Similar observations have been made in prior work [50, 57]. We also observe that the throughput of the remote storage is 1.1× to 5× lower on AWS and 3.9× to 6.6× lower on Azure compared to the local storage (Figure 1b).

Thus, remote cloud storage has limited and variable I/O throughput. As a result, without significant over-provisioning, remote storage cannot meet the needs of big data frameworks that require strict SLOs. Local storage, on the other hand, has higher throughput and lower variability. This motivates Netco's use of local storage to build cache tiers and help jobs meet their deadline SLOs.

2.2 Analysis of production workloads

We next analyze the characteristics of typical big data workloads and provide insights that motivate the design of Netco. As customer telemetry data is hard to obtain from public clouds due to privacy concerns, we analyze a private production data analytics cluster at Microsoft along with few publicly available workloads. The cluster being analyzed consists of thousands of machines; we use logs collected over one week which contain tens of thousands of jobs and hundreds of thousands of input files.

Job characteristics can be predictable. As noted in several prior works, various characteristics of analytics jobs can be inferred from prior execution logs [24, 47, 54]. In particular, prior work shows that nearly 40–75% jobs are *recurring* (i.e., the same code or script is run on different/changing input datasets), and that their submission times, deadlines, and input reuse times can be inferred with high accuracy [54].

Caching files saves network bandwidth. Figure 2a shows a histogram of the fraction of files (and bytes) that are accessed by a certain number of jobs (x-axis). We find that about 25% of the files are accessed by more than 10 jobs. These files contribute to more than 50% of the bytes accessed from the store. Similar observations have been made for other workloads (e.g., Scarlett [26], PACMan [28]). Caching such frequently accessed files can significantly reduce the data read from remote storage.

File access recency or frequency is insufficient to determine which files to cache. Different analytics workloads can process data at very different rates, e.g., reading a compressed file vs. uncompressed, JSON parsing vs. structured data. Jobs that are capable of processing data at higher rates or I/O bound jobs can be sped-up more by caching (or prefetching) their input files as they can take advantage of the higher I/O throughput local storage offers. However, standard caching policies (e.g., LRU, LFU) depend on file access recency and/or frequency, and do not take the data processing rates into account. Thus, such policies are insufficient to determine which files to cache. Indeed, as shown in Figure 2b, we observe in practice a low correlation (pearson correlation coefficient = 0.018) between the number of jobs that read a file (x-axis) and the rate at which it is processed (y-axis).

Data can be prefetched before job execution. We find that the period between data creation and the earliest job execution using the data varies from a few minutes to several hours. If the data is *prefetched* to local storage before the dependent jobs start, these jobs will benefit from the higher throughput and lower variability of local storage.

To quantify such opportunities, we define the notion of "prefetch slackness" of a file as the ratio between (a) the time elapsed since file creation to when it is first accessed, and (b) the time required to fetch the file from remote storage. While the former is a characteristic of the workload, the latter depends on bandwidth available to transfer the file. We measure the prefetch slackness of files in the examined workload (Figure 2c) using bandwidth values of 480Mbps and 1Gbps per VM, based on the measured average throughputs for Azure Blob Store and Amazon S3, respectively (see §2.1). We observe that 95% of the files have a prefetch slackness greater than one, i.e., they can be fully prefetched before being read by a job.[3]

2.3 An illustrative example

In this section, we illustrate how Netco differs from caching policies such as LRU; by considering job characteristics and prefetching inputs into the cache, Netco can perform much better.

Consider a workload with six jobs, $J_1, \ldots, J_6$ which process three files f_1, f_2, f_3 (Table 1). The jobs run on a compute cluster separated

[2] We observe similar results for data sizes of 64MB, 128MB, and 256MB; HDFS-like filesystems typically use such block sizes.

[3] Note that this under-estimates prefetching opportunities as job start can also be delayed till input is prefetched; once input is in the cache the job can execute more quickly and finish within its deadline.

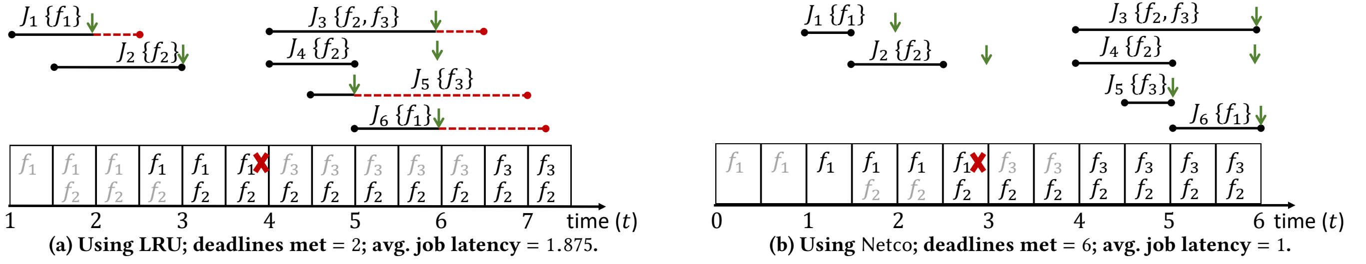


Figure 3: Execution time-lapse of workload in Table 1. A running job is shown using solid (black) lines and job deadlines are indicated by green arrows. If a job misses its deadline, the execution after the deadline is shown by dashed (red) lines. The tables indicate which files are present in the cache; full files are shown in black and partial files are shown in grey. A (red) cross indicates files being evicted from the cache. Files not in the cache on job start are read from the remote store and can be cached.

Jobs	Start	Deadline	Inputs, max. I/O rate
J_1	1	2	$\{f_1, 2.0\}$
J_2	1.5	3	$\{f_2, 1.0\}$
J_3	4	6	$\{\{f_2, f_3\}, 1.0\}$
J_4	4	6	$\{f_2, 1.0\}$
J_5	4.5	5	$\{f_3, 2.0\}$
J_6	5	6	$\{f_1, 1.0\}$

Table 1: Workload with 6 jobs and 3 files. All files are unit size.

from the store containing the files. Assume that all files have unit size, the cache tier has two units of capacity, the network between the store and compute has unit bandwidth, and the I/O bandwidth from the local cache is three units.

Figure 3a shows a time-lapse of job execution when the cache is managed using LRU; recall that in LRU, every cache-miss is added to cache by evicting, if necessary, the least recently used file. In this example, for simplicity, we assume that the interconnect bandwidth is divided equally across all jobs running at any point in time. Similar examples exist for other methods to share the I/O bandwidth. When J_2 starts at $t = 1.5$, the I/O bandwidth to the store is shared equally between J_1 and J_2 causing J_1 to miss its deadline; note that J_1 reads half of f_1 in $[1.0, 1.5]$ but the other half takes a full unit as it shares the I/O bandwidth. When J_3, J_4 start at $t = 4$, f_1 is evicted to make room for f_3. J_4 benefits from a cache-hit and finishes in one unit time; reading from the cache is faster but this job is limited by its own maximum processing rate of the file f_2. J_3 also benefits from cache hit on f_2 and would have finished at $t = 6$ because it takes two units of time to read f_3 from remote store. However, when J_5 and J_6 start, J_3's bandwidth to the store drops to a half and a third respectively causing J_3 to miss its deadline. J_5 and J_6 both suffer from cache misses and receive small shares of the I/O bandwidth to the store causing them to also miss their deadlines. In summary, four out of six jobs miss their deadlines, the average job latency is 1.875 and 5 units are read from the remote store.

