IMPROVING THE END-TO-END LATENCY OF DATACENTER APPLICATIONS USING COORDINATION ACROSS APPLICATION COMPONENTS

BY

VIRAJITH JALAPARTI

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

Doctoral Committee:

Associate Professor Matthew Caesar, Chair
Associate Professor Indranil Gupta
Professor Klara Nahrstedt
Dr. Srikanth Kandula, Microsoft Research

Abstract

To handle millions of user requests every second and process hundreds of terabytes of data each day, many organizations have turned to large datacenter-scale computing systems. The applications running in these datacenters consist of a multitude of dependent logical components or *stages* which perform specific functionality. These stages are connected to form a directed acyclic graph (DAG), with edges representing input-output dependencies. Each stage can run over tens to thousands of machines, and involves multiple cluster *sub-systems* such as storage, network and compute. The scale and complexity of these applications can lead to significant delays in their *end-to-end latency*. However, the organizations running these applications have strict requirements on this latency as it directly affects their revenue and operational costs.

Addressing this problem, the goal of this dissertation is to develop scheduling and resource allocation techniques to optimize for the end-to-end latency of datacenter applications. The key idea behind these techniques is to utilize coordination between different application components, allowing us to efficiently allocate cluster resources. In particular, we develop planning algorithms that coordinate the storage and compute sub-systems in datacenters to determine *how many* resources should be allocated to each stage in an application along with *where* in the cluster should they be allocated, to meet application requirements (e.g., completion time goals, minimize average completion time etc.). To further speed up applications at runtime, we develop a few latency reduction techniques: *reissuing* laggards elsewhere in the cluster, returning partial results and speeding up laggards by giving them extra resources. We perform a global optimization to coordinate across all the stages in an application DAG and determine which of these techniques works best for each stage, while ensuring that the cost incurred by these techniques is within a given *end-to-end budget*. We use application characteristics to predict and determine how resources should be allocated to different application components to meet the end-to-end latency requirements.

We evaluate our techniques on two different kinds of datacenter applications: (a)

web services, and (b) data analytics. With large-scale simulations and an implementation in Apache Yarn (Hadoop 2.0), we use workloads derived from production traces to show that our techniques can achieve more than 50% reduction in the 99th percentile latency of web services and up to 56% reduction in the median latency of data analytics jobs.

To my mom and dad,
my grandmother and grandfather.

Acknowledgments

I am extremely grateful to my advisor, Matthew Caesar. He has greatly influenced my growth as a researcher, helping me understand how to find important research directions. He was always questioning my assumptions which made me think more clearly and concretely about problems. His attention to clarity and detail allowed me to greatly improve my writing and presentation skills, and mature as an academic. He has been very supportive during my PhD and gave me the freedom to work on any problem that interested me. I would like to thank Matt for being an amazing advisor.

During my PhD, I have had the privilege of working with some great researchers. In fact, this dissertation is largely part of collaborations with Hitesh Ballani, Peter Bodik, Paolo Costa, Srikanth Kandula, Thomas Karagiannis, Ishai Menache, Sriram Rao, Ant Rowstron, Konstantin Makarychev, and Chenyu Yan. The initial ideas in this dissertation (part of Chapter 3 and Chapter 4) were developed when I was an intern at Microsoft Research-Cambridge working with Hitesh, Paolo, Thomas and Ant. I continued to pursue similar ideas and extended them to other applications (part of Chapter 4 and Chapter 5) working as an intern at Microsoft Research-Redmond with Peter, Srikanth, and Ishai along with Sriram, Kostya, and Chenyu.

I would like to thank Hitesh for his mentorship over the years. During my internship at Cambridge, he gave me ample space and time to think, and come up with my own solutions to problems. He was very approachable, and helped make my internship a fun and memorable one. He has continued to give me great advice and feedback, which was always valuable.

I would also like to thank Ishai for giving me the opportunity to intern with him at Redmond. Ishai was more than an internship mentor to me over the past three years, and was very helpful and supportive during my stay in Redmond. He had helped me tremendously during my job search and his advice was always extremely useful.

I have also learned a great deal from Srikanth and Peter. Srikanth's approach to

research helped me understand how to work effectively and efficiently, question any results I obtained and precisely articulate my thoughts. Srikanth had also kindly agreed to be part of my PhD committee. Peter was very helpful with his attention to details, which was especially important to quickly get up to speed during the start of my internship. Thank you Srikanth and Peter.

I would like to thank Indranil Gupta and Klara Nahrstedt for serving on my PhD committee, and for their invaluable feedback on this dissertation.

Kobus van der Merwe, Jeffery Pang and Seungjoon Lee were all very helpful and supportive during my first internship at AT&T Labs - Research. I feel fortunate to have worked with them early on in my research career. I have learned a great deal from them. Even though I never got a chance to work closely with Brighten Godfrey, he had given me very useful advice and feedback over the years and for this, I would like to thank him.

I am thankful to my labmates — Rachit Agarwal, Jason Croft, Mo Dong, Fred Douglas, Soudeh Ghorbani, Chi-Yao Hong, Sangeetha Abdu Jyothi, Ahmed Khurshid, Qingxi Li, Ankit Singla, Rashid Tahir, Ashish Vulimiri, and Wenxuan Zhou — for their feedback and conversations, which made the last six years of my life in Seibel Center a bit easier. Rachit has been a great friend, collaborator and mentor to me during the whole time. I fondly remember working on my first paper with him, the long walks and the amazing, often challenging, chats. I greatly appreciate his help and advice, especially during the early years of my PhD.

My life during PhD would not have been easy without so many amazing friends. Although we started out as just flatmates, Parasara Sridhar Duggirala, Naveen Cherukuri and I have grown to become really great friends. Anirudh Vemula and I had many interesting conversations over the years, and we quickly became good friends. The three of them have been very supportive and helpful at all the highs and lows during my PhD. I would also like to thank Aditya Pabbaraju, Bhargava Reddy, Nanda Kishore, Sangeetha Abdu Jyothi, Prasanna Giridhar, Dileep Kini and Pranav Garg for their friendship and companionship over the last three years, and great trips together. I have also made some amazing friends and reconnected with old ones during my internships at AT&T, MSR Cambridge and MSR Redmond — Shankaranarayanan P N, Tejaswini Narayanan, Shashank Chakelam, Rahul Potharaju, Ramik Sadana, Amit Soni, and Harsh Verma. Thank you all for the great times. I would like to particularly thank Shashank and Amit for all the amazing hikes.

I would like to thank all my friends from IIT Kanpur who helped me get through

four years of undergrad, and who challenge me even to this day. Thank you Dilip, Gopi, Pavan, Subhash, Sandeep, Vamsi, Veerendra and Venkatesh.

This dissertation would not have been possible without two people — my mother, Krishna Kumari and my father, Dr. Nageswara Rao. There are not enough words to express my gratitude to them for their love, unceasing support, and encouragement. My mother worked very hard for us, and I really appreciate her tolerance and patience with me. My dad held me to very high standards, and taught me the value of education and hard work. They have been very instrumental in my growth and success over the years. My grandfather, Bala Kishore and my grandmother, Ayushmati Devi always trusted that I would be successful in anything I got involved with. I thank them for their unwavering faith and for all their pampering. I want to thank my brother, Virinchi and my sisters, Dedeepya and Amulya, for helping me grow up. I must apologize to them for not always being there. I am also grateful for the support of my uncles and aunts, Satyanarayana, Krishna Jyothi, Vijay Kumar and Bharathi. This dissertation is dedicated to everyone in my family.

Table of Contents

List of Figures x

Chapter 1 Introduction 1
1.1 Problem statement 4
1.2 Dissertation scope 4
1.3 Challenges 6
1.4 Dissertation outline and contributions 7
1.5 Roadmap 14

Chapter 2 Characterizing datacenter applications 15
2.1 Web services 16
2.2 Data analytics applications 21
2.3 Resource malleability 22

Chapter 3 Estimating end-to-end latency using application characteristics . . 27
3.1 Data analytics applications 29
3.2 Web services 35
3.3 Evaluation 37
3.4 Related work 41
3.5 Conclusion 42

Chapter 4 Resource planning for datacenter applications 43
4.1 Overview 46
4.2 Resource selection 49
4.3 Offline planning 52
4.4 Evaluation of resource selector 60
4.5 Evaluation of offline planner 69
4.6 Discussion 82
4.7 Related work 83
4.8 Conclusion 85

Chapter 5 Speeding up datacenter applications at runtime 86
5.1 Causes for high latencies in web service applications 88
5.2 Key ideas in Kwiken 90

5.3 Design of Kwiken 93
5.4 Evaluation 100
5.5 Discussion 108
5.6 Related work 109
5.7 Conclusion 110

Chapter 6 Conclusions and future work 111
6.1 Future work 112

References 115

List of Figures

1.1 Architecture of Concorde. 8

2.1 Timeline diagram of the processing involved for web search stage at Microsoft Bing. 16
2.2 A few characteristics of the analyzed web service DAGs. 17
2.3 Scatter plots of different topology metrics with the number of stages in each web service DAG at Microsoft Bing. Each circle corresponds to one DAG in the dataset. 18
2.4 Estimating the variability in latency: (a) CDF of 99th percentile to median latency and (b) mean vs. standard deviation of latency, for stages and web service DAGs. 19
2.5 For a subset of stages in production, this plot shows normalized latency variance of the stage as a function of fraction of requests that are reissued. 20
2.6 CDF of the number of compute slots requested by jobs across three production clusters. The vertical line represents 240 slots, the size of one rack. 21
2.7 Normalized amount of input data read by six different jobs during a ten-day period. The x-axis shows time; each tick is a day. The y-axis shows input size in $\log_{10}$ scale; each tick is a 10x increase. . . . 22
2.8 Completion time for jobs with varying network bandwidth. Error bars represent Min–Max values. 23
2.9 Performance of datacenter applications varies with both N and B. . . 25

3.1 Timeline of a MapReduce job and description of resources involved at each stage. 31
3.2 Predicted completion time for `Sort` (an I/O intensive job) and `WordCount` (a CPU intensive job) matches the observed time. 38
3.3 Per-stage breakdown of the observed (*Obs*) and predicted (*Pred*) completion time for `Sort` with bandwidth = 300 Mbps. *Hom* represents the predicted time assuming homogeneous disk performance. . 39
3.4 Impact of skew. 40

4.1 Selecting amongst two feasible resource tuples. Each physical machine has 2 VM slots and an outbound link of capacity 1000 Mbps. Each link is annotated with its residual bandwidth. 52
4.2 Prioritization phase. 56
4.3 Running time of the offline planner heuristic in Corral for a 4000 machine cluster with varying number of jobs. 58
4.4 Percentage of rejected requests, varying mean bandwidth and target occupancy. 63
4.5 Datacenter goodput with varying mean bandwidth and varying target occupancy. 64
4.6 Varying network oversubscription (occupancy is 75%). 65
4.7 Comparing against today's setup (Mean BW is 500 Mbps). 66
4.8 On testbed, Corral-I can accept more Hadoop jobs and increase goodput relative to Baseline. 68
4.9 Datacenter goodput when exploiting time malleability. 69
4.10 Reduction in makespan for different workloads, compared to Yarn-CS in the batch scenario. 73
4.11 Comparing (a) reduction in cross-rack data transferred and (b) compute hours relative to Yarn-CS, and (c) cumulative fraction of average reduce time, for workload W1 in the batch scenario. 74
4.12 Cumulative fraction of job completion times for different workloads, when jobs arrive online. 75
4.13 Reduction in average job completion time relative to Yarn-CS (binned by job size), for workload W1 in the online scenario. 76
4.14 Benefits of running TPC-H queries with Corral. 76
4.15 Using Corral with a mix of jobs. 77
4.16 Improvements in makespan of ad hoc jobs with Corral (compared to Yarn-CS), as their fraction in the workload is varied. 78
4.17 Benefits of using Corral relative to Yarn-CS at different loads. 80
4.18 Variation in benefits of Corral (relative to Yarn-CS) with error in job characteristics for workload W1. 80
4.19 Simulation results: Cumulative fraction of job completion times with different flow-level and job schedulers. 81

5.1 Stages that receive blame for the slowest 5% queries in the web search DAG. 88
5.2 Heatmap showing how the latency varies across machines and time (for queries to the web service in Figure 2.1). 88
5.3 This plot shows the results of a random search in the reissue budget space (circles) and using SumVar on the same DAG (line with crosses). It illustrates that as sum of variances (x-axis, normalized) decreases, so does the 99th percentile of latency (y-axis, normalized). 93
5.4 Trading off incompleteness for latency. 95
5.5 Reduction in latency using per-stage reissues in Kwiken. 102

5.6 Latency reductions achieved by Kwiken and variants when trading off completeness for latency. 104
5.7 Latency improvements from using different catch-up techniques. . . . 105
5.8 A detailed example to illustrate how Kwiken works. 107
5.9 For 45 DAGs, this figure compares the 99th percentile latency improvement and budget values of the training and test datasets. . . . 108

Chapter 1

Introduction

With the exponential growth of tablet PCs and mobile devices over the past decade [1, 2], our lives have increasingly moved online. This has resulted in unprecedented amount of user data and traffic. Internet companies like Google and Facebook process millions of queries every minute and generate several petabytes of data every day [3, 4]. The advent of cheaper and novel technologies has also significantly increased the amount of data being generated and processed in various fields of science. The Large Hadron Collider produces nearly 41 TB of data daily [5]. The Large Synoptic Survey Telescope is expected to generate over 55 PB of data every year [5]. With the increasing popularity of genomic medicine, similar demands have developed in the field of biology [6]. To handle such scale, these organizations have moved to using clusters of thousands of computers or *datacenters* [7].

Typical datacenter applications include (a) *web services* such as web search, social networks and e-commerce, and (b) *data analytics* such as *jobs* run on frameworks like Hadoop [8], Hive [9], Spark [10] and Cosmos [11]. These applications consist of several dependent logical components and can be represented as directed acyclic graphs (DAGs). Nodes in the DAG, referred to as *stages*, correspond to specific functionalities in the application. The edges in the DAG represent the input-output dependencies between these stages. Each stage can use multiple *sub-systems* in a datacenter; e.g., memory, network, storage and compute. Examples of stages include components such as spell checker or ad generator in a web search engine [12], and *map* or *reduce* stages in a Hadoop job. While some application DAGs (e.g., those in Hadoop) have fixed input dataset and run only once, others (e.g., web services) can operate on continuous input[1] (e.g., user queries) and run indefinitely.

The performance of these applications is of paramount importance to the organizations running them. Studies from companies such as Google, Amazon and Bing [13–15] report that a 100-500 ms increase in server-side latency can result in

[1]Input arrives over time.

up to 1–2% loss in revenue, which translates to several tens of millions of dollars. Thus, such companies have strong requirements on the *end-to-end* latency of these datacenter applications, i.e., latency from when the application starts running on a particular input (or request) to when it generates the corresponding output (or response). Several business-critical applications such as generation of the web-index in Bing or Google, also require strict service level objectives (SLOs) on the end-to-end latency. Any missed deadline can have significant effects (e.g., delays in updating the web index information or web page content), resulting in financial penalties [16, 17]. Further, these services represent sizable investments in terms of cluster hardware and software. Thus, any improvements in performance would be a competitive advantage to the organizations running them.

Existing systems fall short of achieving these goals as they optimize for the latency within individual application stages, and lack coordination across different application stages and sub-systems. For example, several flow-level techniques have been proposed recently for datacenter applications (e.g., [18–21]). They aim to meet flow deadlines or minimize the completion time of groups of flows (or *coflows*). Such techniques can improve the network latency within or between individual stages in an application (e.g., *shuffle* in a MapReduce job [22]) but they do not provide any benefits for the rest of the application. Further, their benefits are limited as they assume fixed flow end-points, and lack coordination between the flow and task schedulers. Systems such as Jockey [16] which aim to meet application deadlines provision only the compute resources to meet application SLOs and do not account for delays caused by the network or storage sub-system.

Thus, our *thesis* is that *coordination and joint scheduling across different stages and sub-systems in datacenter applications is required to optimize for their goals on end-to-end latency.*

To this end, we develop *Concorde* (COordinator of applicatioN COmponents in Datacenters), a scheduling and resource allocation framework which takes an end-to-end view of application latency and performs *joint* optimization across the different components of datacenter applications to meet their goals. Concorde consists of two components: (a) an offline component, *Corral*, which determines *how many* and *where* (in the cluster) should resources be allocated to an application, and (b) an online component, *Kwiken*, which uses various latency reduction techniques to overcome runtime issues (e.g., slow machines) and speed up applications at runtime.

Corral coordinates the storage and compute sub-systems in a datacenter to achieve

joint data and compute placement for applications. Such coordination allows us to achieve better data locality and isolation between applications, both spatially (by scheduling them in different parts of the cluster) and temporally (by scheduling them during different periods of time), improving their performance. Given a set of applications and their end-to-end latency goals (e.g., latency deadlines, minimizing average latency), Corral determines *allocation rules* which are used by the cluster scheduler to allocate resources to these applications and meet their goals.

Kwiken, the online component of Concorde, uses various latency reduction techniques to deal with runtime issues such as slow machines, slow disks etc., which result in *laggards* in different stages in an application. We explore three different latency reduction techniques as part of Kwiken — (a) reissuing laggards elsewhere in the cluster, (b) returning incomplete results, and (c) providing extra resources to laggards to speed them up. Each of these techniques has an associated *cost*, e.g., extra resources used, loss in completeness of the response. Kwiken provides an end-to-end view of the costs and corresponding latency improvements of these techniques in different stages in the application. It determines which technique best works for each stage, and how much of the global cost budget must be allocated to that stage to minimize the end-to-end latency of the application DAG (especially, the higher percentiles).

Each of the above components of Concorde relies on application *latency-response* functions which characterize application latency based on the resources allocated. We determine these functions using application characteristics, which are derived from application history or profiling runs. While the techniques in Concorde apply to a variety of DAG-structured datacenter applications, we focus on data analytics applications and web services in this dissertation. These applications account for a large fraction of datacenter applications, and are widely used by many organizations and companies (e.g., [22–26]).

In the rest of this chapter, we first outline the problem this dissertation aims to solve (Section 1.1) and its scope (Section 1.2). We then illustrate the challenges in achieving our goals (Section 1.3) and finally, elaborate on our approach and contributions (Section 1.4).

1.1 Problem statement

Many datacenter applications can be modeled as directed acyclic graphs (DAG) with their logical components, called stages, represented as nodes in the graph. The edges in the graph represent the dependencies between these stages. The *source* stage has no input dependencies and processes the input data from external sources[2] (e.g., a search query from a user, input dataset of a Hadoop job stored in HDFS [27]). The *sink* stage has no output dependencies and produces the final output (e.g., response to the search query, output of a Hadoop job). For example, in a MapReduce application [22], the map stage is the source and the reduce stage is the sink.

Each stage in the DAG can consist of several *tasks*, which process the input data in parallel. The time from when the source starts processing the input data to when the sink produces the final output is defined as the *end-to-end latency* of the application DAG. As mentioned before, application users and datacenter operators are mainly concerned with this end-to-end latency. Note that as cluster resources are shared across multiple applications, a particular application might have to wait from its *submission time* to when the source stage (i.e., a task in the source stage) starts execution. This queuing delay is also considered part of the application's end-to-end latency. In this dissertation, whenever we refer to application latency, we mean this end-to-end latency.

Given a set of such application DAGs, our primary goal is to schedule them on a cluster to meet *one* of the following application requirements: (a) minimize the average application latency or makespan of a batch of applications, (b) meet latency deadline or (c) minimize a particular latency percentile. While the first two goals are relevant for applications such as data analytics, the last one is relevant to web service applications which process several requests over time[3]. A secondary goal is to maximize cluster throughput.

1.2 Dissertation scope

In this dissertation, we focus on addressing the above problem for two major datacenter applications: (i) data analytics applications which can be decomposed into a DAG

[2]The input data of any non-source stage in an application DAG is generated by other stage(s) in the DAG itself.

[3]The latency percentile is calculated over requests received during a particular period of time; e.g., the previous hour or day.

of MapReduce jobs (e.g., those generated by frameworks such as Hadoop [8], Hive [9], Pig [28] and Spark [10]), and (ii) interactive web services such as web search, social networks and e-commerce applications. Our system can be extended to other data analytics applications such as those run on frameworks like Scope [11] – this requires developing techniques (or extending previous systems like Jockey [16]) to predict the latency of such applications based on the compute and network resources allocated to them.

Further, we limit ourselves to exploring two types of cross-layer coordination techniques to address the above problem:

(a) **Coordination between the storage and compute sub-systems in datacenter.** This allows us to reduce the dependence of applications on the core network and thus, provides benefits to applications that can run into network bottlenecks (e.g., shuffle in a MapReduce job). Here, we assume that the network is a constrained resource. In particular, we focus on using this technique for data analytics applications. To achieve such coordination, we plan ahead of application execution and hence, we need to be able to predict how the latency of an application varies depending on the resources allocated to it. We develop the relevant prediction techniques for DAGs of MapReduce jobs. These techniques rely on either (a) previous runs of the application or (b) the ability to profile the application on a sample dataset. They assume that various job characteristics such as average task processing rate, amount of data processed by different stages in the application, task input to output ratio etc. can be estimated from these previous or profiling runs. We show that these assumptions hold for a variety of data analytics applications used in practice (Sections 2.2 and 3.3). This coordination is exploited in the offline component of Concorde, Corral, to determine how should resources be allocated to applications.

(b) **Coordination across multiple application stages.** This technique, used in Kwiken, allows us to capture the end-to-end view of an application and decide how various latency reduction techniques affect the end-to-end application latency when used *within* different application stages. In particular, we explore such coordination for web services when using runtime latency reduction techniques. Here, we rely on the assumption that the reduction in latency variance of application stages, when using different latency reduction techniques (e.g., reissues), can be predicted from application execution history (on queries in the past), and develop

techniques to do so (Section 3.2).

We note that for applications such as web services where a user's query is sent over the wide-area network (WAN) or Internet, we do not include the time spent over the WAN i.e., the *WAN latency*, in its end-to-end latency. While the WAN latency can affect user experience [29, 30], in this dissertation, we deal with only the datacenter-side latency of applications which accounts for a large fraction of the user-experienced latency [31]. Optimizing WAN latency is orthogonal to our research goals, and we refer the reader to existing work for the relevant techniques [29, 30].

1.3 Challenges

While several scheduling techniques have been proposed to speed up individual stages or sub-systems in various datacenter applications (e.g., [16, 18–21, 32–35]), there are several challenges in directly optimizing for the end-to-end latency.

Complex structures of application DAGs. Datacenter applications typically have complex input-output dependencies across the different stages. While we will discuss the structure of these applications in more detail later (Chapter 2), our measurements show that the median application in production at Microsoft Bing has 15 stages and 10% of the stages process the query in parallel on 1000s of servers. Further, these applications can have stages with in or out degree as high as 40. Due to this complexity, random delays at any of these servers or stages can manifest as significant end-to-end delays. To see why, we note that the 99th percentile of an n-way parallel stage depends on the 99.999^{th} percentile of the individual server latencies for $n = 1000$. Reducing the 99th percentile of such a stage could require us to reduce the 99.999^{th} percentile of each individual server, which is hard as it is influenced by a large number of runtime factors (e.g., delays in process or thread scheduling in the operating system).

Further, different stages within an application can have widely different latency characteristics (Section 2.1.2). Thus, a particular technique to reduce latency (e.g., launching a duplicate request after a fixed timeout) may be effective on one particular stage but not work for another.

Modeling end-to-end latency. The end-to-end latency of an application DAG is determined by its critical path. Due to runtime issues such as laggards, failures etc. the critical path of an application can change dynamically. Thus, determining

the effect of any particular per-stage latency reduction technique on the end-to-end latency of the DAG is hard. Further, the latency of each stage is affected by its dependence of the different sub-systems (e.g., network, compute etc.) — for example, while a Hadoop Sort job is I/O-bound[4], a WordCount job is typically compute-bound [36]. Any scheduling technique which optimizes for application latency must consider these properties to meet its goals while efficiently using cluster resources.

