Optimizing Parallel Job Performance in Data-Intensive Clusters

Ganesh Ananthanarayanan

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2014-9
http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-9.html

January 26, 2014

Optimizing Parallel Job Performance in Data-Intensive Clusters

by

Ganesh Ananthanarayanan

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

Graduate Division
of the
University of California, Berkeley

Committee in charge:
Professor Ion Stoica, Chair
Professor Alexandre Bayen
Professor John Chuang
Professor Randy Katz
Professor Scott Shenker

Fall 2013

Optimizing Parallel Job Performance in Data-Intensive Clusters

Abstract

Optimizing Parallel Job Performance in Data-Intensive Clusters

by

Ganesh Ananthanarayanan
Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Ion Stoica, Chair

Extensive data analysis has become the enabler for diagnostics and decision making in many modern systems. These analyses have both competitive as well as social benefits. To cope with the deluge in data that is growing faster than Moores law, computation frameworks have resorted to massive parallelization of analytics jobs into many fine-grained tasks. These frameworks promised to provide efficient and fault-tolerant execution of these tasks. However, meeting this promise in clusters spanning hundreds of thousands of machines is challenging and a key departure from earlier work on parallel computing.

A simple but key aspect of parallel jobs is the all-or-nothing property: unless all tasks of a job are provided equal improvement, there is no speedup in the completion of the job. The all-or-nothing property is critical for the promise of efficient and fault-tolerant parallel computations on large clusters. Meeting this promise in clusters of these scales is challenging and a key departure from prior work on distributed systems. This talk will look at the execution of a job from first principles and propose techniques spanning the software stack of data analytics systems such that its tasks achieve homogeneous performance while overcoming the various heterogeneities.

To that end, we will propose techniques for (i) caching and cache replacement for parallel jobs, which outperforms even Belady's MIN (that uses an oracle), (ii) data locality, and (iii) straggler mitigation. Our analyses and evaluation are performed using workloads from Facebook and Bing production datacenters Along the way, we will also describe how we broke the myth of disk-locality's importance in datacenter computing.

To everyone who has believed in me, especially my family.

Contents

List of Figures

List of Tables

Acknowledgements

I am indebted to my advisor Ion Stoica for his sustained guidance and advise on my projects as well as the qualities aiding good research. I am also thankful to Scott Shenker for his valuable advice and encouragement during my Ph.D. The feedback I received from my committee members — Alexandre Bayen, John Chuang, and Randy Katz has also been important in presenting and positioning my work. A special thanks to Ali Ghodsi for being a solid collaborator and also highlighting many nuances in conceiving projects.

My colleagues in the RAD Lab and AMP Lab have been invaluable and fun, and I will forever be thankful to them for their inputs on my ideas, paper drafts and experiments. In no particular order, thanks to Sameer Agarwal, Mosharaf Chowdhury, David Zats, Prashanth Mohan, Shivaram Venkataraman, Aurojit Panda, Andrew Wang, Arka Bhattacharya, Tathagata Das and Haoyuan Li (HY).

I have been fortunate to have been mentored by some amazing researchers at various stages of my career. Venkat Padmanabhan and Chandramohan Thekkath were instrumental in me pursuing a Ph.D. Srikanth Kandula played a huge role in shaping my research tastes and guiding my research methodology. I benefited considerably from Sriram Rao's many practical insights and inputs to my projects. Ramarathnam "Venkie" Venkatesan's encouragement and positive spirit helped me throughout my Ph.D.

I also want to thank the support of my friends Kedar Hippalgaonkar (and Oink), Shivang Patwa, Gopal Vaswani, Parul Jain, Deepti Chittamuru, Hastagiri Prakash, Manjari Jain, Yamini Kannan, Amey Kaloti and Vamsi Krishna.

Finally, and most importantly, the love and support of my family, has been and will continue to remain my backbone.

Chapter 1

Introduction

Analyzing large volumes of data has become the major source for innovation behind large Internet services as well as scientific applications. Examples of such “big data analytics” occur in personalized recommendation systems, online social networks, genomic analyses, and legal investigations for fraud detection. A key property of the algorithms employed for such analyses is that they provide better results with increasing amount of data processed. In fact, in certain domains there is a trend to use relatively simpler algorithms and instead rely on more data to produce better results.

However, while the amount of data to be analyzed increases on the one hand, the acceptable time to produce results is shrinking on the other hand. Timely analyses have significant ramifications for revenue as well as productivity. Low latency results in online services leads to improved user satisfaction and revenue. Ability to crunch large datasets in short periods results in faster iterations and progress in scientific theories.

Many *compute frameworks* have been built for large scale data analyses. Some of the widely used frameworks are MapReduce [36], Dryad [70], Spark [72], Dremel [88] and Pregel [45]. To cope with the dichotomy of ever-growing datasets and shrinking times to analyze them, these frameworks parallelize computations on large distributed clusters consisting of many machines. Such parallelization enables compute frameworks to cope with growth in datasets being faster than Moore’s law.

Frameworks compose a computation, referred to as a *job*, in to a DAG of *phases*, where each phase consists of many fine grained *tasks*. Tasks of a phase have no dependencies among them and can execute in parallel. The job’s input (*file*) is divided into many *blocks* and stored in the cluster using a distributed file system. Every task’s input is one or more blocks of the file and a centralized scheduler assigns a *compute slot* to every task [1]. Tasks in the input phase produce *intermediate* outputs that are passed to other tasks downstream in the DAG. Table 1.1 defines the terminologies.

[1] Slot is a virtual token, akin to a quota, for sharing cluster resources among multiple jobs. One task can run per slot at a time.

Term	Description
Task	Atomic unit of computation with a fixed input
Phase	A collection of tasks that can run in parallel, e.g., map, aggregate
Workflow	A directed acyclic graph denoting how data flows between phases
Job	An execution of the workflow
Block	Atomic unit of storage by the distributed file system
File	Collection of blocks
Slot	Computational resources allotted to a task on a machine

Table 1.1: **Definitions of terms used in data analytics frameworks.**

1.1 Job Model

We begin with a model of a job's execution by building from first principles. We present our model with a single phase in the job, and then extend it to a DAG of phases. The objective of developing our model is, in addition to explaining the functioning of parallel jobs, also to highlight the implications of task completions on job durations.

1.1.1 Phase Execution

Assume a phase consists of n tasks and has s slots. The phase completes when its last task finishes.

The completion time of task i, t_i, is a function of the size of the data it processes, the code it runs, the resources available on the machine it executes, and the bandwidth available on the network paths involved:

$$t_i = f\left(\text{datasize}, \text{code}, \text{machine}, \text{network}\right). \tag{1.1}$$

The input data size of the task dictates the time it spends on reading the data from the distributed file system. For IO-intensive tasks, this constitutes a significant fraction of its overall duration. Note that even within a phase, the amount of data processed by tasks varies due to limitations in dividing work evenly. While the code executed by tasks of a phase is the same, they differ significantly across phases. For instance, some phases parse the records in the input data and create a schema of them, while other phases aggregate the outputs produced by tasks of upstream phases. Finally, the resources (memory, CPU cores, disk bandwidth) present on the machine on which the task executes, along with contention for those resources, dictates the duration of the task. Placing a task on a machine that has other resource hungry tasks inflates its completion time, as does reading data across congested network links.

In the ideal scenario, where every task takes the same amount of time, say t, scheduling is simple. Any work-conserving schedule would complete the phase in $\left(\lceil \frac{n}{s} \rceil \times t\right)$. When the task completion time varies, however, a naive work-conserving scheduler can take up to

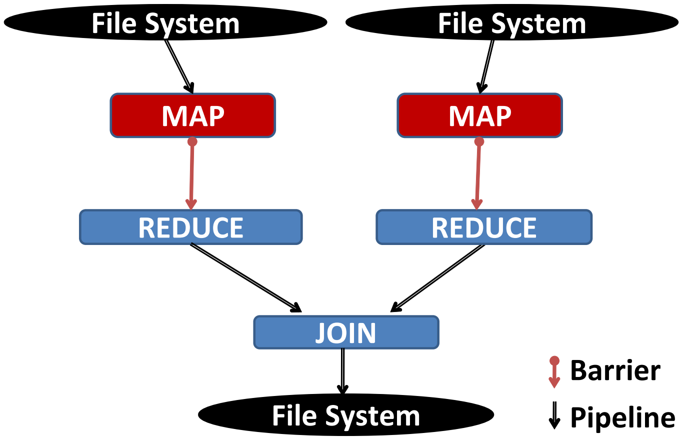


Figure 1.1: **An example of a DAG in a Dryad job. The job has two parallel input phases reading data from the distributed file system, the outputs of which are independently aggregated and then joined before being written back to the file system.**

$\left(\frac{\sum_n t_i}{s} + \max t_i\right)$. A large variation in t_i increases the term $(\max t_i)$. Thus, the goal of a scheduler is to minimize the phase completion time and make it closer to $\frac{\sum_n t_i}{s}$. Sometimes, it can do even better. By placing tasks at less congested machines or network locations, the t_i's themselves can be lowered. The challenge lies in recognizing the aspects that can be changed and scheduling accordingly.

1.1.2 Extending from a phase to a job

A job typically consists of many phases, as illustrated in Figure 1.2a. Tasks of downstream phases read outputs written by tasks of upstream phases. The output of upstream tasks are typically written only to local storage and not replicated to avoid network and storage overheads. Often, phases in a job can have barriers between them. Barriers are points in a job's workflow where none of the tasks in successive phase(s) can begin until all of the tasks in the preceding phase(s) finish. Barriers occur primarily due to aggregation operations that are neither commutative nor associative [100], for instance, computing the median of records that have the same key. Such an aggregation task can begin only when all tasks in the preceding phase(s) finish. In the absence of barriers, a task can start when all its inputs, generated by tasks in earlier phases, are available. [2]

[2] There is a variant in implementation where a slot is reserved for a task even before all its inputs are ready. This is either to amortize the latency of network transfer by moving data over the network as soon as it is generated [36,52], or compute partial results and present answers *online* even before the job is complete [94]. However, pre-allocation of slots can hog more resources for longer periods.

The phase structure of these jobs adds to the challenges in scheduling. Delays in completion of tasks in early phases limit when tasks that use its output downstream may start, thereby cumulatively affecting the whole job. At barriers in the workflow, such delays can bring the job to a standstill. We next proceed to see the reasons behind such variations in completion times of tasks.

1.2 Heterogeneity in Clusters

From the job model described above, it is clear that variations in task completion times has significant implications for job completion. Variations in task completions are induced by many heterogeneities in clusters. Our focus is on heterogeneities induced by the dynamically changing availabilities of resources in the cluster. In the context of the model in §1.1, we do not concern ourselves with the code executed by the task and rely on complementary techniques to optimize their performance.

1.2.1 Data Skew

We briefly describe our technique to deal with skews among tasks in the size of data they read, and then turn our focus to the systemic aspects.

It is natural to ask why data size varies across tasks in a phase. Across phases in the Bing production traces (described shortly), the coefficient of variation ($\frac{stdev}{mean}$) in data size is 0.34 and 3.1 at the 50th and 90th percentiles, respectively. Dividing work evenly is non-trivial for a few reasons. First, scheduling each additional task has overhead at the job manager. Network bandwidth is another reason. There might be too much data on a machine for a task to process, but it may be worse to split the work into multiple tasks and move data over the network. A third reason is poor coding practice. If the data is partitioned on a key space that has too little entropy, i.e., a few keys correspond to a lot of data, then the partitions will differ in size. Reduce tasks are not amenable to splitting (neither commutative nor associative [80, 100]), and hence each partition has to be processed by one task. Some joins and sorts are similarly constrained.

Skews in task durations, however, do delay job completion. We mitigate the effect of such skews by controlling the order in which tasks are scheduled. In particular, given a set of n tasks, s slots and data sizes $d[1 \cdots n]$, computing the optimal schedule that minimizes the job completion time is known to be NP-hard. We approximate this using the heuristic of scheduling tasks in a phase in descending order of their data size. Prior work on scheduling theory has shown that doing so results in the job finishing in time T that has a bounded approximation factor to the optimal completion time (T_O).

$$\frac{T}{T_O} \leq \frac{4}{3} - \frac{1}{3s} \qquad \text{from [48]} \tag{1.2}$$

This means that scheduling tasks with the longest processing time first is at most 30% worse than the optimal.

1.2.2 Systemic Heterogeneities

A variety of systemic factors influence progress of tasks during their IO (reading their inputs and writing their outputs) and computation.

Tasks read their inputs either from local storage or remotely across the network, depending on the location on which it is scheduled. As a result, in clusters with substantial over-subscription in their network topology [66], scheduling tasks with *data locality*, i.e., on the same machine as its input data, significantly impacts its completion. The value of locality only increases with the trend of storing inputs of tasks in memory; memory bandwidths are much higher than network throughputs. When tasks are scheduled without locality, they face contention on the network links depending on their location. Further, when dealing with intermediate data, its availability becomes crucial. Intermediate data is not replicated for efficiency reasons, hence, when the only available copy becomes inaccessible (say, due to loss of connectivity to the machine), it has to be regenerated using expensive recomputation of the task that generated it in the first place.

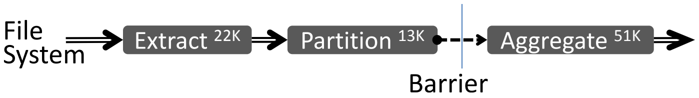
Despite being careful about the location on which tasks are scheduled, their progress is dictated by local resources (memory/CPU) as well as changing contentions to them. For instance, if tasks with high memory footprints are scheduled on the same machine, they cause slowdowns due to paging and other follow-on effects. Even when tasks are scheduled with locality, contentions to local disk can slow them down.

Illustration of Outliers

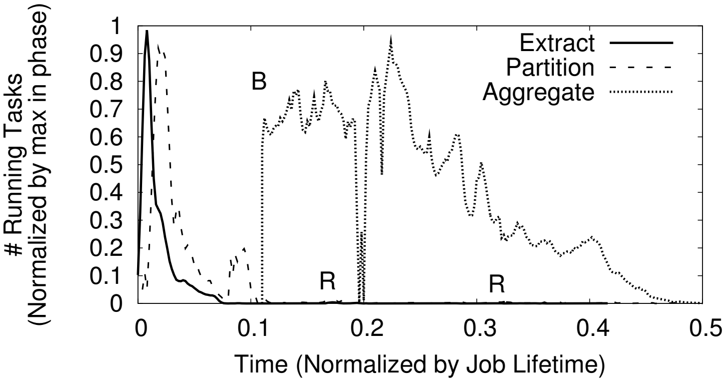
Figure 1.2a shows the workflow for *Log_Merge*, a job whose structure is typical of those in the cluster. The job reads a dataset of search usage and derives an index. It consists of two map-reduce operations and a join, but for clarity we only show the first map-reduce here. Phase names follow the Dryad [70] convention– *extract* reads raw blocks, *partition* divides data on the key and *aggregate* reduces items that share a key.

Figure 1.2b depicts a timeline of an execution of this workflow. It plots the number of tasks of each phase that are active, normalized by the maximum tasks active at any time in that phase, over the lifetime of the job. Tasks in the first two phases start in quick succession to each other at ∼.05, whereas the reduce starts after a barrier.

Some of the outliers are evident in the long lulls before a phase ends when only a few of its tasks are active. In particular, note the regions before x∼.1 and x∼.5. The spike in phase #2 here is due to the outliers in phase #1 holding on to the job's slots. At the barrier, x∼.1, just a few outliers hold back the job from making forward progress. Though most aggregate tasks finish at x∼.3, the phase persists for another 20%.



(a) Partial workflow with the number of tasks in each phase



(b) Time lapse of task execution (R=Recomputes, B=Barrier).

Figure 1.2: **An example job from the production cluster at Microsoft Bing.**

The worst cases of waiting immediately follow recomputations of lost intermediate data marked by R. Recomputations manifest as tiny blips near the x axes for phases that had finished earlier, *e.g.*, phase #2 sees recomputes at x∼.2 though it finished at x∼.1. At x∼.2, note that aggregate almost stops due to a few recomputations.

To summarize, tasks vary in their completions due to the many heterogeneities faced by them in their IO as well as their computation. Ideally, when normalized for skew in inputs, we want tasks to *progress* at the same rate. The central focus of this work is *homogenizing progress rates of computational tasks in large heterogeneous clusters.*

1.2.3 Solution Overview

Towards the goal of achieving homogeneous task durations, we design and implement the following systems. These systems involve the scheduler as well as storage. Figure 1.3 explains the positioning of the different systems.

PACMan: PACMan is an in-memory caching layer for storing input files of jobs. It leverages the trend of machines in clusters having large memories to speed up these analytics jobs. The key challenge, however, is that a job is sped up only when inputs of all its parallel tasks are cached. Indeed, a single task whose input is not cached can slow down the entire job. To meet this all-or-nothing property, PACMan coordinates access to the distributed caches. This coordination is essential to improve job completion times and cluster efficiency. To this end, we have implemented two cache replacement policies on top of PACMans coordinated infrastructure—LIFE that minimizes average completion time by evicting large incomplete inputs, and LFU-F that maximizes cluster efficiency by evicting less frequently accessed

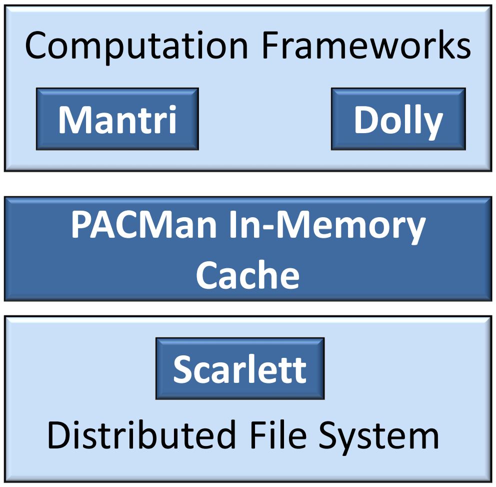


Figure 1.3: **Logical positioning of the different systems within the standard software stack of data analytics frameworks. Mantri and Dolly work with the scheduler while PACMan and Scarlett deal with input data management.**

inputs.

Scarlett: Despite storing blocks in cache, tasks sometimes do not obtain the benefits of caching because they are not scheduled to execute locally. The miss in locality often occurs when a task accesses popular input blocks or when its input is stored on machines storing other popular blocks. When tasks contend for compute slots on popular machines, locality suffers. To avoid contentions and improve data locality, we design a system, Scarlett, that replicates files based on their access patterns and spreads them out to avoid hotspots, while minimally interfering with running jobs. Scarlett accurately predicts file popularity and works within hard bounds on additional storage.

Mantri: Mantri is a system that monitors tasks and prevents stragglers based on their causes. It uses the following techniques. First, it schedules speculative copies of straggler tasks while being cognizant of resource constraints and work imbalances. Second, it places tasks based on the locations of their data sources as well as the current utilization of network links. Finally, on a tasks completion, Mantri replicates its output (intermediate data) if the benefit of not having to recompute outweighs the cost of replication.

Dolly: While Mantri's speculation strategies work well for large jobs, the small jobs require a fundamentally different technique. This is because Mantri involves an element of waiting and speculation. Such waiting limits their agility when dealing with stragglers in small jobs as they often start all their tasks simultaneously. Dolly instead proposes full cloning of small jobs, avoiding waiting and speculation altogether. Cloning of small jobs only marginally increases utilization because workloads show that while the majority of jobs are small, they only consume a small fraction of the resources. The main challenge of cloning is, however, that extra clones can cause contention for intermediate data. We use a technique, delay assignment, which efficiently avoids such contention.

1.3 Methodology: Analysis and Evaluation

The insights and evaluations in this work are based on traces of computational frameworks from production clusters at Facebook and Microsoft Bing. We obtain task level traces from Facebook's production Hadoop cluster and Bing's Dryad cluster. These are large clusters consisting of thousands of well-provisioned machines.

1.3.1 Cluster Details

Microsoft Bing Cluster: We monitored the cluster and software systems that support the Bing search engine for over twelve months. This is a cluster of tens of thousands of commodity servers managed by Cosmos, a fork off Dryad [70].

While programmers can write native code, most of the jobs in the examined cluster are written in Scope [80], a mash-up language that mixes SQL-like declarative statements with user code. The Scope compiler transforms a job into a workflow where each node is a phase and each edge joins a phase that produces data to another that uses it. Compiler optimizations can merge different functionality into one phase or divide functionality across phases. The number of tasks in a phase is chosen at compile time. A task will read its input over the network if it is not available on the local disk but outputs are written to the local disk. The eventual outputs of a job (as well as raw data) are stored in a reliable block storage system implemented on the same servers that do computation. Blocks are replicated three way for reliability. A run-time scheduler assigns tasks to machines, based on data locations, dependence patterns and cluster-wide resource availability. The network layout is such that there is more bandwidth within a rack than across racks.

We obtain detailed logs from the Scope compiler and the Cosmos scheduler. At each of the job, phase and task levels, we record the execution behavior as represented by begin and end times, the machines(s) involved, the sizes of input and output data, the fraction of data that was read across racks and a code denoting the success or type of failure. We also record the workflow of jobs. Table 1.2 depicts the random subset of logs that we analyze here.

Facebook Cluster: The Facebook cluster consisted of over 4000 machines. Each of the machines was well-provisioned with 16 cores and 64GB of memory each. The cluster had networks with an over-subscription factor of ~ 10.

The Facebook cluster executed the Hadoop computing framework. Jobs were submitted using the Hive [8] engine that composed a query in to a series of Hadoop MapReduce jobs. The number of tasks in each of the phases is decided before the job begins its execution. A centralized job scheduler executed these jobs by assigning tasks to slots based on the Fair Scheduler [104]. Just like the Bing cluster, a distributed file system (Hadoop Distributed File System or HDFS [6]) stored the data in the cluster, replicated for reliability.

The jobs executed on these clusters were a mix of production as well as experimental analyses. Their output and performance had significant impact on productivity and revenue. Overall, these traces represent over half a million jobs reading and transferring nearly 0.5

Dates	**Phases** **x 10^3**	**Jobs**	**Compute** **(years)**	**Data** **(PB)**	**Network** **(PB)**
May 2010	19.0	938	49.1	12.6	.66
Jun 2010	16.5	991	88.0	22.7	1.22
Jul 2010	22.0	1183	51.6	14.3	.67
Aug 2010	29.2	1873	60.6	18.7	.76
Sep 2010	27.4	1653	73.0	22.8	.73
Oct 2010	20.4	1362	84.1	25.3	.86
Nov 2010	37.8	1834	88.4	25.0	.68
Dec 2010	18.7	1777	96.2	18.6	.72
Jan 2011	24.4	1842	79.5	21.5	1.99

Table 1.2: **Details of the Bing production Dryad logs.**

	Facebook	**Microsoft Bing**
Dates	Oct 2010	May 2010 – Dec 2011
Framework	Hadoop	Dryad
File System	HDFS [6]	Cosmos
Script	Hive [8]	Scope [80]
Jobs	375K	200K
Input Data	150PB	310PB
Cluster Size	3,500	Thousands
Memory per machine	48GB	N/A

Table 1.3: **Summary of Facebook and Bing clusters and traces.**

Exabyte of data. Table 1.3 summarizes the details of the two clusters and the collected traces.

1.3.2 Evaluation Workload

Evaluation of our system is done on a 200 machine cluster on Amazon's EC2 [2]. Workloads for our evaluation are derived from the Facebook and Bing traces described above, representative of Hadoop and Dryad systems. The key goals during this derivation was to preserve the original workload's characteristics, including the distribution of job input sizes, variable popularity of input files, and load proportional to the original clusters.

We meet these goals as follows. We replay jobs with the same inter-arrival times and input files as in the original workload. However, we scale down the file sizes proportionately to reflect the smaller size of our cluster. This scaling down helps us mimic the proportional load experienced by the original clusters as well as the access patterns of files. Additionally, it also ensures that the memory in our experimental cluster is sufficient for the same fraction of jobs' input as in the original workload.

We confirmed by simulation (described shortly) that performance improvements with the scaled down version matched that of the full-sized cluster.

1.3.3 Trace-driven Simulator

We use a trace-driven simulator to evaluate at larger scales and longer durations. The simulator performed a detailed and faithful replay of the task-level traces of Hadoop jobs from Facebook and Dryad jobs from Bing. It simulated the thousands of machines as there were in the original cluster and preserved the read/write sizes of tasks, replica locations of input data as well as job characteristics of failures, stragglers and recomputations [19]. The simulator also mimicked fairness restrictions on the number of permissible concurrent slots as well as fairness based evictions. For the network, it uses a fluid model rather than simulating individual packets. We use the simulator to test performance at the scale of the original datacenters, as well as to mimic ideal schemes.

1.4 Roadmap

The rest of the thesis is arranged as follows. Chapter 2 describes the work on handling input data of jobs. The two main pieces of work are building a distributed in-memory cache, PACMan, for parallel jobs and ensuring that blocks are replicated in proportion to their popularity to achieve data locality for tasks (Scarlett). Chapter 3 deals with handling intermediate data of jobs. In addition to efficiently scheduling the network to transfer intermediate data, we also deal with inaccessibility of the data. Despite the above measures, tasks experience many contentions for resources during their execution, causing them to straggle. Chapter 4 tackles the occurrence of such runtime stragglers. The techniques in Chapters 2, 3 and 4 together present a comprehensive solution to achieve uniform and efficient progress of tasks of a job. Chapter 5 concludes with some future directions for research.

Chapter 2

Input Data

2.1 Introduction

Data in analytics clusters is typically stored by the distributed file system across disks of many machines. Hardware trends, however, driven by falling costs indicate a steep increase in memory capacities of large clusters. This presents an opportunity to store the input data of the analytics jobs in memory and speed them up. Storing all the data currently present in disks is, however, infeasible because of the three orders of magnitude difference in the available capacities between disk and memory, notwithstanding the growing memory sizes. Therefore, we investigate the use of *memory locality* to speed-up data-intensive jobs by caching their input data.

As mentioned earlier, data-intensive jobs, typically, have a phase where they process the input data (e.g., *map* in MapReduce [36], *extract* in Dryad [70]). This phase simply reads the raw input and writes out parsed output to be consumed during further computations. Naturally, this phase is IO-intensive. Workloads from Facebook and Microsoft Bing datacenters, consisting of thousands of servers, show that this IO-intensive phase constitutes 79% of a job's duration and consumes 69% of its resources. Our proposal is to speed up these IO-intensive phases by caching their input data in memory. Data is cached after the first access thereby speeding up subsequent accesses.

Using memory caching to improve performance has a long history in computer systems, e.g., [11,40,51,57]. We argue,however, that the *parallel* nature of data-intensive jobs differentiates them from previous systems. Frameworks split jobs in to multiple *tasks* that are run in parallel. There are often enough idle compute slots for small jobs, consisting of few tasks, to run all their tasks in parallel. Such tasks start at roughly the same time and run in a single *wave*. In contrast, large jobs, consisting of many tasks, seldom find enough compute slots to run all their tasks at the same time. Thus, only a subset of their tasks run in parallel.[1] As and when tasks finish and vacate slots, new tasks get scheduled on them. We define the number of parallel tasks as the *wave-width* of the job.

[1] We use the terms "small" and "large" jobs to refer to their input size and/or numbers of tasks.

The wave-based execution implies that small single-waved jobs have an *all-or-nothing* property – unless all the tasks get memory locality, there is no improvement in completion time. They run all their tasks in one wave and their completion time is proportional to the duration of the longest task. Large jobs, on the other hand, improve their completion time with every wave-width of their input being cached. Note that the exact set of tasks that run in a wave is not of concern, we only care about the wave-width, i.e., how many of them run simultaneously.

Our position is that *coordinated management* of the distributed caches is required to ensure that enough tasks of a parallel job have memory locality to improve their completion time. Coordination provides a global view that can be used to decide what to evict from the cache, as well as where to place tasks so that they get memory locality. To this end, we have developed PACMan – Parallel All-or-nothing Cache MANager – an in-memory caching system for parallel jobs. On top of PACMan's coordination infrastructure, appropriate placement and eviction policies can be implemented to speed-up parallel jobs.

One such coordinated eviction policy we built, LIFE, aims to *minimize the average completion time of jobs.* In a nutshell, LIFE calculates the wave-width of every job and favors input files of jobs with small waves, i.e., lower wave-widths. It replaces cached blocks of the incomplete file with the largest wave-width. The design of LIFE is driven by two observations. First, a small wave requires caching less data than a large wave to get the same decrease in completion time. This is because the amount of cache required by a job is proportional to its wave-width. Second, we need to retain the entire input of a wave to decrease the completion time. Hence the heuristic of replacing blocks from incomplete files.

Note that maximizing cache hit-ratio – the metric of choice of traditional replacement policies – does not necessarily minimize average completion time, as it ignores the wave-width constraint of parallel jobs. For instance, consider a simple workload consisting of 10 equal-sized single-waved jobs. A policy that caches only the inputs of five jobs will provide a better average completion time, than a policy that caches 90% of the inputs of each job, which will not provide any completion time improvement over the case in which no inputs are cached. However, the first policy will achieve only 50% hit-ratio, compared to 90% hit-ratio for the second policy.

In addition to LIFE, we implemented a second eviction policy, LFU-F, which aims to *maximize the efficiency of the cluster.* Cluster efficiency is defined as finishing the jobs by using the least amount of resources. LFU-F favors popular files and evicts blocks from the least accessed files. Efficiency improves every time data is accessed from cache. So files that are accessed more frequently contribute more to cluster efficiency than files that will be accessed fewer number of times.

A subtle aspect is that the all-or-nothing property is important even for cluster efficiency. This is because tasks of subsequent phases often overlap with the IO-intensive phase. For example, in MapReduce jobs, reduce tasks begin after a certain fraction of map tasks finish. The reduce tasks start reading the output of completed map tasks. Hence a delay in the completion of a few map tasks, when their data is not cached, results in all the reduce tasks waiting. These waiting reduce tasks waste computation slots effectively hurting efficiency.

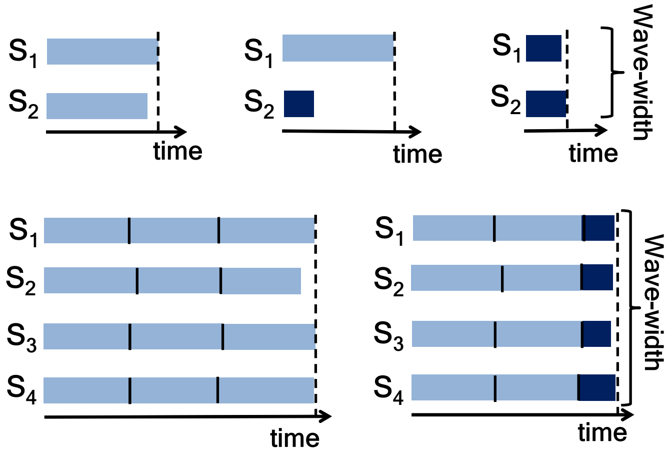


Figure 2.1: **Example of a single-wave (2 tasks, simultaneously) and multi-wave job (12 tasks, 4 at a time). S_i's are slots. Memory local tasks are dark blocks. Completion time (dotted line) reduces when a wave-width of input is cached.**

Finally, a key component towards effectively utilizing the cache is scheduling tasks on machines that stores their input data in memory. Remote reads are limited by network throughputs which are typically lower than memory bandwidths. Unfortunately, tasks contend for compute slots on machines containing popular data blocks. To avoid such contention, we design a selective replication system Scarlett that automatically infers popular data blocks and replicates them. Scarlett uses historical access patterns to predict popularity and impending contentions. While creating extra replicas, Scarlett is intelligent about spreading them out and minimally contending with existing network traffic.

This chapter is outlined as follows. §2.2 describes the fundamental properties of parallel jobs that influence the design of the caching algorithms. We describe PACMan's system design in §2.3. Scarlett's replication algorithm is described in §2.4. We present evaluation results in §2.5 by deploying our system on Amazon's EC2 and evaluating using production workloads. Finally, §2.6 contrasts PACMan and Scarlett with prior work.

2.2 Cache Replacement for Parallel Jobs

In this section, we first explain how the concept of wave-width is important for parallel jobs, and argue that maximizing cache hit-ratio neither minimizes the average completion time of parallel jobs nor maximizes efficiency of a cluster executing parallel jobs. From first principles, we then derive the ideas behind LIFE and LFU-F cache replacement schemes.

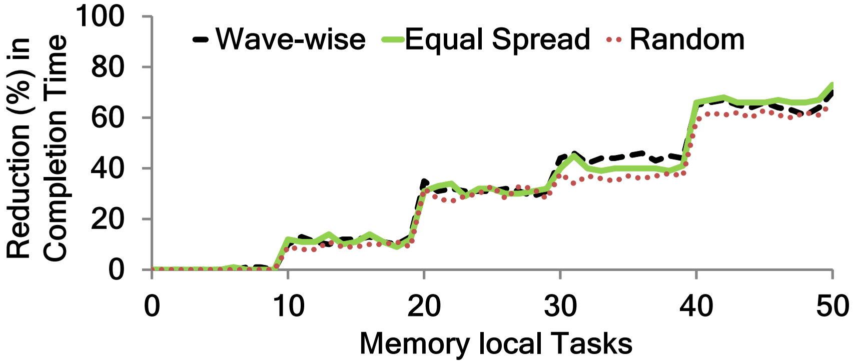


Figure 2.2: **Reduction in completion time of a job with** 50 **tasks running on** 10 **slots. The job speeds up, in steps, when its number of memory local tasks crosses multiples of the wave-width (i.e.,** 10**), regardless of how they are scheduled.**

2.2.1 All-or-Nothing Property

Achieving memory locality for a task will shorten its completion time. But this need not speed up the job. Jobs speed up when an entire wave-width of input is cached (Figure 2.1). The wave-width of a job is defined as the number of simultaneously executing tasks. Therefore, jobs that consist of a single wave need 100% memory locality to reduce their completion time. We refer to this as the *all-or-nothing* property. Jobs consisting of many waves improve as we incrementally cache inputs in multiples of their wave-width. In Figure 2.1, the single-waved job runs both its tasks simultaneously and will speed up only if the inputs of both tasks are cached. The multi-waved job, on the other hand, consists of 12 tasks and can run 4 of them at a time. Its completion time improves in steps when any 4, 8 and 12 tasks run memory locally.

We confirmed the hypothesis of wave-widths by executing a sample job on a cluster with 10 slots (see Figure 2.2). The job operated on 3GB of input and consisted of 50 tasks each working on 60MB of data. Our experiment varied the number of memory-local tasks of the job and measured the reduction in completion time. The baseline was the job running with no caching. Memory local tasks were spread uniformly among the waves ("Equal Spread"). We observed the job speeding up when its number of memory-local tasks crossed 10, 20 and so forth, i.e., multiples of its wave-width, thereby verifying the hypothesis. Further, we tried two other scheduling strategies. "Wave-wise" scheduled the non-memory-local tasks before memory local tasks, i.e., memory local tasks ran simultaneously, and "Random" scheduled the memory local tasks in an arbitrary order. We see that the speed-up in steps of wave-width holds in both cases, albeit with slightly reduced gains for "Random". Surprisingly, the wave-width property holds even when memory local tasks are randomly scheduled because a task is allotted a slot only when slots become vacant, not a priori. This automatically balances memory local tasks and non-memory-local tasks across the compute slots.