Figure 3b shows a time-lapse to execute the same jobs using Netco. Netco decides to prefetch two files: f_1 because it is read at a high I/O rate by J_1 and f_3 because J_5 has a strict deadline. Even though f_2 is used by three different jobs, it is not prefetched as J_2 has a loose deadline. However, f_2 is cached after J_2 reads it from remote store (because f_2 is more useful than f_1 after $t = 3.5$). Note that both J_2 and J_6 finish faster even though neither benefits directly from the cache because Netco's actions ensure that more I/O bandwidth is available to them. Netco also ensures that both inputs are in the cache for J_3. In summary, all six jobs meet their deadlines, the average job latency is 1 and 4 units are read from the store; Netco improves on all of these metrics compared to LRU.

This example shows how Netco uses job characteristics to determine a cache and network-use schedule that lets more jobs meet their deadlines.

2.4 Takeaways

Our analysis above indicates that:

- Job and input characteristics are predictable before job submission, and can be used for network and storage resource planning.
- Files can be prefetched ahead of job execution allowing jobs to benefit from the higher throughput and predictability of reading from local storage.
- I/O management for analytics in disaggregated environments should consider both the bandwidth to the remote store and the capacity of local storage.

3 NETCO OVERVIEW

Netco focuses on deployments where (i) a compute cluster (e.g., Azure Compute [19]) executes multiple analytics jobs over (ii) input data that is stored in a separate store such as Azure Blob Storage [23] and (iii) a distributed filesystem manages the storage available on the compute nodes (e.g., local VM disks, memory, SSDs).

The key idea behind Netco is to use the characteristics of recurring jobs to *plan* how I/O resources should be allocated so that more jobs finish within deadlines. In particular, Netco explicitly manages (i) the I/O bandwidth available to the primary cloud storage (also referred to as *remote* store), and (ii) the storage capacity of the secondary storage volumes (referred to as *local* store or the *cache*). An optimal solution to this *planning problem* requires joint optimization across these two resources. This, in turn, necessitates decisions along multiple dimensions — for each (job, input file) pair determine if the file has to be cached, when and at what rate should the file be transferred from remote to local store, and when to evict it from the cache.

We model this optimization as a linear program. While we defer the details to §4, in this section, we describe the architecture of

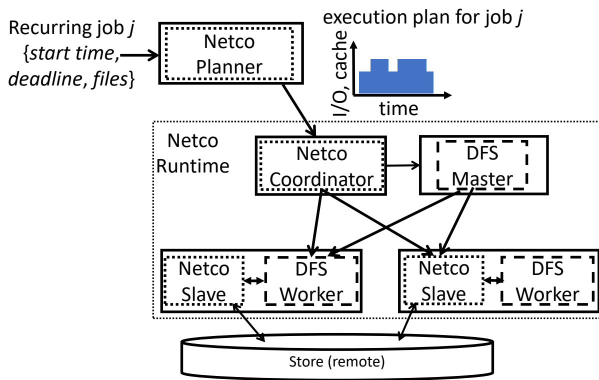


Figure 4: Netco **architecture**

Netco (§3.1), the design choices that result in a practically scalable optimization framework (§3.2) and, various deployment details (§3.3).

3.1 System architecture

Figure 4 illustrates the architecture of Netco which consists of a coordinator and a collection of slaves; this architecture is chosen to work well with existing distributed file systems (DFS) such as HDFS [16] and Alluxio [3]. As shown in the figure, the Netco coordinator is conjoined with the file system master and the Netco slaves serve as an intermediary between the DFS workers and the remote store. We describe each of these components below.

Planner. Recurring job arrivals, their deadlines and inputs are obtained from analyzing logs of previous job executions [54]. With this input, the planner determines a *cache and I/O resource assignment* for jobs using the algorithms in §4. As distributed file systems like HDFS divide files into a sequence of *blocks*, this plan specifies how each input block is processed by a job during its execution — whether (i) the block is prefetched before job start and if so, when the block should be prefetched or (ii) the block is to be read from the remote store during job execution, in which case it is specified whether the block should be cached. In either case, the plan also specifies the I/O rate to use to read/transfer the block and if the block is to be cached, it specifies which other block(s) to evict.

The planner runs at the start of every *planning window* (a configurable parameter, e.g., every hour) to plan for newly arriving jobs. It also maintains the expected state of the cluster — what files (or portions thereof) are cached locally and how much bandwidth to the remote store is assigned to individual file transfers at future times. If the deadline of a job cannot be satisfied, the job can either execute as a best effort job or the user may submit it at a later time. The planner can also be invoked, on demand, to handle changes in the cluster or workload.

Netco runtime. The runtime, as shown in Figure 4, consists of a cluster-wide coordinator and per-node slaves. The coordinator coordinates I/O and cache activities. To prefetch a file, the coordinator performs the necessary metadata operations with the file system master to ensure that file blocks can be cached. For example, in HDFS, this involves setting the replication factor for the file so that the master does not delete the newly cached blocks. Next, the coordinator issues `fetch` commands to individual workers (chosen at random) to fetch file blocks from remote store. The workers use the Netco slaves to read data from the remote store at the specified rate. The coordinator tracks the progress of prefetching and also handles evictions. Evicting a file requires metadata operations on the file system master and `evict` commands are issued to the workers to delete cached blocks.

3.2 Design choices

Modeling each possible I/O action (prefetch a file, demand-paging, . . .) at the granularity of file blocks results in an intractable optimization problem. Consequently, we make some design choices which lead us towards a scalable hierarchical optimization for the planning problem while also accounting for practical constraints imposed by big data filesystems.

Decouple demand paging decisions from the central optimization framework. A job can read files in multiple ways: (a) from the remote store, either without caching the data (*remote read*) or after caching it locally (*demand paging*), or (b) from the cache, if the files are prefetched into the cache before it starts (*prefetch read*) or cached by an earlier job (*cache hit*).

The various read methods interact in complex ways. For example if two contemporaneous jobs access the same file, each can remote-read half of the file and benefit from a cache hit on the other half. However, an optimization problem that considers such complex interactions becomes intractable. An earlier formulation that accounts for all the different kinds of reads was over 10× slower than the formulation described in this paper. Thus, to obtain a practically tractable solution, we trade-off some accuracy for much better performance – our optimization problem ignores demand paging and only models prefetch, cache-hits (due to prefetches) and remote reads. A later *cache augmentation phase* (§4.3) is used to take advantage of demand paging opportunities.

Plan at the granularity of files. Analytics frameworks store files as a sequence of *blocks*, and jobs consist of tasks which read one or more blocks. Hence, planning at block granularity is useful. For example, because even when a file is not fully available in the cache many of its blocks may be in the cache. However, this results in trillions of variables and constraints making the optimization intractable at scale (Table 2 offers some typical problem sizes). Hence, our planner only optimizes at the granularity of jobs and files but the Netco runtime greedily avails of additional cache hit opportunities.

Translating a file-level plan to a block-level plan. One simple translation would be to assign to each block $1/n^{\text{th}}$ of the rate assigned to the file if the file has n blocks. However, our formulation (§4.2) allocates time-varying I/O rates to files which will translate into a time-varying rate for each block. Enforcing a time-varying rate requires tight coordination across the machines that work on each block. To circumvent this complexity, we enforce a fixed but different rate for each block in the file; these rates are computed by fitting as many rectangles as the number of blocks into the "skyline" of file's allocation (height at time t is the rate assigned to the file at time t) (see §4.3).

3.3 Deployment considerations

Replica placement on local storage, and task placement. Netco only considers what to cache and how to transfer data from

remote stores to the cache but does not model replica placement; that is, which machines contain each block. Replica placement is an involved problem in its own right because it has to account for load balance, robustness to machine faults etc. Our implementation (§5) uses the default replica placement policy in HDFS [16] and uses the locality-aware scheduler in Yarn [8] for task placement. Better replica placement policies (e.g., Corral [53]) and task placement (e.g., Tetris [48]) can lead to better results.