Fast estimation of application latency. To meet latency goals of applications (e.g., completion time deadline), we need to determine the amount of resources to be allocated to them and predict their latency as a function of the resources allocated. Application latency is not only dependent on the number of machines allocated to it but also on the network bandwidth available, the location of the data, the rate at which the data can be read, the amount of extra resources any stage in the application can use etc. To explore a variety of resource allocation options, we need to be able to quickly predict this latency. Even though extensive modeling approaches have been developed to accurately predict the latency of certain datacenter applications [37–39], they are prohibitively expensive to use for our purpose.

1.4 Dissertation outline and contributions

Figure 1.1 shows the architecture of our scheduling framework, Concorde, which consists of three parts: (a) the latency estimator, which determines latency-response functions of applications using their characteristics from previous executions or profiling runs, (b) the offline component, *Corral*, which uses these latency-response functions, application goals, and datacenter state to determine how should resources be allocated to each application, and (c) the online component, *Kwiken*, which uses runtime techniques such as request reissues to speed up execution of applications. In this section, we briefly describe each of these three components (Section 1.4.1–Section 1.4.3) and then, summarize our main contributions (Section 1.4.4).

1.4.1 Using application characteristics to estimate end-to-end latency

Estimating the latency of an application allows us to plan ahead on the resources that need to be allocated to it while meeting its individual goals and efficiently utilizing

[4]Latency is predominantly determined by the available disk and network I/O bandwidth.

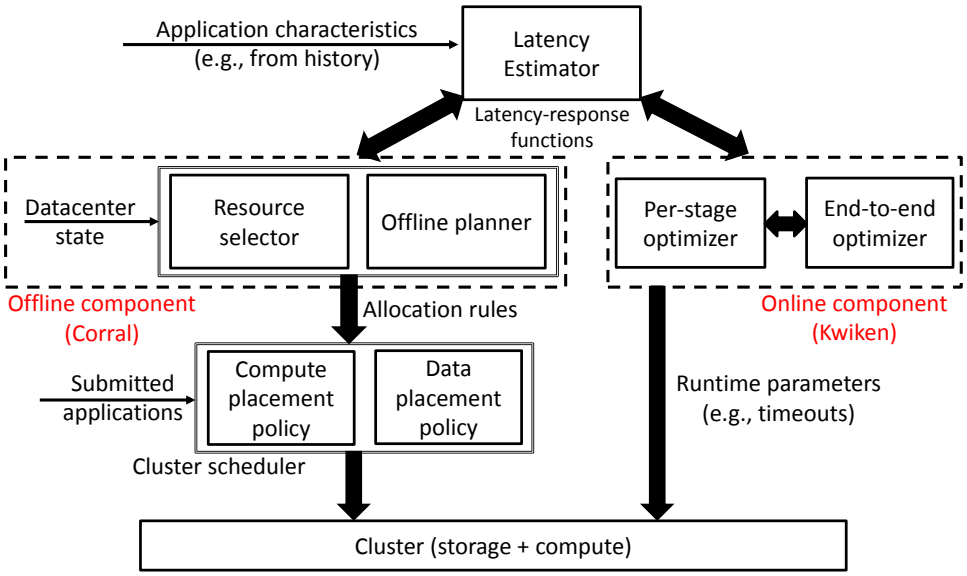


Figure 1.1: Architecture of Concorde.

cluster resources. As mentioned above, in this dissertation, we focus on scheduling two major types of applications which make up a large fraction of workloads in a typical datacenter: (a) data analytics jobs which can be decomposed into a DAG of MapReduce jobs (e.g., those generated by frameworks such as Hadoop [8], Hive [9], Spark [10], Pig [28]), and (b) web services such as web search which can be arbitrary DAGs. In this section, we summarize how we use characteristics of these applications to estimate their latency. The output of the latency estimator is a *latency-response function*, $L(\vec{r})$, which gives us the latency of the application as a function of the abstract resource vector $\vec{r}$. These techniques have appeared in [12, 36, 40] and are described in Chapter 3 in detail.

Data analytics applications. We estimate the latency of MapReduce jobs using a simple *bandwidth-model*, i.e., the latency of any stage in the job is modeled as the ratio of the amount of data processed by it to the average rate at which the data is processed. This rate depends on the bottleneck resource for the stage (e.g., disk, network or compute), and we use previous executions of the job to determine it. Such estimation is feasible for a large fraction of production workloads (up to 40%) which are recurring, and are run periodically as new data becomes available [16,41]. For jobs which are ad-hoc or are running for the first time, we obtain these rates by profiling them on a smaller subset of the input data.

Frameworks such as Hive and Pig typically generate a DAG of MapReduce jobs. In this case, we estimate the latency of each job separately and model the end-to-end

latency as the latency of the critical path of the DAG. In this model, we trade-off accuracy for speed of estimation, and account for the compute, storage and network bandwidth available for jobs. While we do not model runtime issues such as failures and stragglers, we found that errors in our estimates are tolerable in practice. We also add "slack" to the predictions to deal with any inaccuracies in our models.

Web services. Unlike data analytics applications, web services are continuously running applications, processing user queries over time. The resource requirements for these applications (e.g., number of machines to host different stages) are typically determined by the application provider (e.g., the company which runs the web service), offline. Thus, determining the number of resources to allocate to these applications is straightforward. However, the execution time of a query in these applications is not only determined by the number of resources allocated to the application but also runtime issues such as slow machines, network loses etc.

To overcome delays caused by such issues, we develop several latency reduction techniques (Chapter 5) which use extra resources, e.g., reissues increase machine load, partial answers result in loss in relevance of the answer. Our goal in modeling the latency of these applications is to understand how it varies as a function of these extra resources. We first determine stage-level latency-response functions using the characteristics of each stage (from previous executions of the stage, and application logs) and the latency reduction technique used. Next, we use the DAG structure to compose the per-stage latency-response functions to determine the end-to-end application-level latency-response function.

1.4.2 Resource planning

Corral, the offline component of Concorde aims to determine *how many* and *which* resources (in the cluster) should be allocated to application DAGs, given their goals on end-to-end latency. Corral uses coordination between the storage, and compute subsystems in a cluster while scheduling applications. Corral was published in [36, 40] and is described in detail in Chapter 4. We briefly summarize its techniques and benefits here.

As input to Corral, each application specifies its *goals* — completion time deadlines or *none*, in the absence of any — along with *constraints* on its resource requirements — number of machines to be allocated to an application stage, amount of network bandwidth required across machines running a stage, *none*, etc. Depending on these

goals and constraints, Corral uses two components to schedule applications. For applications with latency goals and resource constraints, the *resource selector* (Figure 1.1) determines the amount of resources required to satisfy them. Due to the *resource malleability* property of datacenter applications (Section 2.3), different resource combinations can achieve the same application goal. The resource selector selects the resource combination with the least cost to the cluster to improve cluster efficiency, and specifies it using allocation rules to the cluster scheduler. In this dissertation, we consider storage, compute and network as the resources which are explicitly reserved for an application. We use *virtual clusters* [42] to provide guaranteed network bandwidth.

Next, to schedule applications without any specific goals or constraints, Corral uses an *offline planner* (Figure 1.1) to plan ahead and reduce aggregate metrics such as makespan or average application completion time. To this end, it formulates a *planning problem* to determine which resources and where in the cluster these resources must be allocated to applications. We model this problem as a variant of the malleable job scheduling problem [43–45]. While this problem is NP-hard in general, we develop efficient heuristics to solve it in practice. These heuristics achieve performance appealingly close (within 3-15%) to the solution of a Linear Program (LP) relaxation, which serves as a lower bound to any solution of the planning problem which allocates resources at the granularity of racks.

In developing the heuristics to the planning problem, we focus on data analytics applications without deadlines. We deal with both simple MapReduce jobs as well as complex DAG-structured workloads such as Hive [9] or Scope [11] queries. The resource selector focuses on data analytics applications with deadlines, and web services with resource requirements (specified by the application provider).

We have implemented Corral as an extension of Yarn [46] and deployed it on a 210 machine cluster. Using four different workloads, including traces from Yahoo [47] and large-scale production clusters at Microsoft, and Hive queries derived from TPC-H benchmarks [48], we show that Corral reduces the makespan of a batch of jobs by 10-33% and the median job completion time by 30-56% compared to the Yarn capacity scheduler [49]. When using virtual clusters to provide guaranteed resources to applications and meet their completion time goals, Corral's ability to exploit application resource malleability yields significant gains to the provider as well — the provider can accept 3-14% more requests, with 7-87% improvements in cluster goodput. Further, using large-scale simulations over a 2000 machine topology, we show that Corral

achieves better performance than flow-level techniques like Varys [18] and that the gains from such techniques are orthogonal to those of Corral.

1.4.3 Speeding up application stages at runtime

Chapter 5 describes how Concorde deals with runtime issues such as slow machines or disks, network loses, etc., which can lead to a significant increase in end-to-end application latencies. To overcome such issues, we use three techniques: (a) *reissues* i.e., issue a duplicate of the original query to a replica of the stage processing the query, (b) *incomplete responses*, where we respond to the original application request before all the machines processing the request finish (while ensuring that the loss in answer quality is within an acceptable bound), and (c) *catch-up*, where laggard queries are given higher priority or extra resources (e.g., threads) at later stages in the application. In this dissertation, we focus on using these techniques for web services. Exploration of similar techniques for data analytics applications is part of our future work.

Each of the above techniques have costs associated with them; for example, reissues increase system load, incomplete responses can reduce answer quality etc. Further, as different stages have different characteristics, the benefits from using these techniques can vary significantly. To address these challenges and limit the cost incurred from the above techniques within a given budget, we develop Kwiken (published in [12]), a *holistic framework* that considers (i) the latency characteristics of each stage, (ii) the cost of applying individual techniques and (iii) the structure of the application DAG, to determine how to use different techniques in various application stages to minimize the end-to-end application latency.

Kwiken formulates the overall latency reduction problem as a layered optimization problem, relying on the fact that query latencies across stages are only minimally correlated. The first layer determines latency-response functions which estimate how the latency in individual stages changes as a function of budget allocated to that stage (e.g., fraction of extra resources which can be used for reissues). These functions abstract the actual intra-stage optimizations performed and provide a simple model of how various latency reduction techniques effect the latency of the stage. The second layer integrates these per-stage functions into a single global objective function designed such that its minimization is well-correlated to minimizing higher percentiles of the end-to-end latency. The objective function also has a simple separable structure

that allows us to develop efficient gradient-like methods for its minimization.

We evaluated Kwiken with 45 production web service DAGs at Bing. By appropriately apportioning reissue budget, Kwiken improves the 99th percentile of latency by an average of 29% with just 5% extra resources. This is over half the gains possible from reissuing every request (budget=100%). At stages that are many-way parallel, we show that Kwiken can improve the 99th percentile latency by about 50% when partial answers are allowed for just 0.1% of the queries. We, further, show that reissues and partial answers provide complementary benefits; allowing partial answers for 0.1% queries lets a reissue budget of 1% provide more gains than could be achieved by increasing the reissue budget to 10%.

1.4.4 Contributions

Our key contributions are as follows.

- **Characteristics of web services.** We describe low-latency application DAGs at a large search engine, Microsoft Bing, analyze the structure of its web service DAGs in detail, and report on the causes for high variability in the latency of these DAGs (Section 2.1 and Section 5.1).

- **Predictability of data analytics applications.** We quantify the high predictability in the characteristics of data analytics by analyzing traces from Microsoft Cosmos (Section 2.2).

- **Resource malleability of datacenter applications.** We measure the malleability of several representative datacenter applications, including data analytics, web services and HPC applications. We show that different combinations of compute and network resources for these applications can achieve the same performance (Section 2.3).

- **Abstraction of latency-response functions.** We characterize the latency of datacenter applications as a function of the resources allocated to them. This provides us the strong abstraction of latency-response functions which can be used to make scheduling decisions for applications (Chapter 3).

- We design, implement and evaluate Corral, a data-driven planning framework, which leverages the predictability of future workloads and coordination between

the storage and compute sub-systems in datacenters, leading to significant application performance improvements (Chapter 4)

- **Resource imbalance metric.** We devise a metric to quantify the cost of accommodating multi-resource requests from a cluster provider's perspective. As different resource combinations can be used to meet a particular application goal (due to resource malleability), this allows us to compare one resource tuple against another (Section 4.2) and select the better one.

- **Joint data and compute placement for scheduling datacenter applications.** We formulate the complex data and compute placement problem for datacenter applications as a malleable job scheduling problem and solve it using efficient heuristics, which lead to significant benefits in practice (Section 4.3).

- **Exploiting malleability across time.** We extend Corral to exploit malleability along the time domain by finishing jobs earlier than required, using idle resources. We find this can reduce median job completion time by more than 50% (Section 4.4).

- **Benefits from joint scheduling with network flow-level techniques.** We show that the benefits of Corral are more than those from just using flow-level techniques such as Varys [18] and that Corral can work together with such techniques to provide further performance benefits (Section 4.5).

- **End-to-end runtime framework to reduce latency.** We present a holistic runtime optimization framework that treats each stage as a latency-response curve to apportion the overall cost budget across stages in an application DAG. We evaluate the framework on real-world web services and demonstrate significant reductions in their end-to-end latencies, especially in the higher percentiles i.e., tail latencies (Chapter 5).

- **Novel per-stage latency reduction techniques.** We provide novel policies for bounding quality loss incurred due to partial answers and for catching-up on laggards for web services such as web search (Section 5.3).

1.5 Roadmap

This dissertation is organized as follows: Chapter 2 presents the background and characteristics of two major datacenter applications — data analytics and web services. Chapter 3 discusses how we utilize the characteristics of applications to predict their latency. Chapter 4 presents Corral, the offline component of Concorde, which uses application requirements and goals to determine how should resources be allocated to applications in the datacenter. Chapter 5 presents Kwiken, the online component of Concorde, which uses various runtime techniques to speed up datacenter applications ensuring that the cost of using these techniques is within a given cost budget. We finally present our conclusions and directions for future work in Chapter 6.

Chapter 2

Characterizing datacenter applications

Datacenter applications are often complex, running over 1000s of machines and consist of tens to hundreds of logical components or *stages*. These stages typically form a directed acyclic graph (DAG) with edges representing input-output dependencies. In this chapter, we provide the background and describe several characteristics of these applications, which motivate the design of the scheduling and resource allocation techniques in Concorde. In particular, we analyze the characteristics of two widely used datacenter applications, *web services* and *data analytics*, using traces from two large-scale production systems — Microsoft Bing [50] and Microsoft Cosmos [11].

Our main observations are as follows.

- Web service DAGs are long and many way parallel; about 20% of DAGs we examined have 10 or more stages in sequence with the 90th percentile in-degree as high as 9.
- Web service DAGs have high tail latencies with the 99th percentile latency of nearly 10% DAGs being 10 times their median latency, in the applications examined. Further, latencies across different stages for a single request and those within a stage across multiple requests are mostly uncorrelated.
- Business-critical data analytics applications are often recurring with highly predictable characteristics (e.g., input data size) and more than 90% of these applications require less than one rack worth of compute resources.
- Several datacenter applications exhibit *resource malleability* i.e., multiple resource combinations achieve the same end-to-end application latency.

In the rest of this chapter, we first elaborate on the characteristics of web services (Section 2.1), characterize the predictability of data analytics applications (Section 2.2) and finally, show the resource malleability of various datacenter applications (Section 2.3).

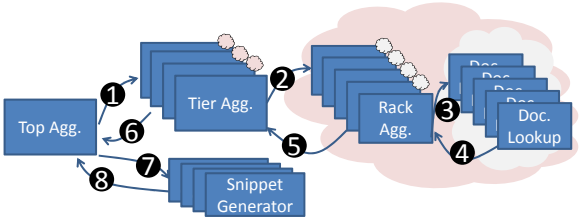


Figure 2.1: Timeline diagram of the processing involved for web search stage at Microsoft Bing.

2.1 Web services

Web services include a variety of interactive applications such as social networks, search engines, and e-commerce web sites. Each of these services are architected as a collection of stages with input-output dependencies; for example, responding to a user search query on Microsoft Bing involves accessing a spell checker stage and then in parallel, a web search stage that looks up documents in an index, and similar video- and image-search stages. We use the term *workflow* to refer to services designed in this manner. Architecting datacenter services in this way allows easy reuse of common functionality encapsulated in stages, similar to the layering argument in the network stack.

Web services can be hierarchical; i.e., complex stages may internally be architected as DAGs themselves. For example, the web search stage at Microsoft Bing consists of multiple tiers which correspond to indexes of different sizes and freshness. Each tier has a document-lookup stage consisting of thousands of servers that each return the best document for the phrase among their sliver of the index. These documents are aggregated at rack and at tier level, and the most relevant results are passed along. This stage is followed by a snippet generation stage that extracts a two sentence snippet for each of the documents that make it to the final answer. Figure 2.1 shows a timelapse of the processing involved in this web service DAG; every search at Microsoft Bing passes through this DAG. While this is one of the most complex DAGs at Microsoft Bing, it is still represented as a single stage at the highest level DAG.

The sheer number of components involved in these applications ensures that each request has a non-trivial likelihood of encountering a runtime anomaly, leading to higher latencies. The causes we observed for high and variable latency include slow servers, network anomalies, complex queries, congestion due to improper load balance or unpredictable events, and software artifacts such as buffering (discussed further in Section 5.1).

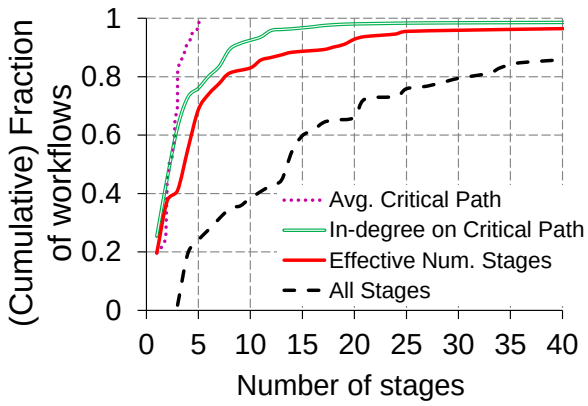
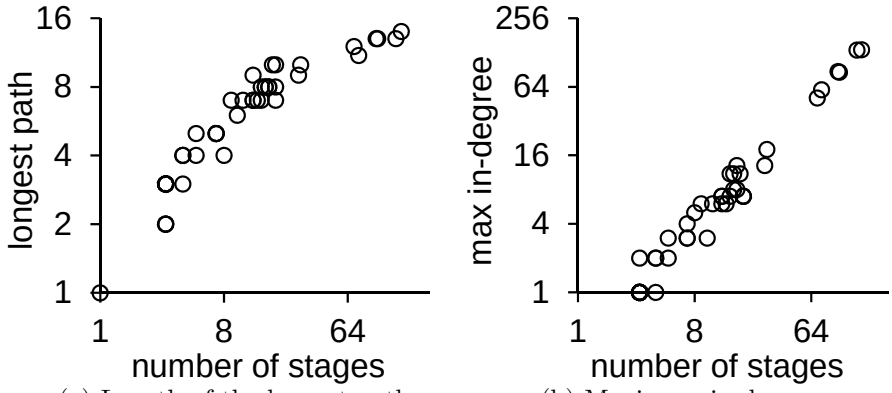


Figure 2.2: A few characteristics of the analyzed web service DAGs.

To characterize these web service DAGs, we use traces from Microsoft Bing which consist of descriptions of the various DAGs used to answer user requests along with the latency associated with the query across the different stages in the DAG. In this section, we report on the topology (Section 2.1.1) and latency properties (Section 2.1.2) of these DAGs. We use request latencies from 64 distinct DAGs over a period of 30 days during Dec 2012. We only consider DAGs and stages that were accessed at least 100 times each day — the 25th and 75th percentile number of requests per stage per day are 635 and 71428 respectively. In all, we report results from thousands of stages and tens of thousands of servers.

2.1.1 Topology characteristics

Most web service DAGs have several tens of stages, with a median value of 14 and 90th percentile of 81 as shown by the "all stages" line in Figure 2.2. To understand how many of these stages determine the latency of the query, we look at the *critical path* of the DAG, which is the sequence of dependent stages in the DAG that finished last for that query. As queries can have different critical paths, we consider the most frequently occurring critical paths that account for 90% of the queries. Along each critical path, we consider the smallest subset of stages that together account for over 90% of the query's latency and call these the *effective* stages. Stages that have a very small latency and rarely occur on the critical path of a request have negligible impact on the request latency. Ignoring these stages, Figure 2.2 plots the number of effective stages across the web service DAGs we considered. We see that the number of effective stages in a DAG is sizable but much smaller than the total number of stages in it;



(a) Length of the longest path (b) Maximum in-degree

Figure 2.3: Scatter plots of different topology metrics with the number of stages in each web service DAG at Microsoft Bing. Each circle corresponds to one DAG in the dataset.

median is 4 and 90th percentile is 18. The figure also plots the average number of effective stages on these critical paths (labelled "Avg. Critical Path"); median is 2.2. Finally, we plot a distribution of the in-degree of these effective stages on the critical paths; median is 2 and 90th percentile is 9. Hence, we see that production DAGs even when counting only the effective stages are long and many way parallel.

Figure 2.3a shows a scatter plot of the length of the longest path in the web service DAG and the number of stages in the DAG. We see that about 20% of the DAGs have stage sequences of length 10 or more. Further, the max in-degree across stages is proportional to the number of stages in the application DAG (Figure 2.3b). That is, most DAGs are parallel.

We point out that stages with high in-degree, that aggregate responses from many other stages, are a significant source of variability. Whenever one of the input stages is slow, the output would be slow. We see two types of such fan-in in our data: (a) at the stage-level, which is accounted for in the above results, and (b) internal to a stage, where an *aggregator* collects responses from many servers. For example, the web search stage above aggregates responses from 100s-1000s of servers, where each server retrieves the documents matching the user request from their shard of the index.

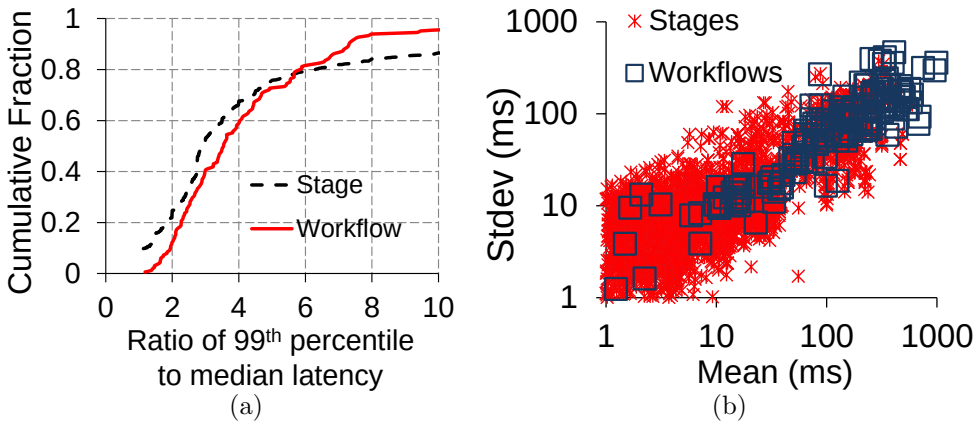


Figure 2.4: Estimating the variability in latency: (a) CDF of 99th percentile to median latency and (b) mean vs. standard deviation of latency, for stages and web service DAGs.

2.1.2 Latency characteristics

Web service DAGs have high tail latencies. To understand variation of latency in different web service DAGs and in individual stages, Figure 2.4a plots a CDF of the ratio of the latency of the 99th percentile request to that of the median request across the stages and DAGs in our dataset. We see that stages have high latency variability; roughly 10% have 99th percentile 10X larger than their median. When the stages are composed into DAGs, the variability increases at the lower percentiles because more DAGs have high variability stages. However, the variability at the higher percentiles decreases.