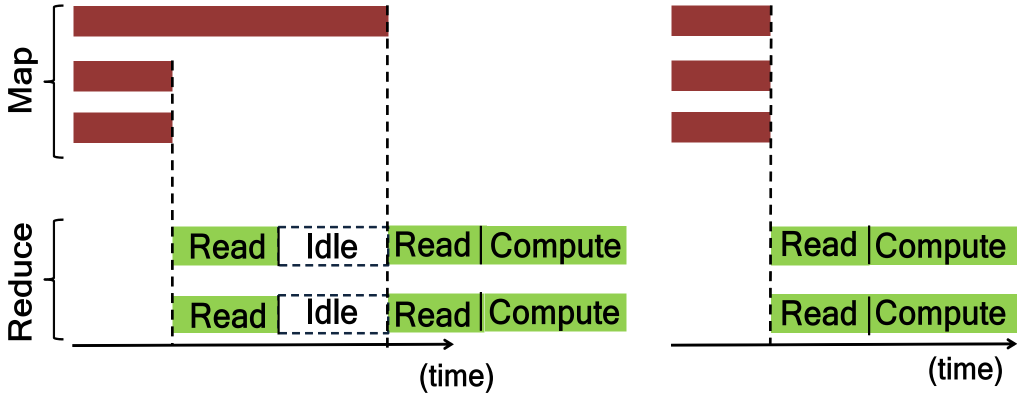


Figure 2.3: **All-or-nothing property matters for efficiency. In this example of a job with 3 map tasks and 2 reduce tasks, even if one map task is delayed (due to lack of memory locality), reduce tasks idle and hurt efficiency.**

That achieving memory locality will lower resource usage is obvious – tasks whose inputs are available in memory run faster and occupy the cluster for fewer hours. A subtler point is that the all-or-nothing constraint can also be important for cluster efficiency. This is because some of the schedulers in parallel frameworks (e.g., Hadoop and MPI) allow tasks of subsequent stages to begin even before all tasks in the earlier stages finish. Such "pipelining" can hide away some data transfer latency, for example, when reduce tasks start running even before the last task in the map stage completes [7]. However, this means that a delay in the completion of some map tasks, perhaps due to lack of memory locality, results in all the reduce tasks waiting. These waiting reduce tasks waste computation slots and adversely affect efficiency. Figure 2.3 illustrates this overlap with an example job of three map tasks and two reduce tasks.

In summary, meeting the all-or-nothing constraint improves completion time and efficiency of parallel jobs.

2.2.2 Sticky Policy

Traditional cache replacement schemes that maximize cache hit-ratio do not consider the wave-width constraint of all-or-nothing parallel jobs. Consider the situation depicted in Figure 2.4 of a 4-entry cache storing blocks A, B, C and D. Job J_1's two tasks will access blocks A and B, while job J_2's two tasks will access C and D. Both jobs consist of just a single wave and hence their job completion time improves only if their entire input is cached.

Now, pretend a third job J_3 with inputs F and G is scheduled before J_1 and J_2, requiring the eviction of two blocks currently in the cache. Given the oracular knowledge that the future block access pattern will be A, C, B, then D, MIN [26] will evict the blocks accessed farthest in the future: B and D. Then, when J_1 and J_2 execute, they both experience a

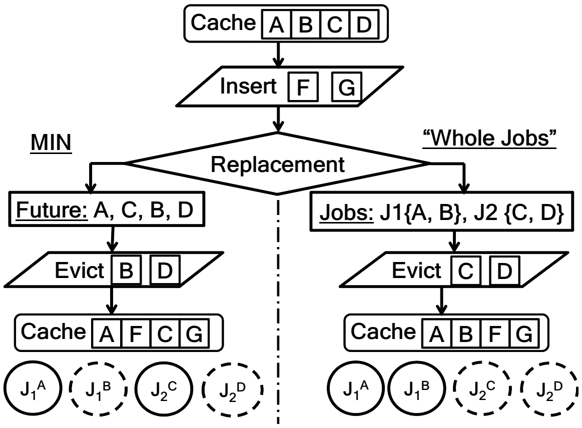


Figure 2.4: **Cache hit-ratio does not necessarily improve job completion. We consider a cache that has to replace two out of its four blocks. MIN evicts blocks to be accessed farthest in future. "Whole jobs" preserves complete inputs of jobs.**

cache miss on one of their tasks. These cache misses bound their completion times, meaning that MIN cache replacement resulted in no reduction in completion time for either J_1 or J_2. Consider an alternate replacement scheme that chooses to evict the input set of J_2 (C and D). This results in a reduction in completion time for J_1 (since its entire input set of A and B is cached). J_2's completion time is unaffected. Note that the cache hit-ratio remains the same as for MIN (50%).

Further, maximizing hit-ratio does not maximize efficiency of the cluster. In the same example as in Figure 2.4, let us add a reduce task to each job. Both J_1 and J_2 have two map tasks and one reduce task. Let the reduce task start after 5% of the map tasks have completed (as in Hadoop [7]). We now compare the resource consumption of the two jobs with MIN and "whole jobs" which evicts inputs of J_2. With MIN, the total resource consumption is $2\,(1+\mu)\,m + 2\,(0.95)\,m$, where m is the duration of a non-memory-local task and μ reflects the speed-up when its input is cached; we have omitted the computation of the reduce task. The policy of "whole jobs", on the other hand, expends $2\,(1+\mu)\,m + (0.95\mu + 0.05)\,m$ resources. As long as memory locality produces a speed-up, i.e., $\mu \leq 1$, MIN consumes more resources.

The above example, in addition to illustrating that cache hit-ratios are insufficient for both speeding up jobs and improving cluster efficiency, also highlights the importance of retaining *complete* sets of inputs. Improving completion time requires retaining complete wave-widths of inputs, while improving efficiency requires retaining complete inputs of jobs. Note that retaining the complete inputs of jobs automatically meets the wave-width constraint to reduce completion times. Therefore, instead of evicting the blocks accessed farthest in the future, replacement schemes for parallel jobs should recognize the commonality between inputs of the same job and evict at the granularity of a job's input.

This intuition gives rise to the *sticky* policy. *The sticky policy preferentially evicts blocks* *of incomplete files.* If there is an incomplete file in cache, it sticks to its blocks for eviction until the file is completely removed from cache. The sticky policy is crucial as it disturbs the fewest completely cached inputs and evicts the incomplete files which are not beneficial for jobs.[2]

Given the sticky policy to achieve the all-or-nothing requirement, we now address the question of which inputs to retain in cache such that we minimize average completion time of jobs and maximize cluster efficiency.

2.2.3 Average Completion Time – LIFE

We show that in a cluster with multiple jobs, favoring jobs with the smallest wave-widths minimizes the average completion time of jobs. Assume that all jobs in the cluster are single-waved. Every job j has a wave-width of w and an input size of I. Let us assume the input of a job is equally distributed among its tasks. Each task's input size is $\left(\frac{I}{w}\right)$ and its duration is proportional to its input size. As before, memory locality reduces its duration by a factor of μ. The factor μ is dictated by the difference between memory and disk bandwidths, but limited by additional overheads such as deserialization and decompression of the data after reading it.

To speed up a single-waved job, we need I units of cache space. On spending I units of cache space, tasks would complete in $\mu\left(\frac{I}{w}\right)$ time. Therefore the saving in completion time would be $(1-\mu)\left(\frac{I}{w}\right)$. Counting this savings for every access of the file, it becomes $f\,(1-\mu)\left(\frac{I}{w}\right)$, where f is the frequency of access of the file. Therefore, the ratio of the job's benefit to its cost is $f\,(1-\mu)\left(\frac{1}{w}\right)$. In other words, it is directly proportional to the frequency and inversely proportional to the wave-width. The smaller the wave-width, the larger the savings in completion time per unit of cache spent. This is illustrated in Figure 2.5 comparing two jobs with the same input size (and of the same frequency), but wave-widths of 2 and 4. Clearly, it is better to use I units of cache space to store the input of the job with a wave-width of two. This is because its work per task is higher and so the savings are proportionately more. Note that even if the two inputs are unequal (say, I_1 and I_2, and $I_1 > I_2$), caching the input of the job with lower wave-width (I_1) is preferred despite its larger input size. Therefore, in a cluster with multiple jobs, *average completion time is best reduced by favoring the jobs with smallest wave-widths* (LIFE).

This can be easily extended to a multi-waved jobs. Let the job have n waves, c of which have their inputs cached. This uses $cw\left(\frac{I}{nw}\right)$ of cache space. The benefit in completion time is $f\,(1-\mu)\,c\left(\frac{I}{nw}\right)$. The ratio of the job's benefit to its cost is $f\,(1-\mu)\left(\frac{1}{w}\right)$, hence best reduced by still picking the jobs that have the smallest wave-widths.

[2] When there are strict barriers between phases, the sticky policy does not improve efficiency. Nonetheless, such barriers are akin to "application-level stickiness" and hence stickiness at the caching layer beneath, expectedly, does not add value.

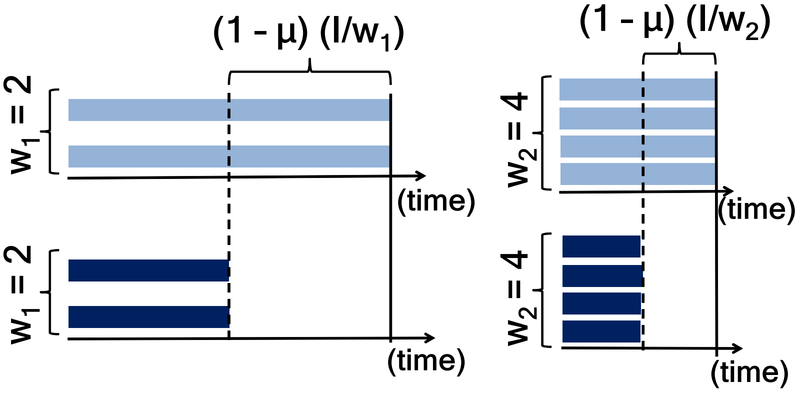


Figure 2.5: **Gains in completion time due to caching decreases as wave-width increases. Solid and dotted lines show completion times without and with caching (for two jobs with input of I but wave-widths of 2 and 4). Memory local tasks are dark blocks, sped up by a factor of μ.**

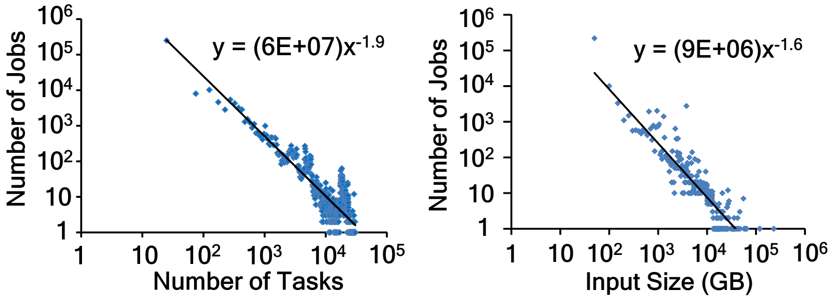
2.2.4 Cluster Efficiency – LFU-F

In this section, we derive that retaining frequently accessed files maximizes efficiency of the cluster. We use the same model for the cluster and its jobs as before. The cluster consists of single-waved jobs, and each job j has wave-width w and input size I. Duration of tasks are proportional to their input sizes, $\left(\frac{I}{w}\right)$, and achieving memory locality reduces its duration by a factor of μ.

When the input of this job is cached, we use I units of cache. In return, the savings in efficiency is $(1-\mu)\,I$. The savings is obtained by summing the reduction in completion time across all the tasks in the wave, i.e., $w \cdot (1-\mu)\left(\frac{I}{w}\right)$. Every memory local task contributes to improvement in efficiency. Further, the savings of $(1-\mu)\,I$ is obtained on every access of the file, thereby making its aggregate value $f\,(1-\mu)\,I$ where f is the frequency of access of the file. Hence, the ratio of the benefit to every unit of cache space spent on this job is $f\,(1-\mu)$, or a function of only the frequency of access of the file. Therefore, *cluster efficiency is best improved by retaining the most frequently accessed files* (LFU-F).

This naturally extends to multi-waved jobs. As jobs in data-intensive clusters typically read entire files, frequency of access of inputs across the different waves of a job is the same. Hence cluster efficiency is best improved by still favoring the frequently accessed files.

To summarize, we have shown that, (*i*) the all-or-nothing property is crucial for improving completion time of jobs as well as efficiency, (*ii*) average completion time is minimized by retaining inputs of jobs with low wave-widths, and (*iii*) cluster efficiency is maximized by retaining the frequently used inputs. We next show some relevant characteristics from production workloads, before moving on to explain the details of PACMan.



(a) Number of tasks

(b) Input Size

Figure 2.6: **Power-law distribution of jobs (Facebook) in the number of tasks and input sizes. Power-law exponents are** 1.9 **and** 1.6 **when fitted with least squares regression.**

2.3 PACMan: System Design

We first present PACMan's architecture that enables the implementation of the sticky policy, and then discuss the details involved in realizing LIFE and LFU-F. Before presenting the architecture, we briefly highlight characteristics in workloads—heavy tail distribution of input sizes of jobs, and correlation between file size and popularity—that are relevant for LIFE and LFU-F.

2.3.1 Workload Characteristics

Heavy-tailed Input Sizes of Jobs

Datacenter jobs exhibit a heavy-tailed distribution of input sizes. Workloads consist of many small jobs and relatively few large jobs. In fact, 10% of overall data read is accounted by a disproportionate 96% and 90% of the smallest jobs in the Facebook and Bing workloads. As Figure 4.6 shows, job sizes – input sizes and number of tasks – indeed follow a power-law distribution, as the log-log plot shows a linear relationship.

The skew in job input sizes is so pronounced that a large fraction of active jobs can simultaneously fit their entire data in memory.[3] We perform a simple simulation that looks at jobs in the order of their arrival time. The simulator assumes the memory and computation slots across all the machines in the cluster to be aggregated. It loads a job's entire input into memory when it starts and deletes it when the job completes. If the available memory is insufficient for a job's entire input, none of it is loaded. Figure 2.7 plots the results of

[3] By active jobs we mean jobs that have at least one task running.

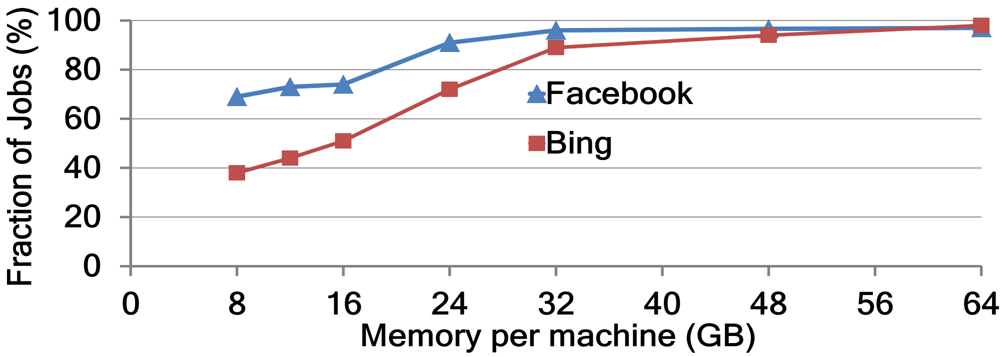


Figure 2.7: **Fraction of active jobs whose data fits in the aggregate cluster memory, as the memory per machine varies.**

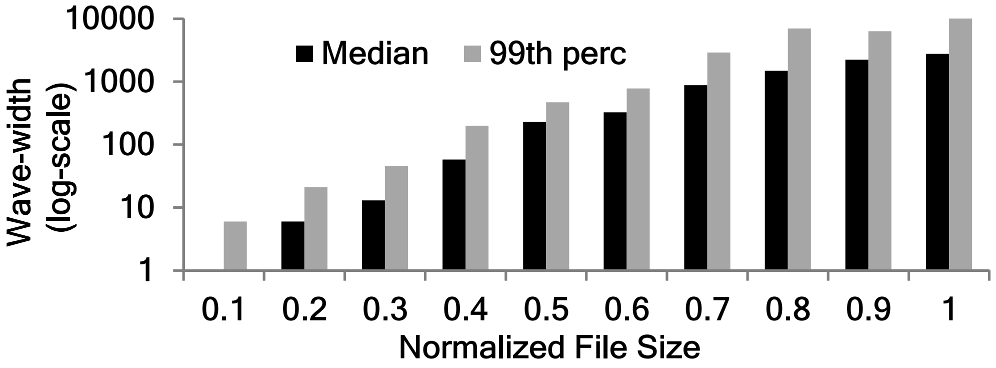


Figure 2.8: **Wave-width, i.e., number of simultaneous tasks, of jobs as a function of sizes of files accessed. File sizes are normalized to the largest file; the largest file has size 1.**

the simulation. For the workloads from Facebook and Bing, we see that 96% and 89% of the active jobs respectively can have their data entirely fit in memory, given an allowance of 32GB memory per server for caching. This bodes well for satisfying the all-or-nothing constraint of jobs, crucial for the efficacy of LIFE and LFU-F.

In addition to being easier to fit a small job's input in memory, its wave-width is smaller. In our workloads, wave-widths roughly correlate with the input file size of the job. Figure 2.8 plots the wave-width of jobs binned by the size of their input files. Small jobs, accessing the smaller files, have lower wave-widths. This is because, typically, small jobs do not have sufficient number of tasks to utilize the slots allocated by the scheduler. This correlation helps to explore an approximation for LIFE to use file sizes instead of estimating wave-widths (§2.3.3).

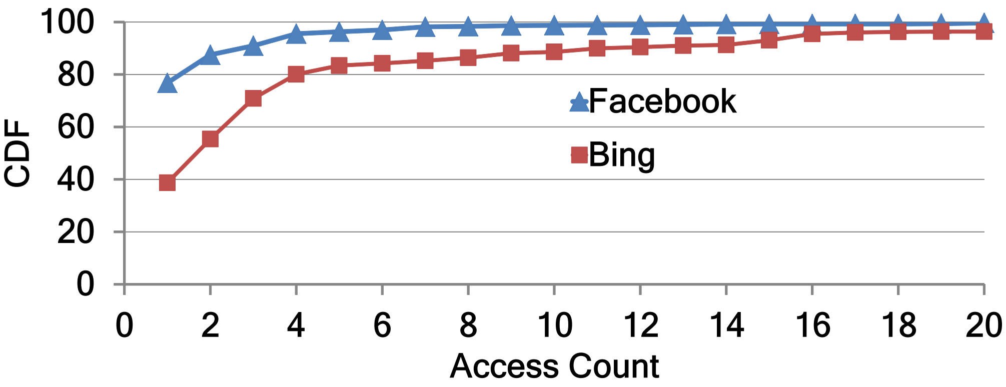


Figure 2.9: **Skewed popularity of data. CDF of the access counts of the input blocks stored in the cluster.**

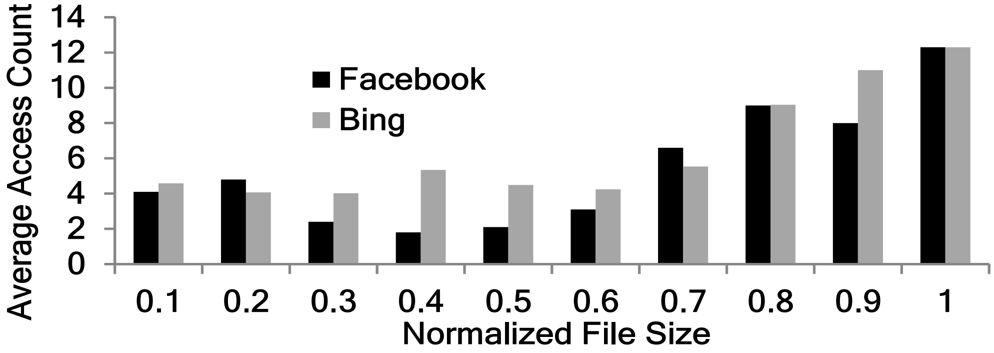


Figure 2.10: **Access count of files as a function of their sizes, normalized to the largest file; largest file has size 1. Large files, accessed by production jobs, have higher access count.**

Large Files are Popular

Now, we look at popularity skew in data access patterns. As noted in prior work, the popularity of input data is skewed in data-intensive clusters [43]. A small fraction of the data is highly popular, while the rest is accessed less frequently. Figure 2.9 shows that the top 12% of popular data is accessed 10× more than the bottom third in the Bing cluster. The Facebook cluster demonstrates a similar skew. The top 5% of the blocks are seven times more popular than the bottom three-quarters.

Interestingly, large files have higher access counts (see Figure 2.10). Often they are accessed by production jobs to generate periodic (hourly) summaries, e.g., financial and performance metrics, from various large logs over consolidated time intervals in the past. These intervals could be as large as weeks and months, directly leading to many of the logs in that interval being repeatedly accessed. The popularity of large files, whose jobs consume most resources, strengthens the idea from §2.2.4 that favoring frequently accessed files is best for cluster efficiency.

We also observe repeatability in the data accesses. *Single-accessed files are spread across only* 11% *and* 6% *of jobs* in the Facebook and Bing workloads. Even in these jobs, not all the data they access is singly-accessed. Hence, we have sufficient repeatability to improve job performance by caching their inputs.

2.3.2 Coordination Architecture

PACMan globally coordinates access to its caches. Global coordination ensures that a job's different input blocks, distributed across machines, are viewed in unison to satisfy the all-or-nothing constraint. To that end, the two requirements from PACMan are, (*a*) support queries for the set of machines where a block is cached, and (*b*) mediate cache replacement globally across the machines.

PACMan's architecture consists of a central *coordinator* and a set of *clients* located at the storage nodes of the cluster (see Figure 2.11). Blocks are added to the PACMan clients. PACMan clients update the coordinator when the state of their cache changes (i.e., when a block is added or removed). The coordinator uses these updates to maintain a mapping between every cached block and the machines that cache it. As part of the map, it also stores the file that a block belongs to and the wave-width of jobs when accessing that file (§2.3.3). This global map is leveraged by LIFE and LFU-F in implementing the sticky policy to look for incomplete files. Frameworks work with the coordinator to achieve memory locality for tasks.

The client's main role is to serve cached blocks, as well as cache new blocks. We choose to cache blocks at the *destination*, i.e., the machine where the task executes as opposed to the *source*, i.e., the machine where the input is stored. This allows an uncapped number of replicas in cache, which in turn increases the chances of achieving memory locality especially when there are hotspots due to popularity skew [43]. Memory local tasks contact the local PACMan client to check if its input data is present. If not, they fetch it from the distributed file system (DFS). If the task reads data from the DFS, it puts it in cache of the local PACMan client and the client updates the coordinator about the newly cached block. Data flow is designed to be local in PACMan as remote memory access could be constrained by the network.

Fault Tolerance: The coordinator's failure does not hamper the job's execution as data can always be read from disk. However, we include a secondary coordinator that functions as a cold standby. Since the secondary coordinator has no cache view when it starts, clients periodically send updates to the coordinator informing it of the state of their cache. The secondary coordinator uses these updates to construct the global cache view. Clients do not update their cache when the coordinator is down.

Scalability: Nothing precludes distributing the central coordinator across different ma-

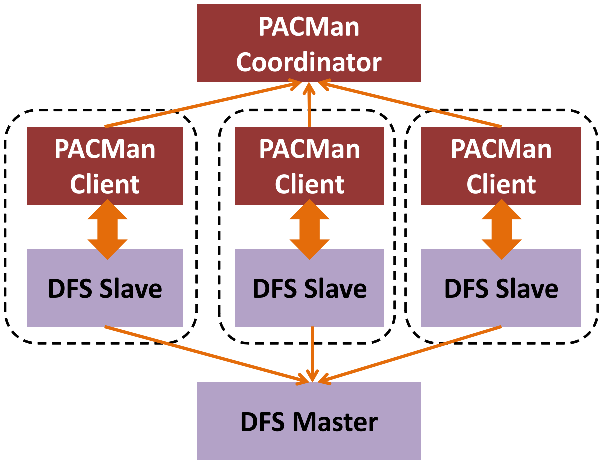


Figure 2.11: **PACMan architecture. The central *coordinator* manages the distributed *clients*. Thick arrows represent data flow while thin arrows denote meta-data flow.**

chines to avoid having it be a bottleneck. We have, however, found that the scalability of our system suffices for our workloads (see §2.5.6).

2.3.3 Wave-width

Wave-width is important for LIFE as it aims to retain inputs of jobs with lower wave-widths. However, both defining and calculating wave-widths is non-trivial because tasks do not strictly follow wave boundaries. Tasks get scheduled as and when previous tasks finish and slots become available. This makes modeling the tasks that run in a wave complicated. Slots also open up for scheduling when the scheduler allots extra slots to a job during periods of low utilization in the cluster. Therefore, wave-widths are not static during a job's execution. They are decided based on slot availabilities, fairness restrictions and scheduler policies. Unlike MIN, which is concerned only with the *order* in which requests arrive, our setting requires knowing the exact time of the request, which in turn requires estimating the speed-up due to memory locality for each task. All these factors are highly variable and hard to model accurately.

Given such a fluid model, we propose the following approximation. We make periodic measurements of the number of concurrent tasks of a job. When a job completes, we get a set of values, $\langle (w, t(w)) \rangle$, such that $0 < t(w) \leq 1$. For every value of wave-width, w, $t(w)$ shows the fraction of time spent by the job with that wave-width. We take measurements every 1s in practice.

Note that while $\sum t(w) = 1$, $\sum w$ is not necessarily equal to the number of tasks in the job. This is because wave boundaries are not strict and tasks overlap between measurements. This led us to drop the idea of using the measurements of $\langle (w, t(w)) \rangle$ to divide blocks of a file into different *waves*. Also, such an explicit division requires the scheduler to collectively

Pseudocode 1 Implementation of LIFE and LFU-F – from the perspective of the PACMan coordinator.

```
procedure FILETOEVICT_LIFE(Client c)
    cFiles = fileSet.filter(c)                              ▷ Consider only c's files
    f = cFiles.olderThan(window).oldest()                   ▷ Aging
    if f == null then                                       ▷ No old files to age out
        f = cFiles.getLargestIncompleteFile()
    if f == null then                                       ▷ Only complete files left
        f = cFiles.getLargestCompleteFile()
    return f.name                                           ▷ File to evict

procedure FILETOEVICT_LFU-F(Client c)
    cFiles = fileSet.filter(c)                              ▷ Consider only c's files
    f = cFiles.olderThan(window).oldest()                   ▷ Aging
    if f == null then                                       ▷ No old files to age out
        f = cFiles.getLeastAccessedIncompleteFile()
    if f == null then                                       ▷ Only complete files left
        f = cFiles.getLeastAccessedCompleteFile()
    return f.name                                           ▷ File to evict

procedure ADD(Client c, String name, Block bId)
    File f = fileSet.getByName(name)
    if f == null then
        f = new File(name)
        fileSet.add(f)
    f.addLocation(c, bId)                                   ▷ Update properties
```

schedule the tasks operating on a wave. Therefore, despite the potential benefits, to sidestep the above problems, we assign a single value for the wave-width of a file. We define the wave-width of a file as a weighted average across the different observed wave-widths, $\sum w \cdot t\,(w)$. The wave-width is included in the mapping maintained by the coordinator.

A noteworthy approximation to wave-widths is to simply consider small and large jobs instead, based on their input sizes. As Figure 2.8 showed, there is a correlation between input sizes of jobs and their wave-widths. Therefore, such an approximation mostly maintains the relative ordering between small and large waves despite approximating them to small and large job input sizes. We evaluate this approximation in §2.5.3.

2.3.4 LIFE and LFU-F within PACMan

We now describe how LIFE and LFU-F are implemented inside PACMan's coordinated architecture.

The coordinated infrastructure's global view is fundamental to implementing the sticky policy. Since LIFE and LFU-F are global cache replacement policies, they are implemented at the coordinator. Pseudocode 1 describes the steps in implementing LIFE and LFU-F. In the following description, we use the terms file and input interchangeably. If all blocks of a file are cached, we call it a *complete file*; otherwise it is an *incomplete file*. When a client runs out of cache memory it asks the coordinator for a file whose blocks it can evict, by calling FILETOEVICT() (LIFE or LFU-F).

To make this decision, LIFE first checks whether the client's machine caches the blocks of any incomplete file. If there are many such incomplete files, LIFE picks the one with the largest wave-width and returns it to the client. There are two points worth noting. First, by picking an incomplete file, LIFE ensures that the number of fully cached files does not decrease. Second, by picking the largest incomplete file, LIFE increases the opportunity for more small files to remain in cache. If the client does not store the blocks of any incomplete file, LIFE looks at the list of complete files whose blocks are cached by the client. Among these files, it picks the one with the largest wave-width. This increases the probability of multiple small files being cached in future.

LFU-F rids the cache of less frequently used files. It assumes that the future accesses of a file is predicted by its current frequency of access. To evict a block, it first checks if there are incomplete files and picks the *least accessed* among them. If there are no incomplete files, it picks the complete file with the smallest access count.

To avoid cache pollution with files that are never evicted, we also implement a window based aging mechanism. Before checking for incomplete and complete files, both LIFE and LFU-F check whether the client stores any blocks of a file that has not been referred to for at least *window* time period. Among these files, it picks the one that has been accessed the least number of times. This makes it flush out the aged and less popular blocks. In practice, we set the window to be large (e.g., hours), and it has had limited impact on most workloads.

PACMan operates in conjunction with the DFS. However, in practice, they are insulated from the job identifiers that access them. Therefore, we approximate the policy of maximizing the number of whole job inputs to maximizing the number of whole *files* that are present in cache, an approximation that works well (§2.5.3).

Finally, upon caching a block, a client contacts the coordinator by calling ADD(). This allows the coordinator to maintain a global view of the cache, including the access count for every file for implementing LFU-F. Similarly, when a block is evicted, the client calls REMOVE() to update the coordinator's global view. We have omitted the pseudocode for this for brevity.

Pluggable policies: PACMan's architecture is agnostic to replacement algorithms. Its global cache view can support any replacement policy that needs coordination.

2.4 Scarlett: Diffusing Hotspots

Crucial to effective functioning of the cache is co-locating the task on the same machine as its input data. Unfortunately, it is not always possible to co-locate a task with its input data because of *contention* for slots on machines storing the more popular blocks. Simply increasing the replication factor of all the blocks is not a good solution, as data access patterns vary widely in terms of the total number of accesses, the number of concurrent accesses, and the access rate over time. Our analysis of logs from Microsoft Bing's Dryad clusters shows that the top 12% of the most popular data is accessed over ten times more than the bottom third of the data. Some data exhibits high access concurrency, with 18% of the data being accessed by at least three unique jobs at a time.

Contention for slots on machines storing popular data may hurt job performance. If the number of jobs concurrently accessing a popular file exceeds the number of replicas available in cache (typically one, to begin with), some of these jobs may have to access data remotely and/or compete for the same replica. We estimate that as a direct consequence of contentions to popular files, the median duration of jobs increases by 16%.

To avoid contentions and improve memory locality, we design a system, Scarlett, that *replicates files based on their access patterns and spreads them out to avoid hotspots, while minimally interfering with running jobs.* To implement this approach it is critical to accurately predict data popularity. If we don't, we may either create too few replicas thus failing to alleviate contention, or create too many replicas thus wasting both storage and network bandwidth. To guide replication, Scarlett uses a combination of historical usage statistics, online predictors based on recent past, and information about the jobs that have been submitted for execution. To minimize interference with jobs running in the cluster, Scarlett operates within a storage budget, replicates data lazily, and uses compression to trade processing time for network bandwidth. Finally, Scarlett benefits from spreading out the extra replicas, and hence cluster load, on machines and racks that are lightly loaded.

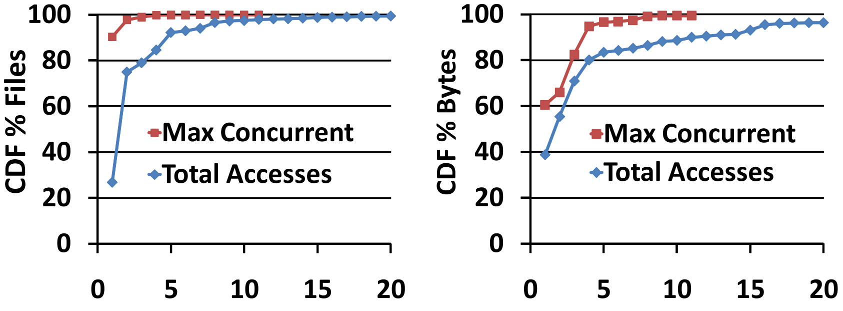
2.4.1 Motivation

In this section, we examine the variation in popularity across files and how popularity changes over time. We also quantify the effects of popularity skew – hotspots in the cluster.

Variation in Popularity

Since accesses to content are made by jobs, we examine popularity at the smallest granularity of content that can be addressed by them. We colloquially refer to this unit as a *file*. In practice, this smallest unit is a collection of many blocks and often has semantic meaning associated with it such as records within a certain time range from a data stream.

There is a large variation among files in their number of accesses as well as in their number of concurrent accesses. Figure 2.12a plots CDFs over files of the total number of



(a) File Accesses (b) Weighted by File Size

Figure 2.12: **Variation in number of jobs accessing and concurrent accesses on input files. The x-axis goes up to a value of 70, we have truncated it to 20 for clarity.**

tasks that access each file and the maximum number of tasks that concurrently access each file. The figure shows that 2.5% of the files are accessed more than 10 times and 1.5% of the files are accessed more than three times concurrently. On the other hand, a substantial fraction of the files are accessed by no more than one task at a time (90%) and no more than once over the entire duration of the dataset (26%).

Files vary in size, so to examine the byte popularity, Figure 2.12b weights each file by its size. We see that 38% of all data is accessed just once in the five-day interval. On the other hand, 12% of the data is accessed more than 10 times, i.e., 12% of the data is 10x more popular than roughly a third of the data. Recall that each block in the file system is replicated thrice. The figure shows that 18% of the data have ≥ 3 concurrent accesses, i.e., are operating *at brim*, while 6% of them have more concurrent accesses than replicas.

When multiple tasks contend for a few replicas, the machines hosting the replicas become hotspots. Even if these tasks ran elsewhere in the cluster, they compete for disk bandwidth at the machines hosting the replicas. When the cluster is highly utilized, a machine can have more than one popular block. Due to collisions between tasks reading different popular blocks, the effective number of replicas per block can be fewer as some of the machines hosting its replicas are busy serving other blocks.