Ad hoc jobs. While a large fraction (40–75%) of the workload in production clusters is recurring and known in advance [47, 54], big data clusters also run ad hoc jobs. Such ad hoc jobs can compete with SLO jobs for compute, cache and network resources. To protect the SLO jobs from such interference by ad hoc jobs, Netco runs ad hoc jobs at lower priority; several frameworks support priority scheduling (e.g., Yarn [8], Mesos [49]). Further, Netco prevents ad-hoc jobs from evicting data that is cached for SLO jobs.

Prediction errors. Netco relies on the ability to predict submission times of jobs and the time their input files are available using techniques such as the ones used in Morpheus [54]. When the runtime behavior of a job diverges significantly from Netco's plan (e.g., a file is not available for prefetch when expected), Netco executes the job using existing techniques; for example, caching the job's input on-demand using PACMAN [28]. Dynamically adapting Netco's plan to meet job SLOs with such runtime deviations is left for future work. Even without such dynamic plan adaptation, our experiments indicate that Netco is fairly robust to runtime variations under typical conditions (§6.3).

Exogenous concerns. Netco does not consider the problems of capacity planning or auto-scaling resource reservations with cluster load; prior work on these problems [54, 60] can work in conjunction with Netco. Furthermore, Netco only considers I/O reads but not writes; writes can be accommodated by setting aside a portion of the I/O bandwidth and using techniques such as Sinbad [39] or by specifying some time-varying write rate in the Netco planner. We leave further investigation to future work.

4 ALGORITHM DESIGN

In this section, we first formulate the algorithmic setting for Netco (§4.1). We then develop a unified Linear Programming (LP) optimization framework that allows Netco to plan for the I/O resource allocation to meet end-to-end job level goals. Finally, we describe the lower-level mechanisms that translate the solution of the LP into an efficient and practical execution plan (§4.3).

4.1 Preliminaries

We next formulate an offline planning problem, in which the algorithm has full information about all jobs submitted and all files required within the planning window T. The input to the problem consists of a set of N jobs $j = 1, \ldots, N$ and L files $\ell = 1, \ldots, L$. All files are stored in the remote store (to start off) and the jobs are run in a separate compute cluster. Each job j has a submission time a_j and deadline d_j. Each file ℓ has size s_ℓ, and is required by a subset of jobs J_ℓ. We also denote by F_j the subset of files required by job j. The local store has fixed capacity C. The maximum bandwidth available to transfer data from the remote storage to the local store is B – this limit can be enforced by the remote storage itself [23] or can be because of the limits on the (virtual) network cards of the compute instances.

We model two ways in which files can be read:

Prefetch read. If file ℓ (or parts of it) is prefetched and cached in the local store, then all jobs in J_ℓ that start after the prefetch can access it. We assume that prefetching can be done with no rate restrictions, i.e., a file can be prefetched using any amount of available network bandwidth (the total bandwidth used should be less than B). Further, to fully benefit from prefetching, we require that all file data is cached before job start. We also assume that a cached file ℓ cannot be evicted during time window $[a_j, d_j]$ if there is a job j that requires ℓ. While these assumptions might affect the quality of the solution (e.g., parts of a file that are processed can be evicted to free up cache space), they allow us to formulate a tractable optimization problem. Overall, Netco still significantly outperforms state-of-the-art techniques as shown in our evaluation (§6).

Remote read. If the file (or parts of it) is read from the remote store, each job j has to read the file separately. Due to practical restrictions (§3.2) and simplicity of implementation, we require that remote read take place at a fixed rate $r_{j,\ell}$, determined by the solution.

Objectives. We consider two variants of our problem, corresponding to different load regimes:
(i) Light/medium load regime, where there is enough network bandwidth to accommodate all job deadlines. The objective is to minimize peak bandwidth utilization while meeting all job deadlines. This objective also allows us to free up the network for unplanned/ad hoc jobs.
(ii) High load regime, where all production jobs may not finish by their deadline. Hence, the objective is to maximize the number of jobs that meet their deadlines given a fixed network bandwidth.

4.2 Linear programming formulation

In this section, we describe a unified linear programming formulation that is used for the two objective functions described above. We use the following variables:

- $r_{j,\ell}$: the rate at which file ℓ is read by job j from remote (as remote read).
- $C_{\ell,t}$: number of bytes of file ℓ in cache at time t
- $X_{\ell,t}$: number of bytes of file ℓ prefetched to cache at time t
- B: available network bandwidth.

The LP includes the following constraints:

(1) $r_{j,\ell}(d_j - a_j) + C_{\ell,a_j} \geq s_\ell, \forall j, \ell.$
(all data is read, either from cache or remotely.)

(2) $C_{\ell,t} \leq C_{\ell,t-1} + X_{\ell,t}, \forall t, \ell.$
(caching requires prefetching.)

(3) $C_{\ell,t+1} \geq C_{\ell,t},\ \forall t \in [a_j, d_j], \forall \ell \in F_j, j.$
(prevent cache evictions while job j is running).

(4) $\sum_\ell C_{\ell,t} \leq C, \forall t.$
(cache capacity cannot be exceeded.)

(5) $\sum_{j|t\in[a_j,d_j]} \sum_\ell \left(r_{j,\ell} + X_{\ell,t}\right) \leq B, \forall t.$
(bandwidth used cannot exceed the capacity B.)

A solution to the above linear program provides a prefetching plan of files to the cache. We note that it allows only part of a file to be prefetched to the cache and the rest to be read from the remote

store. By the above assumptions, a (part of) file ℓ read from the remote store defines a *rectangle* whose base is the window $[a_j, d_j]$ and its height is the rate $r_{j,\ell}$ at which the file is read.

Bandwidth minimization. Under this scenario, the network bandwidth B is a variable, and the objective is to minimize B under the above constraints. A solution to this LP can be used as an execution plan, as the files of every job are fully read, either from the cache or the remote store (§4.3).

Maximizing number of jobs satisfied. In high load scenarios, our objective is to maximize the number of jobs that are fully satisfied, when the network bandwidth B is fixed. This problem can be shown to be NP-hard by reducing the *densest-k-subgraph* problem [32, 46] to it (proof in Appendix). The densest-k-subgraph problem is NP-hard and its approximability has remained wide open despite many efforts [32, 59].

We formulate a mixed integer linear program (MILP) to maximize number of jobs satisfied. First, constraints 2–5 described above also apply here. For each job j, we introduce a new binary variable p_j such that $p_j = 1$ iff job j is fully satisfied (i.e., its input files are read completely). Formally, this requirement is captured through the constraint: $p_j \leq \frac{r_{j,\ell}(d_j - a_j) + C_{\ell,a_j}}{s_\ell}$ for every j, ℓ.

The objective now is to maximize $\sum_j p_j$. In our experiments, we find that this MILP is not scalable for problem instances of around thousand jobs (or more). Consequently, we use a solution which is based on the following relaxation of the problem: p_j as a continuous variable between zero and one, i.e., p_j now stands for the *fraction* of job j executed. This results in a LP with the same objective and constraints as the above MILP.

However, the fractional solution obtained from this LP for maximizing number of jobs does not directly yield an execution plan as job j will not process its files fully when $0 < p_j < 1$. Thus, our goal now is to translate this fractional solution into a solution of the MILP above (jobs either execute fully or not at all). For this purpose, we use the following rounding procedure:

Randomized rounding procedure. The execution plan is divided between prefetching files to the cache and reading files from the remote store. We follow the prefetch plan for files as given by the fractional solution. The remaining parts of the files may not be fully transferred from the remote store. Hence, we need a procedure for choosing which content should be transferred from the remote store. To that end, we use a procedure called *randomized rounding* [62] to the remote reading of files. Intuitively, the idea here is to pick files with probability which is proportional to their value in the fractional LP, while ensuring that the channel capacity is not violated (with high probability).