Figure 2.4b compares, on a log scale, the mean latency (x-axis) and standard deviation (y-axis) of each stage and application from 64 different application DAGs at Microsoft Bing. We see that the larger the mean latency in a stage, the larger is the variability (standard deviation). Further, stages with similar mean latency still have substantial differences in variability. We also find a lot of variability in the per-stage latencies, with means varying from 1 millisecond to 100 milliseconds.

Latencies of individual stages are uncorrelated. We ran a benchmark against the most frequent web service at Microsoft Bing, where we executed two concurrent requests with same parameters and specified they should not use any cached results. These requests execute the same set of stages with identical input parameters and thus, allow us to study correlation of latencies in individual stages. We used 100 different input parameters and executed a total of 10,000 request pairs. For each of

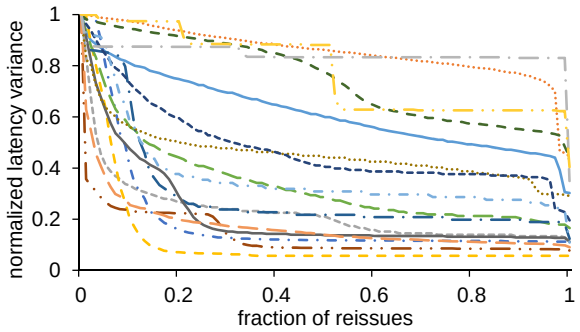


Figure 2.5: For a subset of stages in production, this plot shows normalized latency variance of the stage as a function of fraction of requests that are reissued.

the 380 stages in this web service DAG, we compute the Pearson correlation coefficient (PCC). About 90% of the stages have PCC below 0.1 and only 1% of stages have PCC above 0.5. Hence, we can treat the latency of two copies of the same request as independent random variables.

Latencies across stages are mostly uncorrelated. To understand correlation of latencies across stages, we compute the PCC of latencies of all stage pairs in one of the major DAGs with tens of thousands of stage pairs. We find that about 90% of stage pairs have PCC below 0.1. However 9% of the stage pairs have PCC above 0.5. We hypothesize that this is because some of the stages run back-to-back on the same server when processing a request; if the server is slow for some reason, all the stages will be slow. In spite of this mild correlation, we can treat the inherent processing latency across stages to be independent.

Stages benefit differently from reissues. Figure 2.5 illustrates how reissuing requests impacts the latency for a subset of the stages from production. It shows the normalized variance in latency (y-axis) for these stages when a particular fraction of the slowest queries are reissued (x-axis). Clearly, more reissues lead to lower variance. However, notice that stages respond to reissues differently. In some stages, 10% reissues significantly reduce variance, whereas in other stages even 50% reissues do not achieve similar gains. This is because the reduction in variance at a stage depends on its latency distribution: stages with low mean and high variance benefit a lot from reissues but the benefits decrease as the mean increases or the variance decreases.

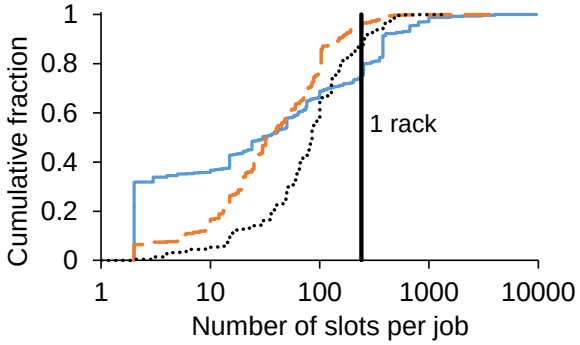


Figure 2.6: CDF of the number of compute slots requested by jobs across three production clusters. The vertical line represents 240 slots, the size of one rack.

2.2 Data analytics applications

Data analytics applications use frameworks like Hadoop [8], Spark [10], Hive [9], or Scope [11] to run MapReduce [22] *jobs*, or more complex, DAG-structured jobs. Each job consists of stages, linked by data dependencies. Each stage consists of tasks that process the input data of the stage in parallel. MapReduce jobs are the simplest of such jobs with a single *map* stage and a single *reduce* stage, with an all-to-all data *shuffle* between their tasks. When a task has all its inputs ready, it can start running and is allocated a free *slot* on a machine in the cluster by the cluster task scheduler, whenever possible. Each slot is assigned a pre-defined amount of memory and CPU for the task to execute. The input data for these jobs is stored in a distributed file system like HDFS [27]. The job input dataset consists of multiple *chunks*, each of which is typically replicated three times across different nodes in the cluster.

In this section, using traces from Microsoft Cosmos, we first present the resource requirements of data analytics applications, and then quantify their predictability.

A bulk of jobs can run within a few racks without losing parallelism. To understand the resource requirements of production jobs, we analyzed jobs from three large production clusters used in Microsoft, each of them with more than 10,000 machines. The users communicate their job requirements by specifying number of *slots* required, where one slot is typically equivalent to two CPU cores. We plot the CDF of the requested number of slots in Figure 2.6. While some jobs require up to 10,000 slots, we find that across these three clusters, 75%, 87%, and 95% of jobs require less than one rack worth of compute resources. Similar observations have been reported before [51, 52].

Job characteristics can often be accurately predicted. Cluster workloads are

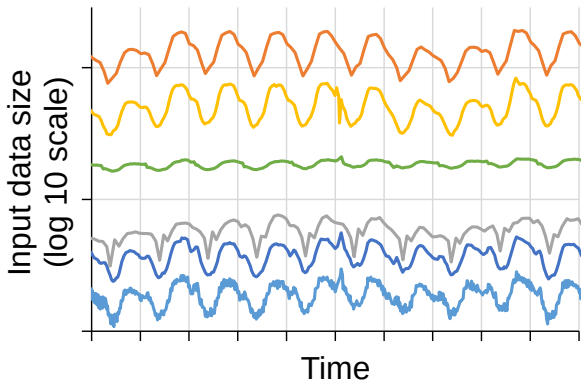


Figure 2.7: Normalized amount of input data read by six different jobs during a ten-day period. The x-axis shows time; each tick is a day. The y-axis shows input size in $\log_{10}$ scale; each tick is a 10x increase.

known to contain a significant number (up to 40%) of *recurring* jobs [16,41]. A recurring job is one in which the same script runs whenever new data becomes available. Consequently, for every instance of that job, it has a fixed structure and similar characteristics (e.g., amount of data moved in a shuffle, or CPU and memory demands).

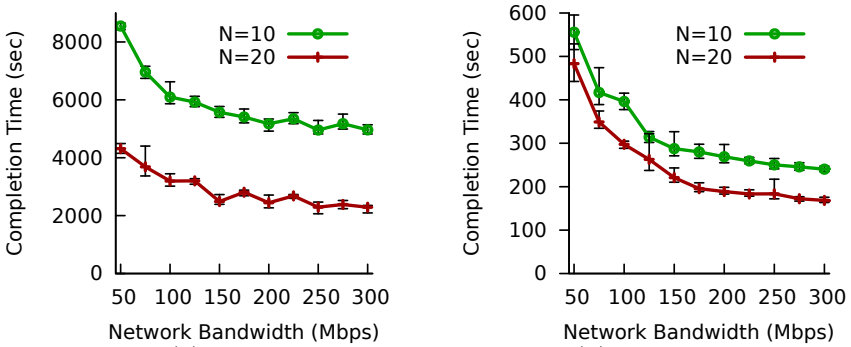
We confirm and quantify this intuition by examining twenty business-critical jobs from Microsoft Cosmos. For each job, we compute the input data size of the recurring instances of these jobs, during the month of December 2013. Figure 2.7 shows the normalized job sizes as a time series for six of those jobs over a 10-day period. Overall, these jobs have input sizes ranging from several gigabytes to hundreds of terabytes. To predict the input size of a job which is submitted at a particular time (e.g., 2PM), we average the input size of the same job type at the same time during several previous days. In particular, if the current day of the week is a weekday (weekend), we average only over weekday (weekend) instances. Using this, we can estimate the job input data size with a small error of 6.5% on average. We observe similar predictability of the intermediate (or shuffle) data and output data. In turn, this lets us predict the amount of data transferred during job execution and utilization of network links.

2.3 Resource malleability

In this section, we show that several representative datacenter applications exhibit the property of resource malleability, and that several resource combinations can achieve the same end-to-end latency for these applications.

Hadoop Job	Input Data Set
`Sort`	200GB using Hadoop's RandomWriter
`WordCount`	68GB of Wikipedia articles
`Gridmix`	200GB using Hadoop's RandomTextWriter
`TF-IDF`	68GB of Wikipedia articles
`LinkGraph`	10GB of Wikipedia articles

Table 2.1: MapReduce jobs and the size of their input data.



(a) `Sort` (b) `LinkGraph`

Figure 2.8: Completion time for jobs with varying network bandwidth. Error bars represent Min–Max values.

Malleability of MapReduce jobs. We experimented with a small yet representative set of MapReduce jobs listed in Table 2.1. These jobs capture the use of data analytics in different domains and the varying complexity of such workloads (through multi-stage jobs). `Sort` and `WordCount` are popular for MapReduce performance benchmarking, not to mention their use in business data processing and text analysis respectively [53]. `Gridmix` is a synthetic benchmark modeling production workloads, Term Frequency-Inverse Document Frequency or `TF-IDF` is used in information retrieval, and `LinkGraph` is used to create large hyperlink graphs. Of these, `Gridmix`, `LinkGraph`, and `TF-IDF` are multi-stage jobs.

We used Hadoop MapReduce on Emulab to execute the jobs while varying the number of nodes (**N**) devoted to them. We also used rate-limiting on the nodes to control the network bandwidth (**B**) between them. For each $<N, B>$ tuple, we executed a job fives times to measure the completion time for the job and its individual stages. While the experiment setup is further detailed in Section 4.4 , here we just focus on the performance trends.

Figure 2.8a shows the completion time for `Sort` on a cluster with 10 and 20 nodes, and varying network bandwidth. As the bandwidth between the nodes increases, the

time to shuffle the intermediate data between map and reduce tasks shrinks, and thus, the completion time reduces. However, the total completion time stagnates beyond 250 Mbps. This is because the local disk on each node provides an aggregate bandwidth of 250 Mbps. Hence, increasing the network bandwidth beyond this value does not help as the job completion time is dictated by the disk performance. This is an artifact of the disks on the testbed nodes. If the disks were to offer higher bandwidth, increasing the network bandwidth beyond 250 Mbps would still shrink the completion time.

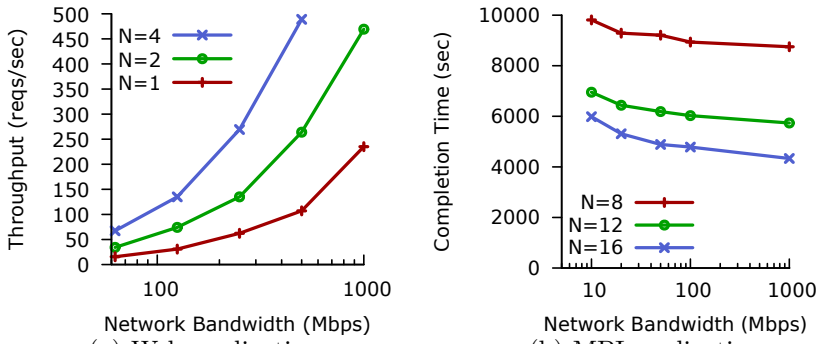
The same trend holds for the other jobs we tested. For instance, Figure 2.8b shows that the completion time for `LinkGraph` reduces as the number of nodes and the network bandwidth between the nodes is increased. However, note that the precise impact of either resource is job-specific. For instance, we found that the relative drop in completion time with increasing network bandwidth is greater for `Sort` than for `WordCount`. This is because `Sort` can be I/O intensive with a lot of data shuffled which means that its performance is heavily influenced by the network bandwidth between the nodes.

Apart from varying network bandwidth, we also executed the jobs with varying number of nodes. The results are detailed in Section 3.3 (Figures 3.2a and 3.2b) and show that the completion time for a job is inversely proportional to the number of nodes devoted to it. This is a direct consequence of the data-parallel nature of MapReduce.

Malleability of other datacenter applications. Our findings for MapReduce above also extend to other datacenter applications. We briefly discuss two examples below.

Three-tier, web application. We used a simple, unoptimized ASP.net web application with a SQL backend as a representative of web applications. We varied the number of nodes (N) running the middle application tier and the bandwidth (B) between the application tier (middle nodes) and the database-storage tier (backend nodes). We used the Apache benchmarking tool (`ab`) to generate web requests and determine the peak throughput for any given resource combination. Figure 2.9a shows that the application throughput improves as expected when either resource is increased.

MPI application. We used an MPI application generating the Crout-LU decomposition of an input matrix as an example of HPC and scientific workloads. Figure 2.9b shows the completion time for a 8000x8000 matrix with varying N and B. Given the



(a) Web application (b) MPI application

Figure 2.9: Performance of datacenter applications varies with both N and B.

Hadoop Job – Completion Time (sec)	<Nodes, Bandwidth (Mbps)> alternatives
`LinkGraph` – 300 (±5%)	<34, 75>, <20, 100> <10, 150>, <8, 250>
`LinkGraph` – 400 (± 5%)	<30, 60>, <10, 75> <8, 150>, <6, 200>
`WordCount` – 900 (± 3%)	<30, 45>, <20, 50> <10, 100>, <8, 300>
`WordCount` – 630 (± 3%)	<32, 50>, <20, 75> <14, 100>, <12, 300>

Table 2.2: Examples of `WordCount` and `LinkGraph` jobs achieving similar completion times with different resource combinations.

CPU-intensive nature of the application, increasing the number of nodes improves performance significantly. As a contrast, the impact of the network is limited. For instance, improving the bandwidth from 10 to 100 Mbps improves completion time only by 15-25%.

Overall, the experiments above lead to two key findings.

1. The performance of typical datacenter applications depends on resources beyond just the number of compute instances.

2. They confirm the *resource malleability* of such applications; an application can achieve the same performance with different resource combinations, thus allowing one resource to be traded-off for the other. For instance, the throughput for the web application above with two nodes and 250 Mbps of network bandwidth is very similar to that with four nodes and 125 Mbps of network. Table 2.2 further emphasizes this for data analytic applications by showing examples where

a number of different compute node and bandwidth combinations achieve almost the same completion time for the `LinkGraph` and `WordCount` jobs. This flexibility is important as it allows for improved cluster efficiency.

Chapter 3

Estimating end-to-end latency using application characteristics

Datacenter applications can have widely different characteristics (e.g., I/O bound, compute-bound) and goals. Many of these applications can simultaneously run on the same physical cluster. To efficiently utilize the underlying resources, we need to allocate them based on the characteristics of the individual applications. For example, an MapReduce job with large amount of shuffle data can have significantly lower latency if its reducers and maps are allocated in the same rack as opposed to across multiple racks as cross-rack links are typically oversubscribed [18,35,54]. This in turn frees up the cross-rack bandwidth which can be used by other applications. However, if the same job has (say) 1000 tasks instead of 100, running it across multiple racks might be better as the benefits of increased parallelism for the tasks can outweigh the benefits of higher network bandwidth for the shuffle stage.

To deal with such complexities and to meet application goals, this chapter develops techniques to predict the performance of datacenter applications based on the amount of resources (e.g., number of machines or compute slots) allocated to them and where in the cluster these resources are allocated. In particular, we focus on data analytics applications and web services.

Our techniques use the characteristics of applications (e.g., data processing rate of a map task, ratio of input to output data of reduce stage) determined from application *history*, i.e., previous execution(s), to predict their performance. Such characteristics are readily available for data analytics jobs as a large fraction of them are recurring (Section 2.2). For those applications without such history, we use *profiling-based* approaches to determine these characteristics. Using such characteristics, we determine *latency-response functions* $L(\vec{r})$ for datacenter applications, which quantify the latency of the application as a function of the resource vector $\vec{r}$. In the context of data analytics applications, we develop *MRCute* which models $\vec{r}$ as either (i) a scalar which specifies the number of racks allocated to the application, or (ii) a two dimensional vector which specifies the number of machines allocated to the application and

the amount of network bandwidth between each of these machines. The granularity of the model used depends on application requirements (e.g., enforcing completion time deadlines or not). For web services, $\vec{r}$ indicates the amount of resources or cost budget available per stage for runtime latency reduction techniques such as reissues or partial responses.

The above response functions provide us a strong abstraction — they hide the complex dependencies within an application, how different techniques are used within an application to utilize the resources provided, and interaction between the components within an application. They allow us to determine how the latency of the application changes based on the resources allocated, without having to deal with application-specific nuances. Given a set of datacenter applications where each application can have its own goals (e.g., end-to-end latency deadline to be met), the latency-response functions allow us to determine how should resources be allocated to each application, to meet their specified goal(s).

Performance prediction has been extensively studied in a variety of domains such as operating systems [55], user software [56], and databases [57]. In the context of MapReduce applications, efforts like Mumak [37] and MRPerf [38] have built detailed MapReduce simulators that can be used for prediction. However, these result in non-trivial prediction times. To allow for an exploration of different resource combinations, we require fast prediction. Thus, we explicitly chose to simplify our prediction model, favoring fast prediction over accuracy. This choice is also motivated by the observation that even the most accurate model will not be able to account for runtime issues such as outliers etc. Instead, to compensate for any model inaccuracies, we introduce "slack" into our prediction. This helps us deal with common sources of prediction errors such as hardware heterogeneity and workload imbalance.

The rest of the chapter is organized as follows. First, we present our prediction techniques for data analytics applications (Section 3.1). We then explore techniques for latency prediction of web services (Section 3.2), evaluate the performance of our prediction techniques (Section 3.3), describe related work (Section 3.4) and finally, conclude (Section 3.5).

3.1 Data analytics applications

Applications running on data analytics frameworks such as Hadoop [8], Spark [10], Hive [9], Pig [28], Tez [58] etc. can be modeled as a DAG of MapReduce jobs. As described in Section 2.2, MapReduce jobs have a very simple structure and lend themselves easily for performance predictability. In this section, we first describe our assumptions on modeling the execution and latency properties of MapReduce jobs (Section 3.1.1). We then present our predictor MRCute, which is used to determine the latency-response functions for MapReduce jobs (Section 3.1.2) and describe how to generalize it to incorporate DAGs of MapReduce jobs (Section 3.1.3).

3.1.1 Assumptions

To quickly predict the latency of MapReduce jobs, we make several assumptions in modeling them. First, the latency of each stage (map, shuffle and reduce) is assumed to be proportional to the amount of data processed by it and similar across all tasks within a particular stage. While this is valid for a wide variety of jobs [36], it may not hold for cases where the computation latency depends on the value of the data read or in the presence of significant *data skew* across tasks [59–62]. Techniques that have been proposed elsewhere [59] can be adopted to handle such issues. Second, the resource demands of the map and reduce stages are assumed to be similar to previous runs of the job (on different datasets). We observed this assumption to hold for a variety of workloads we examined (Section 2.2 and Section 4.5), and has also been shown to hold elsewhere [16, 41]. Finally, we assume that the data (both input and intermediate) and tasks of the various stages in a job are spread uniformly across all the machines allocated to the job.

While deviations from the above assumptions can lead to errors in predicting the latency of a job, our results (Section 4.4 and Section 4.5) show that Concorde is robust to such errors.

3.1.2 MapReduce jobs

Inspired by profiling-based approaches for fast database query optimization [57], we capitalize on the well-defined nature of the MapReduce framework to predict the performance of MapReduce jobs. To this end, we designed a prediction tool called

MRCute or MapReduce Completion Time Estimator. MRCute takes a gray-box approach to performance prediction by complementing an analytical model with job profiling. We first develop a high-level model of the operation of MapReduce jobs and construct an analytical expression for a job's latency (white-box analysis). The resulting expression consists of job-specific and infrastructure-specific parameters such as map processing rate, disk I/O bandwidth, network bandwidth etc. We determine these parameters from previous runs of the jobs (e.g., in case of recurring jobs) or by profiling the job on a sample dataset (black-box analysis).

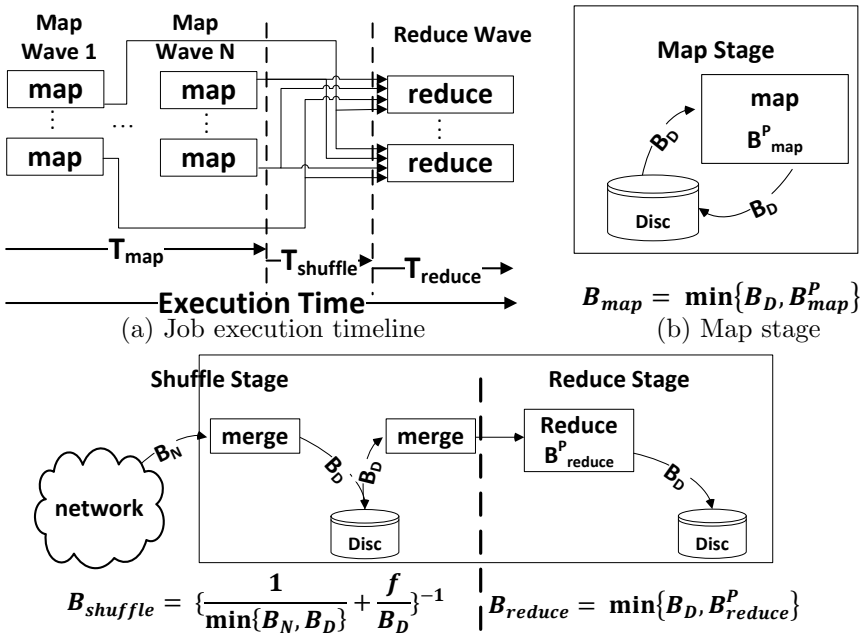
Given a MapReduce job $\mathcal{P}$, size of its input data $|D_i|$, a sample of the input data D_s^i, and a resource vector $\vec{r}$, MRCute estimates the application latency:

$$MRCute(\mathcal{P}, |D_i|, D_s^i, \vec{r}) \rightarrow T_{estimate}.$$

For jobs with deadline requirements, we reserve virtual clusters and their resource vector is specified as $\vec{r} = <N, B>$, where N is the number of machines allocated to the job and B is the amount of network bandwidth between these machines using the virtual cluster abstraction [42]. For the remaining jobs (without deadlines), we model at the granularity of racks and $\vec{r} =< r >$, where r the number of racks allocated to the job. Varying the value of $\vec{r}$ (i.e., its components), we get the latency-response function of the job $L(\vec{r})$.

Analytical model. As shown in Figure 3.1a, the execution of MapReduce jobs comprises of three stages. All tasks in a stage may not run simultaneously. Instead, the tasks execute in *waves*. For instance, Figure 3.1a consists of N map waves. We assume that these three stages execute sequentially. This implies that the completion time for a job is the sum of the time to complete individual stages, i.e., $T_{estimate} = T_{map} + T_{shuffle} + T_{reduce}$. We note that, in practice, it is possible for the shuffle and reduce stage to run in parallel with the map stage. Our techniques can be extended to handle this also.

To determine the completion time of individual stages, one needs to estimate the rate at which a given stage processes data. We use the term *stage bandwidth* to describe this rate. Since each stage uses multiple resources (CPU, disk, network), the slowest or the bottleneck resource governs the stage bandwidth. Hence, we apply bottleneck analysis to the MapReduce framework to determine the bandwidth for individual stages [63]. Overall, the completion time for each stage depends on the number of waves in the stage, the amount of data consumed or generated by each



(a) Job execution timeline (b) Map stage

(c) Shuffle - Reduce stages

Figure 3.1: Timeline of a MapReduce job and description of resources involved at each stage.

task in the stage and the stage bandwidth. We first discuss how to model each of these components for jobs with network reservations (using virtual clusters [42]) and then, for jobs without such reservations.

Jobs with network bandwidth reservations. The completion time of the different stages in a MapReduce job is modeled using the following components.