We find high correlation between the total number of accesses and number of concurrent accesses with a Pearson correlation factor of 0.78, implying that either of these metrics is sufficient to capture file popularity.

We also find that larger files experience more accesses. Figure 2.13 bins files by their size, with the largest file having a normalized size of 1, and plots the average number of accesses (total and concurrent) to files in each bin. Owing to their disproportionately high access counts, focusing on just the larger files is likely to yield most of the benefits.

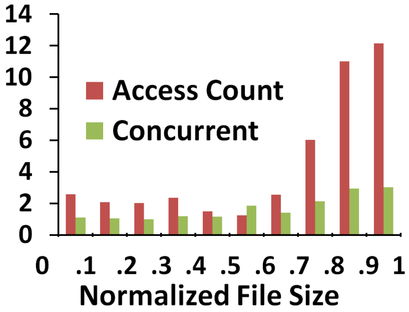


Figure 2.13: **Popularity of files as a function of their sizes, normalized to the largest file; the largest file is of size 1.**

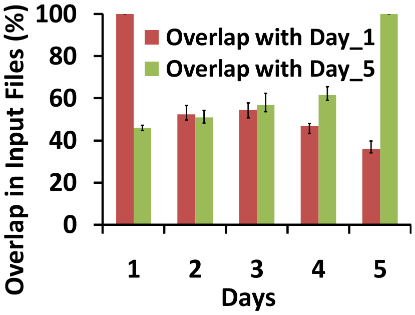


Figure 2.14: **Overlap in files accessed across five days. With the first and fifth day as references, we plot the fraction of bytes accessed on those days that were also accessed in the subsequent and preceding days, respectively.**

Change in Popularity

Files change in popularity over time. Figure 2.14 plots the overlap in the set of files accessed across five consecutive days, with *day-1* and *day-5* as references. We observe a strong day effect – only 50% of the files accessed on any given day are accessed in the next or the previous days. Beyond this initial drop, files exhibit a gradual ascent and decline in popularity. Roughly 40% of the files accessed on day 1 are also accessed four days before or after. The relatively stable popularity across days indicates that prediction techniques that learn access patterns will be effective.

On an hourly basis, however, access patterns exhibit not only the gradual ascent and decline in popularity that we see over days but also periodic bursts in popularity. Figure 2.15 plots hourly overlap in the set of files accessed, with two illustrative reference hours. The figure on top shows gradual variation while the bottom figure shows that some sets of files are accessed in bursts. We conjecture that the difference is due to the types of files involved – the hour on the top likely consists of a time-sensitive set of files used by many different users or groups, so their popularity decays faster and more smoothly, while the bottom hour

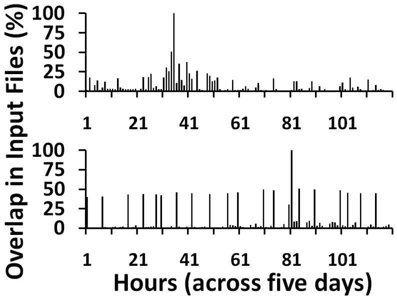


Figure 2.15: **Hourly overlap in the set of files accessed with two sample reference hours (*hour-35* and *hour-82*). The graph on top shows a gradual change while the bottom graph shows periodically accessed files.**

likely consists of a set of files used by fewer but more frequent users explaining the periodic bursts.

Effect of Popularity Skew: Hotspots

When more tasks want to run simultaneously on a machine than that machine can support, we will say a *contention event* has happened. MapReduce frameworks vary in how they deal with contention events. Some queue up tasks, others give up on locality and execute task elsewhere and some others evict the less preferred tasks. Regardless of the coping mechanism, contention events slow down the job and waste cluster resources.

Figure 2.16 plots a CDF of how contentions are distributed across the machines in the cluster. The figure shows that 50% of contentions are concentrated on a small fraction of machines (less than $(\frac{1}{6})^{th}$) in the cluster. Across periods of low and high cluster utilization (5AM and 12PM respectively), the pattern of hotspots is similar.

We attribute these hotspots to skew in popularity of files. Hotspots occur on machines containing replicas of files that have many concurrent accesses. Further, current placement schemes are agnostic to correlations in popularity, they do not avoid co-locating popular files, and hence increase the chance of contentions.

Summary

From analysis of production logs, we take away these lessons for the design of Scarlett:

1. The number of concurrent accesses is a sufficient metric to capture popularity of files.
2. Large files contribute to most accesses in the cluster, so reducing contention for such files improves overall performance.
3. Recent logs are a good indicator of future access patterns.

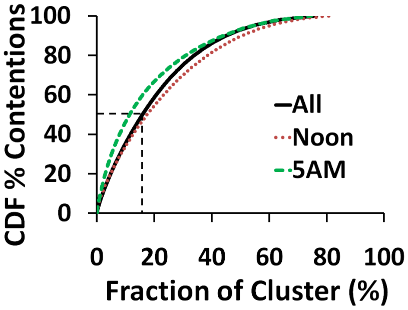


Figure 2.16: **Hotspots: One-sixth of the machines account for half the contentions in the cluster.**

4. Hotspots in the cluster can be smoothened via appropriate placement of files.

2.4.2 Scarlett: System Design

We makes the following two design choices. First, Scarlett considers replicating content at the smallest granularity at which jobs can address content. Recall that we call this a *file*. Scarlett does so because a job will access all blocks or none in a file. Even if some blocks in a file have more replicas, the block(s) with the fewest replicas become the bottleneck, i.e., tasks in the job that access these hot blocks will straggle [19] and hold back the job. Second, Scarlett adopts a *proactive* replication scheme, i.e., replicate files based on predicted popularity. While we considered the reactive alternative of simply caching data when tasks executed non-locally (thereby increasing its replication factor), we discarded it as it was unsuited to handle frameworks that deal with slot contentions using task evictions (e.g., as in Dryad, § 2.4.3). In addition, the choice of replicating at the granularity of files and doing so proactively has the advantage of simplicity of implementation.

Scarlett captures the popularity of files and uses that to increase the replication factor of oft-accessed files, while avoiding hotspots in the cluster and causing minimal interference to the cross-rack network traffic of jobs. To do so, Scarlett computes a replication factor r_f for each file that is proportional to its popularity (§2.4.2) while remaining within a budget on extra storage due to additional replicas. Scarlett smooths out placement of replicas across machines in the cluster so that the expected load on each machine (and rack) is uniform (§2.4.2). Finally, Scarlett uses compression and memoization to reduce the cost to creating replicas (§2.4.2).

Recall that current file systems [6,85] divide files into blocks and uniformly replicate each block three times for reliability. Two replicas are placed on machines connected to the same rack switch, and the third is on a different rack. Placing more replicas within a rack allows tasks to stay within their desired rack. The third replica ensures data availability despite rack-wide failures. Datacenter topologies are such that there is more bandwidth within a rack than across racks [86]. Our analysis in §2.4.1 shows sizable room for improvement over the policy of uniform replication.

Computing File Replication Factor

Scarlett replicates data at the granularity of files. For every file, Scarlett maintains a count of the maximum number of concurrent accesses (c_f) in a *learning window* of length T_L. Once every *rearrangement period*, T_R, Scarlett computes appropriate replication factors for all the files. By default, $T_L = 24$ hours and $T_R = 12$ hours. The choice of these values is guided by observations in the production logs that show relative stability in popularity during a day. It also indicates Scarlett's preference to conservatively replicate files that have a consistent skew in popularity over long periods.

Scarlett chooses to replicate files proportional to their expected usage c_f. The intuition here is that the expected load at a machine due to each replica that it stores be constant – the load for content that is more popular is distributed across a proportional number of replicas. To provide a cushion against under-estimates, Scarlett creates δ more replicas. By default $\delta = 1$. Scarlett lower bounds the replication by one, thereby maintaining at least one copy. Hence the desired replication factor is $\max(c_f + \delta, 1)$.

Scarlett operates within a fixed budget B on the storage used by extra replicas. We note that storage while available is not a free resource, production clusters routinely compress data before storing to lower their usage. How should this budget be apportioned among the various files?

Scarlett employs two approaches. In the *priority* approach, Scarlett traverses the files in descending order of their size and increases each file's replication factor up to the desired value of $c_f + \delta$ until it runs out of budget. The intuition here is that since files with larger size contribute most of the accesses (see Fig 2.13), it is better to spend the limited budget on replication on those files. Pseudocode 2 summarizes this approach. We would like to emphasize that while looking at files by descending order of size is suited for our environment, the design of Scarlett allows any ordering to be plugged in.

The second *round-robin* approach alleviates the concern that most of the budget can be spent on just a few files. Hence, in this approach, Scarlett increases the replication factor of each file by at most 1 in each iteration and iterates over the files until it runs out of budget. Pseudocode 3 depicts this approach. The round-robin approach provides improvements to many more files while the priority approach focuses on just a few files but can improve their accesses by a larger amount. We evaluate both distribution approaches for different values of the budget in §2.5.

The following desirable properties follow from Scarlett's strategy to choose different replication factors for files:

- Files that are accessed more frequently have more replicas to smooth their load over.

Pseudocode 2 **Scarlett computes the file replication factor r_f based on their popularity and budget B. c_f is the observed number of concurrent accesses. Here files with larger size have a strictly higher priority of getting their desired number of replicas.**

```
Used Budget, $B_{used} \leftarrow 0$
$F \leftarrow$ Set of files sorted in descending order of size
Set $r_f \leftarrow 1 \quad \forall f \in F$                              ▷ Base Replication
for file $f \in F$ do
    $r_f \leftarrow \max(c_f + \delta, 1)$                           ▷ Increase $r_f$ to $c_f + \delta$
    $B_{used} \leftarrow B_{used} + f_{size} \cdot (r_f - 1)$
    break if $B_{used} \geq B$
```

Pseudocode 3 **Round-robin distribution of the replication budget B among the set of files F.**

```
Used Budget, $B_{used} \leftarrow 0$
$F \leftarrow$ Set of files sorted in descending order of size
Set $r_f \leftarrow 3 \quad \forall f \in F$                              ▷ Base Replication
while $B_{used} < B$ do
    for file $f \in F$ do
        if $r_f < c_f + \delta$ then
            $r_f \leftarrow r_f + 1$                                  ▷ Increase $r_f$ by 1
            $B_{used} \leftarrow B_{used} + f_{size}$
            break if $B_{used} \geq B$
```

- Together, δ, T_R and T_L track changes in file popularity while being robust to short-lived effects.

- Choosing appropriate values for the budget on extra storage B and the period at which replication factors change T_R can limit the impact of Scarlett on the cluster.

Smooth Placement of Replicas

We just saw which files are worthwhile to replicate but where to place these replicas? A machine that contains blocks from many popular files will become a hotspot, even though as shown above, there may be enough replicas for each block such that the per-block load is roughly uniform. Here, we show how Scarlett smooths the load across machines.

In current and future hardware SKUs, reading from the local disk is comparable to reading within the rack, since top-of-rack switches have enough backplane bandwidth to support all intra-rack transfers. Reading across racks however continues to remain costly due to network over-subscription. Hence, Scarlett spreads replicas of a block over as many racks as possible to provide many reasonable locations for placing the task.

Scarlett's placement of replicas rests on this principle: place the desired number of replicas of a block on as many distinct machines and racks as possible while ensuring that the expected load is uniform across all machines and racks.

A strawman approach to achieve these goals would begin with random circular permutations of racks and machines within each rack. It would place the first replica at the first machine on the first rack. Advancing the rack permutation would ensure that the next replica is placed on a different rack. Advancing to the next machine in this rack ensures that when this rack next gets a replica, i.e., after all racks have taken a turn, that replica will be placed on a different machine in the rack. It is easy to see that this approach smooths out the replicas across machines and racks. The trouble with this approach, however, is that even one change in the replication factor changes the entire placement leading to needless shuffling of replicas across machines. Such shuffling wastes time and cross-rack network bandwidth.

Scarlett minimizes the number of replicas shuffled when replication factors change while satisfying the objective of smooth placement in this way. It maintains a *load* factor for each machine, l_m. The load factor for each rack, l_r, is the sum of load factors of machines in the rack. Each replica is placed on the the rack with the least load and the machine with the least load in that rack. Placing a replica increases both these factors by the expected load due to that replica $-\frac{c_f}{r_f}$. The intuition is to keep track of the current load via the load factor, and make the desired changes in replicas (increase or decrease) such that the lightly loaded machines and racks shoulder more of the load. This approach is motivated by the Deficit Round Robin (DRR) scheme that employs a similar technique to spread load across multiple queues in arbitrary proportions.

Pseudocode 4 shows how, once every T_R, after obtaining a new set of replication factors $r_f^{desired}$, Scarlett places those replicas. Files whose replication factors have to be reduced are processed first so that we get an updated view of the load factors of racks and machines. We defer how replicas are actually created and deleted to the next subsection. Traversing the list of desired replicas, Scarlett places each replica on the next lightly loaded machine and rack. Replicas of the same block are ensured to not end up on a machine or rack that contains another replica.

Creating Replicas Efficiently

Replication of files cause data movement over already over-subscribed cross-rack links [86]. This interferes with the performance of tasks, especially those of network-intensive phases like reduce and joins. A skew in the bandwidth utilization of racks leads to tasks that read data over them lagging behind the other tasks in their phase, eventually inflating job completion times [19]. While our policy of placing one replica per rack makes cross-rack data movement inevitable during replication, we aim to minimize it. The approximation algorithm in Pseudocode 4 takes a first stab by retaining the location of existing blocks. As a next step, we now reduce the interference caused due to replication traffic. In addition to replication traffic running at lower priority compared to network flows of tasks, we employ

Pseudocode 4 Replicating the set of files F with current replication factors r_f to the desired replication factors $r_f^{desired}$. l_m is the current expected load at each machine due to the replicas it stores.

```
for file $f$ in $F$ do
    if $r_f > r_f^{desired}$ then
        Delete Replicas                                    ▷ De-replicate
        Update $l_m$ accordingly
for file $f$ in $F$ do
    while $r_f < r_f^{desired}$ do
        for blocks $b \in f$ do
            $m^* \leftarrow \arg\min(l_m) \forall$ machines not having $b$
            Replicate($b$) at $m^*$
            $l_{m^*} \leftarrow l_{m^*} + \frac{c_f}{r_f}$              ▷ Update load
        $r_f \leftarrow r_f + 1$
```

two techniques that complement each other – (a) equally spread replication traffic across all uplinks of racks, and (b) reduce the volume of replication traffic by trading network usage for computation using compression of data.

While Replicating, Read From Many Sources: We adopt the following simple approach to spread replication traffic equally across all the racks. Suppose the number of replicas increases from r^{old} to r^{new}. The old replicas equally distribute the load of creating new replicas among themselves. Each old replica is a source for $\lceil \frac{r_{new}-r_{old}}{r_{old}} \rceil$ new replicas. In the case of $\frac{r_{new}}{r_{old}} \leq 2$, each rack with old replicas will have only one copy of the block flowing over their uplinks at a time.

When the increase in number of replicas is greater than 2, Scarlett starts from r_{old} and increases the replication factor in steps of two, thereby doubling the number of sources in every step. This strategy ensures that no more than a logarithmic number of steps are required to complete the replication while also keeping the per-link cost for each block being replicated a constant independent of the number of desired replicas.

Compress Data Before Replicating: Recent trends in datacenter hardware and data patterns point to favorable conditions for data compression techniques. Data compression techniques tradeoff computational overhead for network bandwidth [76]. However, the trend of multiple cores on servers (typical datacenter servers now have 8 cores) presents spare cores that can be devoted for compression/decompression purposes. Observations of low utilization in datacenters support this claim [62]. Also, the primary driver of MapReduce jobs are large text data blobs of structured data (e.g., web crawls) [3] which can be substantially compressed.

Libraries for data compression are already used by MapReduce frameworks. Hadoop, the open-source version of MapReduce, includes two compression options [30]. The *gzip* codec implements the DEFLATE algorithm, a combination of Huffman encoding and Lempel-Ziv 1977 (LZ77). The other option is a variant of LZ77 known as LZO. Dryad supports similar

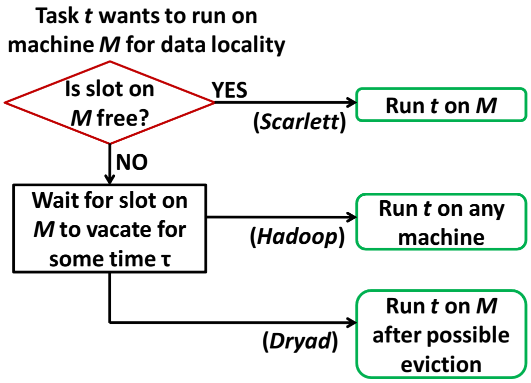

Task *t* wants to run on machine *M* for data locality
Is slot on *M* free?
YES
(*Scarlett*)
Run *t* on *M*
NO
Wait for slot on *M* to vacate for some time τ
(*Hadoop*)
Run *t* on any machine
(*Dryad*)
Run *t* on *M* after possible eviction

Figure 2.17: **Scheduling of tasks, and the different approaches to deal with conflict for slots due to data locality. Scarlett tries to shift the focus to the "YES" part of the decision process by preferentially replicating popular content.**

schemes. Since replication of files is not in the critical path of job executions, our latency constraints are not rigid. With the goal of minimizing network traffic, we employ compression schemes with highest reduction factors albeit at the expense of computational overhead for compression and decompression. We present benchmarks of a few compression schemes as well as the advantages of compressing replication data in §2.5.

Lazy Deletion: Scarlett deletes needless replicas lazily, i.e., by overwriting it when another block or replica that needs to be written to disk. By doing so, the cost to delete is negligible. To ensure that these needless replicas do not contribute unexpected load, they are removed from the list of available replicas.

2.4.3 Effect of Replicas on Frameworks

In this section, we describe how schedulers in frameworks benefit from extra replicas of popular data blocks.

There is a growing trend towards sharing of clusters for economic benefits (e.g., as mentioned in [4]). This naturally raises questions of fairness and MapReduce job managers enforce weighted distribution of resources between the different jobs [56], reflecting the relative importance of jobs. Each job is entitled to a *legitimate* quota of slots. However, each of the frameworks allow jobs to use more than their legitimate share subject to availability, called *bonus* slots. This reduces idling of resources and improves throughput of the cluster.

A natural question that arises is, *how to deal with a bonus task when a legitimate request arrives for its slot?* Job managers confront this question more frequently when dealing with tasks with data locality constraints. Despite the presence of free slots, locality constraints lead to higher contention for certain machines (§2.4.1). Proposed and deployed solutions to dealing with this fall between the following two options: evict the task running on a bonus slot [56], force the newly-arrived task to compromise by running elsewhere [52]. In practice,

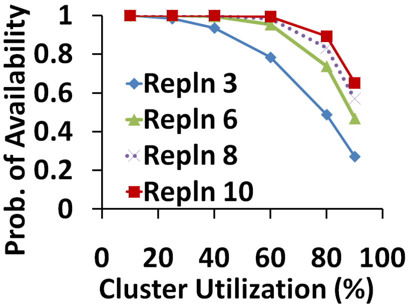


Figure 2.18: **The probability of finding a replica on a free machine for different values of file replication factor and cluster utilization.**

both options wait for a brief period for the bonus task to finish (e.g., as in [104]). Figure 2.17 shows the decision process for contention resolution in different MapReduce frameworks. Each of these solutions are suited to specific environments depending on the service-level agreement constraints, duration of tasks, congestion of network links and popularity skew of input data (elaborated in §2.6). A relative comparison of these solutions is orthogonal to this work; we intend to minimize the occurrence of such contentions in the first place by replicating the popular files thereby ensuring that enough machines with replicas are available.

We start with a simple analysis that demonstrates the intuition behind how increased replication reduces contention. With m machines in the cluster, k of which can be used to run, the probability of finding one of r replicas of a file on the available machines is $1-(1-\frac{k}{m})^r$. This probability increases with the replication factor r, and decreases with cluster utilization $(1-\frac{k}{m})$.

Figure 2.18 plots the results of a numerical analysis to understand how this probability changes with replication factors and cluster utilizations. At a cluster utilization of 80%, with the current replication factor (r=3), we see that the probability of finding a replica among the available machines is less than half. Doubling the replication factor raises the probability to over 75%. Even at higher utilizations of 90%, a file with 10 replicas has a 60% chance of finding a replica on a free machine. By replicating files proportionally to their number of concurrent accesses, Scarlett improves the chances of finding a replica on a free machine.

We now proceed to look at the effect of contention on the Dryad and Hadoop frameworks.

Dryad

Dryad's scheduler pre-emptively evicts a bonus task when a legitimate task makes a request for its slot. In the Dryad cluster under observation, bonus tasks have a 30s notice period before being evicted. Here, we refer to the legitimate and bonus tasks as *evictor* and *evicted* tasks respectively.

Using Dryad logs from Microsoft Bing's clusters, we quantify the magnitude of the

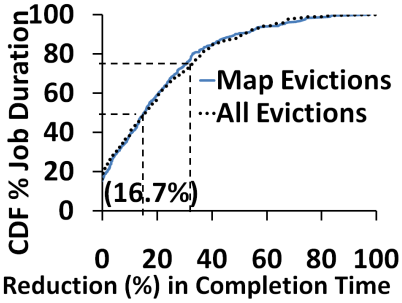


Figure 2.19: **Ideal improvement in job completion times if eviction of tasks did not happen.**

problem due to evictions.

Eviction of Tasks: Of all tasks that run on the cluster, 21.1% of them end up being evicted. Further, an overwhelming majority of the evicted tasks (98.2%) and the evictor tasks (93%) are from map phases. These tasks can execute elsewhere were more replicas available. Reclaiming resources used by killed tasks will reduce the load on the cluster. As a second-order effect, we see from Figure 2.18, that the probability of finding a replica on the available machines improves with lower utilization. In addition, the spare resources can be used for speculative executions to combat outliers.

Inflation of Job Durations: Figure 2.19 plots the ideal improvement in completion times for the jobs in the cluster if all evicted tasks were left to execute to completion and did not have to be re-executed and the evictor tasks achieved locality too. The potential median and third-quartile improvements are 16.7% and 34.1% respectively. In large production clusters, this translates to millions of dollars of savings. We see that the improvements by hypothetically avoiding evictions by only the map tasks is nearly the same (median of 15.2%) as when all evictions are avoided.

Correlation of evictor tasks and files: Figure 2.20 explores the correlation between evictor tasks and the characteristics of the input files they work on – access count, concurrency and size. For clarity, we plot only the top 160 files contributing to evictions, out of a total of 16000 files (or 1%) – these account for 65% of the evictor tasks. As marked in the figure, we see that the files that contribute the most to evictor tasks are directly correlated with popularity – high access counts as well as concurrency. High concurrency naturally leads to contention and eviction while more accesses imply more tasks run overall on them and hence greater probability of evictions. In addition, the worst sources of evictor tasks also are the bigger files. The 1% of the files plotted in Figure 2.20 contribute to a disproportionate 35% of the overall storage's size and account for 65% of all evictions. This validates our design choice in §2.4.2 where we order the files in descending order of size before distributing the replication budget.

That evictions happen even in the presence of idle computational resources points to

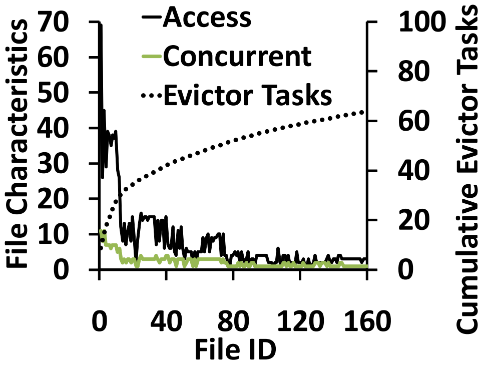


Figure 2.20: **Correlation between file characteristics (y1) and eviction of tasks. We plot only the top 1% of the eviction-causing files for clarity. The cumulative number of evictor tasks are plotted on the right axis (y2). Popular files directly correlate with more evictions. Large files also correlate with evictions – the 1% of the files in this figure account for 35% of the overall storage, and 65% of overall evictions.**

evictions being primarily due to contention for popular data.

Our results in §2.5 show that Scarlett manages to reduce evictions by 83% in Dryad jobs.

Hadoop

Hadoop's policy of dealing with contention for slots is to force the new task to forfeit locality. Delay Scheduling [104] improves on this default policy by making tasks wait briefly before deciding to cede locality. The data from Facebook's Hadoop logs in [104] shows that small jobs (which constitute 58% of all Hadoop jobs) achieve only 5% node locality and 59% rack locality.

As described earlier, Scarlett's replication reduces contention and provides more opportunities for map tasks to attain machine or rack locality. Storing more replicas of popular files provides more machine-local slots (§2.4.2) while spreading out replicas across racks and preventing concentration of popularity (§2.4.2) facilitates rack locality when machine locality is not achievable. Our evaluation in §2.5 shows a 45% increase in locality for map tasks in Hadoop jobs.

We believe that data locality will continue to be a preference in future frameworks built on Hadoop thereby making Scarlett's replication policies generically applicable for contention avoidance of slots.

Bin	Tasks	% of Jobs		% of Resources	
		Facebook	Bing	Facebook	Bing
1	1–10	85%	43%	8%	6%
2	11–50	4%	8%	1%	5%
3	51–150	8%	24%	3%	16%
4	151–500	2%	23%	12%	18%
5	> 500	1%	2%	76%	55%

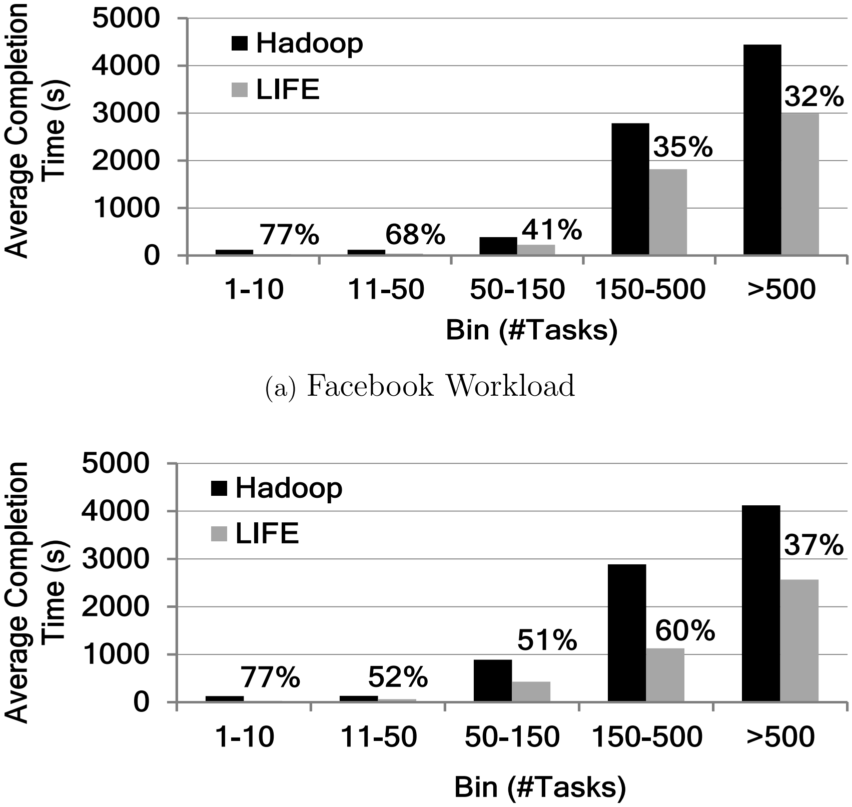
Table 2.1: **Job size distributions. The jobs are binned by their sizes in the scaled-down Facebook and Bing workloads.**

2.5 Evaluation

We built PACMan along with Scarlett's popularity-based replication, and modified HDFS [6] to leverage PACMan's caching service. The prototype is evaluated on a 100-node cluster on Amazon EC2 [2] using workloads derived from the Facebook and Bing traces (§1.3). To compare at a larger scale against a wider set of idealized caching techniques, we use the trace-driven simulator that performs a detailed replay of task logs. We first describe our evaluation setup before presenting our results.

2.5.1 Setup

Cluster: We deploy our prototype on 100 Amazon EC2 nodes, each of them "double-extra-large" machines [2] with 34.2GB of memory, 13 cores and 850GB of storage. PACMan is allotted 20GB of cache per machine; we evaluate PACMan's sensitivity to cache space in §2.5.5.
Compared Schemes: Our implementation and simulator replaced blocks in cache using LIFE and LFU-F,
Metrics: We evaluate PACMan on two metrics that it optimizes – average completion time of jobs and efficiency of the cluster. The baseline for our deployment is Hadoop-0.20.2. The trace-driven simulator compared with currently deployed versions of Hadoop and Dryad.
Job Bins: To separate the effect of PACMan's memory locality on different jobs, we binned them by the number of map tasks they contained in the scaled-down workload. Table 2.1 shows the distribution of jobs by count and resources. The Facebook workload is dominated by small jobs – 85% of them have ≤ 10 tasks. The Bing workload, on the other hand, has the corresponding fraction to be smaller but still sizable at 43%. When viewed by the resources consumed, we obtain a different picture. Large jobs (bin-5), that are only 1% and 2% of all jobs, consume a disproportionate 76% and 55% of all resources. The skew between small and large jobs is higher in the Facebook workload than in the Bing workload.
The following is a summary of our results.



(a) Facebook Workload

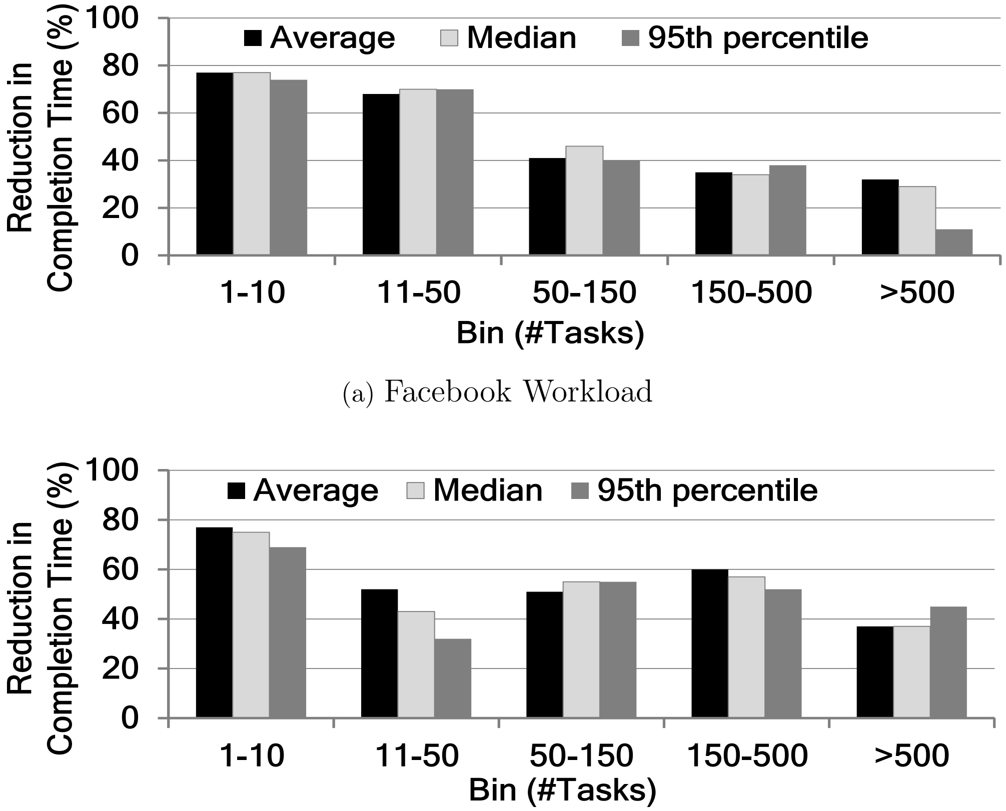
(b) Bing Workload

Figure 2.21: **Average completion times with LIFE, for Facebook and Bing workloads. Relative improvements compared to Hadoop are marked for each bin.**

- Average completion times improve by 53% with LIFE; small jobs improve by 77%. Cluster efficiency improves by 54% with LFU-F (§2.5.2).
- Without the *sticky* policy of evicting from incomplete files, average completion time is 2× more and cluster efficiency is 1.3× worse (§2.5.3).
- LIFE and LFU-F are better than MIN in improving job completion time and cluster efficiency, despite a lower cache hit-ratio (§2.5.4).
- PACMan's performance improves by 15% when Scarlett's selective replication is turned on (§2.5.8).

2.5.2 PACMan's Improvements

LIFE improves the average completion time of jobs by 53% and 51% in the two work-



(a) Facebook Workload

(b) Bing Workload

Figure 2.22: **Distribution of gains for Facebook and Bing workloads. We present the improvement in average, median and 95^{th} percentile completion times.**

loads. As Figure 4.16 shows small jobs (bin-1 and bin-2) benefit considerably. Jobs in bin-1 see their average completion time reduce by 77% with the gains continuing to hold in bin-2. As a direct consequence of the sticky policy, 74% of jobs in the Facebook workload and 48% of jobs in the Bing workload meet the all-or-nothing constraint, i.e.,*all* their tasks being memory local. Large jobs benefit too (bin-5) seeing an improvement of 32% and 37% in the two workloads. This highlights LIFE's automatic adaptability. While it favors small jobs, the benefits automatically spill over to large jobs when there is spare cache space.

Figure 4.17c elaborates on the distribution – median and 95^{th} percentile completion times – of LIFE's gains. The encouraging aspect is the tight distribution in bin-1 with LIFE where the median and 95^{th} percentile values differ by at most 6% and 5%, respectively. Interestingly, the Facebook results in bin-2 and the Bing results in bin-3 are spread tighter compared to the other workload.