Consider a job j; in the fractional solution each file ℓ read by j corresponds to a rectangle whose basis is $[a_j, d_j]$ and height is $r_{j,\ell}$. We can aggregate all such rectangles into a single rectangle of height $h_j = \sum_\ell r_{j,\ell}$. Define $p'_j = \frac{h_j(d_j - a_j)}{\sum_{\ell \in j}(s_\ell - C_{\ell,a_j})}$; p'_j is the fraction of the contents of files read by job j from the remote store, ignoring the cache contribution. We now apply randomized rounding to the remote reading of files. Independently, for each job j, allocate a rectangle of height $(\sum_{\ell \in j}(s_\ell - C_{\ell,a_j}))/(d_j - a_j)$ with probability p'_j. It follows from the work of [35] that the probability of deviating from the network capacity is small, as a result of this randomized rounding procedure. This can be proved under the assumption that for each job j, $\sum_{\ell \in j}(s_\ell - C_{\ell,a_j})$ is not too large relative to B. Unfortunately, one cannot prove any approximation factors for this procedure, since p_j and p'_j cannot be related.

4.3 Determining an execution plan

The pseudo-code below describes a simple mechanism, which supplements our LP solution. The mechanism reclaims some of the lost opportunities due to our design choices (§3.2), and outputs a practical execution plan.

```
function TRANSLATE(start time s, end time T, rates X[], blocks b[])
    i ← min{t | t ≥ s, X_{ℓ,t} > 0 ∨ T}                      ▷ Start of interval
    if i = T then
        return ∅                                               ▷ No capacity in interval
    j ← min{t | t > i, X_{ℓ,t} = 0}                            ▷ End of interval
    r ← min{X_{ℓ,k} | ∀i ≤ k < j}                              ▷ Least common rate to interval
    a ← {(b_x, r) | x = 1, . . . , ⌈r(j − i)/blocksize⌉}       ▷ Assign x blocks at rate r
    b′ ← {b_x | b_x ∈ b, b_x ∉ a}                              ▷ Remaining blocks
    X′ ← [X_i − r | i = 1, . . . , |X|]                        ▷ Remaining bandwidth
    a ← a ∪ TRANSLATE(i, j, X′, b′)                            ▷ Recursively assign remaining
    b′ ← {b_x | b_x ∈ b, b_x ∉ a}                              ▷ Remaining blocks
    X′ ← [X_i − r | i = 1, . . . , |X|]                        ▷ Remaining bandwidth
    return a ∪ TRANSLATE(j, T, X, b′)                          ▷ Assign capacity after this interval
```

Algorithm 1: Assign transfers for file ℓ

Cache augmentation. The optimization framework discussed in §4.2 forgoes opportunities for demand paging, preferring a simple and scalable formulation. However, caching data read directly from the remote store can further reduce the remote I/O. To exploit such opportunities, we leverage the cache space that is not consumed by prefetched data. Thus, the *usable* cache space is given by $\bar{C}_t = C - \sum_\ell C_{\ell,t}$ at any time t. We use a pluggable caching policy to manage this space and cache the data read remotely by each job j, when possible. In our experiments (§6), we used Belady's MIN as the caching policy as we can estimate which files are going to be used furthest in the future. Other caching policies like PACMan [28] can also be implemented.

Translate file-level plan to block-level plan. The solution of the LP determines a file-level plan. However, as discussed earlier, most distributed file systems store files as a sequence of blocks. Thus, a file-level plans needs to be translated into a block-level plan to be practical. This involves translating the following components:

(a) *Cache state.* Suppose the block size of file ℓ is b_ℓ bytes. $C_{\ell,t}$ gives the size of file ℓ in the cache at time t. This translates to the first $\lfloor C_{\ell,t}/b_\ell \rfloor$ blocks of file ℓ being cached at time t. If this value decreases at any time t compared to $t-1$, the corresponding number of blocks should be evicted from the cache. If it increases, then blocks of file ℓ will be added to the cache using the network transfers described next.

(b) *Prefetch network transfers.* A solution of the LP formulation assigns a time-varying rate, $X_{\ell,t}$, to the file ℓ at time t. Thus, its transfer is described by the time series given by $\{X_{\ell,t} | t \in [0, T]\}$ where $X_{\ell,T} = 0$. Suppose file ℓ is fetched into the cache only once. We use Algorithm 1 to translate its time series into a collection of rectangles, where each rectangle corresponds to the transfer of a single block at a constant rate (as mentioned in §3.2). If file ℓ is fetched multiple times, we repeat Algorithm 1 for each time it is fetched.

5 NETCO IMPLEMENTATION

We implement Netco by extending Apache Hadoop/HDFS [8], a widely used data analytics platform[4]. With our changes, HDFS can serve as a cache for cloud stores like Amazon S3 [6], and Azure Blob Store [23]. Users can (a) seamlessly access data in the remote store through HDFS, (b) prefetch data into the local HDFS storage for future jobs at a specified rate, and (c) cache data in local storage as needed. Below, we first provide a brief overview of HDFS and then describe our implementation.

HDFS overview. HDFS exports a hierarchical namespace through a central *NameNode*. Files are managed as a sequence of *blocks*. Each block can have multiple *replicas* stored on a cluster of *DataNode* servers. Each DataNode is configured with a set of attached *storage* devices, each associated with a *storage type*. The storage type is used to identify the type of the storage device — existing types are DISK, SSD, or RAMdisk. When a DataNode starts up, it reports the list of replicas stored on each of its storage devices (and hence, storage type). From these reports, the NameNode constructs a mapping between the blocks and the storage devices on all the DataNodes that contain a replica for the block.

For each file in the namespace, the NameNode also records a *replication factor* specifying the target number of replicas for each block in the file, and a *storage policy* that defines the type of storage in which each replica should be stored. If the number of available replicas for a block are below its expected replication factor, the NameNode schedules the necessary replications.

Modifications to HDFS. Implementing Netco in HDFS required two major extensions. The following description elides engineering details (can be found on JIRA [2]) to focus on conceptual implementation changes.

PROVIDED storage. We add a new storage type called PROVIDED to HDFS to identify data external to HDFS and stored in remote stores. Both the NameNodes and DataNodes are modified to understand the PROVIDED storage type. To address data in the remote store, the remote namespace (or portion thereof) is first *mounted* as a subtree in the NameNode. This is just a metadata operation, and is done by mirroring remote files in the HDFS namespace and configuring a replica of PROVIDED storage type for each block in the namespace.

Subsequently, when any DataNode configured with PROVIDED storage reports to the NameNode, it considers all PROVIDED replicas reachable from that DataNode. Any request to read data from the remote store has to pass through a DataNode with PROVIDED storage type. As data is streamed back to the client, the DataNode can cache a copy of it in the local storage.

Metered block transfers. We add a throttling parameter to block transfer requests, allowing us to control the rate at which block data is sent. The throttling is implemented using token buckets. With this, client read requests and block replications can be limited to a target rate. When blocks of a file are replicated together, concurrent transfers do not exceed the target rate.

Netco realization. The Netco planner and coordinator are implemented as standalone components (run on the same machine as the HDFS NameNode in our experiments). PROVIDED storage devices configured in DataNodes serve as a Netco slave.

Using the above modifications to HDFS, the Netco coordinator can prefetch remote files (into local storage) by adjusting their storage policy and scheduling block replications at the rate determined by the execution plan. The Netco slaves ensure that when jobs read data from the remote store, it is transferred at the rate specified by the execution plan. For cache evictions, the coordinator evicts replicas from local storage by lowering the replication of a file. For example, lowering replication to 1 causes HDFS to delete all replicas but the PROVIDED replica.

[4]We are contributing back our changes to Apache HDFS as part of HDFS-9806 [2] (merged), HDFS-12090 [17] (in-progress), and HDFS-13069 [14] (in-progress).