1. *Stage bandwidth.* During the map stage (Figure 3.1b), each map task reads its input off the local disk, applies the map function and writes the intermediate data to local disk. Thus, a map task involves two resources, the disk and CPU, and the map stage bandwidth is governed by the slowest of the two. Hence, $B_{map} = Min\{B_D, B^{\mathcal{P}}_{map}\}$, where B_D is the disk I/O bandwidth and $B^{\mathcal{P}}_{map}$ is the rate at which data can be processed by the map function of the job $\mathcal{P}$ (assuming no other bottlenecks). For non data-local maps, the network bandwidth is also considered when estimating B_{map}. Following the same logic, the reduce stage also involves the CPU and disk, and $B_{reduce} = Min\{B_D, B^{\mathcal{P}}_{reduce}\}$.

   During the shuffle stage (Figure 3.1c), reduce tasks complete two operations. Each reduce task first reads its partition of the intermediate data across the network, and then merges and writes it to disk. Hence, bandwidth $= Min\{B_D, B\}$

as the slower of the two resources governs the time for this operation, where B is the network bandwidth (as defined above). Next, the data is read off the disk and merge-sorted before being consumed by the reduce stage. This operation is bottlenecked at the disk, i.e., bandwidth = B_D. Given that the two operations occur in series, the shuffle stage bandwidth is

$$B_{shuffle} = \left\{ \frac{1}{Min\{B_D, B\}} + \frac{1}{B_D} \right\}^{-1}.$$

2. *Data consumed.* For a MapReduce job with M map tasks, R reduce tasks and input of size $|D_i|$, each map task consumes $\frac{|D_i|}{M}$ bytes, while each reduce task consumes $\frac{|D_i|}{S_{map} \times R}$ bytes and generates $\frac{|D_i|}{S_{map} \times S_{reduce} \times R}$ bytes. Here, S_{map} and S_{reduce} represent the *data selectivity* (i.e., the ratio of input data size to output data size) of map and reduce tasks respectively. These are assumed to be uniform across all tasks.

3. *Waves.* For a job using N machines with M_c map slots per machine, the maximum number of simultaneous mappers is $N \times M_c$. Consequently, the map tasks execute in $w_{map} = \lceil \frac{M}{N \times M_c} \rceil$ waves. Similarly, the reduce tasks execute in $w_{reduce} = \lceil \frac{R}{N \times R_c} \rceil$ waves, where R_c is the number of reduce slots per machine.

Given these values, the completion time for a task is determined by the ratio of the size of the task input to the task bandwidth. Further, as tasks belonging to a stage execute in waves, the completion time for a stage depends on the number of waves and the completion time for the tasks within each wave. For simplicity, we ignore the overlap of task executions across multiple waves. Thus, the completion time for the map stage is given by

$$T_{map} = w_{map} \times \frac{Input_{map}}{B_{map}} = \lceil \frac{M}{N \times M_c} \rceil \times \left\{ \frac{|D_i|/M}{B_{map}} \right\}.$$

Using similar logic for the shuffle and reduce stage completion time, the estimated job latency is

$$\begin{aligned} T_{estimate} &= T_{map} + T_{shuffle} + T_{reduce} \\ &= w_{map} \times \left\{ \frac{|D_i|/M}{B_{map}} \right\} + w_{reduce} \times \left\{ \frac{|D_i|/\{S_{map} \times R\}}{B_{shuffle}} \right\} \\ &\quad + w_{reduce} \times \left\{ \frac{|D_i|/\{S_{map} \times S_{reduce} \times R\}}{B_{reduce}} \right\}. \end{aligned}$$

Jobs without network bandwidth reservations. Here, we model MapReduce jobs at the granularity of racks, which differs from the above in two aspects.

1. *Waves.* Given r racks and m machines per rack, the map stage runs in $w_{map} = \lceil \frac{M}{r \times m \times M_c} \rceil$ waves and the reduce stage runs in $w_{reduce} = \lceil \frac{R}{r \times m \times R_c} \rceil$ waves.

2. *Shuffle stage.* The latency of the shuffle stage is determined by the maximum of two components:
   (a) Time to transfer data across the core: This is calculated as $T_{core} = \frac{D_i^{core}}{B_M/O}$ where B_M is the bandwidth per machine (i.e., NIC bandwidth), and O (> 1) is the oversubscription factor. D_i^{core} is the amount of data transferred across the core by a machine, which can be expressed as $D_i^{core} = D_i^S \times \frac{1}{r \times m} \times \frac{r-1}{r}$, for $r > 1$, where $D_i^S = \frac{D_i}{S_{map}}$, the total shuffle data of the job. If $r = 1$, i.e., the job is allocated only one rack, $D_i^{core} = 0$.
   (b) Time to transfer data within a rack: Apart from the data transferred across the core, each machine transfers the data D_i^{local} within its rack which is given by $\frac{D_i}{S_{map}} \times \frac{1}{r \times m} \times \frac{1}{r}$. While $\frac{1}{m}^{\text{th}}$ of this data remains on the same machine, the remaining data is transferred to other machines using a bandwidth of $B_M - B_M/O$. Thus, the time for transferring data within the rack is given by $T_{local} = D_i^{local} \times \frac{m-1}{m} \times \frac{1}{B - B/O}$.
   The latency of the shuffle stage is given by $T_{shuffle} = w_{reduce} \times \max\{T_{core}, T_{local}\}$. The latency per map and reduce task remain the same as in the case with network reservations.

Estimating job-specific parameters. The analytical model discussed above involves two types of parameters: (i) those that are specific to MapReduce configuration and are known, such as the number of map slots per machine (M_c), and (ii) those that depend on the infrastructure and the actual job. To estimate the latter, MRCute uses logs from previous runs, $H_{\mathcal{P}}$, of job $\mathcal{P}$ (possibly on different data sets) and determines the execution time for each task and each stage, the amount of data consumed and generated by each task, etc. All this information is used to determine the data selectivity and bandwidth for each stage. In the absence of such history, we profile job $\mathcal{P}$ by executing it on a *single* machine using a sample of the input data D_s^i and use the logs from the profiling run to determine the necessary parameters. Concretely,

$$Estimator(\mathcal{P}, D_s^i, H_{\mathcal{P}}) \rightarrow \{S_{map}, S_{reduce}, B_{map}, B_{reduce}, B_D\}.$$

For instance, the ratio of the data consumed by individual map tasks to the map task completion time yields the bandwidth for the job's map stage (B_{map}). The reduce stage bandwidth is determined similarly. As the profiling involves only a single machine with no network transfers, the observed bandwidth for the shuffle stage is not useful for the model. Instead, we measure the disk I/O bandwidth (B_D) under MapReduce-like access patterns, and use it to determine the shuffle stage bandwidth.

The job profiler assumes that the stage bandwidth observed during profiling is representative of actual job operation. Satisfying this assumption poses two challenges:

1. *Infrastructure heterogeneity.* Ideally, the machine used for profiling should offer the same performance as any other machine in the datacenter. While physical machines in a datacenter often have the same hardware configuration, their performance can vary, especially disk performance [64]. Indeed, we observed variable disk performance which significantly degrades prediction performance. To counter this, MRCute maintains statistics regarding the disk bandwidth of individual machines (see Section 3.3).

2. *Representative sample data.* The sample data used for profiling should be representative and of sufficient size. If too small, intermediate data will not be spilled to disk by the map and reduce tasks, and the observed bandwidth can be different from that seen by the actual job. We use MapReduce configuration parameters regarding the memory dedicated to each task to determine the minimum size of the sample data (see Section 3.3).

3.1.3 DAGs of MapReduce jobs

Any stage in a data-parallel DAG generated by frameworks such as Hive [9], and Tez [58] can be modeled as a MapReduce job. We use this insight to develop the response functions for DAGs. Using the MapReduce model above, we first determine the latency-response function $L_s(\vec{r})$ for every stage s in the DAG. The latency of the DAG is determined by it's critical path CP (i.e., the path from any source to any sink in the DAG that takes the longest time to execute) and can be calculated as $L_{DAG}(\vec{r}) = \sum_{s \in CP} L_s(\vec{r})$.

3.2 Web services

Unlike data analytics applications which are run only once, web services run continuously processing multiple user queries or requests over time. Further, as the processing time of any request depends on the content of the request along with runtime factors such as slow machines, slow disks, and network packet loses, each request can experience a different end-to-end latency. Thus, the latency of a web service is typically specified as a statistical distribution. As a particular latency percentile can vary significantly over time (e.g., 99th latency percentile on a weekend vs. 99th percentile on a weekday), the latency-response curves we develop for web services capture the variance of their latency distributions. These aggregate statistics are more robust to variations in time. Using variance also allows us to decompose the end-to-end latency of the web service DAG into that of its stages (Section 5.2).

The *static* resource requirements for web services (e.g., number of machines to host a particular stage) typically depend on their properties (e.g., number of shards of a dataset) and are specified by the application owner. Thus, we aim to model the latency variance of these applications based on the *extra resources* which can be used by different stages at runtime. For example, using a group of spare machines to handle duplicate requests at a particular stage can reduce the latency variance of the stage and consequently the end-to-end latency variance of the web service. We model these extra resources as a resource budget $\vec{r}$ available for the web service and determine the variance in end-to-end latency as a function of this budget, $\text{Var}(L(\vec{r}))$. Determining the static requirements of web services is outside the scope of this dissertation.

In particular, based on the techniques we use for reducing runtime latencies (Chapter 5), we consider two types of resources (a) extra compute resources available for requests reissues, and (b) incomplete application responses. In the rest of this section, we present how we model the latency variance as a function these resources.

3.2.1 Model

Suppose a web service DAG, w, is specified by (V, E) where $V = \{1, 2, \ldots, n\}$ is the set of stages in the DAG and E denotes the input-output dependencies between these stages. Given a total resource R available for w, suppose $\vec{r} = \{r_1, r_2, \ldots, r_n\}$ denotes the partition of this resource across the different stages in w (r_s is the resource available for stage s). We note that R is an abstract resource and can be used to

specify either extra compute resources or incomplete application responses.

Our goal is to estimate the variance in latency of w, $\mathrm{Var}(L_w(\vec{r}))$, as a function of $\vec{r}$. As described in Section 5.2, we use $\sum_s \mathrm{V}_s(r_s)$ as a proxy for $\mathrm{Var}(L_w(\vec{r}))$, where the variance-response function $\mathrm{V}_s(r_s)$ denotes the variance of stage s with r_s resource allocated to it. In the next two sections, we describe how to estimate these variances for 2 different kinds of resources.

3.2.2 Request reissues

Request reissue is a standard technique to reduce the latency tail in distributed systems at the cost of increasing resource utilization. A typical use of reissues is to start a second copy of the request at a pre-determined time T_s, if there is no response before T_s, and use the first response that returns [65]. Reissuing a request in a particular stage of a web service (i.e., reissuing the work at one or more servers corresponding to a stage) elsewhere in the cluster is feasible as stages are often replicated.

To determine the variance-response functions for request reissues, we first use application-level logs to determine the time spent by a request in different stages in a web service DAG. Aggregating these latencies across several requests (e.g., over the last 24 hours) allows us to determine the latency distribution f_s of each stage s. Given the target fraction of requests r_s to reissue in stage s, the corresponding timeout T_s is equivalent to the $(1 - r_s)^{\text{th}}$ quantile of f_s. For example, to reissue 5% of requests, we set T_s to the 95th percentile of f_s. We can thus obtain the variance-response function $\mathrm{V}_s(r_s)$ for different values of r_s, by computing the corresponding T_s, and then performing an offline simulation using the latencies from past queries at this stage. We use standard interpolation techniques to compute the convex-hull, $\bar{\mathrm{V}}_s(r_s)$, to preclude discretization effects.

3.2.3 Incomplete responses

In many situations, partial answers are useful both to the user and to subsequent stages in the web service DAG. An example is a stage which picks the best results from many responders. Similar to the web search stage shown in Figure 2.1, the image, video and ad search stages at Microsoft Bing consist of many servers that in parallel compute the best matching images, videos, and ads for a search query. In

each, an *aggregator* picks the best few responses overall. Not waiting until the last server responds will speed up the stage while returning an incomplete response.

How to measure the usefulness of an incomplete answer? Some stages have explicit indicators; e.g., each returned document in the web search DAG has a relevance score. For such stages, we say that the answer has utility 1 if it contains the highest ranked document and 0 otherwise (alternatively, we can give weights to each of the top ten results). For other stages, we let utility be the fraction of components that have responded[1]. We define the utility loss of a set of queries as the average loss in utility across all the queries. So, if 1 out of 1000 queries loses the top ranked document, the utility loss incurred is 0.1%.

Various timeout mechanisms can be used to determine when the aggregator should stop waiting for the remaining responders and return its answer. While we discuss various timeout schemes in detail later (Section 5.3.2), for any given utility loss budget r_s for a particular stage s, we use offline simulations and application logs of previous queries (which indicate the relevance scores of the responses from different servers) to choose the timeout parameters such that we achieve the minimum latency variance for the stage while ensuring that the utility loss is at most r_s. Repeating this process for different values of r_s and determining the corresponding variance of the stage s, we get the variance-response curve $\mathrm{V}_s(r_s)$ for the stage when using incomplete responses.

3.3 Evaluation

In this section, we evaluate the performance of our latency estimation techniques for data analytics applications showing that we can predict application completion times with an average prediction error as low as 12%, with low overhead.

3.3.1 Methodology

We use MRCute to predict the job completion of the five MapReduce jobs described in Table 2.1. For each job, MRCute predicts the completion time for varying number of nodes (N) and the network bandwidth between them (B). The prediction involves profiling the job with sample data on a single node, and using the resulting job

[1]If responders are equally likely to yield the most relevant result, both these measures yield the same expected loss.

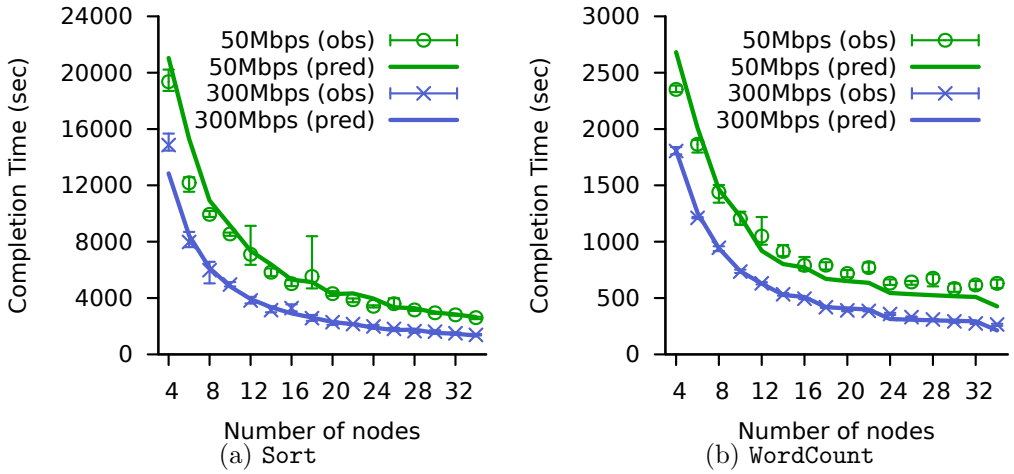


Figure 3.2: Predicted completion time for `Sort` (an I/O intensive job) and `WordCount` (a CPU intensive job) matches the observed time.

parameters to drive the analytical model.

To determine actual job completion times, we executed each job on a 35-node Emulab cluster with Cloudera's distribution of Hadoop MapReduce (version 0.20.2) [66]. Each node has a quad-core Intel Xeon 2.4 GHz processor, 12 GB RAM and a 1 Gbps network interface. The unoptimized jobs were run with default Hadoop configuration parameters. The number of mappers and reducers per node is 8 and 2 respectively, HDFS block size is 128 MB, and the total number of reducers is twice the number of nodes used. While parameter tuning can improve job performance significantly [67], our focus here is not improving individual jobs, but rather predicting the performance for a given configuration. Hence, the results presented here apply as long as the same parameters are used for job profiling and for the actual execution.

3.3.2 Prediction accuracy

We first focus on the results for `Sort` and `WordCount`, two jobs at extreme ends of the spectrum. `Sort` is an I/O intensive job while `WordCount` is processor intensive. Figures 3.2a and 3.2b plot the observed and predicted completion time for five runs of these jobs when varying N and B. The figures show that the predicted and observed completion times are close throughout, with 8.9% prediction error on average for `Sort` and 20.5% at the 95th percentile.

To understand the root cause of the prediction errors, we look at the per-stage completion time. Figure 3.3 presents this breakdown for `Sort` with varying number

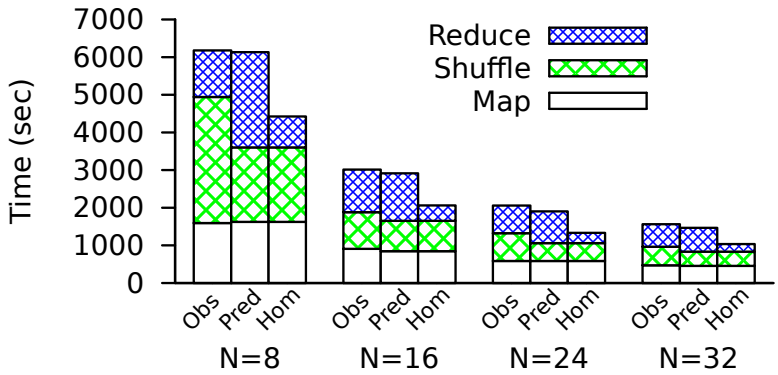


Figure 3.3: Per-stage breakdown of the observed (*Obs*) and predicted (*Pred*) completion time for `Sort` with bandwidth = 300 Mbps. *Hom* represents the predicted time assuming homogeneous disk performance.

of nodes. The bars labeled *Obs* and *Pred* represent the observed and predicted completion time respectively. The figure shows that the predicted time for the map stage is very accurate; most of the prediction error results from the shuffle and reduce stages.

The reason for this difference in the prediction accuracy is that the map stage typically consists of multiple waves. Consequently, any *outlier* map tasks that are straggling in the earlier waves get masked by the latter waves and they do not influence the observed stage completion time significantly. In contrast, the shuffle and reduce stages execute in a single wave as the number of reduce tasks is the same as the number of reduce slots on the nodes. As a result, any outlier reduce tasks inflate the stage completion time and in turn, the job completion time. Overall, such outliers introduce errors in the predicted completion time.

Beyond `Sort` and `WordCount`, the predicted estimates for the other jobs show similar trends. For brevity, we summarize the prediction errors in Table 3.1. Overall, we find a maximum average error of 11.5% and a 95th percentile of 20.5%.

3.3.3 Accounting for outliers

The basic MRCute model predicts job completion time assuming "ideal" operation. However, data analytics in production settings is far from ideal; nodes and tasks fail, and many tasks are outliers. While guaranteed network bandwidth avoids outliers due to network contention, here we describe how MRCute deals with outliers due to bad machines and quantify the impact of workload imbalance.

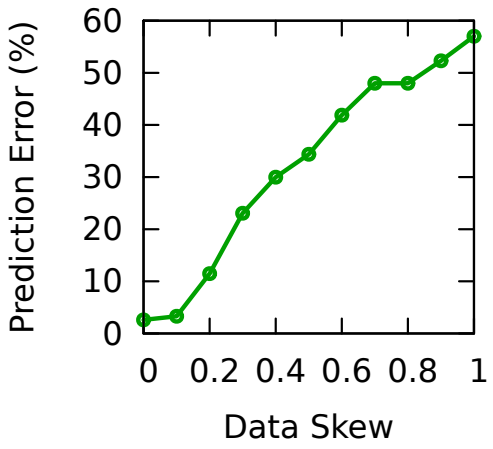


Figure 3.4: Impact of skew.

Bad machines. To account for disk performance variability, MRCute maintains statistics regarding the disk bandwidth of individual machines. To highlight the benefits of such benchmarking, the bars in Figure 3.3 labeled *Hom* (Homogeneous) show the predicted times when MRCute does not account for disk performance heterogeneity, and instead, uses a constant value for the disk bandwidth in the analytical model. As the performance of the disks on individual nodes varies, such an approach underestimates the reduce stage time, which leads to a high prediction error.

Workload imbalance. The amount of data processed by tasks belonging to the same stage can vary significantly which, in turn, leads to outliers. To quantify the impact of such outliers on MRCute, we artificially introduced skew for the `Sort` job by choosing input keys from a skewed distribution. Here skew is the coefficient of variation ($\frac{stdev}{mean}$) for input across tasks belonging to the same stage. Figure 3.4 shows that the prediction error increases almost linearly with increasing skew. This is expected given that MRCute profiles the job only on sample data and does not explicitly account for input data skew. Mantri [32] reported a median data skew of 0.34 which leads to 26% prediction error for MRCute. To account for such outliers resulting from workload imbalance and other factors, we use slack when determining resources for a job. This allows us to satisfy application goals in the presence of outliers.

Hadoop Job	Stages	Sample Data Size	Profiling Time	Prediction error (all runs)	
				Average	95%ile
`Sort`	1	1GB	100.8s	8.9%	20.5%
`WordCount`	1	450MB	67.5s	8.4%	19.7%
`Gridmix`	3	16GB	546s	11.5%	17.8%
`TF-IDF`	3	3GB	335s	5.6%	9.7%
`LinkGraph`	4	3GB	554.8s	8.2%	12.3%

Table 3.1: Prediction overhead and error of MRCute for Hadoop jobs.

3.3.4 Prediction overhead

In the absence of job history, MRCute profiles a job on sample input data to determine the job parameters. This imposes two kinds of overhead.

(1). *Sample data.* We use information about the MapReduce configuration parameters, such as when data is spilled to the disk, to calculate the size of the sample data needed for the job. This is shown in Table 3.1. Other than `Gridmix`, the jobs require <3 GB of sample data, a non-negligible yet small value compared to typical datasets used in data intensive workloads [22]. `Gridmix` is a multi-stage job with high selectivity. Hence, we need more sample data to ensure enough data for the last stage when profiling, as data gets aggregated across stages. This overhead could be reduced by profiling individual stages separately but requires detailed knowledge about the input required by each stage.

(2). *Profiling time.* Figure 3.1 also shows the time to profile individual jobs. For `Sort` and `WordCount`, the profiling takes around 100 seconds. For the multi-stage jobs, profiling time is higher as more data needs to be processed. However, a job needs to be profiled only once to predict the completion time for all resource tuples, and we can also use information from past runs.

3.4 Related work

There has been a lot of progress towards performance prediction of data analytics applications. For example, Mumak [37] and MRPerf [38] are discrete-event simulators for MapReduce, Ganapathi et al. [39] use statistical analysis to discover feature vectors and predict job performance, ParaTimer [68] is a job progress estimator that relies on debug runs of the job. Our system, Concorde, needs fast prediction and hence, cannot use detailed simulators like MRPerf that take minutes per simulation [38];

exploring 100 resource tuples would take 100s of minutes. Instead, we adopt a hybrid approach that trades off accuracy for prediction speed. Tian et. al. [69] use a similar tact involving profiling on sample data and then linear regression for prediction.

Elasticiser [67] and Conductor [70] translate application goals into resource requirements, while Aria [71] and Jockey [16] focus on private settings. All these proposals involve a performance prediction component; Conductor uses a model while Jockey uses a simulator. Like Concorde, Aria and Elasticiser combine profiling with modeling. Aria uses historical information for profiling and has a model that estimates performance bounds. Elasticiser profiles using instrumentation and relies on a statistical model. Unlike MRCute, most of these existing predictors do not model the dependence of job completion time on the network bandwidth available. This is crucial for Concorde which aims to coordinate the storage and compute sub-systems to schedule datacenter applications in a network-aware manner. Finally, there has been work towards performance prediction for other datacenter applications (e.g., web [72] and ERP applications [73]), which do not directly apply to the problem Concorde aims to address.