LFU-F improves cluster efficiency by 47% with the Facebook workload and 54% with the Bing workload. In large clusters, this translates to significant room for executing more

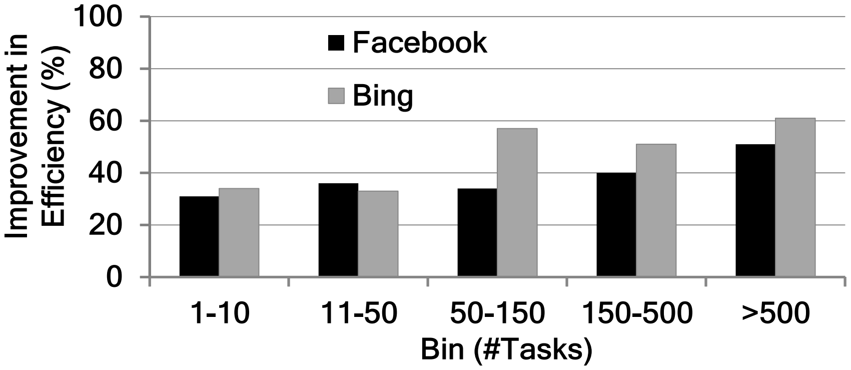


Figure 2.23: **Improvement in cluster efficiency with LFU-F compared to Hadoop. Large jobs contribute more to improving efficiency due to their higher frequency of access.**

Testbed	Scale	LIFE		LFU-F	
		Facebook	Bing	Facebook	Bing
EC2	100	53%	51%	47%	54%
Simulator	$1000's^*$	55%	46%	43%	50%

** Original cluster size*

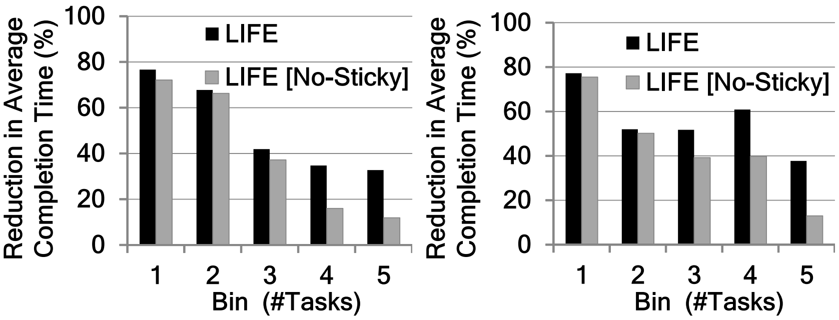
Table 2.2: **Summary of results. We list improvement in completion time with LIFE and cluster efficiency with LFU-F.**

computation. Figure 2.23 shows how this gain in efficiency is derived from different job bins. Large jobs have a higher contribution to improvement in efficiency than the small jobs. This is explained by the observation in §2.3.1 that large files are more frequently accessed.

An interesting question is the effect LIFE and LFU-F have on the metric that is not their target. With LIFE in deployment, cluster efficiency improves by 41% and 43% in the Facebook and Bing workloads. These are comparable to LFU-F because of the power-law distribution of job sizes. Since small jobs require only a little cache space, even after favoring their inputs, there is space remaining for the large (frequently accessed) files. However, the power-law distribution results in LFU-F's poor performance on average completion time. Average completion time of jobs improves by only 15% and 31% with LFU-F. Favoring the large frequently accessed files leaves insufficient space for small jobs, whose improvement has the highest impact on average completion time.

Overall, we see that memory local tasks run 10.8× faster than those that read data from disk.

Simulation: We use the trace-driven simulator to assess PACMan's performance on a larger scale of thousands of machines (same size as in the original clusters). The simulator uses 20GB of memory per machine for PACMan's cache. LIFE improves the average completion



(a) Facebook (b) Bing

Figure 2.24: **Sticky policy. LIFE [No-Sticky] evicts the largest file in cache, and hence is worse off than LIFE.**

times of jobs by 58% and 48% for the Facebook and Bing workloads. LFU-F's results too are comparable to our EC2 deployment with cluster efficiency improving by 45% and 51% in the two workloads. This increases our confidence in the large-scale performance of PACMan as well as our methodology for scaling down the workload. Table 4.2 shows a comparative summary of the results.

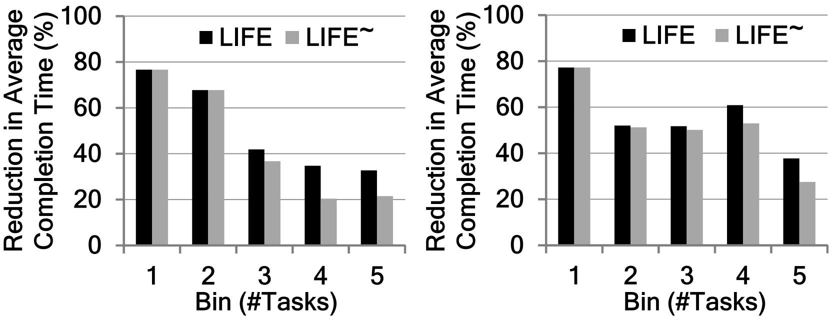
2.5.3 LIFE and LFU-F

In this section, we study different aspects of LIFE and LFU-F. The aspects under focus are the sticky policy, approximation of wave-widths to file size, and using whole file inputs instead of whole job inputs.

Sticky Policy: An important aspect of LIFE and LFU-F is its sticky policy that prefers to evict blocks from already incomplete files. We test its value with two schemes, LIFE[No-Sticky] and LFU-F[No-Sticky]. LIFE[No-Sticky] and LFU-F[No-Sticky] are simple modifications to LIFE and LFU-F to not factor the incompleteness while evicting. LIFE[No-Sticky] evicts the file with the largest wave-width in the cache, LFU-F[No-Sticky] just evicts blocks from the least frequently accessed file. Ties are broken arbitrarily.

Figure 2.24 compares LIFE[No-Sticky] with LIFE. Large jobs are hurt most. The performance of LIFE[No-Sticky] is 2× worse than LIFE in bin-4 and 3× worse in bin-5. Interestingly, jobs in bin-1 are less affected. While LIFE and LIFE[No-Sticky] differ in the way they evict blocks of the large files, there are enough large files to avoid disturbing the inputs of small jobs.

LFU-F[No-Sticky]'s improvement in efficiency is 32% and 39% for the two workloads, sharply reduced values in contrast to LFU-F's 47% and 54% with the Facebook and Bing workloads. These results strongly underline the value of coordinated replacement in PACMan



(a) LIFE∼ vs. LIFE – Facebook

(b) LIFE∼ vs. LIFE – Bing

Figure 2.25: **Approximating LIFE to use file sizes instead of wave-widths. Accurately estimating wave-widths proves important for large jobs.**

by looking at global view of the cache.

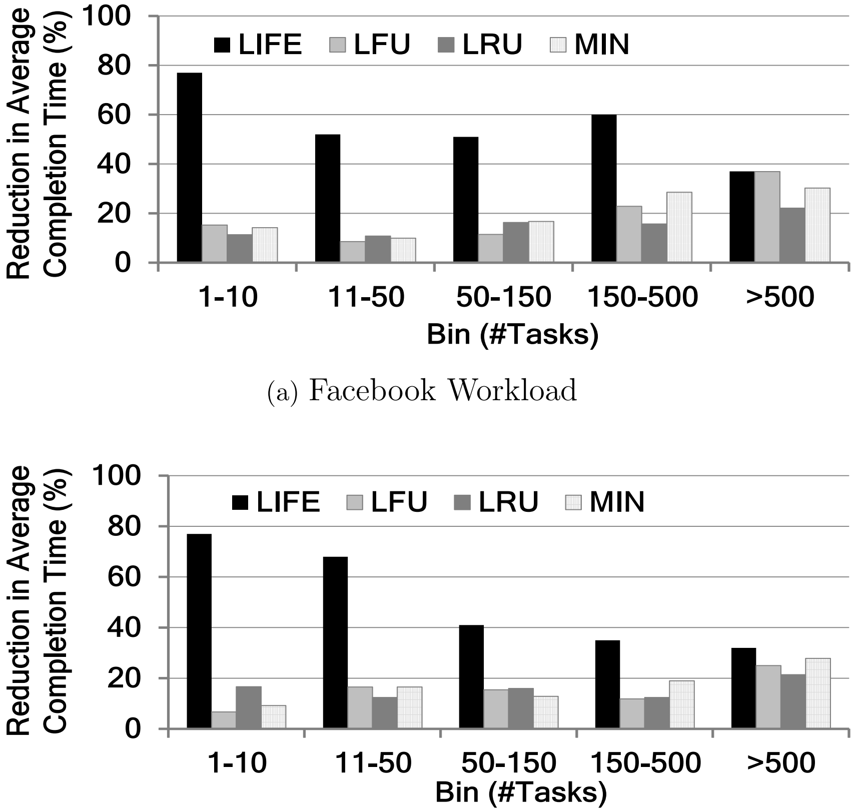
Wave-width vs. File Sizes: As alluded to in §2.3.3, we explore the benefits of using file sizes as a substitute for wave-width. The intuition is founded on the observation in §2.3.1 (Figure 2.8) that wave-widths roughly correlate with file sizes. We call the variant of LIFE that uses file sizes as LIFE∼. The results in Figure 2.25 shows that while LIFE∼ keeps up with LIFE for small jobs, there is significant difference for the large jobs in bin-4 and bin-5. The detailed calculation of the wave-widths pays off with improvements differing by 1.7× for large jobs.

Whole-jobs: Recall from §2.2.2 and §2.3.4 that our desired policy is to retain inputs of as many whole job inputs as possible. As job-level information is typically not available at the file system or caching level, for ease and cleanliness of implementation, LIFE and LFU-F approximate this by retaining as many whole *files* as possible.

Evaluation shows that LIFE is on par with the eviction policy that retains whole job inputs, for small jobs. This is because small jobs typically tend to operate on single files, therefore the approximation does not introduce errors. For larger jobs, that access multiple files, LIFE takes a 11% hit in performance. The difference due to LFU-F using whole files instead of whole job inputs is just a 2% drop in efficiency. The comparable results make us conclude that the approximation is a reasonable trade-off for the significant implementation ease.

2.5.4 Traditional Cache Replacement

Our prototype also implements traditional cache replacement techniques like LRU and LFU. Figure 2.26 and Table 2.3 compare LIFE's performance. Table 2.4 contrasts LFU-F's performance. LIFE outperforms both LRU and LFU for small jobs while achieving



(a) Facebook Workload

(b) Bing Workload

Figure 2.26: **Comparison between LIFE, LFU-F, LFU, LRU and MIN cache replacements.**

comparable performance for large jobs. Likewise, LFU-F is better than LRU and LFU in improving cluster efficiency.

Interestingly, LIFE and LFU-F outperform even MIN [26], the optimal replacement algorithm for cache hit-ratio. MIN deletes the block that is to be accessed farthest in the future. As Figure 2.26 shows, not taking the all-or-nothing constraint of jobs into account hurts MIN, especially with small jobs. LIFE is 7.1× better than MIN in bin-1 and 2.5× better in bin-3 and bin-4. However, MIN's performance for bin-5 is comparable to LIFE. As Table 2.4 shows, the sticky policy also helps in LFU-F outperforming MIN in improving cluster efficiency.

Overall, this is despite a lower cache hit-ratio. This underscores the key principle and differentiator in LIFE and LFU-F – coordinated replacement implementing the sticky policy, as opposed to simply focusing on hit-ratios.

Scheme	Facebook		Bing	
	% Job Saving	Hit Ratio (%)	% Job Saving	Hit Ratio (%)
LIFE	53%	43%	51%	39%
MIN	13%	63%	30%	68%
LRU	15%	36%	16%	34%
LFU	10%	47%	21%	48%

Table 2.3: **Performance of cache replacement schemes in improving average completion times. LIFE beats all its competitors despite a lower hit-ratio.**

Scheme	Facebook		Bing	
	% Cluster Efficiency	Hit Ratio (%)	% Cluster Efficiency	Hit Ratio (%)
LFU-F	47%	58%	54%	62%
MIN	40%	63%	44%	68%
LRU	32%	36%	23%	34%
LFU	41%	47%	46%	48%

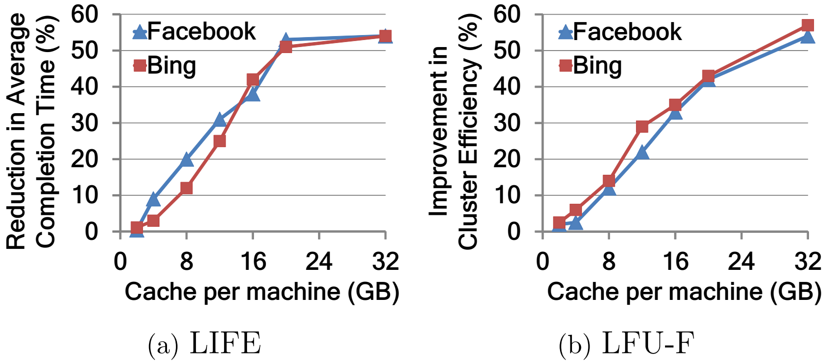
Table 2.4: **Performance of cache replacement schemes in improving cluster efficiency. LFU-F beats all its competitors despite a lower hit-ratio.**

2.5.5 Cache Size

We now evaluate PACMan's sensitivity to available cache size (Figure 2.27) by varying the budgeted cache space at each PACMan client varies between 2GB and 32GB. The encouraging observation is that both LIFE and LFU-F react gracefully to reduction in cache space. As the cache size reduces from 20GB on to 12GB, the performance of LIFE and LFU-F under both workloads hold to provide appreciable reduction of 35% in completion time and 29% improvement in cluster efficiency, respectively.

For lower cache sizes ($\leq$ 12GB), the workloads have a stronger influence on performance. While both workloads have a strong heavy-tailed distribution, recall from Table 2.1 that the skew between the small jobs and large jobs is higher in the Facebook workload. The high fraction of small jobs in the Facebook workload ensures that LIFE's performance drops much more slowly. Even at lower cache sizes, there are sufficient small jobs whose inputs can be retained by LIFE. Contrast with the sharper drop for caches sizes $\leq$ 12GB for the Bing workload.

LFU-F reacts more smoothly to decrease in cache space. Unlike job completion time, cluster efficiency improves even with incomplete files; the sticky policy helps improve it. The correlation between the frequency of access and size of files (§2.3.1), coupled with the fact that the inputs of the large jobs are bigger in the Facebook workload than the Bing workload,



(a) LIFE (b) LFU-F

Figure 2.27: **LIFE's and LFU-F's sensitivity to cache size.**

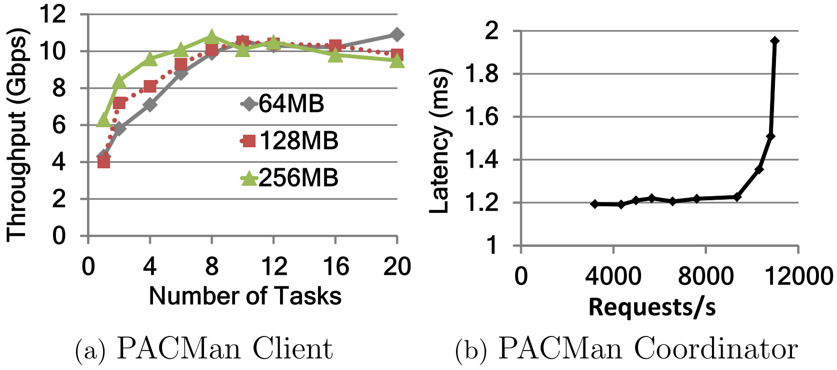
leads to LFU-F's performance deteriorating marginally quicker with reducing cache space in the Facebook workload.

2.5.6 Scalability

We now probe the scalability limits of the PACMan coordinator and client. The client's main functionality is to provide and cache blocks for tasks. We measure the throughput when tasks communicate with the client and latency when clients deal with the coordinator.

PACMan Client: We stress the PACMan client to understand the number of simultaneous tasks it can serve before its capacity saturates. Each task reads a block from the client's cache. Figure 2.28a reports the aggregate throughput for block sizes of 64MB, 128MB and 256MB. For block sizes of 128MB, we see that the client saturates at 10 tasks. Increasing the number of tasks beyond this point results in no increase in aggregate throughput. Halving the block size to 64MB only slightly nudges the saturation point to 12 tasks. We believe this is due to the overheads associated with connection management. Connections with 256MB blocks peak at 8 tasks beyond which the throughput stays constant. The set of block sizes we have tested represent the commonly used settings in many Hadoop installations. Also, since Hadoop installations rarely execute more than 8 or 10 map tasks per machine, we conclude that our client scales sufficiently to handle the expected load.

PACMan Coordinator: Our objective is to understand the latency added to the task because of the PACMan client's communication with the PACMan coordinator. Since we assume a single centralized coordinator, it is crucial that it supports the expected number of client requests (block updates and LIFE eviction). We vary the number of client requests directed at the server per second and observe the average latency to service those requests. As Figure 2.28b shows, the latency experienced by the requests stays constant at ∼1.2ms until 10,300 requests per second. At this point, the coordinator's processing overhead starts increasing the latency. The latency nearly doubles at around 11,000 requests per second. Recently reported research on framework managers [23] show that the number



(a) PACMan Client (b) PACMan Coordinator

Figure 2.28: **Scalability. (a) Simultaneous tasks serviced by client, (b) Simultaneous client updates at the coordinator.**

of requests handled by the centralized job managers of Hadoop is significantly less (3,200 requests/second). Since a task makes a single request to the coordinator via the client, we believe the coordinator scales well to handle expected loads.

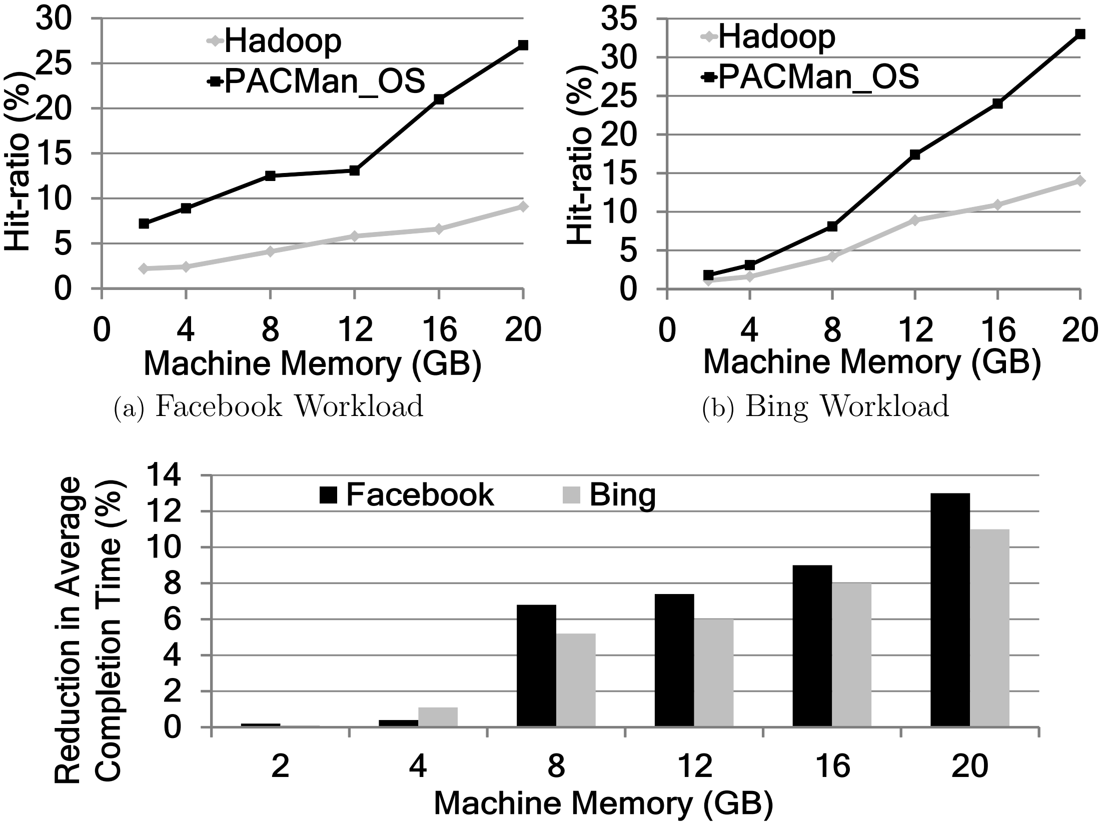
2.5.7 Using File Buffer Cache

We evaluate the extent of achievable gains using only the unmodified local file buffer caches. For this purpose, we use a simplified implementation,PACMan_OS. We turn off the PACMan coordinator and all the clients. The job scheduler in Hadoop itself maintains an estimate of the machines where each data block is cached in the local file buffer caches. After scheduling each task, the job scheduler updates the corresponding input block's cached location to the machine that the task read its data off. Subsequent scheduling of tasks that read the same input is preferably done for memory locality.[4]

Of interest to us is the buffer cache hit-ratio, i.e., *fraction of data that is read from the buffer cache*. We compare PACMan_OS with vanilla Hadoop. Hadoop does not explicitly schedule tasks for memory locality. We verify whether a task read its data from cache or disk using iostat [9] values recorded at the start and end of a task.

Figures 2.29a and 2.29b present hit-ratios for varying memory sizes per machine (2GB to 20GB), indirectly controlling the buffer cache size. Note the low slope of the curve for stock Hadoop – since it is agnostic to cache locations, it does not capitalize on more blocks being cached as memory sizes increase. In both workloads, we see moderate difference in hit-ratios for small cache sizes, while the two curves have a large divergence for higher memory sizes. With limited memory, data is less likely to be retained in cache, so Hadoop's policy of scheduling tasks without being aware of their inputs' cached locations does not cause much harm. However, with larger memory sizes, where more blocks are likely to be retained in

[4] Blocks are assumed to expire off the cache after a time period; 30 minutes gave the maximum hit-ratio in our experiments.



(a) Facebook Workload

(b) Bing Workload

(c) Job Improvement

Figure 2.29: **Using the local file buffer cache. We compare the hit-ratios ((a) and (b)) and reduction in average completion time ((c)), between PACMan_OS and vanilla Hadoop.**

cache, PACMan_OS's targeted scheduling stands out with a 2.2× and 3× difference in the two workloads. A subtle point to note is that since blocks are cached at the source, vanilla Hadoop's performance would be similar to PACMan_OS once all the three replicas of a block are in their respective buffer caches. However, since majority of blocks have an access count of ≤ 4, we do not have the luxury of cache misses (§2.3.1).

Notwithstanding the improved hit-ratio, we do not observe much improvement in completion times (Figure 2.29c). This supports our assertion that improving hit-ratios do not necessarily accelerate parallel jobs.

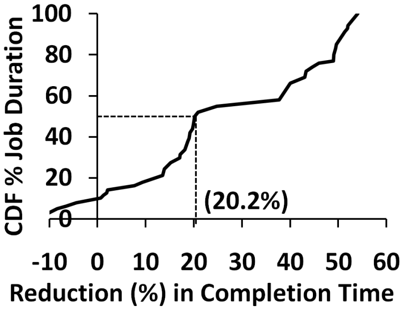


Figure 2.30: **Improvement in data locality for tasks leads to median and third-quartile improvements of 20.2% and 44.6% in job completion times, with forfeiting in place.**

2.5.8 Scarlett: Locality Improvement

Finally, we measure the benefits due to Scarlett's selective replication of popular data blocks. For this measurement, we use PACMan with LIFE as our baseline. We set Scarlett's parameters as follows: $\delta = 1$, let T_L range from 6 to 24 hours, set storage budget $B = 10\%$ and rearrange once at the beginning of the ten hour run ($T_R \geq 10$ hours). As described in §2.4.3, we implement both reactions when the requested slot is currently in use: *forfeit* locality after a brief wait as well as *evict* running tasks.

Figure 2.30 plots the reduction in completion times of jobs. We see that completion times improve by 20.2% and 44.6% at the median and 75^{th} percentile respectively. This is explained by the increase in fraction of map tasks that achieve locality. The fraction of map tasks that achieve locality improves from 57% with vanilla HDFS to 83% with Scarlett, in other words a 45% improvement.

With eviction in place, replicating popular files reduces the necessity for eviction and wastage of work, in turn leading to jobs completing faster. The ideal improvement when all evictions by map tasks are avoided is 15.2% at median, and Scarlett produces a 12.8% median improvement. Here, we set $\delta = 1$, $T_R = 12$ hours, let T_L range from 6 to 24 hours, and set storage budget $B = 10\%$. Figure 2.31 compares the ideal case with our replication scheme where we obtain 84% of ideal performance at median. The ideal case contains no evictions and at the same time assumes that all evictor tasks achieve locality.

A closer look reveals that by replicating popular content, Scarlett avoids 83% of all evictions, i.e., the evictor tasks could be run on another machine containing a replica of their input. Note that this number goes up to 93%, when we consider evictions by tasks operating on the top hundred popular files, confirming the design choice in Scarlett to focus on the more popular files. Increasing the storage budget provides marginal (but smaller improvements) – with an increased storage budget of 20% Scarlett prevents 96% of all evictions.

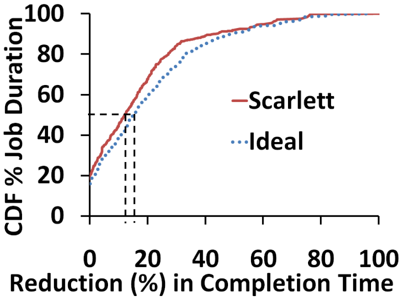


Figure 2.31: **Increased replication reduces eviction of tasks and achieves a median improvement of 12.8% in job completion times or 84% of ideal, with eviction in place.**

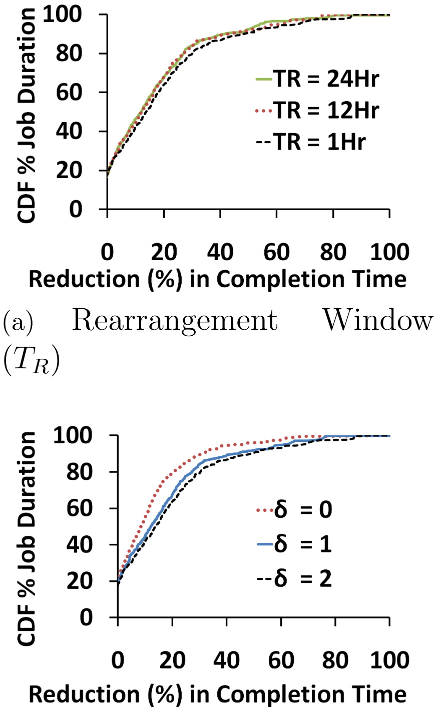
Sensitivity Analysis

We now analyze the sensitivity of Scarlett to the parameters of our learning algorithm– rearrangement window, T_R, and the cushion for replication, δ. T_R decides how often data is moved around, potentially impacting network performance of currently running jobs. δ results in greater storage occupancy and more replication traffic.

Figure 2.32a compares the improvement in job completion times (with eviction) for different rearrangement windows, i.e., $T_R = \{1, 12, 24\}$ hours. Interestingly, we see that T_R has little effect on the performance of jobs. Re-evaluating replication decisions once a day is only marginally worse than doing it once every hour. This points to Scarlett's minimal interference on running jobs. It also points to the fact that most of the gains in the observed workload accrue from replicating files that are consistently popular over long periods. Results with forfeiting in place are similar. For T_R values of 1, 5 and 10 hours, the median improvements are 21.1%, 20.4% and 20.2% respectively. By default, we set T_R to 12 hours, or rearrange files twice a day.

The replication allowance δ impacts performance. Changing δ from 0 to 1 improves performance substantially, but larger values of δ have lower marginal increases. Figure 2.32b shows that for δ values of 0, 1 and 2, the median reductions in job durations are 8.5%, 12.8% and 13.8%. Note the improvement of 42% as δ changes from 0 to 1. Likewise, with forfeiting in place, jobs see a 56% increase from 12.9% to 20.2% in median improvement in completion time as we shift δ from 0 to 1. We believe this is because operating at the brim with a replication factor equal to the observed number of concurrent accesses is inferior to having a cushion, even if that were only one extra replica.

Figure 2.33 shows the cost of increasing δ. Values of storage overhead change upon replication, i.e., once every T_R=12 hours. $\delta = 2$ results in a 24% increase in storage, almost double of the overhead for $\delta = 1$. Combined with the fact that we see the most improvement when moving from δ from 0 to 1, we fix δ as 1.



(a) Rearrangement Window (T_R)

(b) Replication Allowance (δ)

Figure 2.32: **Sensitivity Analysis of T_R and δ. Rearranging files once or twice a day is only marginally worse than doing it at the end of every hour. We set T_R as 12 hours in our system. On the other hand, δ plays a vital role in the effectiveness of Scarlett's replication scheme.**

Storage Budget for Replication

Figure 2.34a plots reduction in job completion times for various budget values. Here, we use the priority distribution, i.e., larger files are preferentially replicated over smaller files within the budget. A budget of 10% improves performance substantially (by 88%) over a 5% limit. As expected, the lower budget reduces storage footprint at the expense of fewer files being replicated or a smaller replication factor for some files. The marginal improvement is smaller as the budget increases to 15%. This indicates that most of the value from replication accrues quickly, i.e., at small replication factors for files. Conversations with datacenter operators confirm that 10% is a reasonable increase in memory usage for storing inputs.

Note however that the improvement going from a budget of 2% to a budget of 5% is smaller than when going from 5% to 10%. This is likely because the distribution policy used by Scarlett is simple and greedy but not optimal. Likely, there are some files, replicating which yields significantly more benefit per unit extra storage, that Scarlett fails to replicate when budgets are small. However, these inefficiences go away with a slightly larger budget

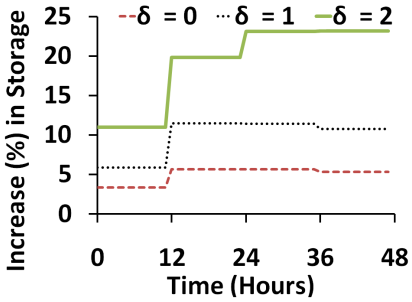


Figure 2.33: **Increasing the value of the replication allowance (δ) leads to Scarlett using more storage space. We fix δ as 1.**

value of 10%, and we choose to persist with the simpler algorithm.

With forfeiting in place, jobs (Figure 2.34b) exhibit a similar trend. The increase in median completion time when moving from a budget of 5% to 10% is much higher (120%). This indicates that how Hadoop deals with contentions (by moving tasks elsewhere) is likely more sensitive to the loss of locality when popular files are not replicated.

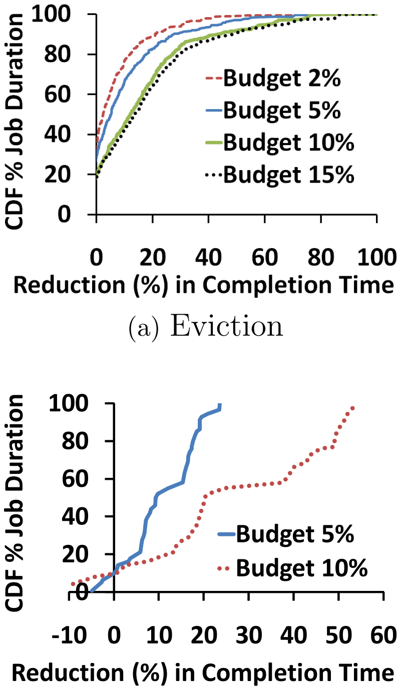
Priority vs. Round-robin Distribution: Recall from §2.4.2 that the replication budget can be spread among the files either in a priority fashion – iterate through the files in decreasing order of size, or distributed iteratively in a round-robin manner. Figure 2.35a plots the performance of jobs with respect to both these allocations. For a replication budget of 10%, we observe that the priority allocation gives a median improvement of 12.8% as opposed to 8.4% with round-robin allocation, or a 52% difference. This is explained by our causal analysis in Figure 2.13 and Figure 2.20 that shows that large files account for a disproportionate fraction of the evicting tasks while also experiencing high levels of concurrent accesses. Hence, giving them a greater share of the replication budget helps avoid more evictions. Hadoop jobs exhibit a greater difference of 63% between the two distributions showing greater sensitivity to loss of locality (Figure 2.35b).

However, the difference in advantage between the two distributions are negligible at small replication budgets. As we see in Figure 2.35a, the limited opportunity to replicate results in there being very little to choose between the two distribution strategies.

Increase in Network Traffic

For $\delta = 1$, the maximum increase in uncompressed network traffic during rearrangement of replicas is 24%. Using the PPMVC compression scheme [90], this reduces to an acceptable overhead of 0.9%.

We also present micro-benchmarks of various compression techniques. Table 2.5 lists the compression and de-compression speeds as well as the compression ratios achieved by a



(a) Eviction

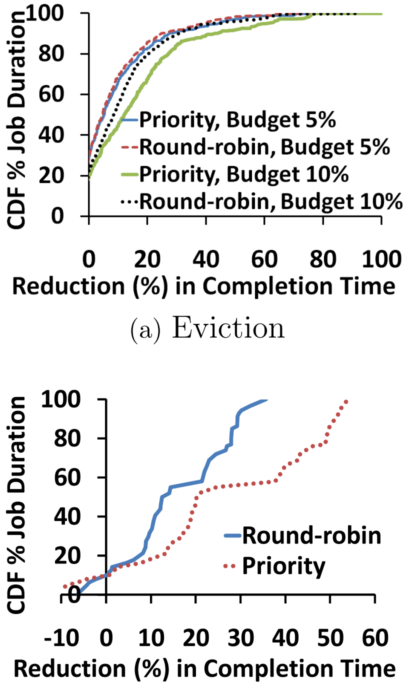
(b) Forfeit

Figure 2.34: **Low budgets lead to little fruitful replication. On the other hand, as the graph below shows, budgets cease to matter beyond a limit.**

Scheme	Throughput (Mbps)		Compression
	Compress	De-compress	Factor
gzip	144	413	12-13X
bzip2	9.7	88.2	19-20X
LZMA	3.6	375	22-23X
PPMVC	30.2	31.4	26-27X

Table 2.5: **Comparison of the computational overhead and compression factors of compression schemes.**

few compression algorithms [76, 90]. There is a clear trend of more computational overhead providing heavier compression. Given the flexible latency constraints for replication and low bandwidth across racks, Scarlett leans toward the choice that results in the least load on the network.



(a) Eviction

(b) Forfeit

Figure 2.35: **Priority distribution of the replication budget among the files improves the median completion time more than round-robin distribution.**

2.6 Related Work

There has been a humbling amount of work on in-memory storage and caches. While our work borrows and builds up on ideas from prior work, the key differences arise from dealing with parallel jobs that led to a coordinated system that improved job completion time and cluster efficiency, as opposed to hit-ratio.

RAMCloud [57] and prior work on databases such as MMDB [51] propose storing all data in RAM. While this is suited for web servers, it is unlikely to work in data-intensive clusters due to capacity reasons – Facebook has 600× more storage on disk than aggregate memory. Our work thus treats memory as a constrained cache.

Global memory systems such as the GMS project [11], NOW project [12] and others [40] use the memory of a remote machine instead of spilling to disk. Based on the vast difference between local memory and network throughputs, PACMan's memory caches only serves tasks on the local node. However, nothing in the design precludes adding a global memory view. Crucially, PACMan considers job access patterns for replacement.

Web caches have identified the difference between byte hit-ratios and request hit-ratios, i.e., the value of having an entire file cached to satisfy a request [22,31,78]. Request hit-ratios are best optimized by retaining small files [96], a notion we borrow. We build up on it by addressing the added challenges in data-intensive clusters. Our distributed setting, unlike web caches, necessitate coordinated replacement. Also, we identify benefits for partial cache hits, e.g., large jobs that benefit with partial memory locality. This leads to more careful replacement like evicting parts of an incomplete file. The analogy with web caches would not be a web request but a web *page* – collection of multiple web objects (.gif, .html). Web caches, to the best of our knowledge, have not considered cache replacment to optimize at that level.