Workload	Jobs	Data processed	Files
B1	9k	720TB	30k
B2	4k	66TB	12k
B3	20k	170TB	66k
B4	1k	200TB	3k

Table 2: Characteristics of workloads from different business units using Microsoft Cosmos (rounded to nearest thousand).

6 EVALUATION

We evaluate Netco on a 50 node cluster on Microsoft Azure [19], a 280 node bare metal cluster, and using large-scale simulations. All experiments are based on workload traces from a production analytics cluster at Microsoft, running on thousands of machines. Compared to various baselines representing state-of-the-art in cloud data analytics, Netco:

- Reduces peak utilization and total data transferred from the remote store to the compute cluster by up to 5×. This, in turn, reduces I/O cost per SLO attained by 1.5×–7×.
- Increases the number of jobs that meet their deadlines up to 80%, under high-load.
- Efficiently allocates I/O resources for the SLO jobs which allows ad hoc jobs to run 20%–68% faster.

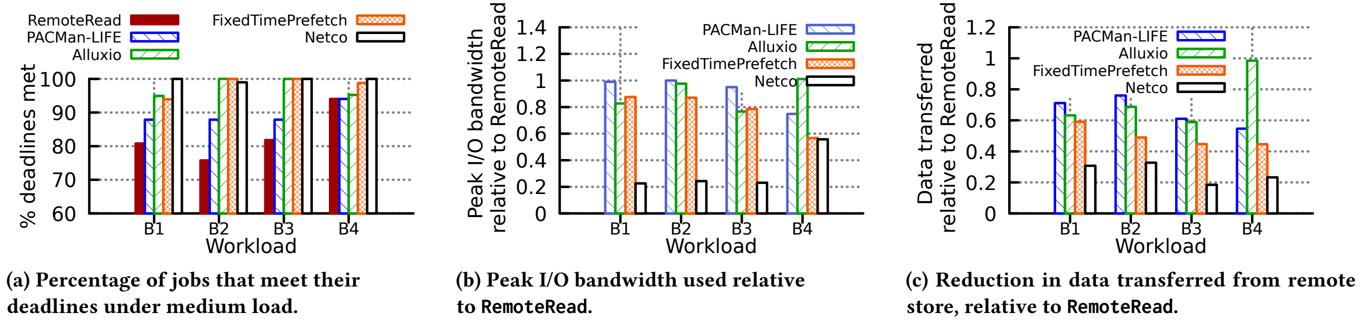
6.1 Methodology

Experimental setup. We deploy our implementation of Netco in two different environments:

50-node VM cluster on Microsoft Azure: We run HDFS with our modifications (§5) on a cluster of 50 Standard_D8s_v3 VMs [21], using YARN as the resource management framework. HDFS DataNodes are configured with the SSD-based local disk (acts as cache), and the Azure Blob Storage as PROVIDED storage. Input files are stored on Azure Blob Storage.

280-node bare metal cluster: We evaluate Netco at larger scale using this cluster. The cluster consists of 7 racks with 40 machines each. Each machine has an Intel Xeon E5 processor, 10Gbps NIC, 128GB RAM and 10 HDDs. Six of the racks are used for compute and one is used for storage. Bandwidth between the two is limited to 60Gbps to emulate a cloud environment. The storage tier runs stock HDFS and stores the input data for the workloads. The compute tier runs HDFS with our modifications.

Workloads. Our workloads are based on a day-long trace from Microsoft Cosmos [33] (Table 2). We consider workloads from 4 different business units (B1, B2, B3 and B4), and scale them down to fit our cluster setup. Job SLOs are derived using techniques from



(a) Percentage of jobs that meet their deadlines under medium load.

(b) Peak I/O bandwidth used relative to `RemoteRead`.

(c) Reduction in data transferred from remote store, relative to `RemoteRead`.

Figure 5: Benefits of running SLO workloads with Netco **on Azure.**

Morpheus [54]. Based on these workloads, we generate a job trace lasting one hour and run them using Gridmix [7].

Metrics. We use the following metrics to measure the benefits of Netco under medium to high load scenarios: (i) peak network bandwidth (averaged over 2 seconds), (ii) total data transferred from remote store and (iii) number of SLO jobs that are admitted, and meet their deadlines. We note that the total data transferred from the remote storage is directly proportional to the cost of I/O (in dollars) to cloud users [6, 23]. Thus, any reduction in this metric reduces the I/O cost for a workload. We also evaluate the scalability of Netco's planning algorithm (§4), and its solution quality by comparing against a Mixed Integer Linear Program (MILP).

Baselines. We compare Netco against the following baselines, which represent how typical data analytics workloads run in public clouds today [12, 20, 22].

(1) `RemoteRead`, in which all workloads read and write data directly from the remote storage.

(2) `PACMan-LIFE`, where data is paged in on demand and cached in local VM storage. We use the PACMan-LIFE algorithm to manage the cache as it outperforms traditional caching algorithms for data analytics [28].

(3) Alluxio [3], an open-source filesystem that allows applications to cache data, from *underlying filesystems*, in locally available storage. In our experiments, we configured Alluxio to use Azure Blob Storage as the underlying filesystem, the local SSD-based storage in the VMs as cache and an LRU eviction policy.

(4) `FixedTimePrefetch`, builds on `PACMan-LIFE` and starts fetching job input files T time units before the job starts (if absent from local cache). Files are fetched at a constant rate and are cached by job start. Comparison with `FixedTimePrefetch` shows how careful planning in Netco compares with a simple prefetching scheme. We set T to be 15 minutes in our experiments.

Netco generates an execution plan using the planning algorithms described in §4 and enforces it using our implementation (§5). We use Gurobi [15] to solve the linear programs. `PACMan-LIFE` and `FixedTimePrefetch` use the same implementation as Netco but their respective caching, and prefetch policies.

6.2 Benefits with Netco

Deployment on Microsoft Azure. We evaluate Netco on Azure under two different load regimes.

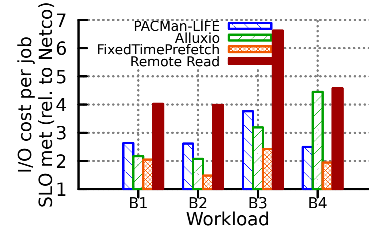
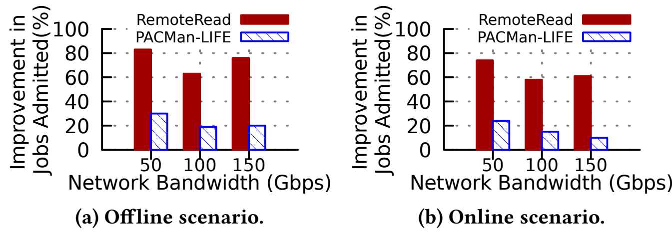


Figure 6: Average I/O cost to meet a job deadline for various baselines, relative to Netco**.**

Medium load regime. When the job arrival rate is low, we expect all jobs to meet their deadline SLOs. In particular, in our experiments, we ensure that the available I/O bandwidth to Azure Blob Storage is sufficient to meet all job deadlines even when each job reads directly from it. However, in practice, due to variance in the I/O bandwidth to the blob store (§2), jobs can miss their deadlines. In particular, we find that `RemoteRead` misses the deadlines of up to 25% jobs (Figure 5a). Using reactive caching techniques such as `PACMan-LIFE` and Alluxio still results in 5–10% jobs missing their SLOs. With its careful planning, Netco eliminates almost all deadline misses (except in the case of workload B2 where fewer than 1% of jobs miss their deadlines).