3.5 Conclusion

The ability to predict application performance as a function of the resources allocated allows us to determine how cluster resources should be partitioned across multiple applications or across multiple stages within an application to meet end-to-end latency goals. In this chapter, we explored how to determine such latency functions for two types of datacenter applications — data analytics and web services. In particular, we proposed MRCute which uses the characteristics of data analytics applications to determine their latency as a function of the number of machines (or racks) allocated and the network bandwidth available across those machines. Further, we show how to quantify the latency of web services as a function of the additional compute resources usable at runtime and loss in relevance tolerable by the application. Using experiments on a 35 node Hadoop cluster and a variety of workloads, we show that MRCute can allow us to predict application latency with less than 12% average error.

Chapter 4

Resource planning for datacenter applications

Datacenter applications utilize several datacenter *sub-systems* such as network, compute, storage, and memory for their execution. Different stages that make up these DAG-structured applications can be bottlenecked on different sub-systems; for example, the map stage in a MapReduce job can be compute-bound while the reduce stage can be disk-bound. Thus, the end-to-end latency of the application can be determined by multiple sub-systems along with how efficiently they function together for executing the application.

Consider a MapReduce job whose input data is stored in a distributed file system such as HDFS [27]. The input data chunks are generally spread across multiple machines in the cluster, randomly and agnostic to the characteristics of the application(s) which reads the data. Using techniques such as delay scheduling [74] or Quincy [75] can result in a large fraction of map tasks running on the machines holding their input data, thus, avoiding the need to utilize the network for the map stage. However, the reducers will have to read their input data (the map output data) from across several machines in the cluster, invariably having to use cross-rack links in the cluster. This can result in increased job latency as, while there is full bisection bandwidth within a rack, modern data intensive computing clusters typically have oversubscription values ranging from 3:1 to 10:1 from the racks to the core [18, 35, 54]. Further, a large fraction of the cross-rack bandwidth (up to 50%) can be used for background data transfers [35], effectively reducing the bandwidth available even more.

The above example shows that the lack of coordination between the storage, network and compute in datacenter applications results in sub-optimal performance, especially as applications like data analytics can read, process and transfer large amounts of data over the network (up to several petabytes each day [4]). Existing cluster schedulers try to overcome these problems by optimizing the placement of compute or by scheduling network flows, while assuming that the input data locations are *fixed*. However, different stages in the applications can still run into network

bottlenecks as illustrated above. Recent techniques like ShuffleWatcher [76] attempt to localize the shuffle of a MapReduce job to one or a few racks, but end up using the cross-rack bandwidth to read input data. The benefits from using flow-level techniques such as Varys [18] or Baraat [19] are also limited, as they only schedule network transfers after both the source and destination are fixed. Sinbad [35] schedules flows around network contention but its benefits are limited to file system writes.

However, a large number of business-critical applications are *recurring*, with predefined submission times and predictable resource requirements, allowing us to carefully place input data to improve network locality. For example, it has been reported that production workloads contain up to 40% recurring jobs, which are run periodically as new data becomes available [16, 41]. Using these characteristics, we can *plan ahead* and determine where the input data of an application can be placed (e.g., in a particular rack). By coordinating placement of data and tasks, most small applications can run completely within one rack with all stages achieving rack-level network locality and no oversubscription. Because the small applications do not use cross-rack links, those running across multiple racks also see benefits due to more available bandwidth.

Using this intuition, we design Corral, which exploits the characteristics of datacenter workloads to *jointly* optimize the placement of data and compute, and improve application latency. Each application is submitted to Corral with a set of *goals* (e.g., completion time deadline) and resource *constraints* (e.g., number of machines to run an application stage). Depending on the presence (or absence) of these goals and constraints, Corral uses two different components. First, for applications with goals, Corral uses prediction tools like MRCute (Chapter 3) to determine the resources required to meet the goals. As a large fraction of datacenter applications exhibit resource malleability (Section 2.3), multiple resource tuples often satisfy these goals. Corral uses a *resource selector* to pick the tuple which improves the cluster's throughput. For applications with constraints, the resource selector picks the tuple which satisfies these constraints. Though our core ideas apply to general multi-resource settings, we focus on two specific resources in this dissertation, namely, compute instances (N) and the network bandwidth (B) between them. Corral reserves network bandwidth[1] using the virtual cluster abstraction [42] to ensure that the goals are met. The cluster scheduler in Concorde uses Oktopus [42] to determine how to provision

[1]To ensure that such reservations are work-conserving, we can use techniques proposed elsewhere [77].

the physical resources (for job execution) for the selected resource tuple.

Second, for applications without specific goals, Corral aims to minimize the average completion time or makespan. Corral uses an *offline planner* to schedule such applications. It uses the applications characteristics — amount of data processed, CPU and memory demands, input to output selectivity etc. — to determine which *set of racks* should be used to run each application and when it should start. To this end, the planner formulates a *planning problem*, which we model as a malleable job scheduling problem [43–45]. We develop simple heuristics to efficiently solve this problem. We show that these heuristics achieve performance appealingly close (within 3-15%) to the solution of a Linear Program (LP) relaxation, which serves as a lower-bound to a particular formulation of the planning problem (Section 4.3.5). The application execution is decoupled from the planner and the planner's output is used as hints for placing data and tasks during data upload and job execution. The offline planner optimizes for both recurring jobs and jobs whose characteristics are known in advance. The techniques in Corral apply to both simple MapReduce jobs as well as complex DAG-structured workloads such as Hive [9] or Scope [11] queries.

We note that during job execution, we *do not exclusively allocate* whole racks to a single job. Instead, given the set of racks assigned to a job by the offline planner, one replica of job input data is placed within those racks and all tasks of the job are *restricted to run within those racks*. This forces all job data transfers to stay within those racks. The remaining slots within these racks are used by ad hoc jobs and also by other planned jobs assigned to the same racks. When a significant fraction of machines in these racks fail (beyond a configured threshold), Corral ignores these constraints and uses any available resources in the cluster to schedule the planned jobs.

The offline planner in Corral improves job performance, when the network is a constrained resource, for the following reasons. First, it runs the small jobs in a single rack, i.e., at least one replica of the job input data and all its compute tasks are placed inside just one rack. These jobs can take advantage of full bisection bandwidth within a rack, thus speeding up their network-heavy stages. Second, while large jobs continue running across multiple racks, they finish faster as they can use the bandwidth freed up by the small jobs. Similarly, even ad hoc jobs benefit as they can also use the additional bandwidth. Finally, by running jobs in their own subset of racks rather than across the whole cluster, we improve isolation across jobs.

We have implemented the techniques in Corral in Hadoop and evaluate them on

testbeds using production workloads and a representative set of MapReduce jobs. We show that when using network reservations, Corral enables the datacenter provider to accept 3-14% more jobs which represents a 7-87% improvement in datacenter goodput. When scheduling workloads without network reservations, Corral achieves a 10-33% reduction in makespan for a batch workload and a 30-56% reduction in median completion time compared to Hadoop's capacity scheduler [49].

4.1 Overview

Corral exploits coordination between the storage, and compute sub-systems in datacenters to meet applications requirements and improve their end-to-end latencies. Each application i submitted to Corral is specified by a 3-tuple $S_i = (\mathcal{A}_i, \mathcal{G}_i, \mathcal{C}_i)$, where

(a) $\mathcal{A}_i$ specifies application information such as the program to run, input data to run on, logs of previous application executions etc.

(b) $\mathcal{G}_i$ specifies the application goals to be met. These goals are used to determine the resources that need to be provisioned for the application. In this dissertation, we consider completion time deadlines as the only goals that can be specified for data analytics applications. For web services, request throughput is a more relevant goal. While we do not focus on developing techniques to determine the resource requirements of web services, existing techniques (e.g., [78–80]) can be used to do so.

(c) $\mathcal{C}_i$ is a set of tuples, where each tuple specifies the constraints on the resource requirements of each application stage. We consider two resources that can be specified — the number of compute nodes and the network bandwidth between these nodes. Thus, for an application with s stages, $\{1, 2, \ldots, s\}$, $\mathcal{C}_i$ is specified as $\{(1, N_1, B_1), (2, N_2, B_2), \ldots, (s, N_s, B_s), B_1^o, B_2^o, \ldots, B_s^o\}$ where N_k and B_k are the compute and bandwidth requirements for stage k and B_k^o is the bandwidth with which stage k communicates with other stages. This is similar to the *virtual oversubscribed cluster* abstraction proposed earlier [42].

Either $\mathcal{G}_i$ or $\mathcal{C}_i$ can be an empty set ($\emptyset$), which suggests the absence of the corresponding goals or constraints respectively. While each of these sets can be specified

in a variety of combinations (e.g., $\mathcal{C}_i$ contains requirements only for a subset of the stages), we deal with only specific requirements in this chapter — $\mathcal{G}_i$ is specified only for data analytics applications to include deadlines while $\mathcal{C}_i$ is typically specified only for web service applications and contains the resource requirements for all stages in the application DAG. While these requirements seem restrictive, they cover most scenarios in practice where (i) users of data analytics applications care about only higher level goals such as completion time and not about low-level details such as the number of machines allocated to them [36], and (ii) the resource requirements for web services are determined offline by application owners.

Given the above application requirements, Corral uses three components to schedule them: (a) *resource selector*, (b) an *offline planner*, and (c) the cluster scheduler (see Figure 1.1). We next describe each of these components briefly and leave the details for later sections.

Resource selector. For data analytics applications with $\mathcal{G}_i \neq \emptyset$, Corral uses MRCute (Section 3.1) to determine multiple resource tuples $<N, B>$ (N is the number of machines and B is the bandwidth between these machines), which satisfy the specified completion time. The resource selector ranks these tuples in terms of the cost to accommodate the tuple on the cluster and selects the tuple with the least cost, thus improving cluster efficiency. For applications with $\mathcal{C}_i \neq \emptyset$, only one resource tuple is feasible and this is selected by the resource selector. If the bandwidth requirements are not specified in these constraints, then it is set equal to the NIC bandwidth. The resource tuples selected are specified using the virtual cluster or the virtual oversubscribed cluster abstraction to the cluster scheduler. We provide the details of the resource selector in Section 4.2.

Offline planner. The offline planner is used to schedule applications with $\mathcal{C}_i = \emptyset$ and $\mathcal{G}_i = \emptyset$. The planner uses the characteristics of applications (e.g., arrival time, input data size etc.) that will be submitted to the cluster *in the future* to estimate their latencies using MRCute (Section 3.1). It solves an offline *planning problem* with the goal of minimizing a specified metric (e.g., makespan or average job completion time). The planner creates an offline schedule which consists of a tuple $\{R_j, p_j\}$ for each job j, where R_j is the set of racks on which job j has to run and p_j is its priority. It provides this schedule as *guidelines* to the cluster scheduler.

The offline planner will periodically receive updated estimates of future workload, rerun the planning problem, and update the guidelines to the cluster scheduler. Note that, as the cluster conditions might change after the plan was generated (e.g.,

racks/machines failing), if the assigned locations are not available for any job, the scheduler will ignore the guidelines for that job and use randomly selected machines in the cluster to hold its data (while ensuring fault tolerance using a policy similar to HDFS) and run its tasks.

The above scheduling approach in Corral applies not only to recurring jobs but also other jobs which are known a priori and whose characteristics can be predicted. For jobs which are not known in advance, the input data is uniformly spread out across the cluster using regular HDFS policies and their tasks are scheduled using existing techniques [74] to any empty slots in the cluster.

Cluster scheduler. For applications with network reservations (e.g., those with completion time requirements), resource requirements are determined by the resource selector as described above. These requirements are specified to the cluster scheduler using existing cluster abstractions (virtual cluster or virtual oversubscribed cluster). The cluster scheduler uses Oktopus [42] to allocate these cluster abstractions.

For applications without network reservations and scheduled using the offline planner, the cluster scheduler uses the offline schedule as guidelines to place their data and tasks during runtime. As the input data of a particular application j is uploaded into the cluster and stored in a distributed filesystem (such as HDFS), the cluster scheduler places one replica of each data chunk in a randomly chosen rack from R_j, and the other replicas on two other racks, which are randomly chosen from the entire set of racks (excluding the one chosen so far). We note that these choices are consistent with the *per-chunk* fault tolerance policy, typically used in HDFS. When application j is submitted to the cluster, the cluster scheduler constraints its tasks to be scheduled within racks in R_j. Whenever a slot becomes empty in any rack, the scheduler examines all jobs which have been assigned this rack and assigns the slot to the job with the highest priority[2].

With such placement of data and tasks in the same set of racks, Corral ensures that not only the initial stage (e.g., map, extract) but also subsequent stages of the jobs (e.g., reduce, join) achieve rack-level data locality. The priority assigned to a job ensures that the order in which jobs are scheduled by the cluster scheduler conforms to that assigned by the offline planner.

[2]To ensure machine-level locality, we use delay scheduling [74].

4.2 Resource selection

The resource malleability property of datacenter applications (Section 2.3) offers flexibility to the cluster provider. As multiple resource tuples achieve similar completion times for an application, the provider can select which resource tuple to allocate. Corral takes advantage of this flexibility by selecting the resource tuple most amenable to the provider's ability to accommodate subsequent applications, thus maximizing the provider revenue. This comprises the two following sub-problems.

The *feasibility problem* involves determining the set of candidate resource tuples that can actually be allocated in the datacenter, given its current utilization. For our two dimensional resource tuples, this requires ensuring that there are both enough unoccupied VM slots on physical machines and enough bandwidth on the network links connecting these machines. Oktopus [42] presents a greedy allocation algorithm for such tuples which ensures that if a feasible allocation exists, it is found. We use this algorithm to determine *feasible resource tuples*.

The *resource selection problem* requires selecting the feasible resource tuple that maximizes the cluster's ability to accept future requests. However, in our setting, the resources required for a given tuple depend not just on the tuple itself, but also on the specific allocation. As an example, consider a tuple $<4, 200>$ requiring 4 VMs each with 200 Mbps of network bandwidth to other VMs. If all these VMs are allocated on a single physical machine, no bandwidth is required on the network link for the machine. On the other hand, if two of the VMs are allocated on one machine and two on another machine, the bandwidth required on the network links between them is 400 Mbps (2*200 Mbps).

To address this, we use the allocation algorithm in Oktopus [42] to convert each feasible resource tuple to a utilization vector capturing the utilization of physical resources in the datacenter after the tuple has been allocated. Specifically,

$$\text{Allocation}(< N, B >) \rightarrow U =< u_1, \ldots, u_d >,$$

where U is a vector with the utilization of all datacenter resources, i.e., all physical machines and links. The vector dimension d is the total number of machines and links in the datacenter. For a machine k, u_k is the fraction of the VM slots on the machine that are occupied while for a link k, u_k is the fraction of the link's capacity that has been reserved for the allocated VMs.

Overall, given the set of utilization vectors corresponding to the feasible tuples,

the *objective* of the resource selector is to select the resource tuple that will *minimize the number of rejected requests* in the future and hence, improve the cluster's throughput. Similar problems have been studied in various contexts, such as online ad allocation [81] and online routing and admission control in virtual circuit networks [82]. Depending on the context, one can show that different cost functions (that measure the cost for accepting a request) yield optimal scheduling for different request allocation models [83]. We experimented with a number of such cost functions and found that a cost function that captures the resource imbalance caused by the allocation of a resource tuple performs very well in terms of minimizing rejected requests. In our setting, *minimizing resource imbalance* translates to choosing the utilization vector that balances the capacity left across all resources after the request has been allocated. Precisely, our selection heuristic aims to minimize the following:

$$\text{minimize} \quad \sum_{j=1}^{d} (1 - u_j)^2.$$

Hence, the resource imbalance is defined as the square of the fractional under-utilization for each resource. The lower this value, the better the residual capacity across resources is balanced. In literature, this is referred to as the Norm-based Greedy heuristic [84]. An extra complication in our setting is the hierarchical nature of typical datacenters. This leads to a hierarchical set of resources corresponding to datacenter hosts, racks and pods. Next, we describe how this heuristic can be extended to such a setting.

4.2.1 Resource imbalance heuristic

The resource imbalance heuristic applies trivially to a single machine scenario. Consider a single machine with a network link. Say the machine has N^{max} VM slots of which N^{left} are unallocated. Further, the outbound link of the machine has a capacity B^{max} of which B^{left} is unallocated. The utilization vector for this machine is

$$< u_1, u_2 > = < 1 - \frac{N^{left}}{N^{max}}, 1 - \frac{B^{left}}{B^{max}} > .$$

Thus, the resource imbalance for the machine is

$$\sum_{j=1}^{2}(1-u_j)^2 = \left\{\frac{N^{left}}{N^{max}}\right\}^2 + \left\{\frac{B^{left}}{B^{max}}\right\}^2.$$

As physical machines in a datacenter are arranged in racks which, in turn, are arranged in pods, there is a hierarchy of resources in the datacenter. To capture the resource imbalance at each level of the datacenter, we extend the set of datacenter resources to include racks and pods. Hence, the datacenter utilization is given by the vector $< u_1, \ldots, u_m >$, where m is the sum of physical machines, racks, pods and links in the datacenter. For a rack k, u_k is the fraction of VM slots in the rack that are occupied and the same for pods. Hence, for a resource tuple being considered, the overall resource imbalance is the sum of the imbalance at individual resources, represented by set C, whose utilization changes because of the tuple being accepted, i.e., $\sum_{j \in C}(1-u_j)^2$.

A lower resource imbalance indicates a better positioned cluster. Hence, Corral chooses the utilization vector and the corresponding resource tuple that minimizes this imbalance. As the allocation algorithm is fast (median allocation time is less than 1 ms), we simply try to allocate all feasible tuples to determine the resulting utilization vector and the imbalance it causes.

4.2.2 Resource selection example

We now use a simple example to illustrate how Corral's imbalance-based resource selection works. Consider a rack of four physical machines, each with 2 VM slots and a Gigabit link. Also, imagine a application request with two feasible tuples $<N, B>$ (B in Mbps): $<3, 500>$ and $<6, 200>$. Figure 4.1 shows allocations for these two resource tuples. Network links in the figure are annotated with the (unreserved) residual bandwidth on the link after the allocation. The figure also shows the imbalance values for the resulting datacenter states. The former tuple has a lower imbalance and is chosen by Corral.

To understand this choice, we focus on the resources left in the datacenter after the allocations. After the allocation of the $<3, 500>$ tuple, the cluster is left with five empty VM slots, each with an average network bandwidth of 500 Mbps (*state-1*). As a contrast, the allocation of $<6, 200>$ results in two empty VM slots, again with

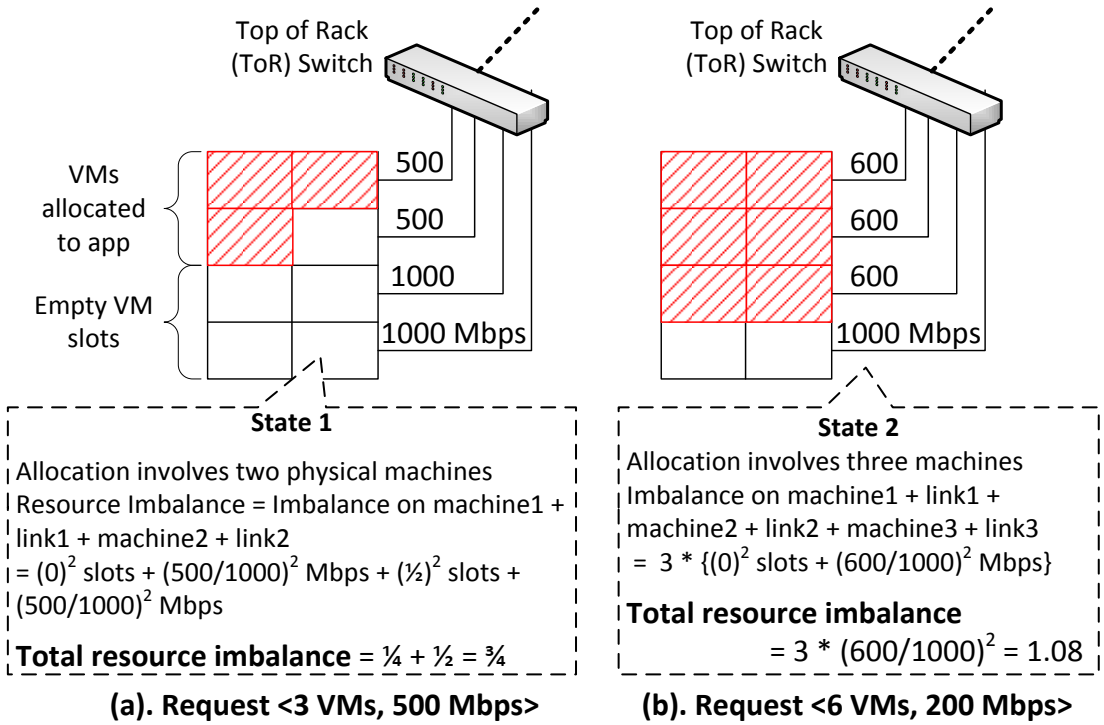


Figure 4.1: Selecting amongst two feasible resource tuples. Each physical machine has 2 VM slots and an outbound link of capacity 1000 Mbps. Each link is annotated with its residual bandwidth.

an average network bandwidth of 500 Mbps (*state-2*). We note that any subsequent application request that can be accommodated by the cluster in state-2 can also be accommodated in state-1. However, the reverse is not true. For instance, a future application request requiring the tuple $<3, 400>$ can be allocated in state-1 but not state-2. Hence, the first tuple is more desirable for the cluster and is the one chosen by the resource imbalance metric.

4.3 Offline planning

In this section, we first describe the design considerations for the offline planner component in Corral (Section 4.3.1), and our approach to formalizing and solving the planning problem (Section 4.3.2). We then present a formal description of the planning problem (Section 4.3.3), our heuristics to solve the problem (Section 4.3.4), and formulate an LP relaxation to evaluate their performance (Section 4.3.5).

4.3.1 Design considerations

Objective. The goal of the offline planner is to provide guidelines (or hints) to the cluster scheduler for placing data and compute tasks. The objective is to optimize the network locality of a job, and thus, improve the overall performance.

Scenarios. Based on the different cases that arise in practice, we consider two scenarios. In the *batch* scenario, we run a batch of jobs that are all submitted to the cluster at the same time (e.g., a collection of log analysis jobs). Our goal is to minimize their *makespan*, i.e., the time to finish the execution of all the jobs in the batch. In the *online* scenario, jobs arrive over a longer period of time with known arrival times. In this scenario, our goal is to minimize the average job completion time.

Challenges. Designing the planner raises three important questions: First, at what granularity should the hints be provided? For example, one option is to provide a target machine for each task of the job. Second, how to formalize the planning problem to make it tractable at the scale of current data analytics systems? Finally, as the planner needs to choose between different data and task placement options, how do we estimate the job latency of a particular configuration?

For the purposes of formalizing a tractable offline planning problem, we make several simplifying assumptions which we discuss next. We note that these assumptions *apply only to the offline planner,* and not to the actual job execution on the cluster.

4.3.2 Solution approach

Planning at granularity of jobs and racks. The solution to the planning problem can be specified at different granularities. At one extreme, it can prescribe which vertex of a job should run on which machine in the cluster. However, generating such a solution is non-trivial. First, while the job input sizes can be predicted with small error, the input sizes of individual vertices depend on how the data is partitioned across vertices in a stage and thus, much less predictable. Second, the number of vertices can be several orders of magnitude higher than the number of jobs, making the problem practically intractable.

Instead of planning at the level of vertices, one can plan at a stage-level, i.e., specify which rack(s) each stage in a job should use. Complex, DAG-structured jobs could potentially benefit from stage-level planning; e.g., two parallel shuffles in a DAG could run in two separate racks, both benefiting from locality. However, after examining a

large number of production jobs in our clusters, we found very few DAG jobs where such stage-level planning provides locality benefits.