LIFE's policy is analogous to servicing small requests in queuing systems, e.g., web servers [69]. In particular, when the workload is heavy-tailed, giving preference to small requests hardly hurts the big requests.

Distributed filesystems such as Zebra [53] and xFS [21] developed for the Sprite operating system [74] make use of client-side in-memory block caching, also suggesting using the cache only for small files. However, these systems make use of relatively simple eviction policies and do not coordinate scheduling with locality since they were designed for usage by a network of workstations.

Cluster computing frameworks such as Piccolo [81] and Spark [72] are optimized for iterative machine learning workloads. They cache data in memory after the first iteration, speeding up further iterations. The key difference with PACMan is that since we assume no application semantics, our cache can benefit multiple and a greater variety of jobs. We operate at the storage level and can serve as a substrate for such frameworks to build upon.

The principle of "replication and placement" of popular data has been employed in different contexts in prior work. Our contributions are to (i) identify (and quantify) the content popularity skew in the MapReduce scenario using production traces, (ii) show how the skew causes contention in two kinds of MapReduce systems (ceding locality for Hadoop vs. eviction for Dryad), and (iii) design solutions that operate under a storage budget for large data volumes common in MapReduce systems. While we have evaluated Scarlett primarily for map tasks, we believe the principle of proactively replicating content based on its expected concurrent access can be extended to intermediate data too (e.g., as in Nectar [50] that stores intermediate data across jobs).

Much recent work focuses on the tussle between data locality and fairness in MapReduce frameworks. Complementary to Scarlett, Quincy [56] arbitrates between multiple jobs. Delay Scheduling [104] on the other hand supports temporary relaxation of fairness while tasks wait to attain locality. This can alleviate contention by steering tasks away from a hotspot. It however makes some assumptions that do not hold universally: (a) task durations are short and bimodal, and (b) one task queue per cluster (as in Hadoop). In the Cosmos clusters at Microsoft, tasks are longer (median of 145s as opposed to the 19s in [104]) to amortize overheads in maintaining task-level state at the job manager, copying task binaries etc. Task lengths are also more variable in Dryad owing to diversity in types of phases. Finally, Dryad uses one task-queue per machine, further reducing the load at the job scheduler to improve scalability. Scarlett does not rely on these assumptions and addresses the root cause of contention by identifying and replicating popular content. Furthermore, Scarlett can be beneficially combined with both Delay Scheduling and Quincy.

The idea of replicating content in accordance to popularity for alleviating hotspots has been used in the past. Caching popular data and placing it closer to the application is used in various content distribution networks (CDNs) [1, 5] in the Internet. Beehive [82] proactively replicates popular data in a DHT to provide constant time look-ups in peer-to-peer overlays. Finally, dynamic placement of popular data has also been recently explored in the context of energy efficiency [95]. To the best of our knowledge, ours is the first work to understand popularity skew and explore the benefits of dynamic data replication in MapReduce clusters. The context of our work is different as file access patterns and sizes in MapReduce significantly differ from web access patterns. It differs from Beehive due to the different application semantics. While Beehive is optimized for lookups, Scarlett aims at parallel computation frameworks like MapReduce. Further, our main goal is to increase performance rather than be energy efficient, so we aim for spreading data across nodes as opposed to compaction.

Bursts in data center workloads often result in peak I/O request rates that are over an order of magnitude higher than average load [35]. A common approach to deal with such bursts is to identify overloaded nodes and offload some of their work to less utilized nodes [35, 44]. In contrast, our approach is geared towards a read-heavy workload (unlike [35]), common to MapReduce clusters. While Dynamo [44] reactively migrates (not replicate) popular data, we replicate and do so proactively, techniques more suited to our setting. Recent work [27,92] on providing semantic context to the file system can be leveraged to implement our replication policies.

A wide variety of work has also been done in the area of predictive pre-fetching of popular files based on historical access patterns [33] as well as elaborate program and user based file prediction models [101]. However, these are in the context of individual systems and deal with small amounts of data unlike our setting with petabytes of distributed storage, the replication and transfer of which require strict storage/network constraints.

Some prior work on dynamic database replication policies [91] is very similar in flavor to ours. However, these policies are reactive in reference to application latency requirements. Our work, on the other hand, focuses on designing proactive replication policies.

Finally, much recent work has gone into designs for full bisection bandwidth networks. By suitably increasing the numbers of switches and links in the core, these designs ensure that the network will not be the bottleneck for well-behaved traffic [15, 66]. Well-behaved refers to the hose model constraint, which requires the traffic incoming to each machine to be no larger than the capacity on its incoming network link. We note that Scarlett's benefits remain even if networks have full bisection bandwidth, since concurrent access of blocks results in a bottleneck at the source machine that stores them. By providing more replicas (as many as the predicted concurrent access), Scarlett alleviates this bottleneck.

Chapter 3

Intermediate Data

3.1 Introduction

Data-intensive jobs typically consist of DAGs of tasks with downstream tasks reading the outputs written by upstream tasks. The communication patterns between tasks can be varied ranging from simple one-to-one communication all the way to an all-to-all transfer. Example scenarios are aggregation of outputs from a subset of upstream tasks (perhaps, within a rack switch) to an shuffle for producing the final result (as with reduce tasks reading map outputs in MapReduce [36] computations). The ability to construct such DAGs of computations enables frameworks to support rich and varied applications.

A significant fraction of the lifetime of jobs is spent on intermediate phases, of which, transferring the data down the DAG occupies a big part. Analysis of the Facebook jobs logs show that 20% to 44% of a job's duration is spent on intermediate phases. The numbers are similar for Bing's Dryad jobs. As a result, inefficiencies in intermediate phases delay job completion. Of particular interest to us are network and disk based inefficiencies while reading intermediate data.

Transferring data over the network can result in flows of a job facing contention with other flows in the cluster as well as amongst themselves. A job relies on the completion of multiple network flows, often a single task involves reading data from different places. It follows from the all-or-nothing property described earlier that even if a subset of the flows traverse contended routes in the network, the overall job's completion suffers. Even with reduction in over-subscription factors of datacenter networks [42], there is considerable transient variation in congestion among different paths.

In our solution, Mantri, we maintain information about the currently active flows. New tasks are scheduled such that their flows face the least contention while also taking care to equalize the contention faced by all the tasks. We develop a detailed analytical model for our task placement and approximate the NP-Hard problem with a simple heuristic suited for online schedulers.

Another key aspect that influences the transfer of intermediate data is reliability of

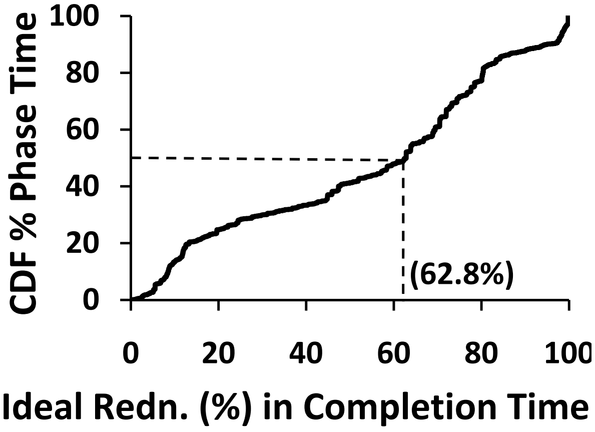


Figure 3.1: **For reduce phases, the reduction in completion time over the current placement by placing tasks in a network-aware fashion.**

accessing them. For reasons of efficiency, intermediate data is not stored in the distributed file system, and hence not replicated. Cluster frameworks make this decision in the interest of causing transient surges in storage as well as network traffic. In fact, if all intermediate data were to be replicated, the overall network traffic in clusters will increase by 22% with short-term spikes as high as 58%. The downside to not replicating the data, however, is that loss of reliability. When the intermediate output is inaccessible, there is no copy to fall back.

Such inaccessibility can occur due to reasons of either disk unreliability (on problematic machines) or unresponsiveness (on overloaded machines). The former occurs due to erratic disk behaviors while the latter results in client processes catering the intermediate data not responding even when the data is present. Consequently, the only option is to *recompute* the upstream task that produced the output. The corresponding downstream task(s) wait for the recomputation to complete before proceeding. Naturally, this leads to them lagging behind the other tasks.

We selectively replicate intermediate outputs by both weighing the benefit due to avoiding recomputation to the cost incurred (time and resources) in replication. Mantri uses simple estimates to model the network costs as well as predict impending failures in machines. Evaluations show that Mantri is able to mitigate the effect of most of the recomputations.

3.2 Workload Analysis

We first quantify the deficiencies in state-of-the-art techniques in handling intermediate data. We cover both techniques for handling network flows while reading the intermediate data (e.g., reduce tasks) as well as the reliability of the intermediate data that is typically stored without replication.

Crossrack Traffic: We find that reduce phases contribute over 70% of the cross rack

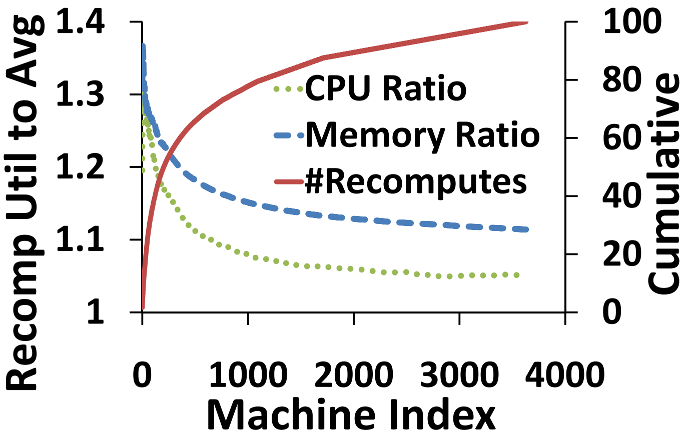


Figure 3.2: **The ratio of processor and memory usage when recomputations happen to the average at that machine (y1). Also, the cumulative percentage of recomputations across machines (y2).**

traffic in the cluster, while most of the rest is due to joins. We focus on cross rack traffic because the network links upstream of the racks have less bandwidth than the cumulative capacity of servers in the rack.

We find that crossrack traffic leads to outliers in two ways. First, in phases where moving data across racks is avoidable (through locality constraints), a task that ends up in a disadvantageous network location runs slower than others. Second, in phases where moving data across racks is unavoidable, not accounting for the competition among tasks within the phase (self-interference) leads to outliers. In a reduce phase, for example, each task reads from every map task. Since the maps are spread across the cluster, regardless of where a reduce task is placed, it will read a lot of data from other racks. Current implementations place reduce tasks on any machine with spare slots. A rack that has too many reduce tasks will be congested on its downlink leading to outliers.

Figure 3.1 compares the current placement with an ideal one that minimizes the cost of network transfer. When possible it avoids reading data across racks, and if not, places tasks such that their competition for bandwidth does not result in hotspots. In over 50% of the jobs, reduce phases account for 17% of the job's lifetime. For the reduce phases, the figure shows that the median phase takes 62% longer under the current placement.

Bad and Busy Machines: We rarely find machines that persistently inflate runtimes. Recomputations, however, are more localized. Half of them happen on 5% of the machines in the cluster. Figure 3.2 plots the cumulative share of recomputes across machines on the axes on the right. The figure also plots the ratio of processor and memory utilization during recomputes to the overall average on that machine. The occurrence of recomputes is correlated with increased use of resources by at least 20%. The subset of machines that triggers most of the recomputes is steady over days but varies over weeks, likely indicative

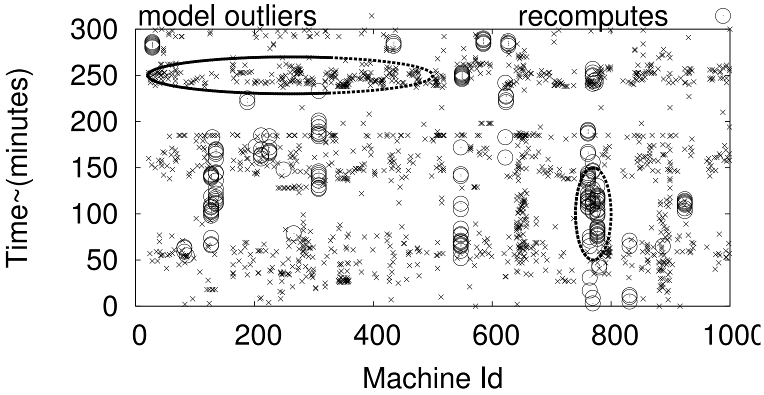


Figure 3.3: **Clustering recomputations and outliers across time**

of changing hotspots in data popularity or corruption in disks [24].

Figure 3.3 investigates the occurrence of "spikes" in outliers. We find that runtime outliers (shown as stars) cluster by time. If outliers were happening at random, there should not be any horizontal bands. Rather it appears that jobs contend for resources at some times. Even at these busy times, other lightly loaded machines exist. Recomputations (shown as circles) cluster by machine. When a machine loses the output of a task, it has a higher chance of losing the output of other tasks.

Rarely does an entire rack of servers experience the same anomaly. When an anomaly happens, the fraction of other machines within the rack that see the same anomaly is less than $\frac{1}{20}$ for recomputes, and $\frac{4}{20}$ for runtime with high probability. So, it is possible to restart a task, or replicate output to protect against loss on another machine within the same rack as the original machine.

3.3 Network-Aware Placement

Reduce tasks, as noted before (§3.2), have to read data across racks. A rack with too many reduce tasks is congested on its downlink and such tasks will straggle. Figure 3.4 illustrates a scenario where placing more reduce tasks on a rack leads to congestion. Input (partitioned) is evenly distributed across three racks and each reduce task consumes its share of the input from each rack. We have three reduce tasks to schedule across racks with equal bandwidths. Assuming a partition size of p and equal uplink and downlink bandwidths of b, the time taken for data transfer for the placement in the left is $4p/b$ whereas it is $2p/b$ when tasks are spread out (right). Our intuition is to evenly distribute the load across the links avoiding bottlenecks and considering network congestion.

Mantri approximates the optimal placement that relieves congestion by the following greedy local algorithm. For a reduce phase, with n reduce tasks, Mantri determines $r = \{r_i\}$, $i = 1, \cdots, n$, the fraction of tasks assigned to rack i. Assume the fraction of map phase output in the i'th rack to be d_i and the available bandwidths on the uplink and downlink to be u_i

and v_i respectively. For each i, compute two terms $c_{2i-1} = \frac{(1-r_i)d_i}{u_i}$ and $c_{2i} = \frac{r_i(1-d_i)}{v_i}$. The first term is the ratio of outgoing traffic and available uplink bandwidth, and the second term is the ratio of incoming traffic and available downlink bandwidth. The algorithm computes the vector $r = \arg\min_r \max_j c_j,\ j = 1, \cdots, 2n$, that minimizes the maximum data transfer time.

Note that the sizes of the map outputs in each rack, d_i, are known to the scheduler prior to placing the tasks of the subsequent reduce phase.

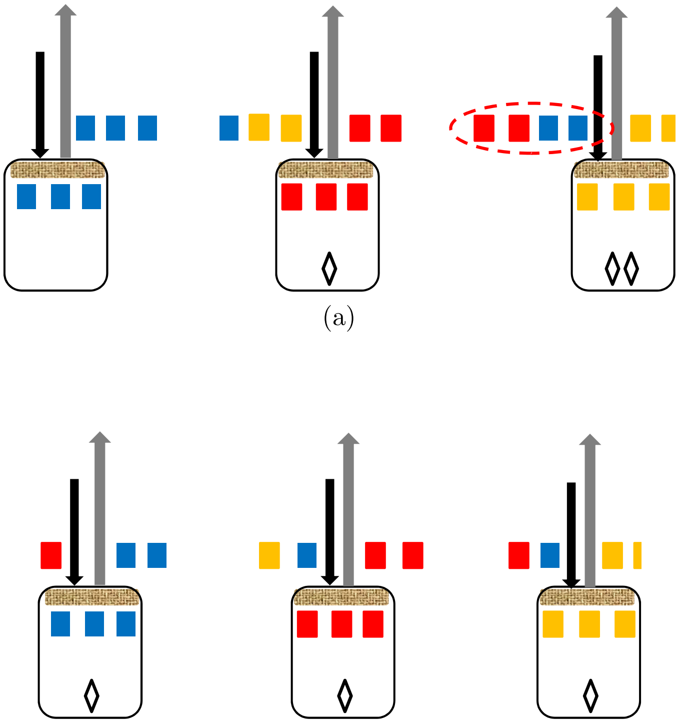
The available bandwidths u_i and v_i change with time and as a function of other jobs in the cluster. Rather than track the changes as an oracle could, Mantri estimates the bandwidths as follows. Phases with small amount of data finish fast, and the bandwidths can be assumed to be constant throughout the execution of the phase. Phases with large amount of data take longer to finish, and the bandwidth averaged over their long lifetime is assumed to be equal for all links. With these estimates, Mantri's placement comes close to the ideal in our experiments (see §3.5.1).

For phases other than reduce, Mantri complements the Cosmos policy of placing a task close to its data [86]. By accounting for the cost of moving data over low bandwidth links in t_{new}, Mantri ensures that no copy is started at a location where it has little chance of finishing earlier thereby not wasting resources.

3.4 Avoiding Recomputation

To mitigate costly recomputations that stall a job, Mantri protects against interim data loss by replicating task output. Mantri acts early by replicating those ouputs whose cost to recompute exceeds the cost to replicate. Mantri estimates the cost to recompute as the product of the probability that the output will be lost and the time to repeat the task. The probability of loss is estimated for a machine over a long period of time. The time to repeat the task is t_{new} with a recursive adjustment that accounts for the task's inputs also being lost. The cost to replicate is the time to move the data to another machine in the rack. Figure 3.5 illustrates the calculation of t_{new} that takes the machines' recompute probabilities (r_i's), time taken by the tasks (t_i's) and recursively looks at prior phases. From our model, expected cost of recomputation automatically decreases if data is replicated (left) as both replicas have to be unavailable for recomputation, and increases if the input pattern is many-to-one (middle) as the unavailability of any of the inputs stalls the task. We assume that in the event of multiple inputs being lost, they can be recomputed in parallel since the overall number of recomputes is small (Figure 4.2a). Recursively looking at prior phases (right) makes us resilient to data loss across any of them.

To avoid excessive replication, Mantri limits the amount of data replicated to 10% of the data processed by the job. This limit is implemented by granting tokens proportional to the amount of data processed by each task. Task output that satisfies the above cost-benefit



(a)

(b)

Figure 3.4: **Placement of reduce tasks across three racks. The rectangle blocks indicate data partitions and rhombus boxes are reduce tasks. Placement (a) results in the uplinks of unutilized racks as bottlenecks, marked with a dotted circle. Placement (b) evenly spreads out the network load.**

check is replicated only if an equal number of tokens are available. Tokens are deducted upon each replication.

In §3.2, we saw that recomputations manifest closely spaced in time. When a recompute is triggered by a machine due to its failure to serve data, Mantri considers this is as the onset of a temporal problem and anticipates future requests to this machine to fail. Hence, all the unread task outputs on that machine (of any job) is proactively generated by scheduling *speculative recomputations* of the corresponding tasks elsewhere. Such speculative recomputations either decrease or eliminate the wait time of a dependent task for lost input to be regenerated.

Replication and speculative recomputation complement each other - long-term failure probabilities of machines decide on replication, and speculative recomputation makes Mantri agile to short-term errors - and well approximate the ideal scheme in our evaluation (§3.5.2), in addition to judicious usage of resources.

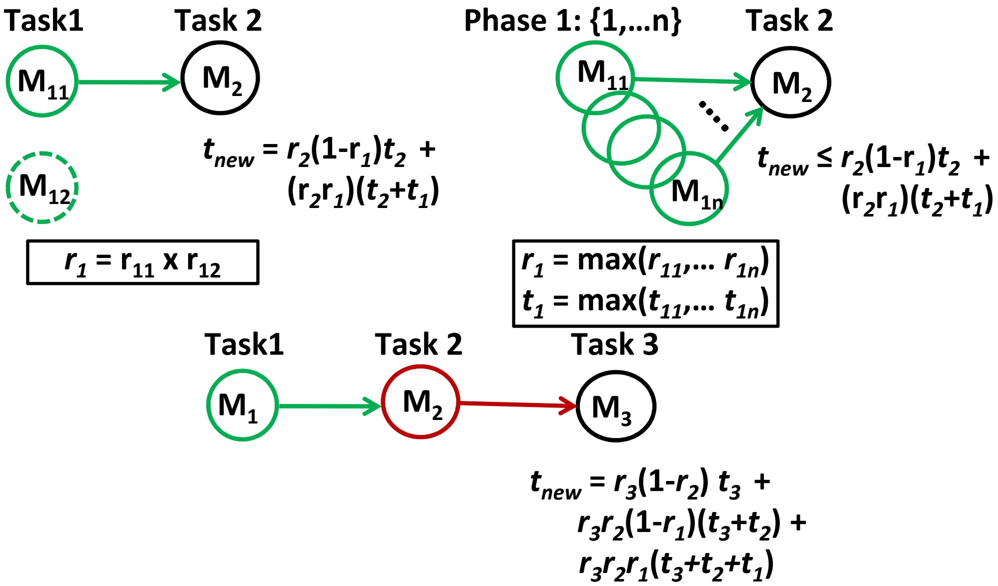

Task1
Task 2
M11
M2
M12
tnew = r2(1-r1)t2 + (r2r1)(t2+t1)
r1 = r11 x r12
Phase 1: {1,…n}
Task 2
M11
M2
M1n
tnew ≤ r2(1-r1)t2 + (r2r1)(t2+t1)
r1 = max(r11,… r1n)
t1 = max(t11,… t1n)
Task1
Task 2
Task 3
M1
M2
M3
tnew = r3(1-r2) t3 + r3r2(1-r1)(t3+t2) + r3r2r1(t3+t2+t1)

Figure 3.5: **Expected Cost of Recomputation. Recompute probabilities of machines are marked r_i. When data is replicated, the effective probability of loss is reduced as we take the minimum (top left). Tasks with many-to-one input patterns have high recomputation cost and are more valuable (top right). The calculation goes recursively back in prior phases (bottom). Finally, the time to replicate, t_{rep} is calculated based on the available rack-local bandwidth and data is replicated only if $t_{rep} < t_{new}$.**

3.5 Evaluation

3.5.1 Does Mantri improve placement?

Figure 3.6 plots the reduction in completion time due to Mantri's placement of reduce tasks as a CDF over all reduce phases in the dataset in Table 1.3. As before, the y-axes weighs phases by their lifetime. The figure shows that Mantri provides a median speed up of 60% or a 2.5× improvement over the current implementation, vindicating our choice of monitoring and judiciously using the available resources (network bandwidths).

The figure also compares Mantri against strategies that estimate available bandwidths differently. The *IdealReduce* strategy tracks perfectly the changes in available bandwidth of links due to the other jobs in the cluster. The *Equal* strategy assumes that the available bandwidths are equal across all links whereas *Start* assumes that the available bandwidths are the same as at the start of the phase. We see a partial order between *Start* and *Equal*.

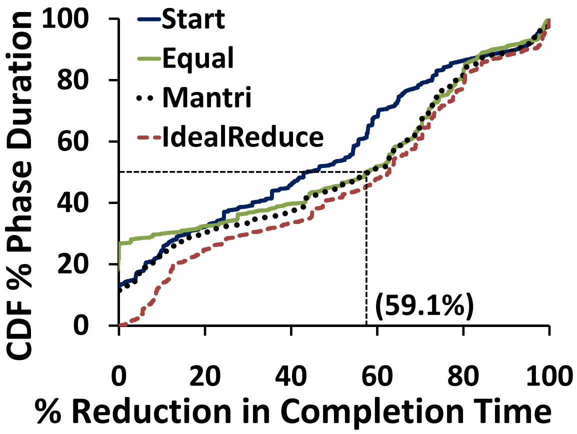


Figure 3.6: **Compared to the current placement, Mantri's network aware placement speeds up the median reduce phase by 60%.**

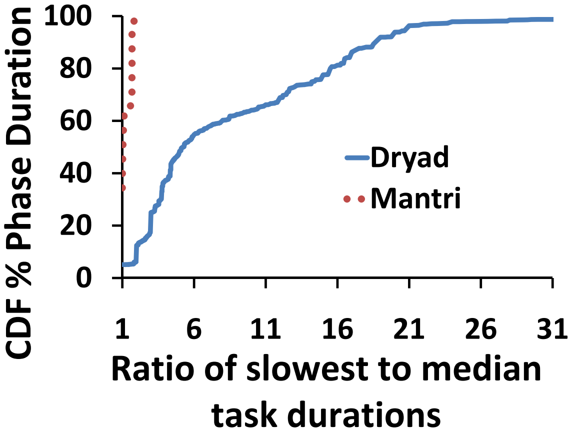


Figure 3.7: **Ratio of slowest to median task durations for Mantri and Dryad.**

Short phases are impacted by transient differences in the available bandwidths and *Start* is a good choice for these phases. However, these differences even out over the lifetime of long phases for whom *Equal* works better. Mantri is a hybrid of *Start* and *Equal*. It achieves a good approximation of *IdealReduce* without re-sampling available bandwidths.

Recall that outliers occur when tasks are placed such that some of them read data over congested links (§3.3). Figure 3.7 plots the ratio of the slowest task duration to the median duration, which roughly corresponds to the ratios in congestions of the network links. By using the available bandwidths and data transfer patterns, Mantri ensures that the median phase's slowest task is just 5% more than its median, with the ratio of slow to median task durations never exceeding 2. Dryad's policy of placing tasks at any available slot results in heavy manifestation of outliers with the ratio of slowest task to the median being 5.25 and 14.33 at 50^{th} and 75^{th} percentiles. We reiterate that identifying the cause of outliers, i.e., network congestion, helped in appropriate placement whereas duplication of tasks without

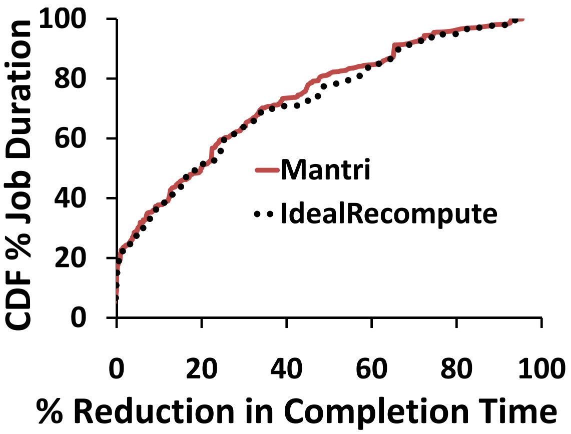


Figure 3.8: **By probabilistically replicating task output and recomputing lost data before it is needed Mantri speeds up jobs by an amount equal to the ideal case of no data loss.**

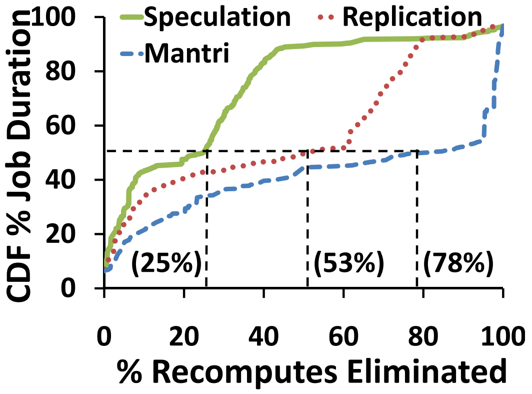
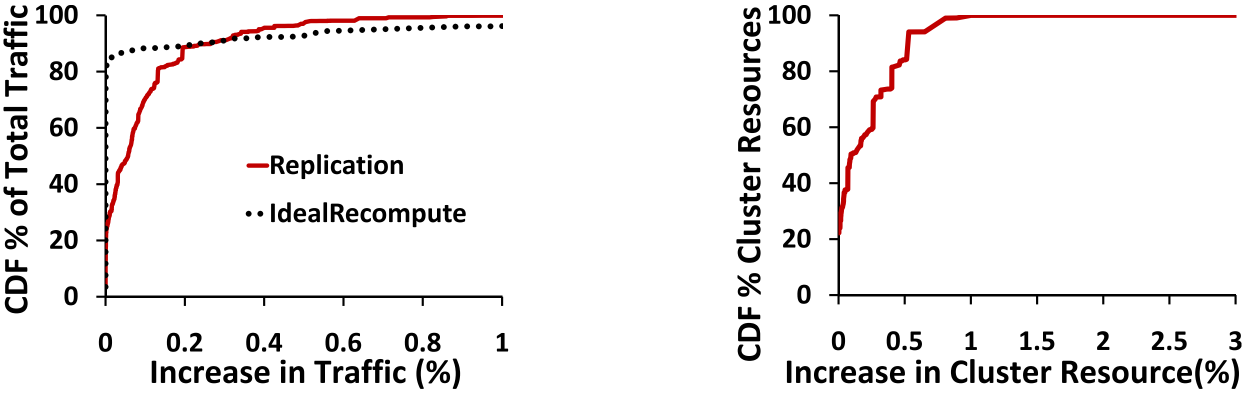


Figure 3.9: **Fraction of recomputations that are eliminated due to Mantri's recomputation mitigation strategy, along with individual contributions from replication and speculative recomputation. Replication and speculative recomputation contribute two-third and one-third of the eliminated recomputations, complementing each other.**

considering available bandwidths would not have helped.

3.5.2 Does Mantri help with recomputations?

The best possible protection against loss of output would (a) eliminate all the increase in job completion time due to tasks waiting for their inputs to be recomputed and (b) do so with little additional cost. Mantri approximates both goals. Fig. 3.8 shows that by selectively



(a) Cost: Network Traffic

(b) Cost: Cluster Time

Figure 3.10: **The cost to protect against recomputes is fewer than a few percentage points in both the extra traffic on the network and cluster time for speculative recomputation.**

replicating tasks that are more likely to have their inputs corrupted (by noting the origin of the problem - problematic machines) and early action using speculatively recomputations for data that has already been lost, Mantri achieves parity with *IdealRecompute* that has perfect future knowledge of loss. The improvement in job completion time is 20% and 40% at the 50th and 75th percentiles. As supporting evidence, we see that Mantri is successful in eliminating the majority of recomputations. As Figure 3.9 shows, 78% of the median job's recomputations are eliminated. Note that despite some jobs having only a small fraction of their recomputes eliminated, Mantri's policy of estimating and protecting the output of more valuable tasks ensures it is almost on par with *IdealRecompute*. Figure 3.9 also shows the contribution of replication and speculative recomputation to the eliminated recomputations. The two techniques contribute roughly two-third and one-third towards the eilimated recomputations, complementing each other.

Now we show that Mantri's accurate predictions ensure that the improvements are despite little increase in resource consumption. Fig. 3.10a shows that the extra network traffic due to replication is (overall negligible and) comparable to a scheme that has perfect future knowledge of which data is lost and replicates just that data. Mantri sometimes replicates more data than the ideal, and at other times misses some tasks that should be replicated. Fig. 3.10b shows that speculative recomputations take no more than a few percentage extra cluster resources. We reinforce the accuracy of our predictions for replication and speculative recomputations by observing their success rate, i.e., the task whose output was replicated or speculatively recomputed did lose its output. Replication turns out to be accurate for 84% of the tasks. The success rate of speculative recomputations increase with greater historical knowledge. Speculative recomputations turn out to be right 76% of the time when we consider the occurrence of a single recompute as a trigger but this rate goes up to 93% when we wait for two recomputes in the last minute window for scheduling speculative recomputes, despite producing only an insignificant drop in improvement of completion times. We consider the high success rate as a validation of our estimation of t_{new}, available replication bandwidth and onset of failure for speculative recomputation.

Chapter 4

Run-time Contentions

4.1 Introduction

While the techniques thus far schedule tasks such that none of them are relatively disadvantaged, tasks can still face unexpected contentions during execution. Such contentions occur due to complex requirements for resources including memory, disk and network. Tasks facing such contentions consequently slow down and we refer to them as *stragglers*. Stragglers are tasks that run much slower than other tasks, and since a job finishes only when its last task finishes, stragglers delay job completion.

Cluster managers combat stragglers by blacklisting problematic machines. Blacklisting identifies machines in bad health (e.g., due to faulty disks) and avoids scheduling tasks on them. The Facebook and Bing clusters, in fact, blacklist roughly 10% of their machines. However, stragglers occur on the non-blacklisted machines, often due to intrinsically complex reasons like IO contentions, interference by periodic maintenance operations and background services, and hardware behaviors [10].

4.1.1 Speculation

An intuitive and effective way to deal with straggler tasks is *speculation*—launching speculative copies for the slower tasks, where a speculative copy is simply a duplicate of the original task. It then becomes a race between the original and the speculative copies.

The key to effective speculation is picking the right task to speculate at the right time. Scheduling tasks to compute slots is a classic scheduling problem with known heuristics [58,63,79]. These heuristics, unfortunately, do not directly carry over to our scenario for the following reasons. First, they calculate the optimal scheduling order statically. Straggling of tasks, on the other hand, is unpredictable and necessitates dynamic modification of the priority ordering of tasks according to the approximation bounds. Second, traditional scheduling techniques assign tasks to slots assuming every task to occupy only one slot. Spawning a speculative copy, however, leads to the same task using two (or multiple) slots simultaneously. Hence, this distills our challenge to *dynamically weighing the gains due to speculation against the cost of using extra resources for speculation.*

Scheduling tasks with speculative copies is NP-Hard and devising good heuristics (i.e., with good approximation factors) is an open theoretical problem. Scheduling a speculative copy helps make immediate progress by finishing a task faster. However, while not scheduling a speculative copy results in the task running slower, many more tasks may be completed using the saved slot. To understand this *opportunity cost*, consider a cluster with one unoccupied slot and a straggler task. Letting the straggler complete takes five more time units while a new copy of it would take four time units. While scheduling a speculative copy for this straggler speeds it up by one time unit, if we were not to, that slot could finish another task (taking five time units too).

We design a heuristic, Mantri, that performs speculation by carefully considering the opportunity cost of the duplicate copy. It schedules a speculative copy only if doing so saves both time and resources.

4.1.2 Cloning

Speculation techniques, however, face a fundamental limitation when dealing with small jobs, i.e., jobs that consist of a few tasks. Such jobs typically get to run all their tasks at once. Therefore, even if a single task is slow, i.e., straggle, the whole job is significantly delayed. Small jobs are pervasive. Conversations with datacenter operators reveal that these small jobs are typically used when performing interactive and exploratory analyses. Achieving low latencies for such jobs is critical to enable data analysts to efficiently explore the search space. To obtain low latencies, analysts already restrict their queries to small but carefully chosen datasets, which results in jobs consisting of only a few short tasks. The trend of such exploratory analytics is evident in Over 80% of the Hadoop jobs and over 60% of the Dryad jobs are small with fewer than ten tasks [1] Achieving low latencies for these small interactive jobs is of prime concern to datacenter operators.