Figures 5b and 5c show (a) peak bandwidth used, and (b) total data transferred from azure blob storage relative to `RemoteRead`. Our observations are three-fold. First, while caching (`PACMan-LIFE` and Alluxio) appreciably reduces the total amount of data transferred from remote storage compared by `RemoteRead` (by 23 – 45%), it has limited impact on the peak I/O bandwidth used — just 2 – 6% reduction on average over all workloads with a maximum of 25% reduction for workload B3. This is a result of files being read directly from the remote store when first accessed, at the rate it is processed by the dependent job(s). This phenomenon is amplified if multiple jobs read the same file concurrently, which was observed in our workloads. Similar behavior has been reported earlier in Scarlett [26], and is expected as most jobs are submitted at the start of an hour [54].

Second, we find that these limitations with caching algorithms can be overcome by prefetching files — `FixedTimePrefetch` reduces the peak I/O bandwidth and total data transferred by 13–43% and 41–51% compared to `RemoteRead`. Finally, using Netco



(a) Offline scenario.

(b) Online scenario.

Figure 7: Increase in jobs that meet their SLOs with Netco.

results in even more improvements – 44–77% reduction in peak I/O bandwidth and 63–81% reduction in data transferred compared to RemoteRead (i.e., up to 5× reduction). Such significant reductions are a result of Netco's use of job and file-access characteristics to carefully plan data prefetch and cache occupancy. Netco prioritizes prefetch of files accessed by larger number jobs, and fetches them at a sufficient rate.

As cloud providers charge users for data transferred from the storage to the compute tiers [6, 23], Netco also helps reduce the I/O cost per job SLO met — as shown in Figure 6, we see a 4–7× reduction in cost compared to RemoteRead, 2–4× relative to PACMan-LIFE and Alluxio, and 1.5–2.5× compared to the FixedTimePrefetch.

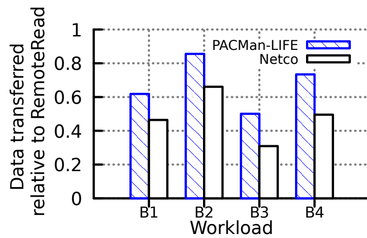


Figure 8: Data transferred from storage to compute (relative to RemoteRead) in 280 node deployment.

High load regime. Under high load when it may not be possible to meet deadlines of all SLO jobs, Netco aims to maximize the number of jobs that meet their deadlines. We compare Netco's planning algorithm (§4) with planning algorithms that use (a) RemoteRead, and (b) demand paging with PACMan-LIFE as the eviction policy, to run jobs. Figure 7 shows the increase in number of jobs admitted by Netco compared to these strategies in (a) an offline scenario, where we assume all jobs are known ahead of time, and (b) an online scenario, where jobs are planned for as they are submitted. Jobs are derived from workload B1, job inter-arrival duration is decreased by a random factor between 2–5×, and planning algorithms are run with different bandwidth limits between the storage and compute.

While Netco completes fewer jobs in the online scenario compared to offline, the difference is small. Overall, Netco accepts up to 80% more jobs than RemoteRead and 10–30% more jobs than demand paging. This increase is a result of (a) the reduction in the peak network bandwidth with prefetching, allowing more jobs to be admitted, and (b) efficient use of the cache and network resources by planning ahead of job submissions.

Deployment on 280-node bare-metal cluster. The results here are qualitatively similar to those observed in the above experiments. As shown in Figure 8, Netco results in up to 70% less data transferred relative to RemoteRead. While caching helps PACMan-LIFE (up to 50% less data transferred compared to RemoteRead), it reads 1.2–1.75× more data than Netco.

6.3 Performance of planning algorithms

Scalability of LP formulation. The scalability of Netco's linear program (Section 4) depends on the number of jobs to plan for, number of files accessed by each job, and the duration of the plan. Figure 9 shows that the LP only takes a few minutes to execute as we increase the number of jobs planned for. For this experiment, the workload lasted for one hour, and the average number of files accessed per job varied between 4.5 to 6.7. As this planning is done ahead of job arrivals and is not in the critical path of the jobs, this overhead is acceptable.

Comparison to a MILP-based lower bound. For better scalability, Netco solves a fractional linear program and approximates the MILP to maximize the number of jobs admitted (§4.2). This can lead to fewer admitted jobs than using the MILP. In practice, we find that this reduction is minimal — Netco is within 3% of the MILP.

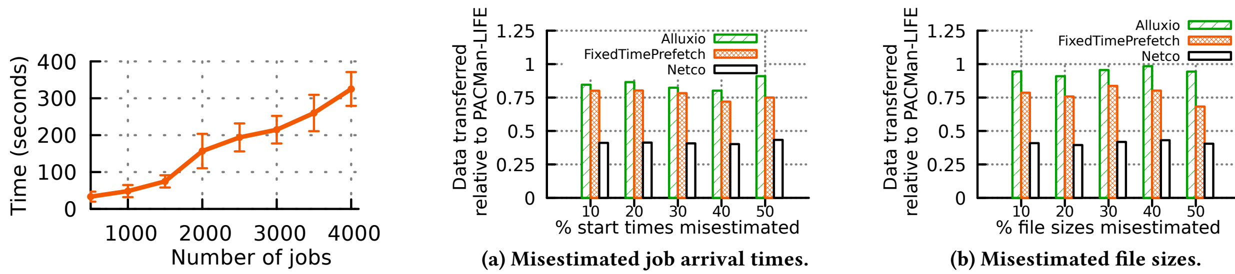
Sensitivity analysis. Netco relies on predictable job characteristics to determine an efficient execution plan. However, even state-of-the-art techniques to predict workload characteristics have prediction errors [54]. To understand how robust Netco is to this, we introduced errors in the following job characteristics for workload B1: (a) job submission times – A certain percentage of jobs are chosen at random, and their submission times are changed by a randomly chosen value between between −5 and 5 minutes (this is more than twice the average job inter-arrival duration), and (b) file sizes – the sizes of a certain percentage of files, chosen at random, are increased or decreased by up to 10%; this represents nearly 3× the typically prediction error [53]. As the percentage of error increases, the benefits of Netco reduce slightly compared to PACMan-LIFE but it performs significantly better than various baselines (Figure 10).

6.4 Benefits for ad hoc jobs

Data analytics clusters run ad hoc jobs along with SLO jobs for data exploration or research purposes. No guarantees are provided to the ad hoc jobs, but users expect them to finish quickly. While Netco does not schedule the ad hoc jobs, its efficient use of resources for SLO jobs allows ad hoc jobs to run faster.

To understand this effect, we perform trace-driven simulations with ad hoc jobs running alongside SLO jobs. The SLO jobs are derived from workloads B1 and B2. Ad hoc jobs are derived from two traces: (a) internal production cluster traces and (b) published traces from Facebook's data analytics clusters [36]. We will label these workloads A1 and A2, respectively. For the traces from Facebook, we randomly sample 40% of the jobs to be ad hoc jobs (this has been shown to be typical percentage of ad hoc jobs in clusters [53, 54]). In these simulations, resources are reserved for the SLO jobs to avoid interference from ad hoc jobs, and ad hoc jobs share the remaining resources based on max-min fairness.

Figure 11 shows the improvement in the runtime percentiles of ad hoc jobs, when the SLO jobs are scheduled with Netco and



(a) Misestimated job arrival times. (b) Misestimated file sizes.

Figure 9: Runtime of Netco's planning algorithm (averaged over 10 runs).

Figure 10: Data transferred from remote store (relative to `PACMan-LIFE`) when job characteristics are misestimated in for workload B1.

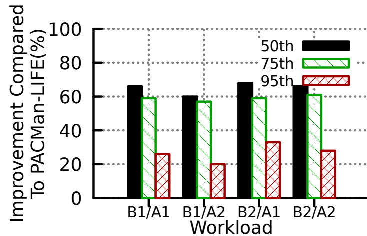


Figure 11: Reduction in ad hoc job runtimes for workloads with both SLO and ad hoc jobs.