Therefore, to improve scalability and robustness of the offline plan, we pose the planning problem at the granularity of racks and jobs. Further, most production clusters have full bisection bandwidth within a rack and oversubscribed links from the racks to the core [18, 35, 54]. Planning at the granularity of racks also allows us to assume that all tasks in a rack can communicate at NIC speeds, which in turn simplifies the modeling of job latency.

Planning as malleable job scheduling. We formulate the planning problem as a malleable job scheduling problem [43–45]. A *malleable* job is a job whose latency depends on the amount of resources allocated to it. While each job typically has a fixed requirement for the total number of compute slots, the planner's decision on how many racks the job will execute can affect its latency.

To illustrate the dependency of job latency on the number of racks assigned to it, consider a map-reduce shuffle operation with a total of S bytes spread over r racks. Each rack has internal bandwidth of B, and racks are connected with oversubscription ratio of V; assume that network is the bottleneck resource affecting latency. Then when $r = 1$, almost all of S bytes have to be transferred across machines in the rack, with total latency of S/B. For $r > 1$, assuming symmetry for simplicity, each rack sends $(r-1)/r$ fraction of its data to other racks, so the whole shuffle has to transfer $S(r-1)/r$ bytes using aggregate bandwidth of rB/V, resulting in approximate latency of $\frac{(r-1)SV}{r^2B}$ which approaches $\frac{V}{r}\frac{S}{B}$ for large r; note that the latency reduces with r for this simple example. More comprehensive latency models are described in Section 3.1.

By using malleable job scheduling, we can tap into previous work in this area to design efficient algorithms for determining which, how many, and at what priority order would racks be assigned to jobs.

Characterizing job latency using simple response functions. Using the above intuition, we model job latency via latency-response functions $L_j(r)$, where j is the job index and r is the number of racks allocated to j. In particular, we model job latency as depending on the number of racks allocated, and also on fixed job characteristics (such as the amount of data transferred between vertices, amount of CPU resources required, and the maximum parallelism of the job). Using these job characteristics, which can be estimated from earlier runs of the job, we derive simple analytical models to compute job latency (Section 3.1).

The actual job latencies obviously depend on additional runtime factors such as

failures, outliers, and other jobs which run simultaneously on the same rack(s). However, as our latency approximation is used only for offline planning (and these factors affect all jobs), we can trade off accurate (absolute) latency values for simpler and practical planning algorithms. As it will become evident from our evaluation, using approximate latency models suffices for significant latency improvements.

4.3.3 Problem formulation

Given the design choices above, Corral's offline planning problem is formulated as follows. A set of jobs $\mathcal{J}$ (of size J) has to be scheduled on a cluster of R racks. Each job j is characterized by a *latency-response function* $L_j : [1, R] \rightarrow \Re^+$, which gives the (expected) job completion time as a function of the number of racks assigned to the job. Our model assumes that once a subset of racks is allocated to a job, it is used by the job until completion. That is, we do not allow preemption or a change in the allocation throughout the execution of the job. This assumption simplifies our problem formulation significantly. To ensure work conservation during job execution, Corral's cluster scheduler does not enforce these constraints. Our evaluation results (Section 4.5) show that even with this deviation from the assumptions, Corral significantly outperforms existing schedulers.

In the batch scenario, the goal is to minimize the makespan, i.e., the time it takes to complete all jobs. In the online scenario, each job has its own arrival time. The goal now is to minimize the average completion time, i.e., the average time from the arrival of a job until its completion. Both problems are NP-hard [44, 45]. Hence, we design efficient heuristics that run in low polynomial time.

4.3.4 Planning heuristics

Solving the planning problem consists of determining (a) the amount of resources (number of racks) to be allocated to a job, and (b) where in the cluster these resources have to be allocated. To address each of these sub-problems and use ideas from existing techniques (e.g., LIST scheduling [85]), we decouple the planning problem into two phases — the *provisioning* phase and the *prioritization* phase. In the provisioning phase, for each job j, we determine r_j, the number of racks allocated to the job. In the prioritization phase, given r_j for all jobs, we determine R_j, the specific subset of racks for j, and T_j, the time when j would start execution. We use T_j to

Input: Set of jobs $\mathcal{J}$ of size J; $\forall j \in \mathcal{J}$, r_j, the number of racks to be assigned to job j for $L_j(r_j)$ time units
and A_j, the arrival time of job j (0 in batch case).
Output: $\forall j \in \mathcal{J}$, R_j, the set of racks allocated to
job j and T_j, the start time of job j.
Initialization: *Sort* and re-index jobs according to the scenario (batch or online).
$j := 0$
$F_i := 0$ for all racks $i = 1 \ldots R$
while $j < J$ **do**
  $R_j :=$set of r_j racks with smallest F_i
  $T_j := \max\{\max_{i \in R_j} F_i, A_j\}$
  for $i \in R_j$, $F_i := T_j + L_j(r_j)$
  $j := j + 1$
end while

Figure 4.2: Prioritization phase.

determine the priority ordering of jobs.

Provisioning phase. Initially, we set $r_j = 1$ for each job. This represents a schedule where each job runs on a single rack. In each iteration of this phase, we find the job j which is allocated less than R racks and has the longest execution time (according to the current r_j allocations), and increase its allocation by one rack. When a job is already allocated R racks it cannot receive additional racks. We proceed iteratively until all jobs reach $r_j = R$.

Intuitively, by spreading the longest job across more racks, we shorten the job that is "sticking out" the most. Note that if the latency of the longest job increases when its allocation is increased by one rack, it will continue to be the longest and thus, its allocation will be increased again in the next iteration. For the latency response curves we observed it practice, we found that the latency of the longest job eventually decreases.

As each job in $\mathcal{J}$ can be allocated any number of racks between 1 and R, a total of R^J different allocations exist for the provisioning phase. The above heuristic is designed to explore a plausible polynomial-size subset of candidate allocations ($J \cdot R$), which can be evaluated within practical time constraints. For each such allocation, we run the prioritization phase described below and pick the allocation (and respective schedule) which yields the lowest value for the relevant objective (makespan for batch scenario, average completion time for online scenario).

We note that this heuristic is similar to the one used in [43] for scheduling malleable

jobs with the objective of makespan minimization. [43] terminates the heuristic when $\sum_{j|r_j>1} r_j = R$, and obtains an approximation ratio of roughly 2 on the makespan, by using LIST scheduling [85] on top of the provisioning heuristic. However, as each iteration is fast, we allow ourselves to run the heuristic for more iterations compared to [43], until reaching $r_j = R$ for every job j. This allows us to explore more options, and to obtain adequate results (including for the objective of average completion time, for which [43] does not provide guarantees).

Prioritization phase. If all jobs are constrained to run on a single rack, the longest processing time first (LPT) algorithm [85] is a well-known scheduling algorithm one could use to minimize makespan. However, jobs can run on multiple racks in Corral. We extend LPT to account for this case (pseudo-code in Figure 4.2).

In the batch scenario, we first sort jobs in decreasing order of number of racks allocated to them (r_j), i.e., *widest-job first*. Then, to break ties, we sort them in decreasing order of their processing times (similar to LPT). The widest-job first order helps us avoid "holes" in the schedule; for example, a job allocated R racks will not have to wait for a job allocated just one rack to complete, which would result in wasted resources. We then iterate over the jobs in this sorted order, assigning the required number of racks. We keep track of F_i, the time when rack i finishes running previously scheduled jobs. For each job j, we allocate the required set of racks by selecting the first r_j racks that are available (based on F_i). We then update F_i for the selected racks based on the start time of the job and its duration.

For the online scenario, we sort jobs in ascending order of their arrival time; in case of ties, we apply the sorting criteria of the batch case in the same order described above, and then use the algorithm described in Figure 4.2.

Complexity. The complexity of the prioritization phase is $O(JR)$, because for each job we make a pass over all R racks to determine the first r_j racks that become available. The provisioning phase has JR iterations, hence the overall complexity of our heuristic is $O(J^2R^2)$. The complexity of calculating the latency-response functions is linear in R and thus does not increase the overall complexity[3].

In terms of actual running time, our heuristic is highly scalable as shown in Figure 4.3. Running our heuristic on a single desktop machine with 6 cores and 24GB RAM, we found that it requires around 55 seconds to generate the schedule for 500

[3]With DAGs, we find critical path via an efficient shortest path algorithm; using BFS, this adds an $O(V + E)$ to the complexity, where V is the number of stages and E is the number of edges in the DAG. However, because the DAGs are typically small compared to number of racks and jobs, this does not change the complexity of our heuristic.

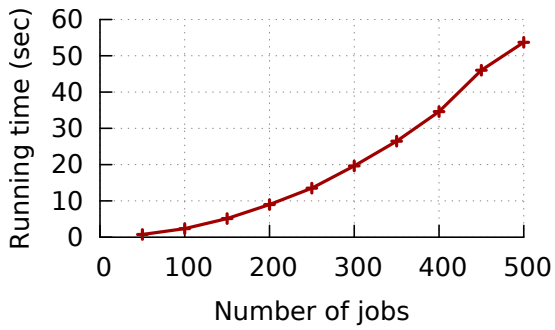


Figure 4.3: Running time of the offline planner heuristic in Corral for a 4000 machine cluster with varying number of jobs.

jobs on a 4000 node cluster with 100 racks (40 machines per rack). As the planner runs offline, this results in minimal overhead to the cluster scheduler.

Accounting for data (im)balance. Using the latency-response functions as described above, Corral would directly optimize for latency-related metrics, but would ignore how the input data is spread across different racks. Consequently, it is possible that a large fraction of the input data would be placed in a single rack, leading to high data imbalance and excess delays (e.g., in reading the input data). To address this issue and achieve better data distribution, we add a penalty term to the latency-response function, given by $\alpha \cdot D_j^I/r$, where D_j^I/r is the amount of input data of job j in a single rack and α is a tradeoff coefficient. Accordingly, the modified latency-response function is given by $L'_j(r) = L_j(r) + \alpha \cdot D_j^I/r$, where $L_j(r)$ is the original response function determined using MRCute (Section 3.1).

In our experiments (Section 4.5), we set α to be the inverse of the bandwidth between an individual rack and the core network. The intuition here is to have the penalty term serve as a proxy for the time taken to upload the input data of a job to a rack. Increasing α favors schedules with better data balance. We note that in practice, we supplement this approach by greedily placing the last two data replicas on the least loaded rack. The combination of these approaches leads to a fairly balanced input data distribution (Section 4.5).

4.3.5 LP relaxation

To estimate the quality of our heuristics for the planning problem, we formulate a related integer linear program (ILP). The ILP would provide a lower bound on the

makespan and the average completion time for *any* algorithm. However, as the ILP is computationally expensive to solve in practice, we relax it to a linear program (LP), whose solution is still a lower bound to our problem. Thus, if the solution from our heuristics is close to that of the LP, it is guaranteed to be close to optimal. We emphasize that the LP relaxation is not used by Corral, but only serves as a benchmark for the solutions we developed in Section 4.3.4.

We first describe the Integer Linear Program (ILP). Let T be the makespan of the (unknown to us) optimal solution to the planning problem. For every job j and every number of racks $r \in \{1, \ldots, R\}$, we introduce a variable $x_{jr} \in \{0, 1\}$. In the ILP, x_{jr} equals 1 if job j is assigned r racks, and 0 otherwise. Relaxing the integrality constraint, we obtain the following LP.

$$\text{Minimize}_{\{x_{jr}\}} \quad T \qquad (LP - Batch) \tag{4.1}$$

$$\text{Subject to} \quad \sum_r x_{jr} = 1, \quad \forall j \tag{4.2}$$

$$T \geq \sum_r x_{jr} L_j(r), \quad \forall j \tag{4.3}$$

$$TR \geq \sum_{j,r} x_{jr} L_j(r) \cdot r, \tag{4.4}$$

$$x_{jr} \in [0, 1], \forall j, \forall r \tag{4.5}$$

The constraint (4.2) ensures that all jobs are completed. In the integral solution corresponding to a feasible schedule, for each j exactly one x_{jr} equals 1, so the constraint (4.2) is satisfied. The constraints (4.3) and (4.4) give a lower bound on the makespan. Constraint (4.3) asserts that the makespan is at least as large as the completion time of every job j (as in the integral solution corresponding to a feasible schedule, the right hand side of (4.3) exactly equals the running time of job j). Constraint (4.4) is a capacity constraint. It indicates that the capacity used by the LP schedule (the right hand side of inequality) is at most the available capacity (the left hand side).

We emphasize that LP-Batch is a *relaxation* of our original planning problem (Section 4.3.3) as it does not give an actual schedule (e.g., does not specify the time when a job should start) but only provides the number of racks a job has to be allocated. Still, any feasible schedule (particularly, an optimal schedule) should satisfy the constraints of the LP and thus, the cost (i.e, makespan) returned by the LP is a lower bound on the cost of any schedule.

In the online scenario, we are interested in minimizing the average completion time. Accordingly, we formulate the following LP.

$$\text{Minimize}_j \quad \frac{1}{J}\sum_j T_j \qquad (LP - Online) \tag{4.6}$$

$$\text{Subject to} \quad \sum_r x_{jr} = 1, \quad \forall j \tag{4.7}$$

$$T_j \geq \sum_r (x_{jr} L_j(r) + d_{jr}), \quad \forall j \tag{4.8}$$

$$\sum_j T_j \cdot R \geq \sum_{j,r} (x_{jr} L_j(r) + d_{jr}) \cdot r, \tag{4.9}$$

$$x_{jr} \in [0,1], \forall j, \forall r \tag{4.10}$$

where T_j is the completion time of job j (time elapsed since its arrival till its completion), $d_{jr} > 0$ is the delay in scheduling job j on r racks, relative to its arrival and x_{jr} is as above. Constraint (4.7) ensures that all jobs are completed with an unique rack allocation. Constraint (4.8) ensures that the completion time of a job is at least as large as its execution time and finally, Constraint (4.9) is the capacity constraint as before.

Near-optimality of Corral's heuristics. We implemented the above LP relaxations along with the heuristics. Our experiments (setup described in Section 4.5) show that Corral's heuristic for the batch (online) scenario finds a schedule with resulting makespan (mean completion time) within 3% (15%) of the LP solution.

These bounds represent the maximal gap between the performance of our heuristics and the optimal solution to the planning problem, as defined under the assumptions of Section 4.3.2. We note that these bounds might not hold for other formalizations of the problem (e.g., allowing allocation at machine granularity). Nevertheless, as discussed previously, these assumptions result in a practical scheduling framework with performance guarantees.

4.4 Evaluation of resource selector

In this section, we evaluate the potential gains resulting from smart resource selection in Corral for applications with completion time requirements. Our evaluation

combines simulations and a testbed deployment. Specifically:

- We use large scale simulations to evaluate the benefits of Corral. Capitalizing on resource malleability significantly improves datacenter goodput (Section 4.4.2).
- We deploy and benchmark our prototype on a 26-node Hadoop cluster. We further use this deployment to cross-validate our simulation results (Section 4.4.3).
- We further show that exploiting malleability in the time domain allows Corral to finish jobs earlier than required by dedicating idle resources to them, with at least 50% improvement in the median job completion time (Section 4.4.4).

4.4.1 Simulation setup

We developed a simulator to evaluate the resource selector in Corral at scale. The simulator coarsely models a multi-tenant datacenter. It uses a three-level tree topology with no path diversity. Racks of 40 machines with one 1 Gbps link each and a Top-of-Rack switch are connected to an aggregation switch. The aggregation switches, in turn, are connected to the datacenter core switch. The results in this section involve a datacenter with 16,000 physical machines and 4 VMs per machine, resulting in a total of 64,000 VMs. The network has an oversubscription of 10:1 and we vary this later. Each VM has a local disk. While high-end SSDs can offer bandwidth in excess of 200 MB/s for even random access patterns [86], we conservatively use a disk I/O bandwidth of 125 MB/s = 1 Gbps such that it can saturate the network interface.

MapReduce jobs. We use a simple model for MapReduce jobs. The program $\mathcal{P}$ associated with a job is characterized by four parameters– the rate at which data can be processed by the map and reduce function when there are no I/O bottlenecks ($B^{\mathcal{P}}_{map}$, $B^{\mathcal{P}}_{reduce}$) and the selectivity of these functions (S_{map}, S_{reduce}). Given the input size, the selectivity parameters are used to determine the size of the intermediate and output data generated by the job. Note that an I/O intensive job like `Sort` can process data fast and has high values for $B^{\mathcal{P}}_{map}$ and $B^{\mathcal{P}}_{reduce}$. To capture the entire spectrum of MapReduce jobs, we choose these parameters from an exponential distribution with a mean of 500 Mbps. We also experiment with other mean values.

Application Requests. Each application request consists of a MapReduce job, input size and a completion time goal. This information is fed to the analytical model to determine the candidate resource tuples for the job. From these candidate tuples,

one tuple $<N, B>$ is chosen based on the selection strategies described below. The corresponding resources, N VMs with B Mbps of network bandwidth, are allocated using the allocation algorithm in [42]. If the request cannot be allocated because of insufficient resources, it is rejected.

We simulate all three stages of MapReduce jobs. We do not model the disk and CPU operations. Instead, the duration of the map and the reduce stage is simply calculated a priori by invoking the MRCute analytical model. As part of the shuffle stage, we simulate all-to-all traffic matrix with N^2 network flows between the N VMs allocated to the application. Given the bandwidth between VMs, we use max-min fairness to calculate the rate achieved by each flow. The shuffle stage completes when all flows complete.

Resource selection strategies. We evaluate three strategies to select a resource tuple.

(1). *Baseline.* This strategy does not take advantage of a job's resource malleability. Instead, one of the candidate tuples is designated as the baseline tuple $<N_{base}, B_{base}>$. The job is executed using this baseline resource tuple.

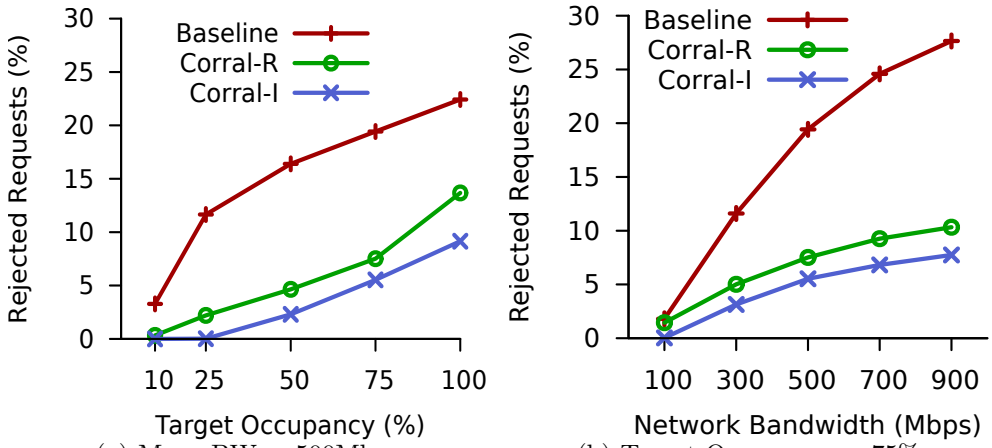
(2). *Corral-R* (random selection). A tuple is randomly selected from the list of candidates, and if it can be allocated in the datacenter, it is chosen. Otherwise the process is repeated. This strategy takes advantage of resource malleability to accommodate requests that otherwise would have been rejected. However, it does not account for the impact that a tuple bears on the provider.

(3). *Corral-I* (imbalance-based selection). For each tuple, we determine how it would be allocated, and calculate the resulting utilization vector and resource imbalance. The tuple with the lowest resource imbalance is chosen.

Workload. To model the operation of datacenters, we simulate application requests arriving over time. By varying the application arrival rate, we vary the *target VM occupancy* for the datacenter. Assuming Poisson application arrivals with a mean arrival rate of λ, the target occupancy on a datacenter with M total VMs is $\frac{\lambda NT}{M}$, where T is the mean completion time for the requests and N is the mean number of requested VMs in the Baseline scenario.

4.4.2 Selection benefits

We simulate the arrival and execution of 15,000 application requests. The desired completion time for each request is chosen such that the number of compute nodes



(a) Mean BW = 500Mbps (b) Target Occupancy = 75%

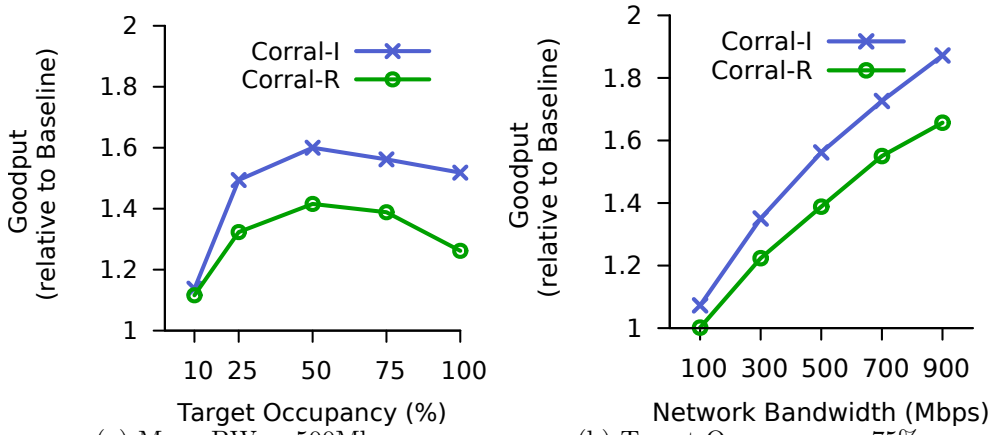
Figure 4.4: Percentage of rejected requests, varying mean bandwidth and target occupancy.

(N_{base}) and network bandwidth (B_{base}) required in the Baseline scenario is exponentially distributed. The mean value for N_{base} is 50, which is consistent with the mean number of VMs that application request in datacenters [87].

Workloads and metrics. Two primary variables are used in the following experiments to capture different workloads. First, we vary the *mean bandwidth* required by applications (B_{base}). This reflects applications having varying completion time requirements. Second, we vary the *target occupancy* to control the application request arrival rate.

From a cluster provider's perspective, we look at two metrics to quantify the potential benefits of resource selection. First is the fraction of requests that are rejected. However, this, by itself, does not represent the full picture as individual requests are of different sizes, i.e., each request processes a different amount of data. To capture this, we also look at the sum of input data consumed across all requests. This represents the total useful work in the datacenter, and we define it as the *datacenter goodput*.

Impact of varying mean bandwidth and target occupancy. Figure 4.4a plots the percentage of rejected requests with varying target occupancy. For all selection strategies, the rejection ratio increases with increasing target occupancy. This is because requests start arriving faster and hence, a greater fraction have to be rejected. The figure shows that, depending on the occupancy, Corral-I results in 3-14% fewer requests being rejected. Corral-R rejects around 2-5% more requests than Corral-I. However, as we explain below, the actual benefit of the imbalance-based selection is



(a) Mean BW = 500Mbps (b) Target Occupancy = 75%

Figure 4.5: Datacenter goodput with varying mean bandwidth and varying target occupancy.

larger.

To put this in perspective, operators like Amazon EC2 target an average occupancy of 70-80% [88]. Figure 4.4b plots the rejected requests for a target occupancy of 75%. The figure shows that the difference between the fraction of requests rejected by both Corral strategies as compared to Baseline increases with increasing mean bandwidth. Increasing the bandwidth required by the job implies tighter completion time requirements which, in turn, means there are greater gains to be had from selecting the appropriate resource combination. At mean bandwidth of 900 Mbps, Corral-I rejects 19.9% fewer requests than Baseline.

Figure 4.5 shows the datacenter goodput for the Corral selection strategies relative to Baseline. Depending on the occupancy and bandwidth, *Corral-I improves the goodput by 7-87% over Baseline, while Corral-R provides improvements of 0-66%.* As an example, at typical occupancy of 75% and a mean bandwidth of 500 Mbps, Corral-I and Corral-R offer 56% and 39% benefits relative to Baseline respectively. Note that the gains with Corral-R show how resource malleability can be used to accommodate application requests that would otherwise have been rejected. The further gains with Corral-I represent the benefits to be had by smartly selecting the resources to use.