While speculation techniques rely on comparing tasks of a job, any meaningful comparison requires waiting to collect statistically significant samples of task performance. Such waiting limits their agility when dealing with stragglers in small jobs as they often start all their tasks simultaneously. The problem is exacerbated when some tasks start straggling when they are well into their execution. Spawning a speculative copy at that point might be too late to help.

We propose a different approach. Instead of waiting and trying to predict stragglers, we take speculative execution to its extreme and propose launching multiple *clones* of every task of a job and only use the result of the clone that finishes first. This technique is both general and robust as it eschews waiting, speculating, and finding complex correlations. Such *proactive* cloning will significantly improve the agility of straggler mitigation when dealing with small interactive jobs.

[1] The length of a task is mostly invariant across small and large jobs.

p_b	**Blacklisted Machines (%)**		**Job Improvement (%)**	
	5 **min**	1 **hour**	5 **min**	1 **hour**
0.3	4%	6%	7.1%	8.4%
0.5	1.6%	2.8%	4.4%	5.2%
0.7	0.8%	1.2%	2.3%	2.8%

Table 4.1: **Blacklisting by predicting straggler probability. We show the fraction of machines that got blacklisted and the improvements in completion times by avoiding them.**

Cloning comes with two main challenges. The first challenge is that extra clones might use a prohibitive amount of extra resources. However, our analysis of production traces shows a strong *heavy-tail distribution* of job sizes: the smallest 90% of jobs consume as less as 6% of the resources. The interactive jobs whose latency we seek to improve all fall in this category of small jobs. We can, hence, improve them by using few extra resources. The second challenge is the potential *contention* that extra clones create on intermediate data, possibly hurting job performance. Efficient cloning requires that we clone each task and use the output from the clone of the task that finishes first. This, however, can cause contention for the intermediate data passed between tasks of the different phases (e.g., map, reduce, join) of the job; frameworks often compose jobs as a graph of *phases* where tasks of downstream phases (e.g., reduce) read the output of tasks of upstream phases (e.g., map). If all downstream clones read from the upstream clone that finishes first, they contend for the IO bandwidth. An alternate that avoids this contention is making each downstream clone read exclusively from only a single upstream clone. But this staggers the start times of the downstream clones.

Our solution to the contention problem, *delay assignment*, is a hybrid solution that aims to get the best of both the above pure approaches. It is based on the intuition that most clones, except few stragglers, finish nearly simultaneously. Using a cost-benefit analysis that captures this small variation among the clones, it checks to see if clones can obtain exclusive copies before assigning downstream clones to the available copies of upstream outputs. The cost-benefit analysis is generic to account for different communication patterns between the phases, including all-to-all (MapReduce), many-to-one (Dryad), and one-to-one (Dryad and Spark).

We have built Dolly, a system that performs cloning to mitigate the effect of stragglers while operating within a resource budget.

4.1.3 Blacklisting is Insufficient

An intuitive solution for mitigating stragglers is to *blacklist* machines that are likely to cause them and avoid scheduling tasks on them. For this analysis, we classify a task as a straggler if its progress rate is less than half of the median progress rate among tasks in its

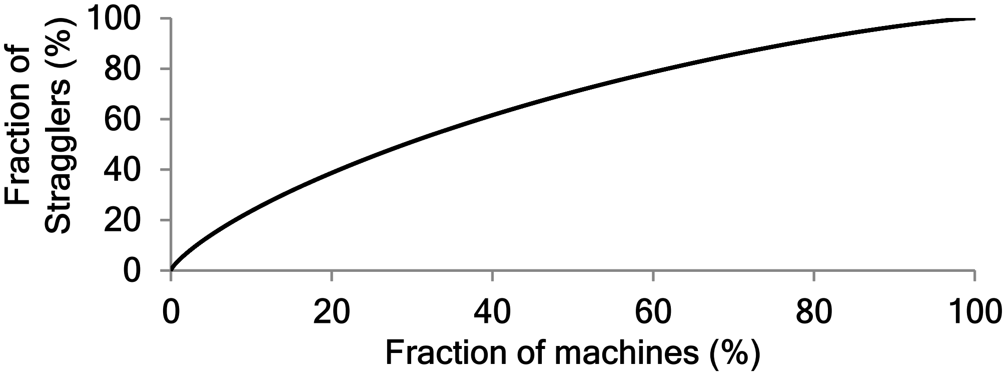


Figure 4.1: **CDF of the total fraction of straggler tasks to the fraction of machines they occur on, over the entire Facebook trace. The nearly linear nature of the graph shows that stragglers are not restricted to a small set of machines.**

phase. In our trace, stragglers are not restricted to a small set of machines but are rather spread out uniformly through the cluster. This is not surprising because both the clusters already blacklist machines with faulty disks and other hardware troubles using periodic diagnostics.

Stragglers are not restricted to a set of problematic machines (for e.g., due to faulty hardware). Figure 4.1 plots the CDF of the total fraction of stragglers to the fraction of machines they occur on, over the entire trace. The curve grows near-linearly with a slope of 2 until $\sim$ 35% of the stragglers beyond which the slope gradually decreases and reaches unity. This shows that just a few machines do not cause the bulk of the stragglers in the cluster. This is not surprising because the Facebook cluster already performs blacklisting of machines to eliminate machines with faulty disks and other hardware troubles.

We enhance this blacklisting by monitoring machines at finer time intervals and employing temporal prediction techniques to warn about straggler occurrences. We use an EWMA to predict stragglers—the probability of a machine causing a straggler in a time window is equally dependent on its straggler probability in the previous window and its long-term average. Machines with a predicted straggler probability greater than a threshold (p_b) are blacklisted for that time window but considered again for scheduling in the next time window.

We try time windows of 5 minutes and 1 hour. Table 4.1 lists the fraction of machines that get blacklisted and the resulting improvement in job completion times by eliminating stragglers on them, in the Facebook trace. The best case eliminates only 12% of the stragglers and improves the average completion time by only 8.4% (in the Bing trace, 11% of stragglers are eliminated leading to an improvement of 6.2%). This is in harsh contrast with potential improvements of 29% to 47% if all stragglers were eliminated, as shown in §4.3.1.

Surprisingly, occurrence of stragglers on a machine does not correlate with the utilization of memory, CPU or slots on it; the Pearson correlation coefficients between the utilizations and probability of stragglers are 0.1, 0.16 and 0.14, respectively, indicating little correlation. This shows that stragglers often occur due to complex and fleeting events (e.g., garbage collection). The lack of simple correlations for idiosyncratic performances is consistent with recent observations in Google's clusters where the complexity of the hardware and the intertwining of the various software and operating system modules precludes realistic modeling and blacklisting solutions [10].

The above results do not prove that effective blacklisting is impossible, but shows that none of the blacklisting techniques that we and, to our best knowledge, others [10] have tried effectively prevent stragglers, suggesting that such correlations either do not exist or are hard to find.

4.2 Speculation with Opportunity Cost

4.2.1 Quantifying Stragglers

We characterize the prevalence and causes of outliers and their impact on job completion times and cluster resource usage. We will argue that three factors – dynamics, concurrency and scale, that are somewhat unique to large map-reduce clusters for efficient and economic operation lie at the core of the outlier problem.

Figure 4.2a plots the fraction of high runtime outliers and recomputes in a phase. For exposition, we arbitrarily say that a task has high runtime if its time to finish is longer than 1.5x the median task duration in its phase. By recomputes, we mean instances where a task output is lost and dependent tasks wait until the output is regenerated.

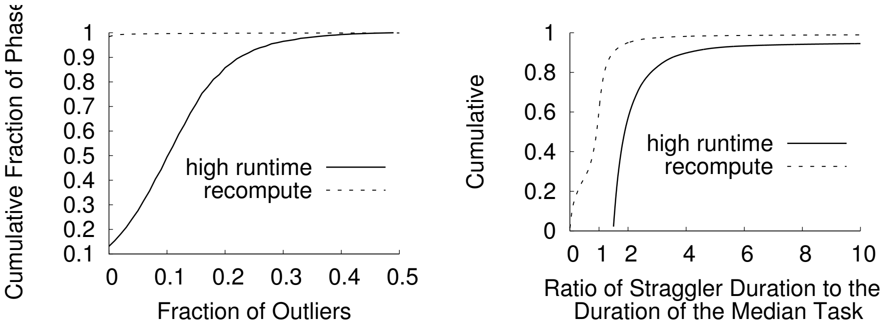
We see in Figure 4.2a that 25% of phases have more than 15% of their tasks as outliers. The figure also shows that 99% of the phases see no recomputes. Though rare, recomputes have a widespread impact. Two out of a thousand phases have over 50% of their tasks waiting for data to be recomputed.

How much longer do outliers run for? Figure 4.3 shows that 80% of the runtime outliers last less than 2.5 times the median task duration in the phase, with a uniform probability of being delayed by between 1.5x to 2.5x.

The tail is heavy and long– 10% take more than 10x the median duration. Ignoring these if they happen early in a phase, as current approaches do, appears wasteful.

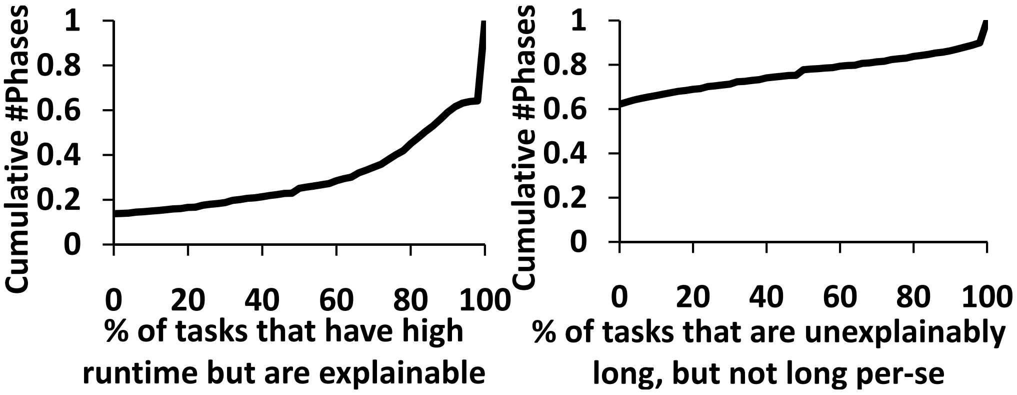
Figure 4.3a shows that in 40% of the phases (top right), all the tasks with high runtimes (i.e., over 1.5x the median task) are well explained by the amount of data they process or move on the network. Duplicating these tasks would not make them run faster and will waste resources.

Figure 4.3b shows tasks that take longer than they should, as predicted by the model based on the data they read, but do not take over 1.5X the median task in their phase. Such tasks present an opportunity for improvement. They may finish faster if run elsewhere, yet



(a) (b)

Figure 4.2: **Contribution of data size to task runtime (see §3.2)**



(a) (b)

Figure 4.3: **How much longer do outliers take to finish?**

current schemes do nothing for them. 30% of the phases (on the top right) have over 50% of such improvable tasks.

4.2.2 Resource-aware Speculation

Based on task progress reports, Mantri estimates for each task the remaining time to finish, t_{rem}, and the predicted completion time of a new copy of the task, t_{new}. Tasks report progress once every 10s or ten times in their lifetime, whichever is smaller. We use Δ to refer to this period. We defer details of the estimation to §4.2.3 and proceed to describe the algorithms for mitigating each of the main causes of outliers. All that matters is that t_{rem} be an accurate estimate and that the predicted distribution t_{new} account for the underlying work that the task has to do, the appropriateness of the network location and any persistent slowness of the new machine.

Duplication of outliers needs to be *resource aware*, *act early* and be *closed-loop*. Resource

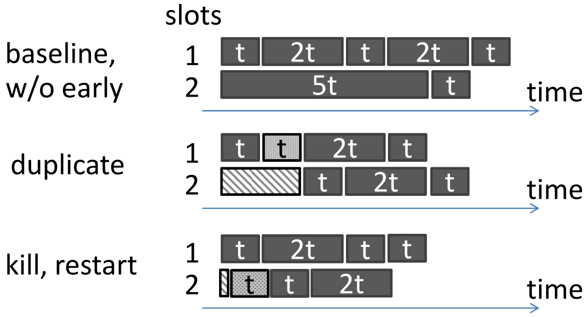


Figure 4.4: **A stylized example to illustrate our main ideas. Tasks that are eventually killed are filled with stripes, repeat instances of a task are filled with a mesh.**

awareness measures the value of duplicate tasks. Note that reducing the completion time of a task may in fact increase the job completion time because twice as many resources are used up for this task now denying resources to other tasks that are waiting to run. Early action on outliers vacates slots for outstanding tasks and can speed up completion. Unlike Mantri, none of the existing approaches act early. Closed-loop action reduces wasted resource and makes Mantri robust to errors involved with the probabilistic predictions. Mantri monitors running copies and kills those whose cost exceeds the benefit, *e.g.*, a restarted task that is slower than expected, or a duplicate that is slower than the original. The freed resource enables other tasks or new copies to start.

Scheduling Example

Figure 4.4 shows a phase that has seven tasks and two slots. Normal tasks run for times t and $2t$. One outlier has a runtime of $5t$. Time increases along the x axes.

The timeline at the top shows a baseline which ignores outliers and finishes at $7t$. Prior approaches that only address outliers at the end of the phase also finish at $7t$.

Note that if this outlier has a large amount of data to process letting the straggling task be is better than killing or duplicating it, both of which waste resources.

If however, the outlier was slowed down by its location, the second and third timelines compare duplication to a restart that kills the original copy. After a short time to identify the outlier, the scheduler can duplicate it at the next available slot (the middle time-line) or restart it in-place (the bottom timeline). If prediction is accurate, restarting is strictly better. However, if slots are going idle, it may be worthwhile to duplicate rather than incur the risk of losing work by killing.

Duplicating the outlier costs a total of $3t$ in resources ($2t$ before the original task is killed and t for the duplicate) which may be wasteful if the outlier were to finish in sooner than $3t$ by itself.

Pseudocode 5 Resource-aware restart.

```
let $\Delta$ = period of progress reports
let $c$ = number of copies of a task
periodically, for each running task, kill all but the fastest $\alpha$ copies after $\Delta$ time has
        passed since begin
while slots are available do
    if tasks are waiting for slots then
        kill, restart task if $t_{rem} > \mathbb{E}(t_{new}) + \Delta$, stop at $\gamma$ restarts
        duplicate if $\mathbb{P}(t_{rem} > t_{new}\frac{c+1}{c}) > \delta$
        start the waiting task that has the largest data to read
    else                                                    ▷ all tasks have begun
        duplicate iff $\mathbb{E}(t_{new} - t_{rem}) > \rho\Delta$
```

Speculation Algorithm

Mantri uses two variants of restart, the first kills a running task and restarts it elsewhere, the second schedules a duplicate copy. In either method, Mantri restarts only when the probability of success, i.e., $\mathbb{P}(t_{new} < t_{rem})$ is high. Since t_{new} accounts for the systematic differences and the expected dynamic variation, Mantri does not restart tasks that are normal (e.g., runtime proportional to work). Pseudocode 1 summarizes the algorithm. Mantri kills and restarts a task if its remaining time is so large that there is a more than even chance that a restart would finish sooner. In particular, Mantri does so when $t_{rem} > \mathbb{E}(t_{new}) + \Delta$.[2] To not thrash on inaccurate estimates, Mantri never kills a task more than $\gamma = 3$ times.

The "kill and restart" scheme drastically improves the job completion time without requiring extra slots as we show analytically. However, the current job scheduler incurs a queueing delay before a task is restarted. This delay can be large and has a high variation. Hence, we consider scheduling duplicates.

Scheduling a duplicate results in the minimum completion time of the two copies and provides a safety net when estimates are noisy or the queueing delay is large. However, it requires an extra slot and if allowed to run to finish, consumes extra computation resource that will increase the job completion time if outstanding tasks are prevented from starting. Hence, when there are outstanding tasks and no spare slots, we schedule a duplicate only if the total amount of computation resource consumed decreases. In particular, if c copies of the task are currently running, a duplicate is scheduled only if $\mathbb{P}(t_{new} < \frac{c}{c+1}t_{rem}) > \delta$. By default, $\delta = .25$. For example, a task with one running copy is duplicated only if t_{new} is less than half of t_{rem}. On the other hand, when spare slots are available, a duplicate is scheduled if the reduction in the job completion time is larger than the extra resource consumed, $\mathbb{E}(t_{new} - t_{rem}) > \rho\Delta$. By default, $\rho = 3$.

Mantri's restart algorithm is independent of the values for its parameters. Setting γ to

[2] Since the median of the heavy tailed task completion time distribution is smaller than the mean, this check implies that $\mathbb{P}(t_{new} < t_{rem}) > \mathbb{P}(t_{new} < \mathbb{E}(t_{new})) \geq .5$

a larger and ρ, δ to a smaller value trades off the risk of wasteful restarts for the reward of a larger speed-up. The default values err on the side of caution. In addition, Mantri kills all but the fastest copy of a task to reduce consumed resources. Hence, the number of running copies of a task is never larger than 3. By scheduling duplicates conservatively and pruning aggressively, Mantri boosts the success rate of its restarts. It reduces completion time and conserves resources (§4.4).

4.2.3 Estimation of t_{rem} and t_{new}

Periodically, every running task informs the job scheduler of its status, including how many bytes it has read or written thus far. Combining progress reports with the size of the input data that each task has to process, d, Mantri predicts how much longer the task would take to finish as follows:

$$t_{rem} = t_{elapsed} * \frac{d}{d_{read}} + t_{wrapup}. \tag{4.1}$$

The first term captures the remaining time to process data. The second term is the time to compute after all the input has been read and is estimated from the behavior of earlier tasks in the phase. Tasks may speed up or slow down and hence, rather than extrapolating from each progress report, Mantri uses a moving average. To be robust against lost progress reports, when a task hasn't reported for a while, Mantri increases t_{rem} by assuming that the task has not progressed since its last report.

Mantri estimates t_{new}, the distribution over time that a new copy of the task will take to run, as follows:

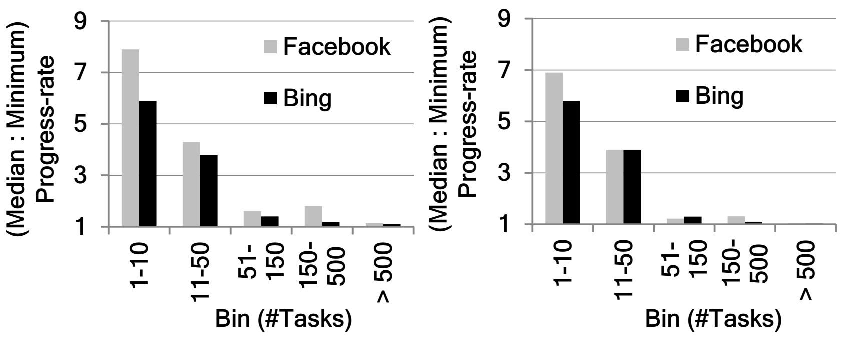
$$t_{new} = \text{processRate} * \text{locationFactor} * d + \text{schedLag}. \tag{4.2}$$

The first term is a distribution of the process rate, i.e., $\frac{\Delta time}{\Delta data}$, of all the tasks in this phase. The second term is a relative factor that accounts for whether the candidate machine for running this task is persistently slower (or faster) than other machines or has smaller (or larger) capacity on the network path to where the task's inputs are located. The third term, as before, is the amount of data the task has to process. The last term is the average delay between a task being scheduled and when it gets to run. We show in §4.4 that these estimates are sufficiently accurate for Mantri's algorithms to function.

4.3 Cloning Tasks

4.3.1 Case for Cloning

In this section we quantify: (*i*) magnitude of stragglers and the potential in eliminating them, and (*ii*) power law distribution of job sizes that facilitate aggressive cloning.



(a) LATE

(b) Mantri

Figure 4.5: **Slowdown ratio *after* applying LATE and Mantri. Small jobs see a higher prevalence of stragglers.**

Magnitude of Stragglers and their Impact

We first quantify the magnitude and impact of stragglers.

A job consists of a graph of *phases* (e.g., map, reduce, and join), with each phase executing the same type of tasks in parallel. We identify stragglers by comparing the *progress rates* of tasks within a phase. The progress rate of a task is defined as the size of its input data divided by its duration. In absence of stragglers, progress rates of tasks of a phase are expected to be similar as they perform similar IO and compute operations. We use the progress rate instead of the task's duration to remain agnostic to skews in work assignment among tasks [19]. Techniques have been developed to deal with the problem of data skews among tasks [17, 60, 99] and our approach is complementary to those techniques.

Within each phase, we measure the *slowdown ratio*, i.e., the ratio of the progress rate of the median task to the slowest task. The negative impact of stragglers increases as the slowdown ratio increases. We measure the slowdown ratio after applying the LATE and Mantri mitigations; a what-if simulation is used for the mitigation strategy that the original trace did not originally deploy.

Figure 4.5a plots the slowdown ratio by binning jobs according to their number of tasks, with LATE in effect. Phases in jobs with fewer than ten tasks, have a median value of this ratio between 6 and 8, i.e., the slowest task is up to 8× slower than the median task in the job. Also, small jobs are hit harder by stragglers.[3] This is similar even if Mantri [19] was deployed. Figure 4.5b shows that the slowest task is still 7× slower than the median task, with Mantri. However, both LATE and Mantri effectively mitigate stragglers in large jobs.

Speculation techniques are not as effective in mitigating stragglers in small jobs as

[3] Implicit in our explanation is that small interactive jobs consist of just a few tasks. While we considered alternate definitions based on input size and durations, in both our traces, we see a high correlation between jobs running for short durations and the number of tasks they contain along with the size of their input.

they are with large jobs because they rely on comparing different tasks of a job to identify stragglers. Comparisons are effective with more samples of task performance. This makes them challenging to do with small jobs because not only do these jobs have fewer tasks but also start all of them simultaneously.

Impact of Stragglers: We measure the potential in speeding up jobs in the trace using the following crude analysis: replace the progress rate of every task of a phase that is slower than the median task with the median task's rate. If this were to happen, the average completion time of jobs improves by 47% and 29% in the Facebook and Bing traces, respectively; small jobs (those with ≤ 10 tasks) improve by 49% and 38%.

Heavy Tail in Job Sizes

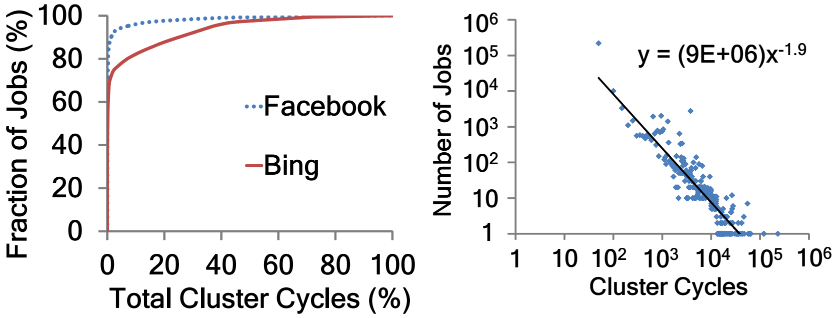
We observed that smaller jobs are most affected by stragglers. These jobs were submitted by users for iterative experimental purposes. For example, researchers tune the parameters of new mining algorithms by evaluating it on a small sample of the dataset. For this reason, these jobs consist of just a few tasks. In fact, in both our traces, we have noted a correlation between a job's duration and the number of tasks it has, i.e., jobs with shorter durations tend to have fewer tasks. Short and predictable response times for these jobs is of prime concern to datacenter operators as they significantly impact productivity.

On the one hand, small interactive jobs absolutely dominate the cluster and have stringent latency demands. In the Facebook and Bing traces, jobs with ≤ 10 tasks account for 82% and 61% of all the jobs, respectively. On the other hand, they are the most affected by stragglers.

Despite this, we can clone all the small jobs using few extra resources. This is because job sizes have a *heavy-tail* distribution. Just a few large jobs consume most of the resources in the cluster, while the cluster is dominated by small interactive jobs. As Figure 4.6a shows, 90% of the smallest jobs consume only 6% and 11% of the total cluster resources in the Facebook and Bing clusters, respectively. Indeed, the distribution of resources consumed by jobs follows a power law (see Figure 4.6b). In fact, at any point in time, the small jobs do not use more than 2% of the overall cluster resources.

The heavy-tail distribution offers potential to speed up these jobs by using few extra resources. For instance, cloning each of the smallest 90% of the jobs three times increases overall utilization by merely 3%. This is well within reach of today's underutilized clusters which are heavily over-provisioned to satisfy their peak demand of over 99%, that leaves them idle at other times [61, 98].

Google recently released traces from their cluster job scheduler that schedules a mixed workload of MapReduce batch jobs, interactive queries and long-running services [13]. Analysis of these traces again reveal a heavy-tail distribution of job sizes, with 92% of the jobs accounting for only 2% of the overall resources [29].



(a) Heavy-tail (b) Power Law

Figure 4.6: **Heavy tail. Figure (a) shows the heavy tail in the fraction of total resources used. Figure (b) shows that the distribution of cluster resources consumed by jobs, in the Facebook trace, follows a power law. Power-law exponents are** 1.9 **and** 1.8 **when fitted with least squares regression in the Facebook and Bing traces.**

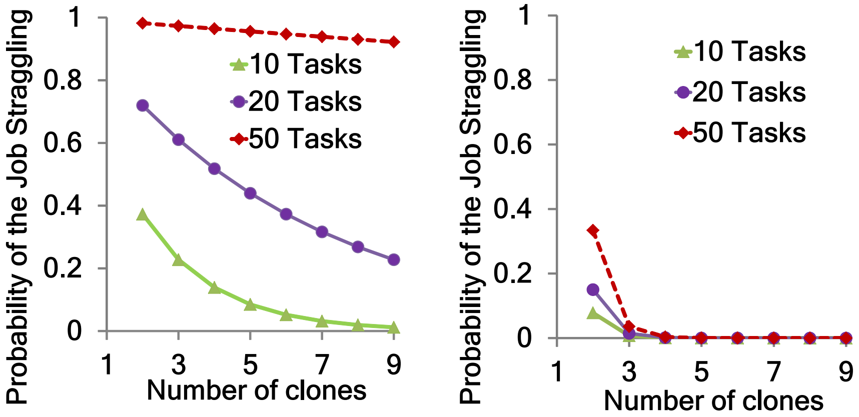
4.3.2 Cloning of Parallel Jobs

We start this section by describing the high-level idea of cloning. After that (§4.3.2) we determine the granularity of cloning, and settle for cloning at the granularity of tasks, rather than entire jobs, as the former requires fewer clones. Thereafter (§4.3.2), we investigate the number of clones needed if we desire the probability of a job straggling to be at most ϵ, while staying within a cloning budget. Finally (§4.3.2), as we are unlikely to have room to clone every job in the cluster, we show a very simple admission control mechanism that decides when to clone jobs. An important challenge of cloning—handling data contention between clones—is dealt with in §4.3.3.

In contrast to *reactive* speculation solutions [19, 36, 71], Dolly advocates a *proactive* approach—straightaway launch multiple clones of a job and use the result of the first clone that finishes. Cloning makes straggler mitigation agile as it does not have to wait and observe a task before acting, and also removes the risk inherent in speculation—speculating the wrong tasks or missing the stragglers. Similar to speculation, we assume that picking the earliest clone does not bias the results, a property that generally holds for data-intensive computations.

Granularity of Cloning

We start with a job consisting of a single phase. A crucial decision affecting efficiency is the granularity of cloning. A simple option is to clone at the granularity of jobs. For every job submitted to the cluster, multiple clones of the entire job are launched. Results are taken from the earliest job that finishes. Such job-level cloning is appealing due to its simplicity and ease of implementation.



(a) Job-level Cloning (b) Task-level Cloning

Figure 4.7: **Probability of a job straggling for varying number of clones, and sample jobs of** 10, 20 **and** 50 **tasks. Task-level cloning requires fewer clones than job-level cloning to achieve the same probability of the job straggling.**

A fine-grained alternative is to clone at the granularity of individual tasks. Thus, multiple clones of each task are launched. We refer to the different clones of the same task as a *clone group*. In every clone group, we then use the result of the clone that finishes first. Therefore, unlike job-level cloning, task-level cloning requires internal changes to the execution engine of the framework.

As a result of the finer granularity, for the same number of clones, task-level cloning provides better probabilistic guarantees for eliminating stragglers compared to job-level cloning. Let p be the probability of a task straggling. For a single-phased job with n parallel tasks and c clones, the probability that it straggles is $(1-(1-p)^n)^c$ with job-level cloning, and $1-(1-p^c)^n$ with task-level cloning. Figure 4.7 compares these probabilities. Task-level cloning gains more per clone and the probability of the job straggling drops off faster.

Task-level cloning's resource efficiency is desirable because it reduces contention on the input data which is read from file systems like HDFS [6]. If replication of input data does not match the number of clones, the clones contend for IO bandwidth in reading the data. Increasing replication, however, is difficult as clusters already face a dearth of storage space [16, 42]. Hence, due to its efficiency, we opt for task-level cloning in Dolly.

Budgeted Cloning Algorithm

Pseudocode 6 describes the cloning algorithm that is executed at the scheduler per job. The algorithm takes as input the cluster-wide probability of a straggler (p) and the acceptable risk of a job straggling (ϵ). We aim for an ϵ of 5% in our experiments. The probability of a straggler, p, is calculated every hour, where the straggler progresses at less than half the median task in the job. This coarse approach suffices for our purpose.

Dolly operates within an allotted resource budget. This budget is a configurable fraction (β) of the total capacity of the cluster (C). At no point does Dolly use more than this cloning budget. Setting a hard limit eases deployment concerns because operators are typically nervous about increasing the average utilization by more than a few percent. Utilization and capacity are measured in number of slots (computation units allotted to tasks).

The pseudocode first calculates the desired number of clones per task (step 2). For a job with n tasks, the number of clones desired by task-level cloning, c, can be derived to be at least $\log\left(1 - (1-\epsilon)^{(1/n)}\right) / \log p$. [4] The number of clones that are eventually spawned is limited by the resource budget ($C \cdot \beta$) and a utilization threshold (τ), as in step 3. The job is cloned only if there is room to clone all its tasks, a policy we explain shortly in §4.3.2. Further, cloning is avoided if the cluster utilization after spawning clones is expected to exceed a ceiling τ. This ceiling avoids cloning during heavily-loaded periods.

Note that Pseudocode 6 spawns the same number of clones to all the tasks of a job. Otherwise, tasks with fewer clones are more likely to lag behind. Also, there are no conflicts between jobs in updating the shared variables B_U and U because the centralized scheduler handles cloning decisions one job at a time.

Multi-phased Jobs: For multi-phased jobs, Dolly uses Pseudocode 6 to decide the number of clones for tasks of *every* phase. However, the number of clones for tasks of a downstream phase (e.g., reduce) never exceeds the number of clones launched its upstream phase (e.g., map). This avoids contention for intermediate data (we revisit this in §4.3.3). In practice, this limit never applies because small jobs have equal number of tasks across their phases. In both our traces, over 91% of the jobs with ≤ 10 tasks have equal number of tasks in their phases.

Admission Control

The limited cloning budget, β, should preferably be utilized to clone the small interactive jobs. Dolly achieves this using a simple policy of *admission control*.

Whenever the first task of a job is to be executed, the admission control mechanism computes, as previously explained, the number of clones c that would be required to reach the target probability ϵ of that job straggling. If, at that moment, there is room in the cloning budget for creating c copies of all the tasks, it admits cloning the job. If there is not enough budget for c clones of all the tasks, the job is simply denied cloning and is executed without Dolly's straggler mitigation. The policy of admission control implicitly biases towards cloning small jobs—the budget will typically be insufficient for creating the required number of clones for the larger jobs. Step 3 in Pseudocode 6 implements this policy.

Many other competing policies are possible. For instance, a job could be partially cloned

[4] The probability of a job straggling can be at most ϵ, i.e., $1 - (1 - p^c)^n \leq \epsilon$. The equation is derived by solving for c.

Pseudocode 6 Task-level cloning for a single-phased job with n parallel tasks, on a cluster with probability of straggler as p, and the acceptable risk of straggler as ϵ.

```
procedure CLONE($n$ tasks, $p$, $\epsilon$)
    $C$: Cluster Capacity, $U$: Cluster Utilization
    $\beta$: Budget in fraction, $B_U$: Utilized budget in #slots
    $c = \lceil \log \left(1 - (1 - \epsilon)^{(1/n)}\right) / \log p \rceil$
    if $(B_U + c \cdot n) \leq (C \cdot \beta)$ and $(U + c \cdot n) \leq \tau$ then
                                    ▷ Admission Control: Sufficient capacity to
            create $c$ clones for each task
        for each task $t$ do
            Create $c$ clones for $t$
        $B_U \leftarrow B_U + c \cdot n$
```

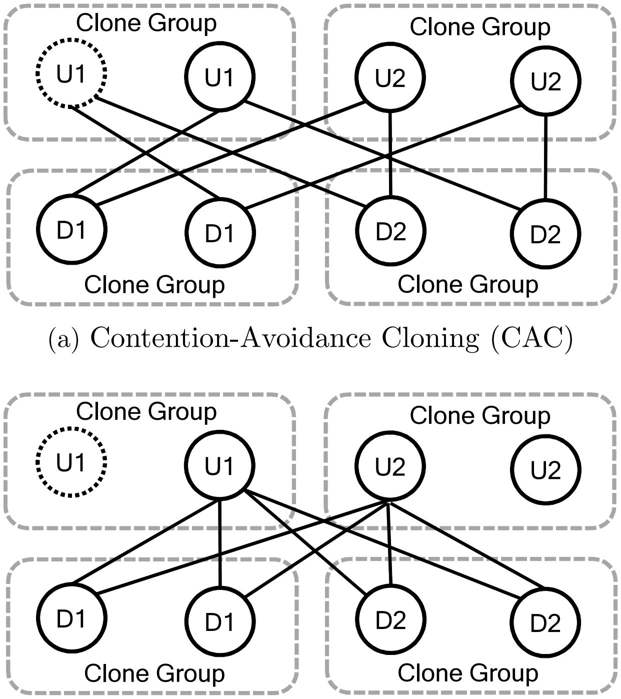
if there is not enough room for c clones. Furthermore, preemption could be used to cancel the clones of an existing job to make way for cloning another job. It turns out that these competing policies buy little performance compared to our simple policy. We compare these policies in §4.4.2.

4.3.3 Intermediate Data Access with Dolly

A fundamental challenge of cloning is the potential contention it creates in reading data. Downstream tasks in a job read intermediate data from upstream tasks according to the communication pattern of that phase (all-to-all, many-to-one, one-to-one). The clones in a downstream clone group would ideally read their intermediate data from the upstream clone that finishes first as this helps them all start together.[5] This, however, can create contention at the upstream clone that finishes first. Dealing with such contentions is the focus of this section.