`PACMan-LIFE`, for different workload combinations (workload Bi/Aj denotes SLO jobs drawn from Bi and ad hoc jobs from Aj). As Netco aims to reduce the network utilization of SLO jobs, it frees up the network resources for ad hoc jobs and significantly improves their runtimes — we observe up to 68% improvement in the 50th percentile.

7 RELATED WORK

Caching and prefetching. Practical [38, 61, 63] and theoretical [25, 30, 34, 51] treatments of general-purpose caching and prefetching techniques are ubiquitous [31, 43, 65, 66, 69, 71]. Big data workloads apply caching techniques to improve locality for applications' working set [26, 28] and to share expensive storage media in multi-tenant workloads [67, 68]. Correlations between datasets are also mined for prefetch heuristics in block stores [74] and cloud storage gateways [75]. In contrast, Netco targets recurring workloads, optimizing for job-level metrics (e.g., deadlines). Netco not only schedules the necessary transfers to cache data, it also performs admission control under high load.

Scheduling network flows. Scheduling network transfers to guarantee deadlines, or increase network utilization has been extensively explored for datacenters [42, 45] and wide-area networks [52, 56, 76]. Recent work also aims to improve end-to-end application runtimes using smart replica placement [53] and the coflow abstraction [40–42, 77]. However, these works do not consider the cache optimization problem tackled by Netco.

Storage systems. Similar to Netco, Alluxio [3, 58] transparently caches data from remote file systems in local storage. CAST [37] and OctopusFS [55] optimize data placement across media tiers to achieve performance and fault tolerance objectives. Systems such as IOFlow [70] focus on the mechanism enforcing fine-grained bandwidth guarantees. These systems are complementary to Netco, which generates an explicit local storage and network I/O schedule for satisfying workload SLOs.

Resource management. Explicit I/O planning in Netco is orthogonal to straggler mitigation strategies [27, 29] in data analytics frameworks. Our work assumes that compute resources may be provisioned either reactively [47] or proactively [44, 54] to meet job SLOs. In principle, these techniques may be combined with Netco to plan for storage, compute, and network resources for recurring workloads. We leave this direction for future work.

Provisioning cloud resources. Sizing cloud resources to meet application, resource, and budget objectives has been widely explored [72, 73, 78]. Netco complements these techniques by performing admission control and schedules I/O for analytics workloads in a given cluster.

8 CONCLUSION

We have designed and implemented Netco which maintains one or more caching tiers to provide predictable data access for analytics jobs over disaggregated stores. Netco has several ideas that can be used individually or together. When workload is predictable, Netco prefills the cache. When job characteristics such as deadlines and I/O rates are available, Netco can tune what it caches and how it allocates the I/O rate so as to let more jobs meet their deadlines. Doing so preferentially caches files used by jobs with tighter deadlines and files processed by jobs that read with high I/O rate. The overall problem, jointly allocating network and cache resources in order to meet job SLOs and/or minimizing the network bandwidth used is of interest primarily in our simplifications; we make the optimization tractable by ignoring carefully chosen aspects of the problem. Our implementation of Netco is open sourced with Apache Hadoop. Our evaluations show promising results (up to 80% more jobs meet their SLOs while using up to 5× less I/O bandwidth).

9 APPENDIX

Proof sketch for NP-hardness of maximizing number of jobs satisfied. The NP-hardness proof follows by reducing the *densest-k-subgraph* problem [32, 46] to a special case of our problem. The densest-k-subgraph problem is NP-hard by a reduction from max clique and its approximability remains an open question [32, 59].

Reduction: Suppose that the network capacity is very small and all jobs have the same window $[a, d]$, where a is far enough into

the future that there is only one time opportunity to transfer files to the cache before the jobs start running. Assume further that all files are of equal size (one unit) and the cache size is k. The goal is to bring in k files so as to satisfy as many jobs as possible. Given an undirected graph, the densest-k-subgraph problem asks for a subset of k vertices that contains the maximum number of edges. For an instance of the densest-k-subgraph problem we construct an instance of the above problem as follows: Given graph G, define for each vertex ℓ a file f_ℓ and for each edge e a job j_e. If edge e is adjacent to vertices ℓ and ℓ', then job j_e requires file f_ℓ and f'_ℓ. Finding a densest-k-subgraph in G is now equivalent to maximizing the number of jobs satisfied with cache size k.