In Figure 4.5a, the relative improvement in goodput with Corral strategies first increases with target occupancy and then declines. This is because, at both low and high occupancy, there is not as much room for improvement. At low occupancy, requests arrive far apart in time and most can also be accepted by Baseline. At high occupancy, the arrival rate is high and the datacenter is heavily utilized. In

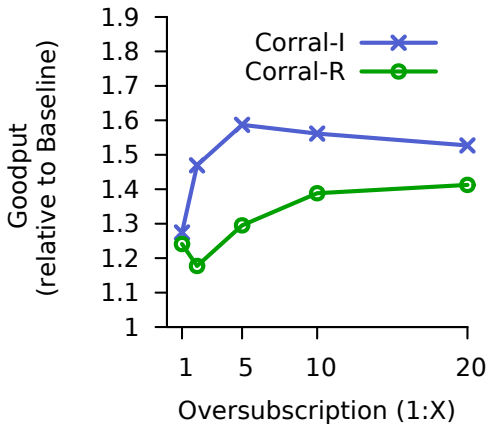


Figure 4.6: Varying network oversubscription (occupancy is 75%).

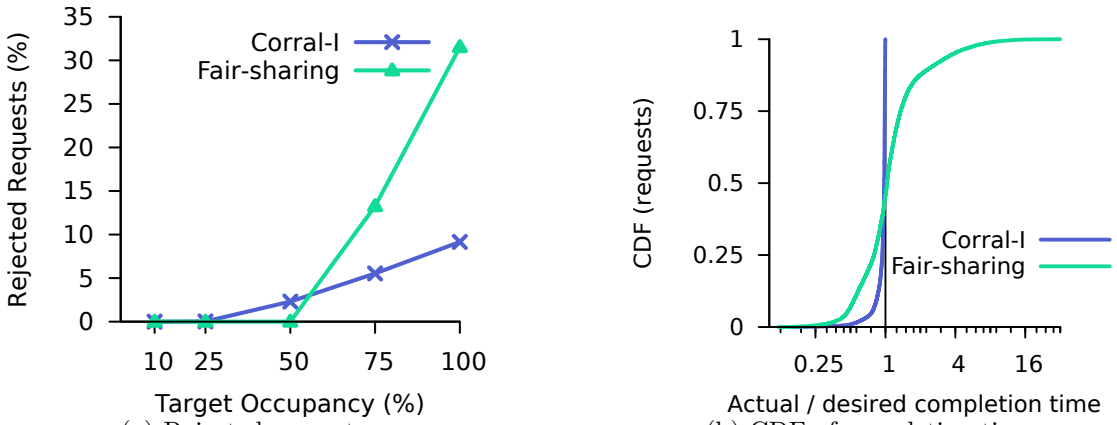
Figure 4.5b, the gains increase with increasing bandwidth. As explained above, this results from shrinking completion time requirements which allow Corral strategies to accept more requests as compared to Baseline. Further, Corral is able to accept bigger requests resulting in even higher relative gains.

Impact of simulation parameters. We also determined the impact of other simulation parameters on Corral performance and the results stay qualitatively the same. Here, we show the results of varying oversubscription, and discuss the impact of varying the mean disk bandwidth and other parameters.

Figure 4.6 shows the relative goodput with varying network oversubscription. Even in a network with no oversubscription, e.g., [89], Corral-I is able to accept 10% more requests (not shown) and improves the goodput by 27% relative to Baseline. Further, the relative improvement with Corral increases with increasing oversubscription before flattening out. This is because the physical network becomes more constrained and Corral can benefit by reducing the network requirements of applications while increasing their VMs.

We also ran experiments using different values of the disk bandwidth. As expected, low values of the disk bandwidth reduce the benefits of Corral-I. When the disk bandwidth is extremely low (250 Mbps), increasing the network bandwidth beyond this value does not improve performance. Thus, there are very few candidate resource tuples and the gains with Corral are small (2% over Baseline). However, as the disk bandwidth improves, there are more candidate tuples to choose from and the performance improves.

Finally, we varied the mean task bandwidth (map and reduce) and also the datacenter size (up to a maximum of 32,000 servers and 128,000 VMs) and the results are



(a) Rejected requests.

(b) CDF of completion time.

Figure 4.7: Comparing against today's setup (Mean BW is 500 Mbps).

similar to the trends observed in Figure 4.4 and 4.5.

Comparison with today's setup. Today, cluster providers do not offer any bandwidth guarantees to applications. VMs are allocated using a greedy locality-aware strategy and bandwidth is fair-shared across applications using TCP. In Figure 4.7a, we compare the performance of Corral-I against a setup representative of today's scenario, which we denote as *Fair-sharing.* For low values of occupancy, Fair-sharing achieves a slight better performance than Corral-I. The reason is that Corral-I reserves the network bandwidth throughout the entire duration of the application request. This also includes the map and reduce stage, which are typically characterized by little or no network activity. In contrast, in Fair-sharing, the network bandwidth is not exclusively assigned to applications and, hence, due to greater multiplexing, it achieves a higher network utilization. Yet, for high values of occupancy, which are typical of today's datacenters [88], rejected requests significantly increase. This is due to the high congestion incurred in the core of the network, caused by the sub-optimal placement of VMs and corresponding flows.

The main drawback of Fair-sharing, however, is highlighted in Figure 4.7b, which shows that the completion time is extended for at least 50% of the jobs and for 12% of the jobs the actual completion time is at least twice the desired completion time.

Mitigating outliers with slack. The results above assume perfect prediction. As our prediction does not account for all possible outliers, the predicted completion time for a job can be off which, in turn, would cause the job to be *late*. To counter this, Corral relies on slack. To evaluate how much slack is needed in practice, we use the outlier distribution from Microsoft Bing's production clusters (reported in [32]) to

introduce outlier tasks in our experiments. Such tasks extend jobs past the predicted completion time. Given this, we measured the impact of varying slack on rejected requests (relevant for the provider) and late jobs (relevant for the applications).

As slack increases, it is harder to accommodate requests as they need to be finished sooner and thus, require more resources. For slack less than 50%, we found that Corral-I rejects fewer requests than an "Oracle" that can do perfect prediction but does not capitalize on resource selection. In effect, smart resource selection allows us to offset prediction inaccuracies. The same trends hold for goodput too.

We further found that that with no slack, approximately 90% of jobs are late, a consequence of 90% of jobs having at least one outlier. As slack increases, the late requests decrease almost linearly, and with a slack of 50%, no requests are late. Overall, the slack parameter gives the cluster provider a knob to satisfy application goals even in the presence of outliers at the expense of greater rejections. For instance, the provider may use historical job information to determine the amount of slack to provision so as to bound the probability of breached SLAs.

4.4.3 Deployment

We complement our simulation analysis with experiments on a small-scale Hadoop cluster using a prototype implementation of the resource selector of Corral. We deployed the resource selector on 26 Emulab servers, using the same hardware setup described in Section 3.3 and the Cloudera distribution of Hadoop. We used the Linux Traffic Control API on individual servers to enforce the rate limits.

We configured one of the testbed servers as the cluster head node and the rest of the servers as compute nodes. The head node is responsible for generating application requests and allocating them on the compute nodes. The workload consisted of 100 `Sort` job requests with an exponentially generated input data size (the mean value was 5.7 GB). As in the previous experiments, we used a target occupancy of 75%, mean B_{base} of 500 Mbps and mean N_{base} equal to 9.

The goal of these experiments is threefold. First, we quantify the benefits of resource selection in Corral. Second, we cross-validate the accuracy of our simulator. Finally, we verify the scalability of our implementation to allocate requests in a much bigger network.

Benefits. Figure 4.8 shows that Corral-I is able to accept 11.43% more `Sort` jobs than Baseline (i.e., 8 extra jobs) and increases goodput by 15.47%. This is despite

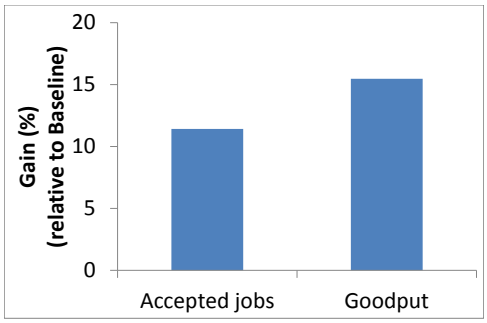


Figure 4.8: On testbed, Corral-I can accept more Hadoop jobs and increase goodput relative to Baseline.

limited opportunities– the deployment is small and the disk bandwidth available is low. Also, with Corral-I, less than 2% of the accepted requests completed later than expected. Note, however, that in these experiments we did not add any slack, which would have enabled all requests to complete on time.

Cross-validation. To validate the accuracy of our simulator, we replicated the same workload in the simulator, i.e. the same stream of jobs arrive in the simulator as on the testbed. Across all cases, the maximum difference in the number of accepted requests and goodput is approximately 5.12%, and 8.43% respectively. This cross-validation gives us confidence in our simulation results.

Scalability analysis. To evaluate the performance of our prototype at scale, we measured the time to allocate application requests on a datacenter with 128,000 VMs. This includes both the time to generate the set of candidate resource tuples using the analytical model (Section 3.1) and to select the resources (Section 4.2). This does not include the job profiling time. Over 10,000 requests, the median allocation time is 950.17 ms with a 99th percentile of 983.29 ms. Note that this only needs to be run when an application is admitted, and, hence, the overhead introduced is negligible.

4.4.4 Beyond two resources: time malleability

Our evaluation so far considered application malleability along two dimensions, i.e., N and B. However, Corral allows the cluster provider to exploit malleability across other resources and across time. Here, we briefly explore this opportunity. The provider can devote additional (idle) resources to applications, so that they complete before the desired time. In this way, the resources used by the job can be reclaimed earlier, and a larger number of requests can potentially be accommodated in the

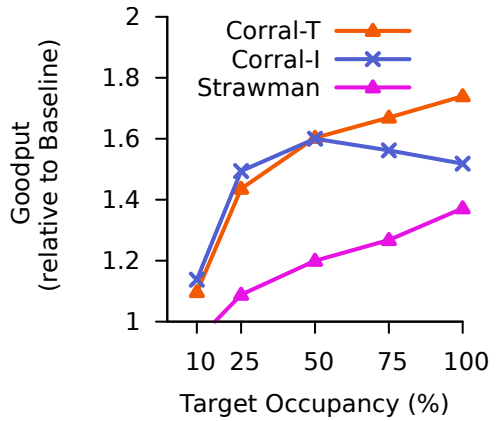


Figure 4.9: Datacenter goodput when exploiting time malleability.

future. Applications would benefit too since they would experience shorter than desired completion times.

We denote this further selection strategy as *Corral-T* . The key difference between Corral-T and Corral-I is that the latter only considers tuples $< N, B >$ that yield a completion time $T = T_{desired}$ while Corral-T also consider tuples where $T < T_{desired}$. Among these, Corral-T selects the tuple that minimizes the product of the tuple resource imbalance and T. Figure 4.9 shows that, at high values of the target occupancy, exploiting time flexibility significantly improves the ability of the provider to accommodate more requests and, hence, the goodput increases. Corral-T is also beneficial for applications as *the median completion time is reduced by more than 50%, and for 20% of the jobs the completion time is reduced by 80%.* In Figure 4.9, we also consider a naive approach, *Strawman* , that always selects the tuple that yields the lowest completion time (irrespective of the resource imbalance). Such a strategy performs poorly as it tends to over-provision the resources for the early requests, which reduces the ability to accommodate future ones.

4.5 Evaluation of offline planner

In this section, we present the results of evaluating the planning component of Corral on a 210 machine cluster. For this purpose, we use a variety of workloads drawn from production traces. Our main results are as follows.

- Compared to Yarn's capacity scheduler, Corral achieves 10-33% reduction in makespan and 26-36% reduction in average job completion time for workloads

consisting of MapReduce jobs (Section 4.5.3). For Hive queries derived from the TPC-H benchmark [48], Corral improves completion times by 21% on average (Section 4.5.4).

- When a workload consists of both recurring and ad hoc jobs, using Corral to schedule the recurring jobs improves the completion times of the recurring and ad hoc jobs by 33% and 20% (respectively), on average (Section 4.5.5). Further, the benefits with Corral hold as we vary the fraction of ad hoc jobs in the mixed workload.

- Corral's improvements increase significantly as the cluster load and network utilization increase. Further, its improvements are robust to errors in predicted job characteristics (Section 4.5.6).

- Using large-scale simulations, we show that the benefits from flow-level schedulers such as Varys [18] improve significantly when used in combination with Corral and that Corral's benefits are orthogonal to those of using Varys alone.

4.5.1 Implementation

We implemented Corral on top of the Apache Yarn framework (Hadoop 2.4) [46] and HDFS. Corral's offline planner determines the set of racks where (a) the input data of a job has to be stored and (b) its tasks have to be executed. To ensure that these rack preferences are respected (as described in Section 4.1), we made the following changes to the different components in Yarn. The changes involve about 300 lines of Java code.

Data placement policy. We modified HDFS's `create()` API to include a set of $\langle \text{rack}, \text{number of replicas} \rangle$ tuples, which allows Corral to specify the racks where different replicas of a file's data chunk are to be placed. These specifications are passed on to the block placement policy in HDFS, which was modified to ensure that at least one replica of the data chunk is placed on a machine which belongs to the rack specified.

Task placement policy. Every job in Yarn uses an Application Manager (AM) to request slots from a centralized Resource Manager (RM). The AM can also specify preferences for locations (e.g., specific machine or rack) where it would like slots to be allocated in the cluster. By default, the MapReduce AM in Yarn specifies location

preferences for map tasks only and not for the reducers. For Corral, we modified the Yarn MapReduce AM to specify locality preferences for *all* tasks of a job.

As the subset of racks where a job needs to be scheduled is determined by Corral's offline planner, we pass this to the AM using a new configuration parameter. This is further passed to the RM as a location preference. The RM makes every effort to respect this locality preference while allocating slots to the job. However, in the event that a majority of the machines in the racks preferred by a job are unreachable, the RM will ignore the locality guidelines and allocate slots on the available nodes in the cluster.

4.5.2 Methodology

Cluster setup. We deployed our implementation of Corral in Yarn/HDFS on a 210 machine cluster, organized into 7 racks with 30 machines per rack. Each machine has 32 cores, 10 Gbps NICs, and runs CentOS 6.4. The racks are connected in a folded CLOS topology at 5:1 oversubscription, i.e., each rack has a 60 Gbps connection to the core. To match network conditions in production clusters, we emulate background traffic, accounting for up to 50% of the core bandwidth usage [4, 35].

Workloads. We first evaluate Corral using the scenario where all jobs in the workload are assumed to be recurring (or have predictable characteristics). We then consider workloads with both recurring and ad hoc jobs. We use jobs from the following workloads for our evaluation.

(a) W1: Starting from the Quantcast workloads [90], we constructed this workload to incorporate a wider range of job types, by varying the job size, and task selectivities (i.e., input to output size ratio). The job size is chosen from small ($\leq$ 50 tasks), medium ($\leq$ 500 tasks) and large ($\geq$ 1000 tasks). The selectivities are chosen between 4:1 and 1:4.

(b) W2: This workload is derived from the SWIM Yahoo workloads [47] and consists of 400 jobs.

(c) W3: We have chosen 200 jobs, randomly, from a 24 hour trace collected from production data analytics clusters at Microsoft Cosmos and constructed this workload. Some characteristics of this workload are given in Table 4.1.

(d) TPC-H: We run queries from the TPC-H benchmark using Hive [9] to evaluate the performance of Corral for general DAG-structured workloads.

Baselines. We compare Corral against three baselines.

	50%-tile	95%-tile
Number of tasks	180	2,060
Input Data Size (GB)	7.1	162.3
Intermediate data size (GB)	6	71.5

Table 4.1: Characteristics of workload W3.

(a) the capacity scheduler in Yarn [49] (referred to as Yarn-CS from now on). The capacity scheduler uses techniques like delay scheduling [74] to achieve locality for map tasks but does not plan for data placement. Comparison with Yarn-CS allows us to show the benefits of achieving better locality for all stages of a job using Corral.
(b) ShuffleWatcher [76], which schedules each job in a subset of racks to reduce the amount of cross-rack data transferred by them. ShuffleWatcher does not place the input data of the job in these racks and as a result, most maps end up reading their input across the core network. It also fails to account for contention between jobs and schedules them independently from each other. Comparison with ShuffleWatcher allows us to show the benefits of careful planning, and joint data and compute placement in Corral.
(c) LocalShuffle, which uses the task placement of Corral but the data placement policy of HDFS. The comparison of Corral with LocalShuffle allows us to quantify the benefits of proper placement of input data. We note that, unlike ShuffleWatcher, LocalShuffle schedules jobs using the same offline planning phase as Corral.

For all the above baselines, the input data is placed using HDFS's default (random) placement. When using Corral, we run its offline planner taking data balance into account (Section 4.3.4). Using the generated schedule, the input data of the jobs is placed on the assigned racks while it is uploaded and jobs are run using Corral's cluster scheduler (Section 4.1).

Metrics. The primary metrics of interest are (a) makespan, in the batch scenario, and (b) average job completion time, in the online scenario.

4.5.3 MapReduce workloads

In this section, we show the benefits of using Corral to schedule MapReduce jobs in the batch and online scenarios, when all jobs have predictable characteristics.

Batch scenario. Figure 4.10 shows the improvement in makespan relative to Yarn-CS, for different workloads, when run as a batch – Corral achieves 10% to 33%

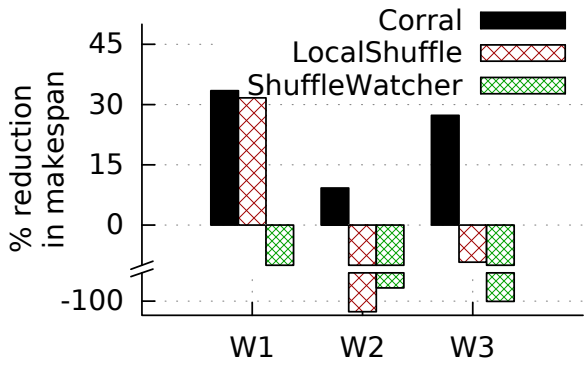


Figure 4.10: Reduction in makespan for different workloads, compared to Yarn-CS in the batch scenario.

reduction. The reduction in makespan for W2 is lower than that for the other workloads because W2 is highly skewed. Nearly, 90% of the jobs are *tiny* with less than 200MB (75MB) of input (shuffle) data and two (out of the 400) jobs are relatively *large*, reading nearly 5.5TB each. These large jobs determine the makespan of W2 and do not suffer significant contention from the tiny jobs. Out of the 7 racks available, Corral allocates 3 racks each to the two large jobs and packs most of the tiny jobs on the remaining rack. Compared to Yarn-CS, the benefits of Corral stem from running each of the large jobs in isolation, on separate subsets of racks.

Corral's improvements are a consequence of its better locality and reduced contention on the core network. Figure 4.11a shows that Corral reduces the amount of cross-rack data transferred by 20-90% compared to Yarn-CS. This, in turn, improves task completion times. To quantify this, we use two additional metrics, namely, (a) compute hours, which measures the total time spent by all the tasks in the workload; compared to Yarn-CS, using Corral reduces the compute hours by up to 20% (Figure 4.11b), and (b) average reduce time, which measures the average execution time of all the reduce tasks in a job. Figure 4.11c plots the cumulative fraction (over jobs) of this metric for Corral and Yarn-CS, showing that Corral is approximately 40% better at the median, with higher benefits at the tail.

Comparison with other baselines: Corral outperforms LocalShuffle showing that proper input data placement is key for good performance (Figure 4.10). Even with better shuffle locality, LocalShuffle performs worse than Yarn-CS for W2 and W3 due to its lack of input data locality.

ShuffleWatcher optimizes for each job individually and ends up scheduling several large jobs on the same subset of racks. This leads to increased completion times for

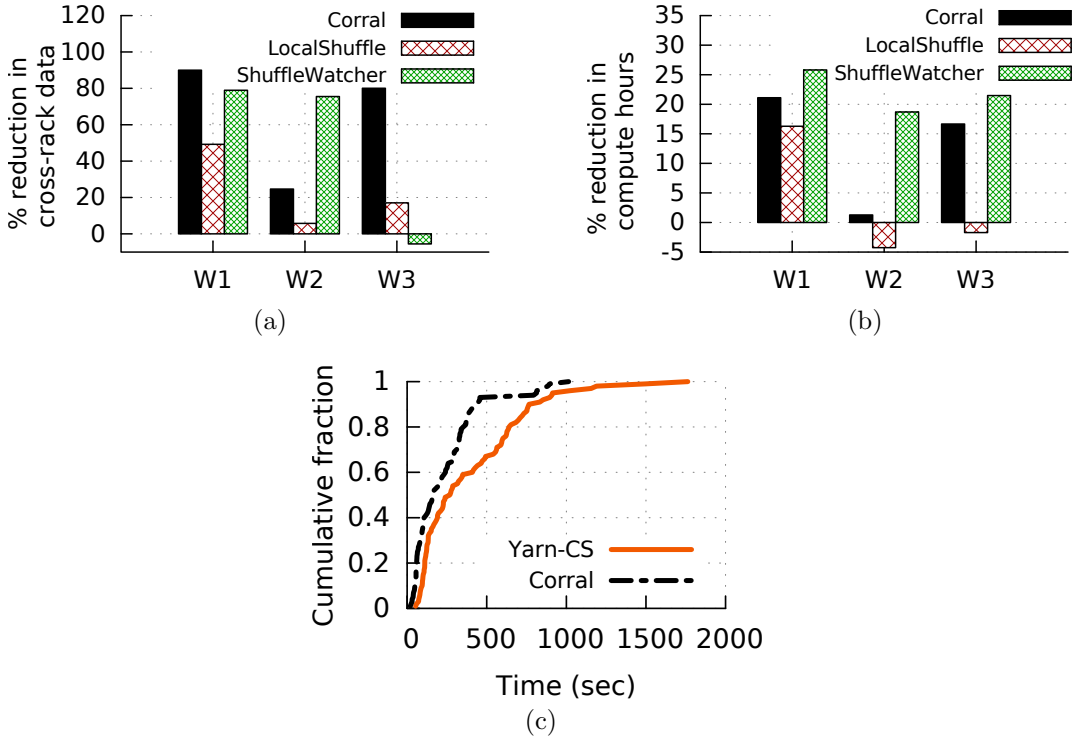


Figure 4.11: Comparing (a) reduction in cross-rack data transferred and (b) compute hours relative to Yarn-CS, and (c) cumulative fraction of average reduce time, for workload W1 in the batch scenario.

all these jobs. In the worst case, it can schedule all jobs on a single rack as it doesn't directly optimize for makespan or job completion time but tries to minimize the cross-rack data transferred. Thus, ShuffleWatcher results in significantly worse makespan compared to Yarn-CS for all workloads (Figure 4.10). Note that using ShuffleWatcher results in lower cross rack data for W2 compared to Corral (Figure 4.11a). This is because the huge jobs in W2 have nearly 1.8 times more shuffle data than input. Corral spreads them on 3 racks each (for better makespan) while ShuffleWatcher places them in a single rack.

ShuffleWatcher loads racks unevenly with some racks running significantly lower number of jobs than others. The tasks of these jobs finish much faster than when run with Corral and hence, ShuffleWatcher achieves better compute hours than Corral (Figure 4.11b). However, as shown in Figure 4.10, it is significantly worse than Corral for makespan.