We first (§4.3.3) explore two pure strategies at opposite ends of the spectrum for dealing with intermediate data contention. At one extreme, we completely avoid contention by assigning each upstream clone, as it finishes, to a new downstream task clone. This avoids contention because it guarantees that every upstream task clone only transfers data to a single clone per downstream clone group. At another extreme, the system ignores the extra contention caused and assumes that the first finished upstream clone in every clone group can sustain transferring its intermediate output to all downstream task clones. As we show (§4.3.3), the latter better mitigates stragglers compared to the former strategy. However, we show (§4.3.3) that the latter may lead to congestion whereas the former completely avoids

[5] Intermediate data typically only exists on a single machine, as it is not replicated to avoid time and resource overheads. Some systems do replicate intermediate data [19, 87] for fault-tolerance but limit this to replicating only a small fraction of the data.



(a) Contention-Avoidance Cloning (CAC)

(b) Contention Cloning (CC)

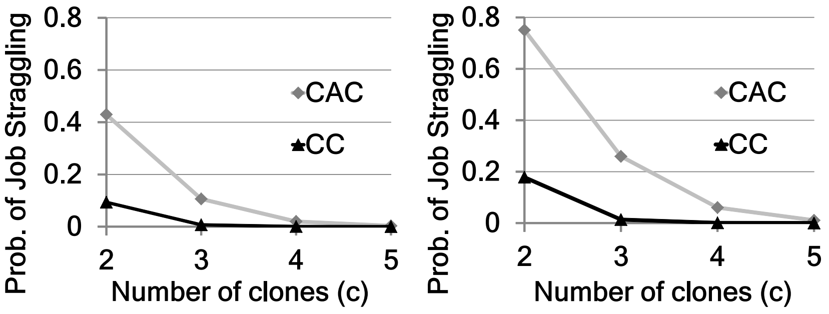
Figure 4.8: **Intermediate data contention. The example job contains two upstream tasks (U1 and U2) and two downstream tasks (D1 and D2), each cloned twice. The clone of U1 is a straggler (marked with a dotted circle). CAC waits for the straggling clone while CC picks the earliest clone.**

it. Finally (§4.3.3), we settle on a hybrid between the two (§4.3.3), *delay assignment* that far outperforms these two pure strategies.

Two Opposite Strategies

We illustrate two approaches at the opposite ends of the spectrum through a simple example. Consider a job with two phases (see Figure 4.8) and an all-to-all (e.g., shuffle) communication pattern between them (§4.3.3 shows how this can be generalized to other patterns). Each of the phases consist of two tasks, and each task has two clones.

The first option (Figure 4.8a), which we call Contention-Avoidance Cloning (CAC) eschews contention altogether. As soon as an upstream task clone finishes, its output is sent to exactly one downstream task clone per clone group. Thus, the other downstream task clones have to wait for another upstream task clone to finish before they can start their computation. We call this Contention-Avoidance Cloning (CAC). Note that in CAC an upstream clone will send its intermediate data to the exact same number of other tasks as if no cloning was done, avoiding contention due to cloning. The disadvantage with CAC



(a) $n = 10$ tasks (b) $n = 20$ tasks

Figure 4.9: **CAC vs. CC: Probability of a job straggling.**

is that when some upstream clones straggle, the corresponding downstream clones that read data from them automatically lag behind.

The alternate option (Figure 4.8b), Contention Cloning (CC), alleviates this problem by making all the tasks in a downstream clone group read the output of the upstream clone that finishes first. This ensures that no downstream clone is disadvantaged, however, all of them may slow down due to contention on disk or network bandwidth.

There are downsides to both CAC and CC. The next two sub-sections quantify these downsides.

Probability of Job Straggling: CAC vs. CC

CAC increases the vulnerability of a job to stragglers by negating the value of some of its clones. We first analytically derive the probability of a job straggling with CAC and CC, and then compare them for some representative job sizes. We use a job with n upstream and n downstream tasks, with c clones of each task.

CAC: A job straggles with CAC when either the upstream clones straggle and consequently handicap the downstream clones, or the downstream clones straggle by themselves. We start with the upstream phase first before moving to the downstream phase.

The probability that at least d upstream clones of every clone group will succeed without straggling is given by the function Ψ; p is the probability of a task straggling.

$$\Psi(n, c, d) = \text{Probability}[n \text{ upstream tasks of } c \text{ clones with } \geq d \text{ non-stragglers per clone group}]$$

$$\Psi(n, c, d) = \left(\sum_{i=0}^{c-d} \binom{c}{i} p^i (1-p)^{c-i} \right)^n \tag{4.3}$$

Therefore, the probability of exactly d upstream clones not straggling is calculated as:

$$\Psi(n, c, d) - \Psi(n, c, d-1)$$

Recall that there are n downstream tasks that are cloned c times each. Therefore, the probability of the whole job straggling is essentially the probability of a straggler occurring in the downstream phase, conditional on the number of upstream clones that are non-stragglers.

$$\begin{gathered}\text{Probability[Job straggling with CAC]} = \\ 1 - \sum_{d=1}^{c} \left[\Psi(n, c, d) - \Psi(n, c, d-1)\right] \left(1 - p^{d}\right)^{n}\end{gathered} \tag{4.4}$$

CC: CC assigns all downstream clones to the output of the first upstream task that finishes in every clone group. As all the downstream clones start at the same time, none of them are handicapped. For a job to succeed without straggling, it only requires that one of the upstream clones in each clone group be a non-straggler. Therefore, the probability of the job straggling is:

$$\begin{gathered}\text{Probability[Job straggling with CC]} = \\ 1 - \Psi(n, c, 1)\left(1 - p^{c}\right)^{n}\end{gathered} \tag{4.5}$$

CAC vs. CC: We now compare the probability of a job straggling with CAC and CC for different job sizes. Figure 4.9 plots this for jobs with 10 and 20 upstream and downstream tasks each. With three clones per task, the probability of the job straggling increases by over 10% and 30% with CAC compared to CC. Contrast this with our algorithm in §4.3.2 which aims for an ϵ of 5%. The gap between CAC and CC diminishes for higher numbers of clones but this is contradictory to our decision to pick task-level cloning as we wanted to limit the number of clones. In summary, CAC significantly increases susceptibility of jobs to stragglers compared to CC.

I/O Contention with CC

By assigning all tasks in a downstream clone group to read the output of the earliest upstream clone, CC causes contention for IO bandwidth. We quantify the impact due to this contention using a micro-benchmark rather than using mathematical analysis to model IO bandwidths, which for contention is likely to be inaccurate.

With the goal of realistically measuring contention, our micro-benchmark replicates the all-to-all data shuffle portion of jobs in the Facebook trace. The experiment is performed on the same 150 node cluster we use for Dolly's evaluation. Every downstream task reads its share of the output from each of the upstream tasks. All the reads start at exactly the same relative time as in the original trace and read the same amount of data from every

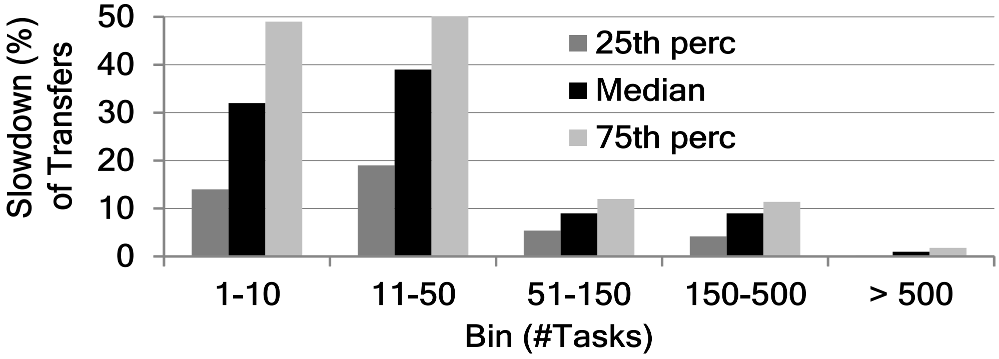


Figure 4.10: **Slowdown (%) of transfer of intermediate data between phases (all-to-all) due to contention by CC.**

upstream task's output. The reads of all the downstream tasks of a job together constitute a *transfer* [68].

The number of clones per upstream and downstream task is decided as in §4.3.2. In the absence of stragglers, there would be as many copies of the upstream outputs as there are downstream clones. However, a fraction of the upstream clones will be stragglers. When upstream clones straggle, we assume their copy of the intermediate data is not available for the transfer. Naturally, this causes contention among the downstream clones.

Reading contended copies of intermediate data likely results in a lower throughput than when there are exclusive copies. Of interest to us is the slowdown in the transfer of the downstream phase due to such contentions, compared to the case where there are as many copies of the intermediate data as there are downstream clones.

Figure 4.10 shows the slowdown of transfers in each bin of jobs. Transfers of jobs in the first two bins slow down by 32% and 39% at median, third quartile values are 50%. Transfers of large jobs are less hurt because tasks of large jobs are often not cloned because of lack of cloning budget. Overall, we see that contentions cause significant slowdown of transfers and are worth avoiding.

Delay Assignment

The analyses in §4.3.3 and §4.3.3 conclude that both CAC and CC have downsides. Contentions with CC are not small enough to be ignored. Following strict CAC is not the solution either because it diminishes the benefits of cloning. A deficiency with both CAC and CC is that they do not distinguish stragglers from tasks that have normal (but minor) variations in their progress. CC errs on the side of assuming that all clones other than the earliest are stragglers, while CAC assumes all variations are normal.

We develop a hybrid approach, *delay assignment*, that first waits to assign the early upstream clones (like CAC), and thereafter proceeds without waiting for any remaining stragglers (like CC). Every downstream clone waits for a small window of time (ω) to see if it can get an exclusive copy of the intermediate data. The wait time of ω allows for normal variations among upstream clones. If the downstream clone does not get its exclusive copy even after waiting for ω, it reads with contention from one of the finished upstream clone's outputs.

Crucial to delay assignment's performance is setting the wait time of ω. We next proceed to discuss the analysis that picks a balanced value of ω.

Setting the delay (ω)**:** The objective of the analysis is to minimize the expected duration of a downstream task, which is the minimum of the durations of its clones.

We reuse the scenario from Figure 4.8. After waiting for ω, the downstream clone either gets its own exclusive copy, or reads the available copy with contention with the other clone. We denote the durations for reading the data in these two cases as T_E and T_C, respectively. In estimating read durations, we eschew detailed modeling of systemic and network performance. Further, we make the simplifying assumption that all downstream clones can read the upstream output (of size r) with a bandwidth of B when there is no contention, and αB in the presence of contention ($\alpha \leq 1$).

Our analysis, then, performs the following three steps.

1. Calculate the clone's expected duration for reading each upstream output using T_C and T_E.

2. Use read durations of all clones of a task to estimate the overall duration of the task.

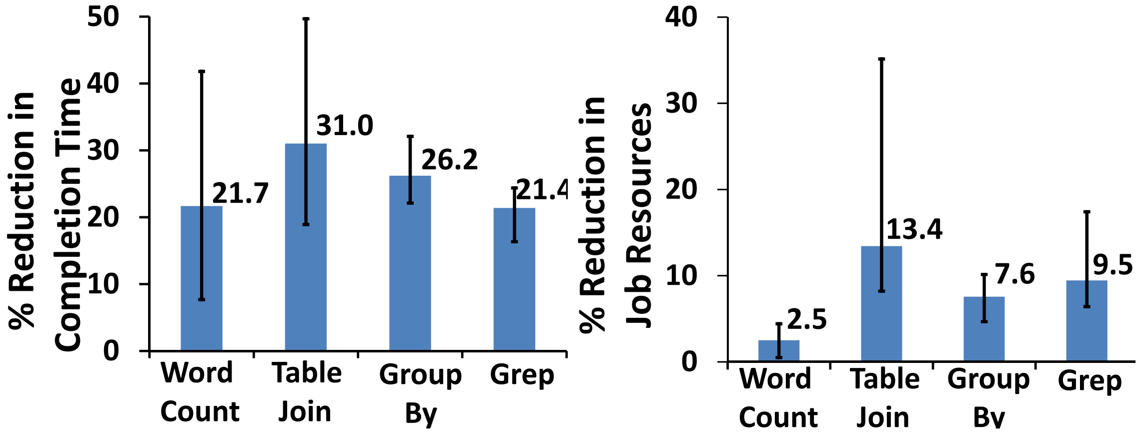
3. Find the delay ω that minimizes the task's duration.

Step (1)**:** We first calculate T_C, i.e., the case where the clone waits for ω but does not get its exclusive copy, and contends with the other clone. The downstream clone that started reading first will complete its read in $\left(\omega + \left(\frac{r-Bw}{\alpha B}\right)\right)$, i.e., it reads for ω by itself and contends with the other clone for the remaining time. The other clone takes $\left(2\omega + \left(\frac{r-Bw}{\alpha B}\right)\right)$ to read the data.

Alternately, if the clone gets its exclusive copy, then the clone that began reading first reads without interruption and completes its read in $\left(\frac{r}{B}\right)$. The other clone, since it gets its own copy too, takes $\left(\frac{r}{B} + \min(\frac{r}{B}, \omega)\right)$ to read the data.[6] Now that we have calculated T_C and T_E, the expected duration of the task for reading this upstream output is simply $p_c T_C + (1 - p_c)\, T_E$, where p_c is the probability of the task not getting an exclusive copy. Note that, regardless of the number of clones, every clone is assigned an input source latest at the end of ω. Unfinished upstream clones at that point are killed.

Step (2)**:** Every clone may have to read the outputs of multiple upstream clones, depending on the intermediate data communication pattern. In all-to-all communication, a task reads data from each upstream task's output. In one-to-one or many-to-one communications, a task reads data from just one or few tasks upstream of it. Therefore, the total time T_i taken by clone i of a task is obtained by considering its read durations from each of the relevant

[6] The wait time of ω is an upper limit. The downstream clone can start as soon as the upstream output arrives.



(a) Completion Time (b) Resource Usage

Figure 4.11: **Comparing Mantri's straggler mitigation with the baseline implementation on a O(10K)-node production cluster for the four representative jobs.**

upstream tasks, along with the expected time for computation. The expected duration of the task is the minimum of all its clones, $\min_i(T_i)$.

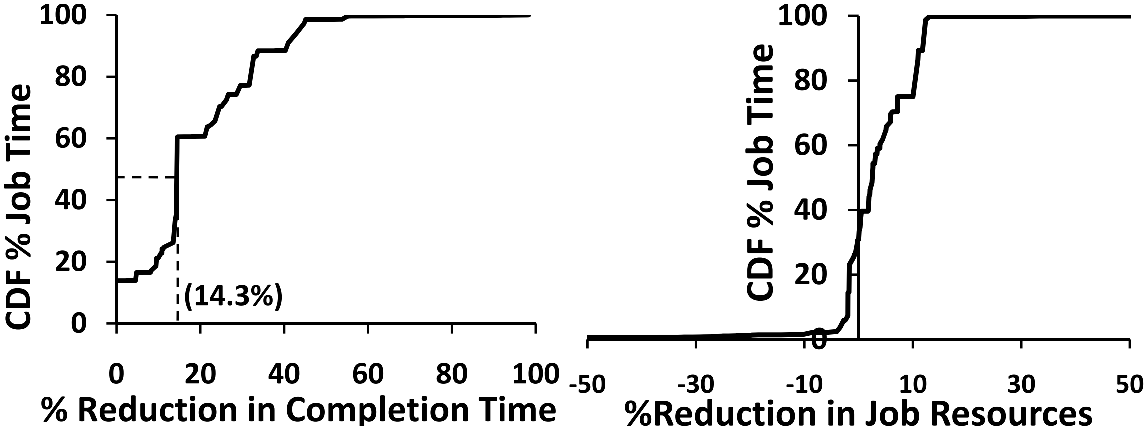
Step (3)**:** The final step is to find ω that minimizes this expected task duration. We sample values of B and α, p_c and the computation times of tasks from samples of completed jobs. The value of B depends on the number of active flows traversing a machine, while the p_c is inversely proportional to ω. Using these, we pick ω that minimizes the duration of a task calculated in step (2). The value of ω is calculated periodically and automatically for different job bins (see §4.4.2). A subtle point with our analysis is that it automatically considers the option where clones read from the available upstream output, one after the other, without contending.

A concern in the strategy of delaying a task is that it is not work-conserving and also somewhat contradicts the observation in §2.4.1 that waiting before deciding to speculate is harmful. Both concerns are ameliorated by the fact that we eventually pick a wait duration that minimizes the completion time. Therefore, our wait is not because we lack data to make a decision but precisely because the data dictates that we wait for the duration of ω.

4.4 Evaluation

4.4.1 Mantri Evaluation

Figure 4.11 compares Mantri with the baseline Cosmos implementation for four jobs running on the larger cluster. Each job was repeated twenty times with and without Mantri. The histograms plot the average reduction, error bars are the 10th and 90th percentiles of samples. We see that Mantri improves job completion times by roughly 25%. Further, by



(a) Completion Time

(b) Resource Usage

Figure 4.12: **Evaluation of Mantri as the default build for all jobs on a pre-production cluster for nine days.**

terminating the largest stragglers early, resource usage *falls* by roughly 10%. As we show later, this is because very few of the duplicates scheduled by the current mitigation scheme based on Dryad are useful.

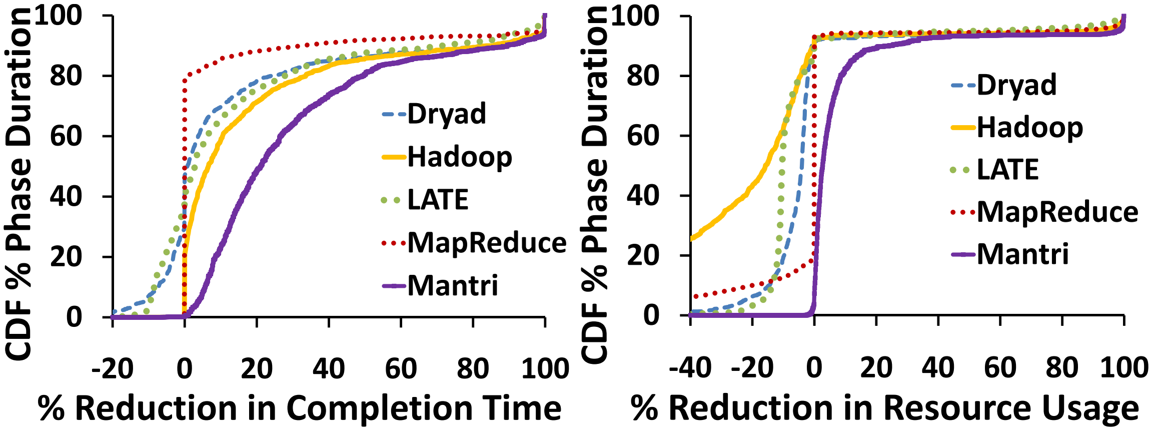
To micro-benchmark Mantri's estimators, we logged progress reports from these production runs. We find that Mantri's predictor, based on reports from the recent past, estimates t_{rem} to within a 2.9% error of the actual completion time. From the results above, we see that this accuracy suffices to see practical gains.

Jobs in the Wild All the jobs submitted to the pre-production cluster ran with Mantri for a nine day period. We compare these job runs with earlier runs of the same jobs that ran with the unmodified build. Figure 4.12a plots the CDF of the net improvement in completion times of 202 jobs. Jobs that occupy the cluster for half the time sped up by at least 14.3%. We see larger gains on the benchmark jobs in the production cluster and in trace driven simulations. This is because, the pre-production being lightly loaded has fewer outliers and hence, less room for Mantri to improve. Figure 4.12b also shows that 60% of jobs see a *reduction* in resource consumption while the others use up a few extra resources.

To compare against alternative schemes and to piece apart gains from the various algorithms in Mantri, we present results from the trace-driven simulator.

Figure 4.13 compares straggler mitigation strategies in their impact on completion time and resource usage. The y-axes weighs phases by their lifetime since improving the longer phases improves cluster efficiency. The figures plots the cumulative reduction in these metrics over each of the 210K phases in Table 1.3 with each phase repeated 3 times.

Figures 4.13a and 4.13b show that Mantri improves completion time by 21% and 42% at the 50th and 75th percentiles and reduces resource usage by 3% and 7% at these percentiles.



(a) Change in Completion Time

(b) Change in Resource Usage

Figure 4.13: **Comparing straggler mitigation strategies. Mantri provides a greater speed-up in completion time while using fewer resources than existing schemes.**

Results from simulation are consistent with those from our production deployment. We attribute these gains to the combination of *early action* and *cause-aware* restarts.

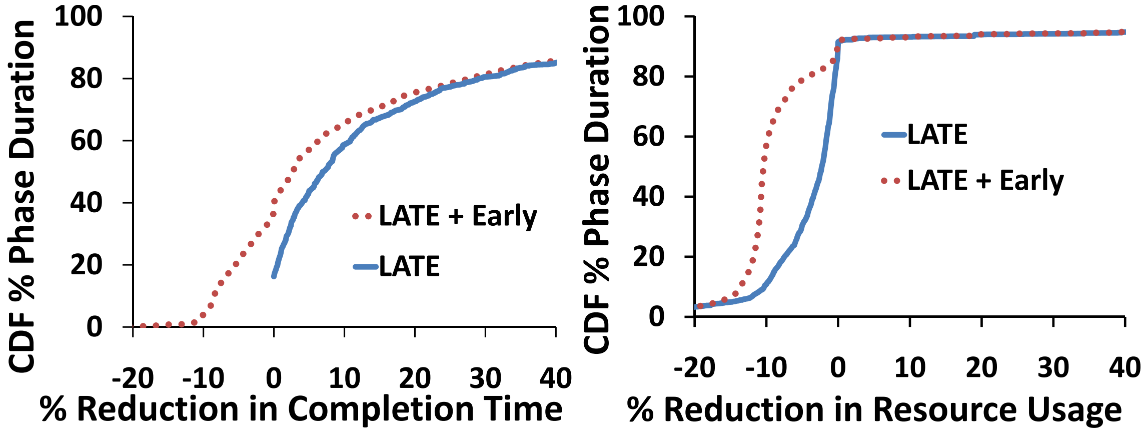
From Figure 4.13a, at the 50th percentile, Mantri sped up phases by 21.1%, an additional 3.1X over the 6.9% improvement of Hadoop, the next best scheme. To achieve this Hadoop uses 15.9% more resources (Fig.4.13b).

MapReduce and Dryad have no positive impact until the 80th and 50th percentile respectively. Up to the 30th percentile Dryad increases the completion time of phases. LATE is similar in its time improvement to Hadoop but does so using fewer resources.

The reason for poor performance is that they miss outliers that happen early in the phase and by not knowing the true causes of outliers, the duplicates they schedule are mostly not useful. Mantri and Dryad schedule .2 restarts per task for the average phase (.06 and .56 for LATE and Hadoop). But, Mantri's restarts have a success rate of 70% compared to the 15% for LATE. The other schemes have lower success rates.

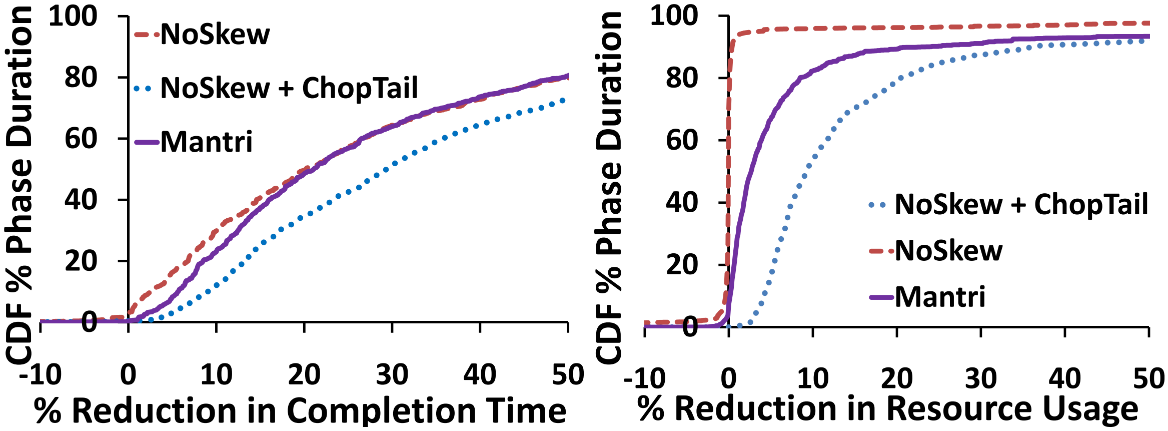
While the insight of *early action* on stragglers is valuable, it is nonetheless non trivial. We evaluate this in Figures 4.14a and 4.14b that present a form of LATE that is identical in all ways except that it addresses stragglers early. We see that addressing stragglers early increases completion time up to the 40th percentile, uses more resources and is worse than vanilla LATE. Being resource aware is crucial to get the best out of early action.

Finally, Fig. 4.15 shows that Mantri is on par with the ideal benchmark that has no variation in tasks *NoSkew* and is slightly worse than the variant that removes all durations in the top quartile, *NoSkew+ChopTail*.



(a) Time (b) Resources

Figure 4.14: **Extending LATE to speculate early results in worse performance**



(a) Time (b) Resources

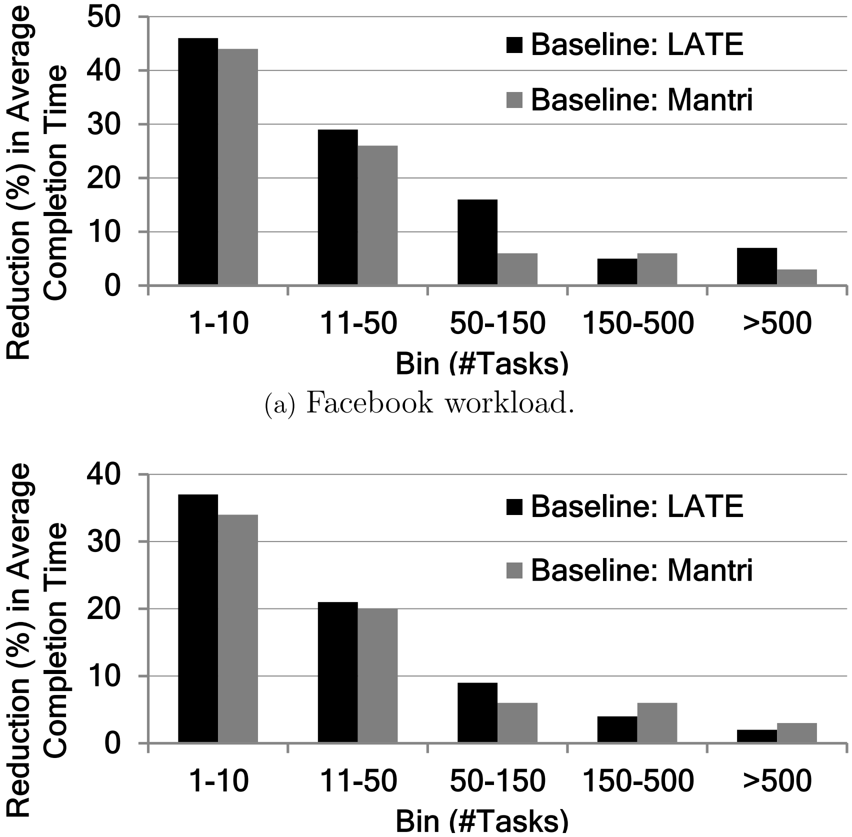
Figure 4.15: **Mantri is on par with an ideal *NoSkew* benchmark and slightly worse than *NoSkew+ChopTail***

4.4.2 Dolly Evaluation

Does Dolly mitigate stragglers?

We first present the improvement in completion time using Dolly. Unless specified otherwise, the cloning budget β is 5% and utilization threshold τ is 80%.

Dolly improves the average completion time of jobs by 42% compared to LATE and 40% compared to Mantri, in the Facebook workload. The corresponding improvements are 27% and 23% in the Bing workload. Figure 4.16 plots the improvement in different job bins.



(a) Facebook workload.

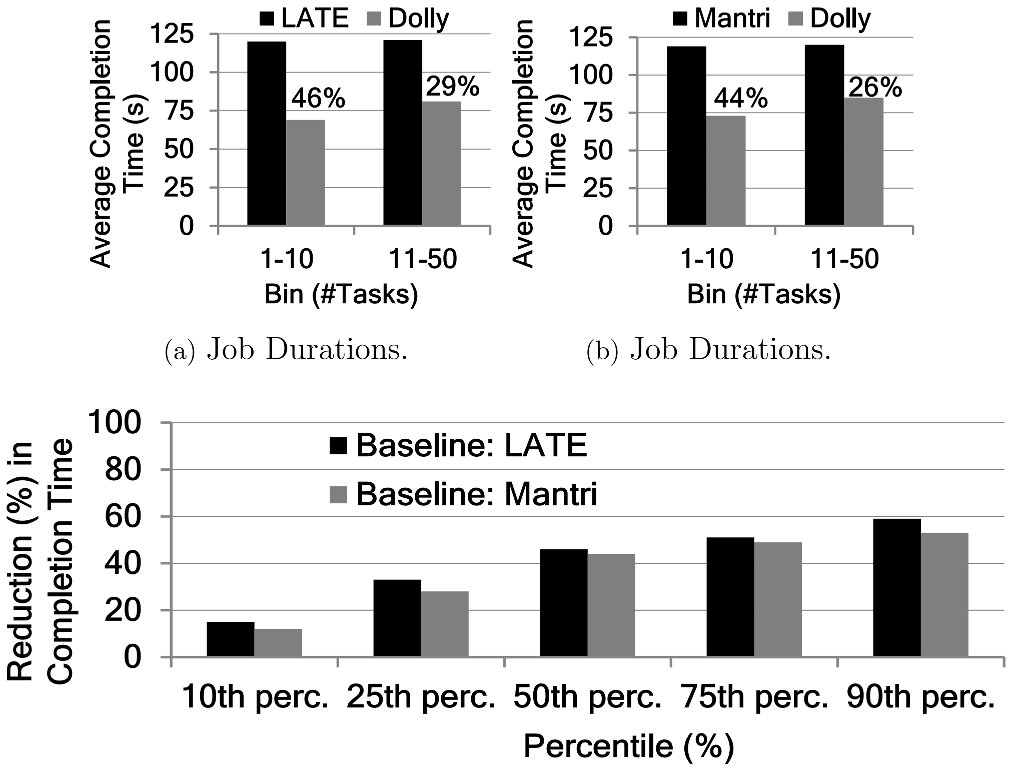
(b) Bing workload.

Figure 4.16: **Dolly's improvement for the Facebook and Bing workloads, with LATE and Mantri as baselines.**

Small jobs (bin-1) benefit the most, improving by 46% and 37% compared to LATE and 44% and 34% compared to Mantri, in the Facebook and Bing workloads. This is because of the power-law in job sizes and the policy of admission control. Figures 4.17a and 4.17b show the average duration of jobs in the smallest two bins with LATE and Mantri, and its reduction due to Dolly's cloning, for the Facebook workload. Figure 4.17c shows the distribution of gains for jobs in bin-1. We see that jobs improve by nearly 50% and 60% at the 75^{th} and 90^{th} percentiles, respectively. Note that even at the 10^{th} percentile, there is a non-zero improvement, demonstrating the seriousness and prevalence of the problem of stragglers in small jobs.

Figure 4.18 presents supporting evidence for the improvements. The ratio of medium to minimum progress rates of tasks, which is over 5 with LATE and Mantri in our deployment, drops to as low as 1.06 with Dolly. Even at the 95^{th} percentile, this ratio is only 1.17, thereby indicating that Dolly effectively mitigates nearly all stragglers.

The ratio not being exactly 1 shows that some stragglers still remain. One reason for



(a) Job Durations.

(b) Job Durations.

(c) Distribution of improvements ($\leq$ 10 tasks).

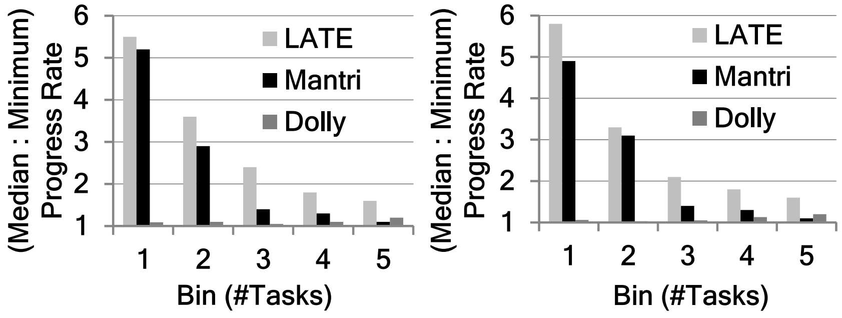
Figure 4.17: **Dissecting Dolly's improvements for the Facebook workload. Figures (a) and (b) show the duration of the small jobs before and after Dolly. Figure (c) expands on the distribution of the gains for jobs with $\leq$ 10 tasks.**

this is that while our policy of admission control is a good approximation (§4.3.2), it does not explicitly prioritize small jobs. Hence a few large jobs possibly deny the budget to some small jobs. Analyzing the consumption of the cloning budget shows that this is indeed the case. Jobs in bin-1 and bin-2 together consume 83% of the cloning budget. However, even jobs in bin-5 get a small share (2%) of the budget.

Improvements with our simulator that replays the entire traces are similar to our deployment results. Table 4.2 summarizes and compares the results.

Delay Assignment

Setting ω: Crucial to the above improvements is delay assignment's dynamic calculation of the wait duration of ω. The value of ω, picked using the analysis in §4.3.3, is updated every hour. It varied between 2.5s and 4.7s for jobs in bin-1, and 3.1s and 5.2s for jobs in bin-2.



(a) Facebook

(b) Bing

Figure 4.18: **Ratio of median to minimum progress rates of tasks within a phase. Bins are as per Table 2.1.**

	Baseline: LATE		Baseline: Mantri	
	Facebook	Bing	Facebook	Bing
Dep.	42% (46%)	27% (37%)	40% (44%)	23% (34%)
Sim.	43% (48%)	28% (41%)	41% (45%)	26% (36%)

* *Original cluster size*

Table 4.2: **Summary of results with the deployment and simulator. We list improvement in completion time due to Dolly; improvement of small jobs are in parentheses.**

The value of ω varies based on job sizes because the number of tasks in a job influences B, α and p_c. Figure 4.19 plots the variation with time. The sensitivity of ω to the periodicity of updating its value is low—using values between 30 minutes to 3 hours causes little change in its value.

CC and CAC: We now compare delay assignment to the two static assignment schemes, Contention Cloning (CC) and Contention Avoidance Cloning (CAC) in Figure 4.20, for the Bing workload. With LATE as the baseline, CAC and CC improve the small jobs by 17% and 26%, in contrast to delay assignment's 37% improvement (or up to 2.1× better). With Mantri as the baseline, delay assignment is again up to 2.1× better. In the Facebook workload, delay assignment is at least 1.7× better.