REFERENCES

[1] Amazon Elastic Compute Cloud: Enhanced Networking on Linux. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/enhanced-networking.html.
[2] Allow HDFS block replicas to be provided by an external storage system. https://issues.apache.org/jira/browse/HDFS-9806.
[3] Alluxio - Open Source Memory Speed Virtual Distributed Storage. http://www.alluxio.org/.
[4] Amazon EC2. https://aws.amazon.com/ec2/.
[5] Amazon Elastic Block Store. https://aws.amazon.com/ebs/.
[6] Amazon S3. https://aws.amazon.com/s3/.
[7] Apache Gridmix. https://hadoop.apache.org/docs/r1.2.1/gridmix.html.
[8] Apache Hadoop. http://hadoop.apache.org/.
[9] Azure Data Lake Analytics. https://azure.microsoft.com/en-us/services/data-lake-analytics/.
[10] Azure Storage Scalability and Performance Targets. https://docs.microsoft.com/en-us/azure/storage/common/storage-scalability-targets.
[11] Best Practices for Amazon EMR. https://d0.awsstatic.com/whitepapers/aws-amazon-emr-best-practices.pdf.
[12] Cloudera Enterprise Reference Architecture for Azure Deployments. http://www.cloudera.com/documentation/other/reference-architecture/PDF/cloudera_ref_arch_azure.pdf.
[13] Databricks IO Cache. https://docs.databricks.com/user-guide/databricks-io-cache.html.
[14] Enable HDFS to cache data read from external storage systems. https://issues.apache.org/jira/browse/HDFS-13069.
[15] Gurobi Optimization. http://www.gurobi.com/.
[16] Hadoop Distributed File System. https://wiki.apache.org/hadoop/HDFS.
[17] Handling writes from HDFS to Provided storages. https://issues.apache.org/jira/browse/HDFS-12090.
[18] High-performance Premium Storage and managed disks for VMs. https://docs.microsoft.com/en-us/azure/virtual-machines/windows/premium-storage.
[19] Microsoft Azure. https://azure.microsoft.com/en-us/.
[20] Moving Data into HDFS from Amazon S3. http://documentation.altiscale.com/moving-data-from-s3-to-hdfs.
[21] Sizes for Windows virtual machines in Azure. https://docs.microsoft.com/en-us/azure/virtual-machines/windows/sizes.
[22] Use HDFS-compatible Azure Blob storage with Hadoop in HDInsight. https://docs.microsoft.com/en-us/azure/hdinsight/hdinsight-hadoop-use-blob-storage.
[23] Windows Azure Storage BLOB. https://azure.microsoft.com/en-us/services/storage/blobs/.
[24] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou. Re-optimizing data-parallel computing. In *NSDI*, 2012.
[25] S. Albers, S. Arora, and S. Khanna. Page replacement for general caching problems. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '99, Philadelphia, PA, USA, 1999. Society for Industrial and Applied Mathematics.
[26] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris. Scarlett: Coping with skewed content popularity in mapreduce clusters. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, New York, NY, USA, 2011. ACM.
[27] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective Straggler Mitigation: Attack of the Clones. In *NSDI*, 2013.
[28] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. Pacman: Coordinated memory caching for parallel jobs. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, Berkeley, CA, USA, 2012. USENIX Association.
[29] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the Outliers in Map-reduce Clusters Using Mantri. In *OSDI*, 2010.
[30] N. Bansal, N. Buchbinder, and J. S. Naor. A primal-dual randomized algorithm for weighted paging. *Journal of the ACM (JACM)*, 59(4):19, 2012.
[31] A. Bestavros. Using speculation to reduce server load and service time on the www. Technical report, Boston, MA, USA, 1995.
[32] A. Bhaskara, M. Charikar, E. Chlamtac, U. Feige, and A. Vijayaraghavan. Detecting high log-densities: an $O(n^{1/4})$ approximation for densest k-subgraph. In *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010, Cambridge, Massachusetts, USA, 5-8 June 2010*, pages 201–210, 2010.
[33] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou. Apollo: Scalable and Coordinated Scheduling for Cloud-scale Computing. In *OSDI*, 2014.
[34] M. Brehob, S. Wagner, E. Torng, and R. Enbody. Optimal replacement is np-hardfor nonstandard caches. *IEEE Trans. Comput.*, 53(1):73–76, Jan. 2004.
[35] G. Călinescu, A. Chakrabarti, H. J. Karloff, and Y. Rabani. An improved approximation algorithm for resource allocation. *ACM Trans. Algorithms*, 7(4):48:1–48:7, 2011.
[36] Y. Chen, S. Alspaugh, and R. Katz. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *Proc. VLDB Endow.*, 5(12):1802–1813, Aug. 2012.
[37] Y. Cheng, M. S. Iqbal, A. Gupta, and A. R. Butt. Cast: Tiering storage for data analytics in the cloud. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, New York, NY, USA, 2015. ACM.
[38] H.-T. Chou and D. J. DeWitt. An evaluation of buffer management strategies for relational database systems. In *Proceedings of the 11th International Conference on Very Large Data Bases - Volume 11*, VLDB '85. VLDB Endowment, 1985.
[39] M. Chowdhury et al. Leveraging Endpoint Flexibility in Data-Intensive Clusters. In *SIGCOMM*, 2013.
[40] M. Chowdhury and I. Stoica. Coflow: A networking abstraction for cluster applications. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, HotNets-XI, New York, NY, USA, 2012. ACM.
[41] M. Chowdhury and I. Stoica. Efficient coflow scheduling without prior knowledge. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, New York, NY, USA, 2015. ACM.
[42] M. Chowdhury, Y. Zhong, and I. Stoica. Efficient coflow scheduling with varys. In *ACM SIGCOMM 2014*.
[43] D. E. Culler, A. Gupta, and J. P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1997.
[44] C. Curino, D. E. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan, and S. Rao. Reservation-based scheduling: If you're late don't blame us! In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, New York, NY, USA, 2014. ACM.
[45] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron. Decentralized task-aware scheduling for data center networks. In *ACM SIGCOMM 2014*.
[46] U. Feige, G. Kortsarz, and D. Peleg. The dense k-subgraph problem. *Algorithmica*, 29(3):410–421, 2001.
[47] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: Guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, New York, NY, USA, 2012. ACM.
[48] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource Packing for Cluster Schedulers. In *SIGCOMM*, 2014.
[49] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *NSDI*, 2011.
[50] A. Iosup, N. Yigitbasi, and D. Epema. On the Performance Variability of Production Cloud Services. In *CCGRID*, 2011.
[51] S. Irani. Page replacement with multi-size pages and applications to web caching. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, New York, NY, USA, 1997. ACM.
[52] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a globally-deployed software defined wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, New York, NY, USA, 2013. ACM.
[53] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar. Network-aware scheduling for data-parallel jobs: Plan when you can. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, New York, NY, USA, 2015. ACM.
[54] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. n. Goiri, S. Krishnan, J. Kulkarni, and S. Rao. Morpheus: Towards automated slos for enterprise clusters. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, Berkeley, CA, USA, 2016. USENIX Association.
[55] E. Kakoulli and H. Herodotou. OctopusFS: A Distributed File System with Tiered Storage Management. In *SIGMOD Conference*, 2017.
[56] S. Kandula, I. Menache, R. Schwartz, and S. R. Babbula. Calendaring for wide area networks. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, New York, NY, USA, 2014. ACM.
[57] P. Leitner and J. Cito. Patterns in the chaos–a study of performance variation and predictability in public iaas clouds. *ACM Transactions on Internet Technology (TOIT)*, 16(3):15, 2016.
[58] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–15. ACM, 2014.
[59] P. Manurangsi. Almost-polynomial ratio eth-hardness of approximating densest k-subgraph. In *Proceedings of the 49th ACM Symposium on Theory of Computing, STOC 2017, Montreal, Quebec, Canada*.
[60] M. Mao and M. Humphrey. Auto-scaling to Minimize Cost and Meet Application Deadlines in Cloud Workflows. In *SC*, 2011.
[61] N. Megiddo and D. S. Modha. Arc: A self-tuning, low overhead replacement cache. In *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies*, FAST '03, Berkeley, CA, USA, 2003. USENIX Association.
[62] R. Motwani and P. Raghavan. *Randomized algorithms*. Chapman & Hall/CRC, 2010.
[63] V. Narasayya, I. Menache, M. Singh, F. Li, M. Syamala, and S. Chaudhuri. Sharing buffer pool memory in multi-tenant relational database-as-a-service. *Proceedings of the VLDB Endowment*, 8(7):726–737, 2015.
[64] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K Page Replacement Algorithm for Database Disk Buffering. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, New York, NY, USA, 1993. ACM.
[65] E. J. O'neil, P. E. O'neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. *ACM SIGMOD Record*, 22(2):297–306, 1993.
[66] V. N. Padmanabhan and J. C. Mogul. Using predictive prefetching to improve world wide web latency. *SIGCOMM Comput. Commun. Rev.*, 26(3):22–36, July 1996.
[67] Q. Pu, H. Li, M. Zaharia, A. Ghodsi, and I. Stoica. Fairride: Near-optimal, fair cache sharing. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, Berkeley, CA, USA, 2016. USENIX Association.
[68] K. V. Rashmi, M. Chowdhury, J. Kosaian, I. Stoica, and K. Ramchandran. EC-cache: Load-balanced, Low-latency Cluster Caching with Online Erasure Coding. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, 2016.
[69] A. S. Tanenbaum and H. Bos. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 4th edition, 2014.
[70] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu. Ioflow: A software-defined storage architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, New York, NY, USA, 2013. ACM.

[71] J. Wang. A survey of web caching schemes for the internet. *SIGCOMM Comput. Commun. Rev.*, 29(5), Oct. 1999.

[72] A. Wieder, P. Bhatotia, A. Post, and R. Rodrigues. Orchestrating the Deployment of Computations in the Cloud with Conductor. In *NSDI*, 2012.

[73] Z. Wu, C. Yu, and H. V. Madhyastha. CosTLO: Cost-effective Redundancy for Lower Latency Variance on Cloud Storage Services. In *NSDI*, 2015.

[74] J. Yang, R. Karimi, T. Sæmundsson, A. Wildani, and Y. Vigfusson. MITHRIL: Mining Sporadic Associations for Cache Prefetching. *CoRR*, abs/1705.07400, 2017.

[75] S. Yang, K. Srinivasan, K. Udayashankar, S. Krishnan, J. Feng, Y. Zhang, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Tombolo: Performance enhancements for cloud storage gateways. In *MSST*, 2016.

[76] H. Zhang, K. Chen, W. Bai, D. Han, C. Tian, H. Wang, H. Guan, and M. Zhang. Guaranteeing deadlines for inter-datacenter transfers. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. ACM.

[77] H. Zhang, L. Chen, B. Yi, K. Chen, M. Chowdhury, and Y. Geng. Coda: Toward automatically identifying and scheduling coflows in the dark. In *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference*, SIGCOMM '16, New York, NY, USA, 2016. ACM.

[78] T. Zou, R. Le Bras, M. V. Salles, A. Demers, and J. Gehrke. ClouDiA: a deployment advisor for public clouds. In *PVLDB'13*, 2013.