The main reason behind delay assignment's better performance is its accurate estimation of the effect of contention and the likelihood of stragglers. It uses sampling from prior runs to estimate both. Bandwidth estimation is 93% accurate without contention and 97% accurate with contention. Also, the probability of an upstream clone straggling is estimated to an accuracy of 95%.

Between the two, CC is a closer competitor to delay assignment than CAC, for small jobs. This is because they transfer only moderate amounts of data. However, contentions

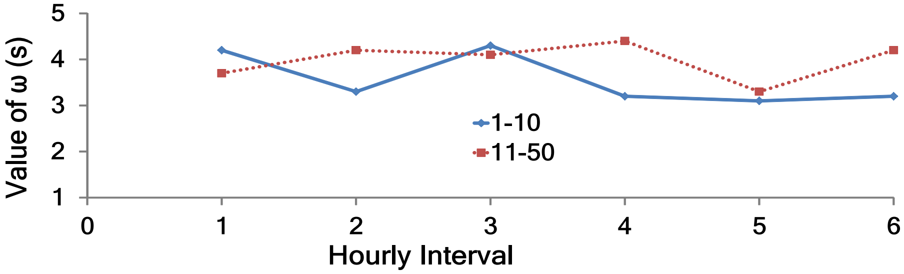


Figure 4.19: **Variation in ω when updated every hour.**

hurt large jobs as they transfer sizable intermediate data. As a result, CC's gains drop below CAC.

Number of Phases: Dryad jobs may have multiple phases (maximum of 6 in our Bing traces), and tasks of different phases have the same number of clones. More phases increases the chances of there being fewer exclusive copies of task outputs, which in turn worsens the effect of both waiting as well as contention. Figure 4.21 measures the consequent drop in performance. CAC's gains drop quickly while CC's performance drops at a moderate rate. Importantly, delay assignment's performance only has a gradual and relatively small drop. Even when the job has six phases, improvement is at 31%, a direct result of its deft cost-benefit analysis (§4.3.3).

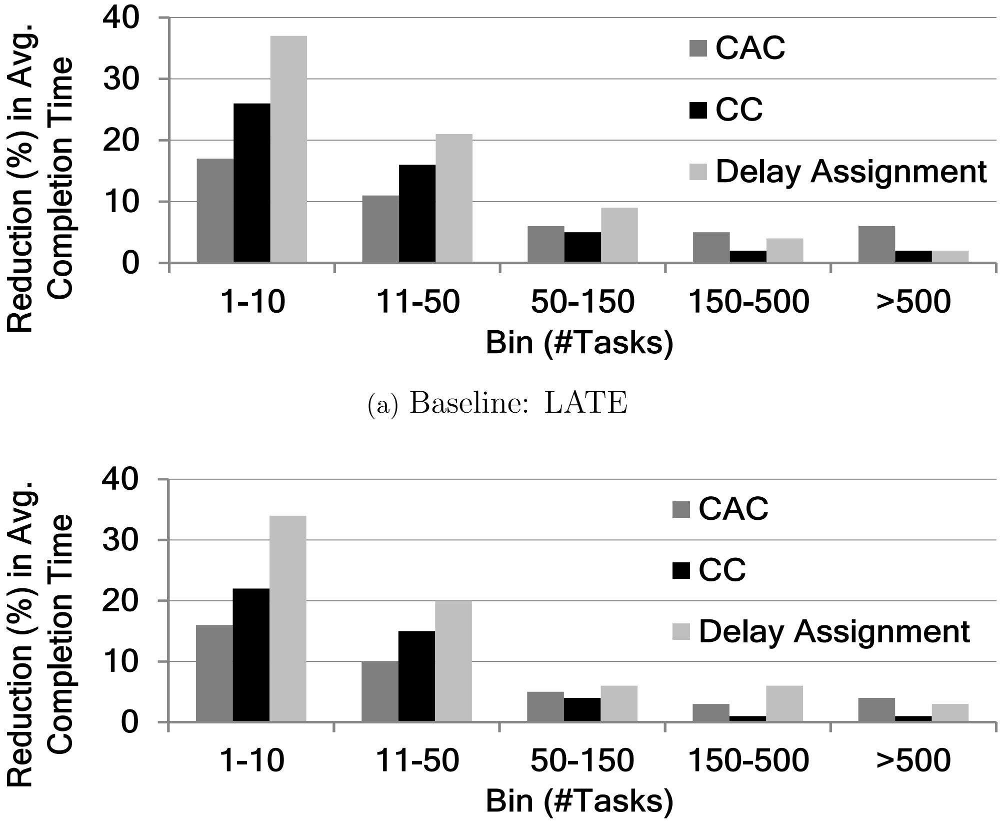
Communication Pattern: Delay assignment is generic to handle any communication pattern between phases. Figure 4.22 differentiates the gains in completion times of the *phases* based on their communication pattern. Results show that delay assignment is significantly more valuable for all-to-all communication patterns than the many-to-one and one-to-one patterns. The higher the dependency among communicating tasks, the greater the value of delay assignment's cost-benefit analysis.

Overall, we believe the above analysis shows the applicability and robust performance of Dolly's mechanisms to different frameworks with varied features.

Cloning Budget

The improvements in the previous sections are based on a cloning budget β of 5%. In this section, we analyze the sensitivity of Dolly's performance to β. We aim to understand whether the gains hold for lower budgets and how much further gains are obtained at higher budgets.

In the Facebook workload, overall improvement remains at 38% compared to LATE



(a) Baseline: LATE

(b) Baseline: Mantri

Figure 4.20: **Intermediate data contention. Delay Assignment is** 2.1× **better than CAC and CC (Bing workload).**

even with a cloning budget of only 3% (Figure 4.23a). Small jobs, in fact, see a negligible drop in gains. This is due to the policy of admission control to favor small jobs. Large jobs take a non-negligible performance hit though. In fact, in the Bing workload, even the small jobs see a drop of 7% when the budget is reduced from 5% to 3%. This is because job sizes in Bing are less heavy-tailed. However, the gains still stand at a significant 28% (Figure 4.23b).

Increasing the budget to 10% does not help much. Most of the gains are obtained by eliminating stragglers in the smaller jobs, which do not require a big budget.

In fact, sweeping the space of β (Figure 4.24) reveals that Dolly requires a cloning budget of at least 2% and 3% for the Facebook and Bing workloads, below which performance drops drastically. Gains in the Facebook workload plateau beyond 5%. In the Bing workload, gains for jobs in bin-1 plateau at 5% but the overall gains cease to grow only at 12%. While this validates our setting of β as 5%, clusters can set their budgets based on their utilizations and the jobs they seek to improve with cloning.

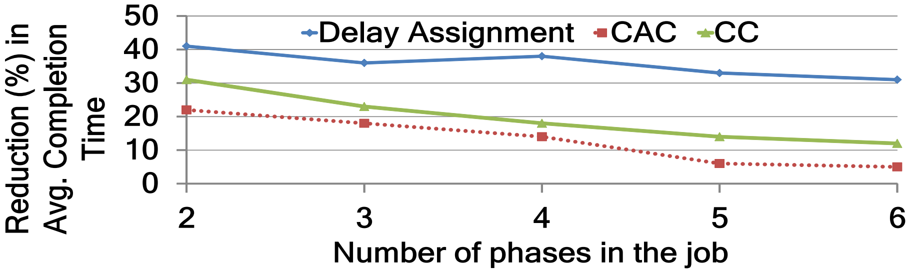


Figure 4.21: **Dolly's gains as the number of phases in jobs in bin-1 varies in the Bing workload, with LATE as baseline.**

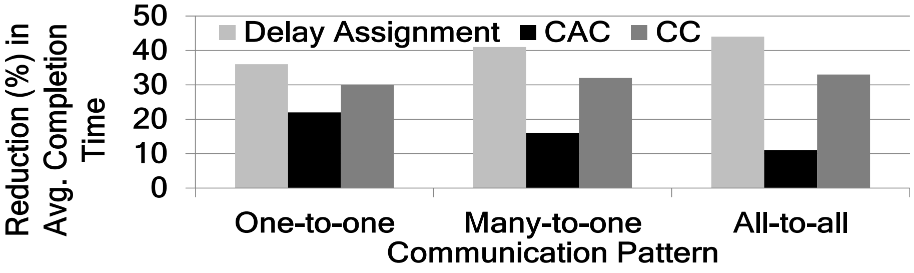
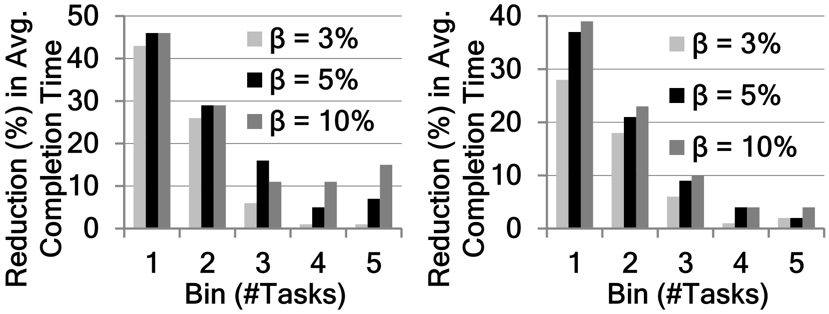


Figure 4.22: **Performance of Dolly across phases with different communication patterns in bin-1, in the Bing workload.**

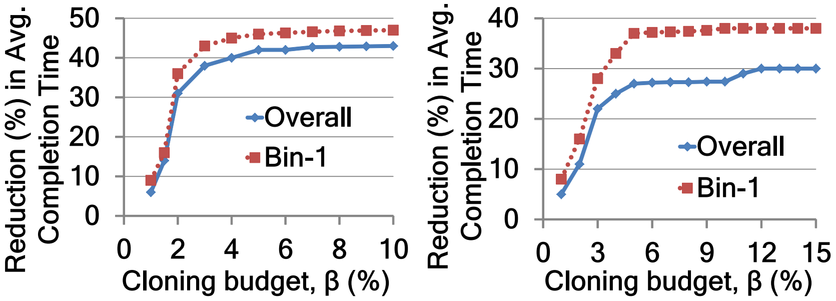


(a) Facebook (b) Bing

Figure 4.23: **Sensitivity to cloning budget (β). Small jobs see a negligible drop in performance even with a 3% budget.**

Admission Control

A competing policy to admission control (§4.3.2) is to preempt clones of larger jobs for the small jobs. Preemption is expected to outperform admission control as it explicitly prioritizes the small jobs; we aim to quantify the gap.



(a) Facebook

(b) Bing

Figure 4.24: **Sweep of β to measure the overall average completion time of all jobs and specifically those within bin-1.**

Figure 4.25 presents the results with LATE as the baseline and cloning budgets of 5% and 3%. The gains with preemption is 43% and 29% in the Facebook and Bing workloads, compared to 42% and 27% with the policy of admission control. This small difference is obtained by preempting 8% and 9% of the tasks in the two workloads. Lowering the cloning budget to 3% further shrinks this difference, even as more tasks are preempted. With a cloning budget of 3%, the improvements are nearly equal, even as 17% of the tasks are preempted, effectively wasting cluster resources. Admission control well approximates preemption due to the heavy tailed distribution. Note the near-identical gains for small jobs.

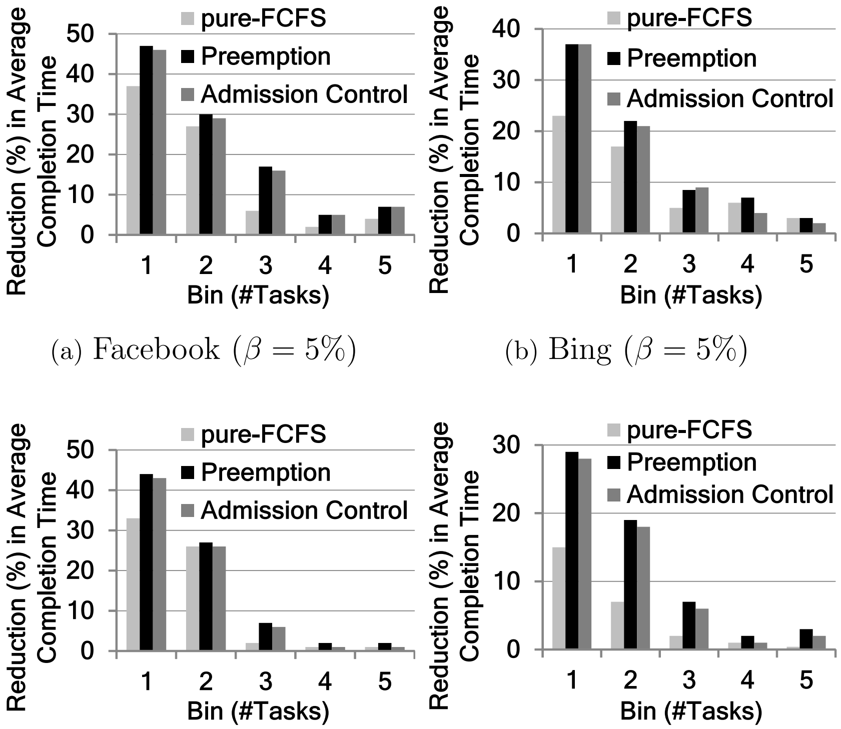
Doing neither preemption or admission control in allocating the cloning budget ("pure-FCFS") reduces the gains by nearly 14%, implying this often results in larger jobs denying the cloning budget to the smaller jobs.

4.5 Related Work

Outliers inevitably occur in systems that compete for a share of resource pools [97], of which mapreduce is one example. OpenDHT [83] and MONET [20] reported outliers over planetlab and wide-area Internet respectively.

Much recent work focuses on large scale data parallel computing. Following on the map-reduce [36] paper, there has been work in improving workflows [52, 70], language design [28, 80, 102], fair schedulers [56, 103], and providing privacy [84]. Our work here takes the next step of understanding how such production clusters behave and can be improved.

Run-time stragglers have been identified by past work [36, 52, 59, 71]. However, this paper is the first to characterize the prevalence of stragglers in production and their various causes. By understanding the causes, addressing stragglers early and scheduling duplicates only when there is a fair chance that the speculation saves both time and resources, our approach provides a greater reduction in job completion time while using fewer resources



(a) Facebook ($\beta = 5\%$)

(b) Bing ($\beta = 5\%$)

(c) Facebook ($\beta = 3\%$)

(d) Bing ($\beta = 3\%$)

Figure 4.25: **Admission Control. The policy of admission control well approximates the policy of preemption and outperforms pure-FCFS in utilizing the cloning budget.**

than prior strategies that duplicate tasks towards the end of a phase.

Current approaches [36,52,70] only duplicate tasks except for Late [71] which also stops using persistently slow machines. Logs from the production cluster show that persistent slowness occurs rarely and duplicates do not counter most of the causes of outliers, e.g., those that do a lot of work. Further, they do not avoid network hotspots or protect against loss of task output.

By only acting at the end of a phase, current approaches [36, 52, 70] miss early outliers. They vary in the choice of which among the tasks that remain at the end of a phase to duplicate. After a threshold number of tasks have finished, MapReduce [36] duplicates all the tasks that remain. Dryad [70] duplicates those that have been running for longer than the 75th percentile of task durations. After all tasks have started, Hadoop [52] uses slots that free up to duplicate any task that has read less data than the others, while Late [71] duplicates only those reading at a slow rate.

Replicating tasks in distributed systems have a long history [25, 34, 67], and have been studied extensively [32, 47, 77] in prior work. These studies conclude that modeling running tasks and using it for predicting and comparing performance of other tasks is the hardest component, errors in which often cause degradation in performance. We concur with a similar observation in our traces.

The problem of stragglers was identified in the original MapReduce paper [36]. Since then solutions have been proposed to fix it using speculative executions [19, 70, 71]. Despite these techniques, stragglers remain a problem in small jobs. Dolly addresses their fundamental limitation—wait to observe before acting—with a proactive approach of cloning jobs. It does so using few extra resources by relying on the power-law of job sizes.

Based on extensive research on detecting faults in machines (e.g., [38, 39, 49, 55, 73]), datacenters periodically check for faulty machines and avoid scheduling jobs on them. However, stragglers continue to occur on the non-blacklisted machines. Further improvements to blacklisting requires a root cause analysis of stragglers in small jobs. However, this is intrinsically hard due to the complexity of the hardware and software modules, a problem recently acknowledged in Google's clusters [10].

In fact, Google's clusters aim to make jobs "predictable out of unpredictable parts" [10]. They overcome vagaries in performance by scheduling backup copies for every job. Such backup requests are also used in Amazon's Dynamo [44]. This notion is similar to Dolly. However, these systems aim to overcome variations in scheduling delays on the machines, not runtime stragglers. Therefore, they cancel the backup copies once one of the copies starts. In contrast, Dolly has to be resilient to runtime variabilities which requires functioning within utilization limits and efficiently handle intermediate data.

Finally, our delay assignment model is similar to the idea of delay scheduling [104] that delays scheduling tasks for locality. We borrow this idea in Dolly, but crucially, pick the value of the delay based on a cost-benefit analysis weighing contention versus waiting for slower tasks.

Chapter 5

Conclusions and Future Work

Several interesting and important issues remain in designing frameworks for parallel jobs, borne out of theoretical complexities as well as evolving usage trends. We detail some of there as future research directions.

Optimal Replacement for Parallel Jobs: An unanswered question is the optimal cache replacement strategy to minimize average completion time of parallel jobs or maximize cluster efficiency. The optimal algorithm picks that block for replacement whose absence hurts the least when the entire trace is replayed. Note the combinatorial explosion as a greedy decision for each replacement will not be optimal. We outline the challenges in formulating such an oracular algorithm.

The completion time of a job is a function of when its tasks get scheduled, which in turn is dependent on the availability of compute slots. An aspect that decides the availability of slots for a job is its fair share. So, when executing tasks of a job finish, slots open up for its unscheduled tasks. Modeling this requires knowing the speed-up due to memory locality but that is non-trivial because it varies across tasks of even the same job. Further, scheduler policies may allow jobs to use some extra slots if available. Hence one has to consider scheduler policies on using extra slots as well as the mechanism to reclaim those extra slots (e.g., killing of running tasks) when new jobs arrive. Precise modeling of the speed-ups of tasks, scheduler policies and job completion times will help formulate the optimal replacement scheme and evaluate room for improvement over LIFE and LFU-F.

Pre-fetching: A challenge for any cache is data that is accessed only once. While our workloads have only a few jobs that read such singly-accessed blocks, they nonetheless account for over 30% of all tasks. Pre-fetching can help provide memory locality for these tasks.

We consider two types of pre-fetching. First, as soon as a job is submitted, the scheduler knows its input blocks. It can inform the PACMan coordinator which can start pre-fetching parts of the input that is not in cache, especially for the later waves of tasks. This approach is helpful for jobs consisting of multiple waves of execution. This also plays nicely with LIFE's policy of favoring small single-waved jobs. The blocks whose files are absent are likely to be those of large multi-waved jobs. Any absence of their input blocks from the cache can be rectified through pre-fetching. Second, recently created data (e.g., output of jobs or logs imported into the file system) can be pre-fetched into memory as they are likely to be accessed in the near future, for example, when there are a chain of jobs. We believe that an investigation and application of different pre-fetching techniques will further improve cluster efficiency.

Speculation for Approximation Jobs: Increasingly, with the deluge of data, analytics applications no longer require processing entire datasets. Instead, they choose to tradeoff accuracy for response time. *Approximate* results obtained early from just part of the dataset are often *good enough* [14, 89, 93]. Approximation is explored across two dimensions—time for obtaining the result (deadline) and error in the result [64].

- *Deadline-bound* jobs strive to maximize the accuracy of their result within a specified time limit. Such jobs are common in real-time advertisement systems and web search engines. Generally, the job is spawned on a large dataset and accuracy is proportional to the fraction of data processed [46,54,65] (or tasks completed, for ease of exposition).

- *Error-bound* jobs strive to minimize the time taken to reach a specified error limit in the result. Again, accuracy is measured in the amount of data processed (or tasks completed). Error-bound jobs are used in scenarios where the value in reducing the error below a limit is marginal, e.g., counting of the number of cars crossing a section of a road to the nearest thousand is sufficient for many purposes.

Approximation jobs require schedulers to *prioritize the appropriate subset of their tasks* depending on the deadline or error bound. Prioritization is important for two reasons. First, due to cluster heterogeneities [18, 37, 71], tasks take different durations even if assigned the same amount of work. Second, jobs are often *multi-waved*, i.e., their number of tasks is much more than available compute slots [41], thereby they run only a fraction of their tasks at a time. The trend of multi-waved jobs is only expected to grow with smaller tasks [75].

Optimally prioritizing tasks of a job to slots is a classic scheduling problem with known heuristics [58, 63, 79]. These heuristics, unfortunately, do not directly carry over to our scenario for the following reasons. First, they calculate the optimal ordering statically. Straggling of tasks, on the other hand, is unpredictable and necessitates dynamic modification of the priority ordering of tasks according to the approximation bounds. Second, and most importantly, traditional prioritization techniques assign tasks to slots assuming every task to occupy only one slot. Spawning a speculative copy, however, leads to the same task using two (or multiple) slots simultaneously. Hence, this distills our challenge to *achieving the approximation bounds by dynamically weighing the gains due to speculation against the cost of using extra resources for speculation.*

Bibliography

[1] Akamai content distribution network. http://www.akamai.com/.

[2] Amazon elastic compute cloud. http://aws.amazon.com/ec2/instance-types/.

[3] Applications and organizations using hadoop. http://wiki.apache.org/hadoop/PoweredBy.

[4] Cloud compute can save govt agencies 25-50% in costs. http://googlepublicpolicy.blogspot.com/2010/04/brookings-cloud-computing-can-save-govt.html.

[5] The coral content distribution network. http://www.coralcdn.org/.

[6] Hadoop distributed file system. http://hadoop.apache.org/hdfs.

[7] Hadoop Slowstart. https://issues.apache.org/jira/browse/MAPREDUCE-1184/.

[8] Hive. http://wiki.apache.org/hadoop/Hive.

[9] iostat - linux users manual. http://linuxcommand.org/man_pages/iostat1.html.

[10] J. Dean. *Achieving Rapid Response Times in Large Online Services.* http://research.google.com/people/jeff/latency.html.

[11] The Global Memory System (GMS) Project. http://www.cs.washington.edu/homes/levy/gms/.

[12] The NOW Project. http://now.cs.berkeley.edu/.

[13] J. Wilkes and C. Reiss., 2011. https://code.google.com/p/googleclusterdata/wiki/ClusterData2011_1.

[14] Interactive Big Data analysis using approximate answers, 2013. http://tinyurl.com/k5favda.

[15] A. Greenberg, N. Jain, J. Hamilton, S. Kandula, C. Kim, P. Lahiri, D. Maltz, P. Patel, S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *ACM SIGCOMM*, 2009.

[16] A. Thusoo . Data warehousing and analytics infrastructure at facebook. In *SIGMOD*, 2010.

[17] G. Ananthanarayanan, C. Douglas, R. Ramakrishnan, S. Rao, and I. Stoica. True Elasticity in Multi-Tenant Clusters through Amoeba. In *ACM SoCC*, 2012.

[18] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective Straggler Mitigation: Attack of the Clones. In *USENIX NSDI*, 2013.

[19] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, E. Harris, and B. Saha. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[20] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Rao. Improving web availability for clients with monet. In *NSDI*, 2005.

[21] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. In *ACM Symposium on Operating Systems Principles (SOSP)*, 1995.

[22] M. Arlitt, L. Cherkasova, J. Dilley, R. Friedrich, and T. Jin. Evaluating Content Management Techniques for Web Proxy Caches. In *WISP*, 1999.

[23] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. Joseph, R. Katz, S. Shenker, I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[24] L. N. Bairavasundaram, G. R. Goodson, B. Schroeder, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. An analysis of data corruption in the storage stack. In *FAST*, 2008.

[25] A. Baratloo, M. Karaul, Z. Kedem, and P. Wycko. Charlotte: Metacomputing on the Web. In *9th Conference on Parallel and Distributed Computing Systems*, 1996.

[26] L. A. Belady. A Study of Replacement Algorithms for Virtual-Storage Computer. *IBM Systems Journal*, 1966.

[27] N. M. Belaramani, J. Zheng, A. Nayate, R. Soul, M. Dahlin, and R. Grimm. Pads: A policy architecture for distributed storage systems. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[28] C. Olston, B. Reed, U. Srivastava, R. Kumar and A. Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In *ACM SIGMOD*, 2008.

[29] C. Reiss, A. Tumanov, G. Ganger, R. H. Katz, M. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *ACM SoCC*, 2012.

[30] Y. Chen, A. Ganapathi, and R. Katz. To Compress or Not To Compress - Compute vs. IO tradeoffs for MapReduce Energy Efficiency. In *Proceedings of the First ACM SIGCOMM Workshop on Green Networking*, 2010.

[31] L. Cherkasova and G. Ciardo. Role of Aging, Frequency, and Size in Web Cache Replacement Policies. In *HPCN EUrope*, 2001.

[32] W. Cirne, D. Paranhos, F. Brasileiro, L. F. W. Goes, and W. Voorsluys. On the Efficacy, Efficiency and Emergent Behavior of Task Replication in Large Distributed Systems. In *Parallel Computing*, 2007.

[33] K. M. Curewitz, P. Krishnan, and J. S. Vitter. Practical prefetching via data compression. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 257–266, New York, NY, USA, 1993. ACM.

[34] E. K. D. Anderson, J. Cobb. SETI@home: An Experiment in Public-Resource Computing. In *Comm. ACM*, 2002.

[35] D. Narayanan and A. Donnelly and E. Thereska and S. Elnikety and A. Rowstron. Everest: Scaling down peak loads through i/o off-loading. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[36] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*,

2004.

[37] E. Bortnikov, A. Frank, E. Hillel, S. Rao. Predicting Execution Bottlenecks in Map-Reduce Clusters. In *USENIX HotCloud*, 2012.

[38] J. G. Elerath and S. Shah. Dependence upon fly-height and quantity of heads. In *Annual Symposium on Reliability and Maintainability*, 2003.

[39] J. G. Elerath and S. Shah. Server class disk drives: How reliable are they? In *Annual Symposium on Reliability and Maintainability*, 2004.

[40] M. J. Franklin, M. J. Carey, and M. Livny. Global memory management in client-server database architectures. In *Very Large Data Bases*, 1992.

[41] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, I. Stoica. PACMan: Coordinated Memory Caching for Parallel Jobs. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[42] G. Ananthanarayanan, A. Ghodsi, S. Shenker, I. Stoica. Disk Locality Considered Irrelevant. In *USENIX HotOS*, 2011.

[43] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, E. Harris. Scarlett: Coping with Skewed Popularity Content in MapReduce Clusters. In *European Conference on Computer Systems*, 2011.

[44] G. DeCandia and D. Hastorun and M. Jampani and G. Kakulapati and A. Lakshman and A. Pilchin and S. Sivasubramanian and P. Vosshall and W. Vogels. Dynamo: Amazons Highly Available Key-value Store. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[45] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, G. Czajkowski. Pregel: A System for Large-Scale Graph Processing. In *ACM SIGMOD*, 2010.

[46] M. Garofalais and P. Gibbons. Approximate Query Processing: Taming the Terabytes. In *VLDB*, 2001.

[47] G. Ghare and S. Leutenegger. Improving Speedup and Response Times by Replicating Parallel Programs on a SNOW. In *JSSPP*, 2004.

[48] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, March 1969.

[49] J. Gray and C. van Ingen. Empirical measurements of disk failure rates and error rates. In *Technical Report MSR-TR-2005-166*, 2005.

[50] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: Automatic Management of Data and Computation in Data Centers. In *OSDI'10: Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[51] H. Garcia-Molina and K. Salem. Main Memory Database Systems: An Overview. In *IEEE Transactions on Knowledge and Data Engineering*, 1992.

[52] http://hadoop.apache.org.

[53] J. H. Hartman and J. K. Ousterhout. The Zebra Striped Network File System. In *ACM SOSP*, 1993.

[54] J. Hellerstin, P. Haas, and H. Wang. Online Aggregation. In *ACM SIGMOD*, 1997.

[55] E. Ipek, M. Krman, N. Krman, and J. F. Martinez. Core Fusion: Accommodating

Software Diversity in Chip Multiprocessors. In *ISCA*, 2007.

[56] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair Scheduling for Distributed Computing Clusters. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[57] J. Ousterhout *et al.* The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM. In *SIGOPS Operating Systems Review*, 2009.

[58] L. Kleinrock. *Queueing systems, volume II: computer applications.* John Wiley & Sons New York, 1976.

[59] S. Ko, I. Hoque, B. Cho, and I. Gupta. On Availability of Intermediate Data in Cloud Comput. In *HotOS*, 2009.

[60] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. A Study of Skew in MapReduce Applications. In *Open Cirrus Summit*, 2011.

[61] L. A. Barroso. Warehouse-scale computing: Entering the teenage decade. In *ISCA*, 2011.

[62] L. A. Barroso and U. Holzle. The Case for Energy-Proportional Computing. In *Computer, 40(12)*, 2007.

[63] C. Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard-real-time Environment. *Journal of the ACM (JACM)*, 1973.

[64] J. Liu, K. Shih, W. Lin, R. Bettati, and J. Chung. Imprecise Computations. *Proceedings of the IEEE*, 1994.

[65] S. Lohr. *Sampling: design and analysis.* Thomson, 2009.

[66] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM*, 2008.

[67] M. C. Rinard and P. C. Diniz. Commutativity Analysis: A New Analysis Framework for Parallelizing Compilers. In *ACM PLDI*, 1996.

[68] M. Chowdhury, M. Zaharia, J. Ma, M. Jordan, I. Stoica. Managing Data Transfers in Computer Clusters with Orchestra. In *ACM SIGCOMM*, 2011.

[69] M. H.-B. M. E. Crovella, R. Frangioso. Connection Scheduling in Web Servers. In *USENIX USITS*, 1999.

[70] M. Isard, M. Budiu, Y. Yu, A. Birrell and D. Fetterly. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *ACM European Conference on Computer Systems*, 2007.

[71] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[72] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M.J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[73] A. Merchant, M. Uysal, P. Padala, X. Zhu, S. Singhal, and K. Shin. Maestro: Quality-of-Service in Large Disk Arrays. In *ACM ICAC*, 2011.

[74] M. N. Nelson, B. B. Welch, and J. K. Ousterhout. Caching in the Sprite Network File

System. *ACM TOCS*, Feb 1988.

[75] K. Ousterhout, A. Panda, J. Rosen, S. Venkataraman, R. Xin, S. Ratnasamy, S. Shenker, and I. Stoica. The Case for Tiny Tasks in Compute Clusters. In *USENIX HotOS*, 2013.

[76] P. Skibinski and J. Swacha. 2007: Fast and efficient log file compression. In *CEUR ADBIS 2007*.

[77] D. Paranhos, W. Cirne, and F. Brasileiro. Trading Cycles for Information: Using Replication to Schedule Bag-of-Tasks Applications on Computational Grids. In *Euro-Par*, 2003.

[78] P.Cao and S.Irani. Cost Aware WWW Proxy Caching Algorithms. In *USENIX USITS*, 1997.

[79] M. Pinedo. *Scheduling: theory, algorithms, and systems*. Springer, 2012.

[80] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, J. Zhou. SCOPE: Easy and Efficient Parallel Processing of Massive Datasets. In *Very Large Data Bases*, 2008.

[81] R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[82] V. Ramasubramanian and E. G. Sirer. Beehive: O(1)lookup performance for power-law query distributions in peer-to-peer overlays. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2004.

[83] S. Rhea, B.-G. Chun, J. Kubiatowicz, and ScottShenker. Fixing the embarrassing slowness of opendht on planetlab. In *WORLDS*, 2005.

[84] I. Roy, S. T. Shetty, A. Kilzer, V. Shmatikov, and E. Witchel. Airavat: Security and privacy for mapreduce. In *NSDI*, 2010.

[85] S. T. L. S. Ghemawat, H. Gobioff. The Google File System. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[86] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, R. Chaiken. Nature of Datacenter Traffic: Measurements and Analysis. In *IMC*, 2009.

[87] S. Ko, I. Hoque, B. Cho, I. Gupta. Making Cloud Intermediate Data Fault-Tolerant. In *ACM SOCC*, 2010.

[88] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, T. Vassilakis. Dremel: Interactive Analysis of Web-Scale Datasets. In *Very Large Data Bases*, 2010.

[89] S.Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *EuroSys*. ACM, 2013.

[90] P. Skibinski and S. Grabowski. Variable-length contexts for PPM. In *DCC'04: Proceedings of IEEE Data Compression Conference*, 2004.

[91] G. Soundararajan, C. Amza, and A. Goel. Database replication policies for dynamic content applications. In *ACM European Conference on Computer Systems*, 2006.

[92] J. Stribling, Y. Sovran, I. Zhang, X. Pretzer, J. Li, M. F. Kaashoek, and R. Morris. Flexible, wide-area storage for distributed systems with wheelfs. In *Symposium on*

Networked Systems Design and Implementation (NSDI), 2009.

[93] T. Condie, N. Conway, P. Alvaro, J. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce Online. In *USENIX NSDI*, 2010.

[94] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, R. Sears. MapReduce Online. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2010.

[95] A. Verma, R. Koller, L. Useche, and R. Rangaswami. Srcmap: Energy proportional storage using dynamic consolidation. In *FAST*, 2010.

[96] S. Williams, M. Abrams, C. R. Standridge, G. Abdulla, and E. A. Fox. Removal Policies in Network Caches for World-Wide Web Documents. In *ACM SIGCOMM*, 1996.

[97] D. Wischik, M. Handley, and M. B. Braun. The resource pooling principle. www.cs.ucl.ac.uk/staff/D.Wischik/Research/respool.html.

[98] Y. Chen, S. Alspaugh, D. Borthakur, R. Katz. Energy Efficiency for Large-Scale MapReduce Workloads with Significant Interactive Analysis. In *ACM European Conference on Computer Systems*, 2012.

[99] Y. Yu *et al.* Distributed Aggregation for Data-Parallel Computing: Interfaces and Implementations. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[100] Y. Yu, P. K. Gunda, M. Isard. Distributed Aggregation for Data-Parallel Computing: Interfaces and Implementations. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[101] T. Yeh, D. D. E. Long, and S. A. Brandt. Increasing predictive accuracy by prefetching multiple program and user specific files. *High Performance Computing Systems and Applications, Annual International Symposium on*, 0:12, 2002.

[102] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A System for General-Purpose Data-Parallel Computing Using a Language. In *OSDI*, 2008.

[103] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Job scheduling for multi-user mapreduce clusters. Technical Report EECS-2009-55, UCBerkeley.

[104] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *ACM European Conference on Computer Systems*, 2010.