Data balance: Corral optimizes for reducing the imbalance in the data allocated to different racks (Section 4.3.4). To evaluate this, we measure the coefficient of variation

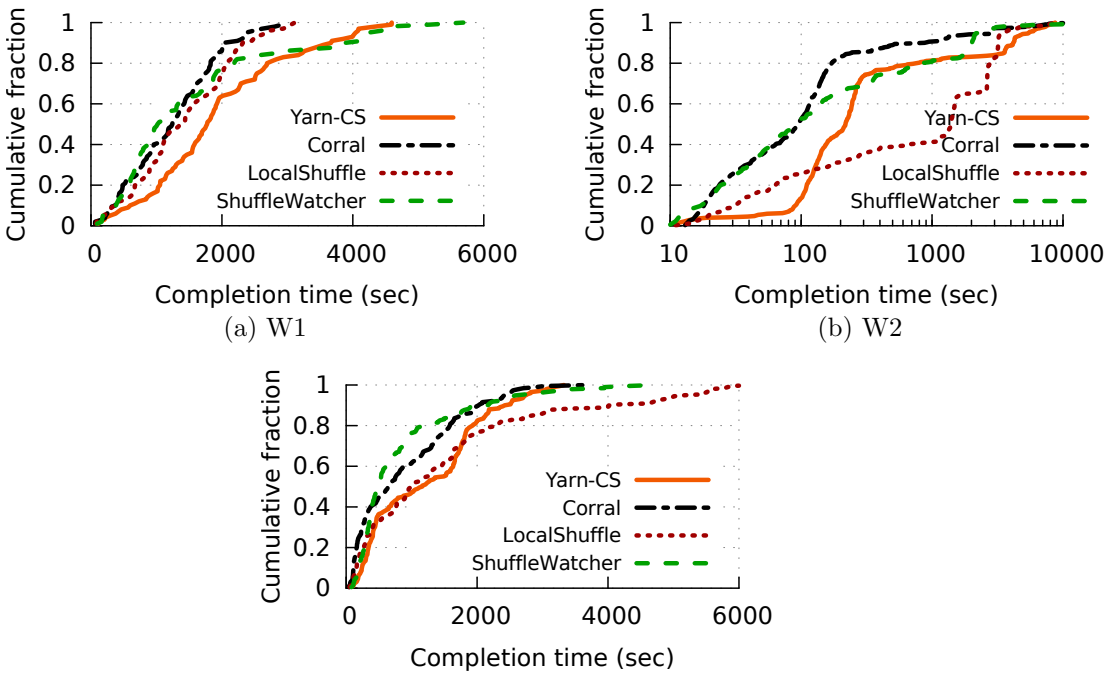


Figure 4.12: Cumulative fraction of job completion times for different workloads, when jobs arrive online.

(CoV) of the size of input data stored on each rack. Our results show that Corral has a low CoV of at most 0.004 and performs better than HDFS, which spreads data randomly, resulting in a CoV of at most 0.01[4].

Online scenario. In this scenario, jobs arrive over a period of time instead of as a batch. We pick the arrival times uniformly at random in $[0, 60\text{min}]$. Figure 4.12 shows the cumulative fraction of job completion time for workloads W1, W2 and W3. Corral outperforms Yarn-CS, with 30%-56% improvement at the median and nearly 26-36% improvement for the average job time (not shown). Further, Corral equally benefits jobs of all sizes. Figure 4.13 shows the reduction in average job completion time for workload W1, binned by the job size. Corral achieves 30-36% reduction in average job time across the various bins.

Comparison with other baselines: Similar to the batch case, LocalShuffle performs worse than Corral due to the lack of proper input data placement (Figure 4.12). While ShuffleWatcher is close to Corral at the lower percentiles, it is significantly worse at the higher percentiles. ShuffleWatcher schedules jobs independently. It ends

[4]Note that the CoV of a random distribution can be higher than that of uniform distribution, which is 0.

Figure 4.13: Reduction in average job completion time relative to Yarn-CS (binned by job size), for workload W1 in the online scenario.

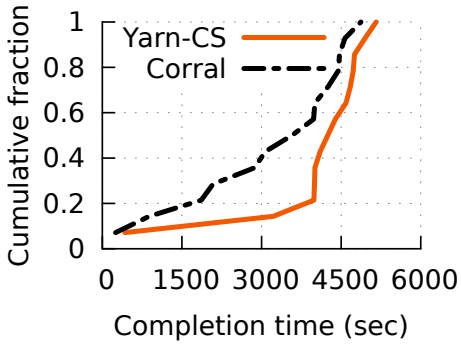


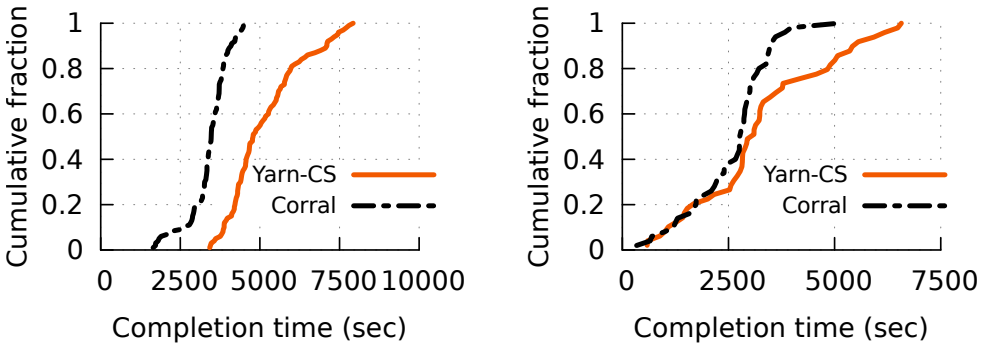
Figure 4.14: Benefits of running TPC-H queries with Corral.

up placing a large fraction of jobs on a few racks and a smaller fraction of jobs on the remaining racks. Jobs on the lightly loaded racks run faster due to lesser contention and those on the heavily loaded racks slow down. Figure 4.13 further confirms this, as ShuffleWatcher reduces the completion times of the small/medium jobs relative to Yarn-CS but performs worse for large jobs.

4.5.4 DAG workloads

To evaluate the benefits of the offline planner in Corral for data-parallel DAGs, we ran 15 queries from the TPC-H benchmark [48], using Hive 0.14.0 [9]. Each query reads from a 200GB database organized in ORC format [91]. The queries are submitted over a period of 25 minutes, with arrival times chosen uniformly at random. To emulate conditions in a real cluster, along with the queries, we also submit a batch of MapReduce jobs chosen from the workload W1, which are run using Yarn-CS.

Figure 4.14 plots the cumulative fraction of the execution times for the queries in



(a) Recurring jobs (b) Ad hoc jobs

Figure 4.15: Using Corral with a mix of jobs.

two cases: (i) the queries are run using Corral (dashed black line) and (ii) the queries are scheduled using Yarn-CS (solid orange line). Corral reduces the median execution time by nearly 18.5% with the average time being reduced by 21%. We found that these queries spend up to only 20% of their time in the shuffle stage, which shows Corral can provide benefits even for workloads that are mostly CPU or disk bound.

4.5.5 Scheduling ad hoc jobs

A significant portion of jobs in a production cluster can be ad hoc, e.g., those run for research or testing purposes. Such jobs arrive at arbitrary times and cannot be planned for in advance. Corral uses the same scheduling policies as Yarn's capacity scheduler (Yarn-CS) for such jobs. To explore the benefits of Corral in this scenario, we run a mix of 50 ad hoc and 100 recurring MapReduce jobs, drawn from W1. The ad hoc jobs are run as a batch with Yarn-CS, while the recurring jobs arrive uniformly over $[0, 60\text{min}]$.

Our observations are two-fold. First, even in the presence of ad hoc jobs, using Corral to schedule the recurring jobs is beneficial. Figure 4.15a shows the cumulative fraction of the completion times for recurring jobs in the workload. Corral reduces the average (median) completion times by 33% (27%). Second, using Corral to schedule the recurring jobs leads to faster execution of ad hoc jobs, especially at the tail with a 37% reduction at the 90th percentile compared to using Yarn-CS (Figure 4.15b). The makespan of the ad hoc jobs reduces by around 28% (not shown). As jobs run with Corral's policy use significantly lower core bandwidth and complete earlier, more network and compute resources are available for the ad hoc jobs, allowing them to

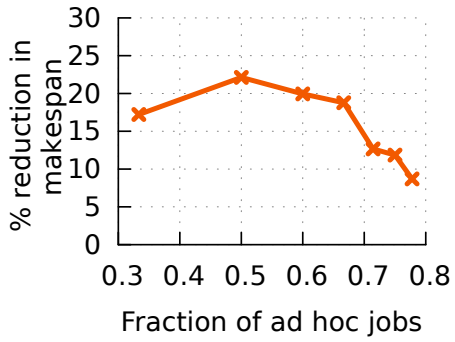


Figure 4.16: Improvements in makespan of ad hoc jobs with Corral (compared to Yarn-CS), as their fraction in the workload is varied.

also finish faster.

Discussion. The improvements in the completion time of ad hoc jobs, with Corral, depend on what fraction of the workload they make up. At one extreme, when all the jobs running on a cluster are ad hoc, Corral defaults to using existing techniques and thus, will not provide any benefits. At the other extreme, when ad hoc jobs make up only a tiny portion of the workload, the benefits from Corral would be limited as these ad hoc jobs can run off the cluster resources that remain after scheduling the recurring jobs. The benefits of Corral are significant when both ad hoc jobs and the recurring jobs make up comparable portions of the workload.

The above intuition is confirmed by the results shown in Figure 4.16. In this experiment, ad hoc jobs are run as a batch while the recurring jobs arrive uniformly over a period of one hour. All jobs are drawn from workload W1. The number of recurring jobs are fixed at 100 while we vary the number of ad hoc jobs between 50 and 350, i.e., they make up between 33% to 77% of the workload. We note that Corral achieves more than 10% benefits in makespan as long as the fraction of ad hoc jobs remains less than 75%. At 60%, which is generally the fraction of ad hoc jobs in production clusters [16,41], Corral achieves nearly 20% reduction in makespan of the ad hoc jobs.

While it is possible to (a) use techniques such as profiling (e.g., [92]) to estimate the latency of ad hoc jobs, or (b) use simple scheduling decisions such as running the next ad hoc job on the least loaded rack, we leave exploration of such *adaptive* techniques to future work.

4.5.6 Sensitivity analysis

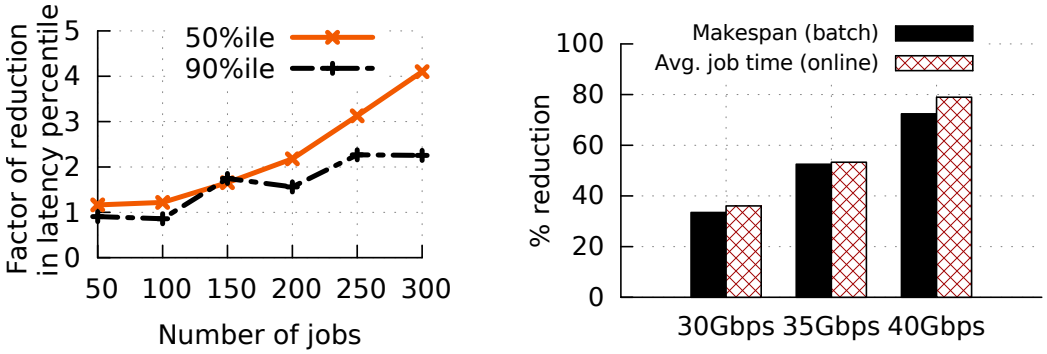
The benefits of Corral depend on (a) cluster load, (b) network utilization and (c) the accuracy with which job characteristics can be predicted. Here, we evaluate the robustness of Corral to variation in these factors.

Varying cluster load. The joint data and compute placement in Corral provides benefits to jobs by improving their data locality and reducing their dependence on network bandwidth. Thus, at low cluster load, when the network may not be fully utilized, the benefits of Corral would be minimal. To investigate how these benefits change with load on the cluster, we vary the number of jobs submitted over a fixed period of time. In particular, in our experiments, we vary the number of jobs submitted over a period of one hour from 50 to 300. The jobs arrive uniformly over that period of time and are drawn from the workload W1. Figure 4.17a shows the result of this variation as a ratio of different latency percentiles when the jobs are run using Yarn-CS compared to when they are run using Corral. With 50 jobs, the maximum cluster load (the ratio of number of slots occupied to total number of slots in the cluster) is near 30% (average is around 8%) and Corral does not provide any appreciable benefits over Yarn-CS. As expected, the benefits increase as load is increased and with 200 jobs, when the maximum cluster load is nearly 70%, Corral achieves nearly 2X (1.5X) improvement over Yarn-CS at the 50th percentile (90th percentile).

Varying background network traffic. Our results indicate that the gains with Corral increase significantly as the network utilization increases. For workload W1, Figure 4.17b shows that as the per-rack core network usage of the background traffic increases from 30 Gbps (50%) to 40 Gbps (67%), Corral achieves more than 2X higher benefits compared to Yarn-CS both in makespan (batch scenario) and average job time (online scenario).

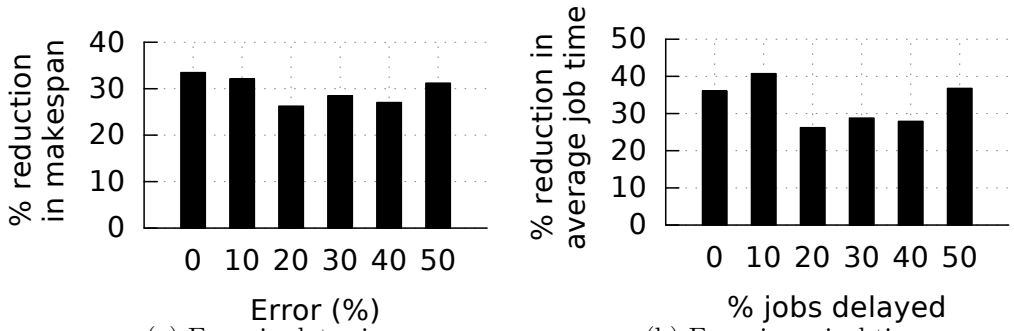
Error in predicted job sizes. Compared to the 6.7% observed in practice (Section 2.2), we varied the amount of data processed by jobs up to 50% and found that the benefits of Corral relative to Yarn-CS remain between 25-35% (Figure 4.18a). This shows that Corral's schedule is robust to errors in job sizes seen in practice.

Error in job start times. In practice, the start of a job can vary due to various reasons such as (a) input data upload does not finish in time, or (b) the jobs on which it depends on, are delayed. To evaluate the effect of such error in job start times, we choose a fraction f of jobs in a workload and add a random delay between $[-t, t]$, for a fixed t, in their start times. Figure 4.18b shows the results of running the



(a) Variation in number of jobs (b) Variation in background traffic

Figure 4.17: Benefits of using Corral relative to Yarn-CS at different loads.



(a) Error in data size (b) Error in arrival times

Figure 4.18: Variation in benefits of Corral (relative to Yarn-CS) with error in job characteristics for workload W1.

online scenario for workload W1 with such perturbation in arrival times. We set t at 4 minutes, which is nearly 6.67 times the expected job inter-arrival time and 20% of the average job completion time (and thus, represents a large error). Varying f from 0% to 50%, we found that the benefits of Corral reduce from 40% to at most 25%.

4.5.7 Using Corral with flow-level schedulers

Corral schedules the tasks in a job with better locality but doesn't explicitly schedule the network flow for them. In all experiments above, we use TCP as the transport protocol for Corral. However, several flow-level techniques have been proposed in literature (e.g., [18,19,21]), which have been shown to outperform TCP in datacenter environments. Here, we evaluate how Corral performs when used together with such flow-level schedulers.

For this purpose, we built a flow-based event simulator which models the execution

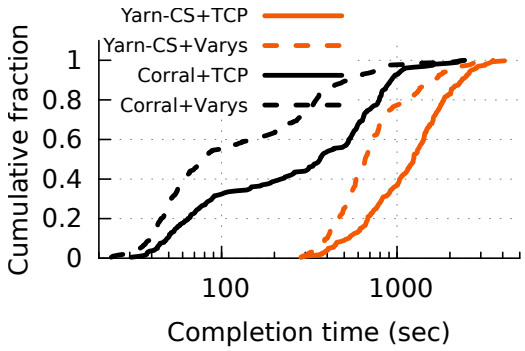


Figure 4.19: Simulation results: Cumulative fraction of job completion times with different flow-level and job schedulers.

of data-parallel jobs. We use pluggable policies for the job and network schedulers. We have implemented Yarn-CS and Corral to represent job schedulers. For network schedulers, we implemented a max-min fair bandwidth allocation mechanism to emulate TCP, and Varys [18], which uses application communication patterns to better schedule flows.

We simulate a topology of 2000 machines, organized into 50 racks with 40 machines each, connected using the folded CLOS topology with 5:1 oversubscription. Each machine can run up to 20 tasks and has a 1 Gbps NIC. We run 200 jobs from W1, arriving uniformly over 15min.

Figure 4.19 shows the cumulative fraction of the job completion times, when run using all the 4 possible combinations of job and network schedulers. Our main observations are as follows. First, using Varys with Yarn-CS improves the median job completion time by 46% compared to Yarn-CS+TCP. This is consistent with the improvements claimed previously [18]. Second, Corral+TCP outperforms Yarn-CS+Varys across all jobs, with nearly 45% gains at the median. This shows that the benefits of using schedulers like Varys are limited if the end-points of a flow are not placed properly. On the other hand, Corral schedules jobs in their own set of racks, reducing their core bandwidth usage and improving completion times. Finally, Corral+Varys results in much better job completion times compared to Corral+TCP or Yarn-CS+Varys. Thus, Corral's benefits are orthogonal to those attained with better flow-level scheduling and combining them performs better than using either one of them alone.

4.6 Discussion

Benefits. We note that the joint data and compute placement in Corral directly provides benefits to those applications that are constrained by network bandwidth. This has been shown to be the case for various data analytics applications [18,35,54], especially in the presence of oversubscribed network topologies and large background network transfers. As the amount of data being processed by these applications grows, this trend will continue to hold. Even if computation is performed on compressed data (e.g., [93]), intermediate data may not be compressed, leading to large volumes of data being shuffled by applications.

The exact benefits of Corral on the end-to-end latency of jobs depends on what fraction (on the critical path) of the total latency is spent in data being transferred over the network. The larger this fraction, the more are the benefits from scheduling jobs using Corral. However, the benefits of Corral are not limited to jobs with network-heavy stages — when less network-intensive jobs are run together with network-heavy jobs (as in Section 4.5.4) on a cluster with insufficient resources[5], Corral can benefit the former as it finishes the latter jobs faster and frees up the compute resources being used by them.

Dealing with failures. While Corral places one copy of job input data in the racks assigned to it, it spreads the other two copies across the rest of the cluster (similar to HDFS). This ensures that even if the assigned racks fail, the dataset can be recovered. Further, in the event that a majority of the machines (above a threshold) in a rack fail or a whole rack fails, Corral reverts back to using the existing placement policies (e.g., [74]) to run the jobs assigned to that rack, ensuring that they are not slowed down due to insufficient resources.

Data-job dependencies. Corral assumes that each job reads its own dataset. This simplifies the offline planning problem and allows each job to be scheduled independently. However, in general, the relation between datasets and jobs can be a complex bipartite graph. This can be incorporated into Corral by using the schedule of the offline planner and formulating a simple LP with variables representing what fraction of each dataset is allocated to each rack and the cost function capturing the amount of cross-rack data transferred, for any partition of datasets across the racks.

In-memory systems. Systems such as Spark [10] try to use memory to store input and intermediate data of jobs. While this decreases the dependence on disk I/O, dif-

[5] All jobs cannot run in parallel on the cluster.

ferent stages in these jobs (e.g., the shuffle) can still be bottlenecked or dependent on the network. The benefits of Corral extend to such scenarios as it reduces dependence on the core network and contention across jobs, by scheduling each of them on only a few racks.

4.7 Related work

The techniques in Corral are related to the following areas of research in the context of datacenter applications.

Scheduling techniques for data analytics systems. Improving data-locality in big data clusters has been the focus of several recent works. Techniques like delay scheduling [74] and Quincy [75] try to improve the locality of individual tasks (e.g., maps) by scheduling them close to their input. ShuffleWatcher [76] tries to improve the locality of the shuffle stage by scheduling both maps and reducers on the same set of racks. Others such as Tetris [94] schedule recurring jobs to ensure better packing of tasks at machines. Contrary to such approaches, Corral couples the placement of data and compute, achieving improved locality for all stages of a job.

Several techniques have been recently proposed with the goal of meeting application goals [16, 67, 70, 71]. The key difference of Corral from such approaches is our focus on multiple resources. Specifically, to meet deadlines of applications, we dedicate a network slice to them. The fact that the network is a shared yet distributed resource makes this hard. First, guaranteed network resources avoid inter-job contention and make model-based prediction tractable. Second, a per-application network slice, combined with the notion of slack, avoids the need for dynamic adaptation [16, 70]. Finally, by exploiting the trade-off between resources (number of machines, network) and between time, the cluster provider achieves greater flexibility and revenue.

Systems like Natjam [95] propose various eviction policies to support (deadlines-based) priorities in consolidated clusters. However, they do not explicitly determine the resources required by applications to meet their deadlines. As opposed to such approaches, Corral uses admission control, prediction slack and guaranteed resources to ensure applications meet deadlines.

Data placement techniques. CoHadoop [96] aims to colocate different datasets processed by a job on the same set of nodes, but does not guarantee locality for subsequent stages (e.g., shuffle). PACMan [33] caches repeatedly accessed data in

memory but does not provide locality for intermediate data transfers. Techniques like Scarlett [34] and GreenHDFS [97] use application access patterns to determine data replication factors or replica placement. However, none of these techniques coordinate data and compute placement for datacenter applications, which is the main focus in Corral. Further, the benefits from such data placement techniques are orthogonal to those of Corral and they can be used along with Corral for better performance.

Cross-layer scheduling techniques. The idea of placing data and compute together has been previously explored in systems like Purlieus [98] and CAM [99]. As opposed to such existing techniques, Corral exploits the recurring nature of datacenter applications and carefully assigns resources to them, to execute them efficiently. Further, Corral deals with complex DAG-structured jobs which have not been considered previously. Techniques such as [100] explore how network-level routing can be coordinated with application task-level dependencies and dynamically assigns network routes to different application flows. As opposed to such approaches, Corral assumes fixed network routes and achieves gains from from proper input data placement for applications.

Flow-level scheduling techniques. Several network-level techniques such as D3 [20], PDQ [21], Varys [18] and Baraat [19], have been proposed to finish network flows or groups of network flows faster. Choreo [101] assigns application tasks to existing VMs in a public cloud setting, to meet their bandwidth requirements. The benefits from such techniques are inherently limited as the end-points of the network transfers are fixed. Corral exploits the flexibility in placing input data and the subsequent stages of the jobs, and provides benefits orthogonal to such network-level schedulers (Section 4.5.7). Sinbad [35] takes advantage of flexibility in the placement of output data in big data clusters, but does not consider other stages in a job.

Malleable job scheduling. The problem of task scheduling across identical servers has been studied for over four decades (e.g., [102]). The basic formulation appears in [85], where tasks have precedence constraints and each task can run on a single server. A different variant of the problem considers malleable tasks, where each task can run on multiple servers [43, 103]. Our offline planning algorithm is inspired by these papers, but none of them addresses the issues of (a) malleability in the context of shared networks, and (b) balancing input data required for executing the jobs.

4.8 Conclusion

Corral, the resource planning component in Concorde, uses coordination between the storage and compute sub-systems of datacenter applications to achieve application goals. Corral uses the characteristics of future workloads and jointly optimizes the location of the job data (during upload) and tasks (during execution). For applications with deadline requirements, Corral exploits their resource malleability to select the resource tuple that will achieve the application goal while improving cluster efficiency. For applications without deadlines, Corral solves an offline planning problem and improves job performance by isolating applications from each other, reducing network contention in the cluster and running applications across fewer racks, thereby improving their data locality. We have implemented Corral in Yarn and with production workloads on a 210 machine cluster, we show that Corral can result in 10-33% reduction in makespan and 30-56% reduction in median job completion time, compared to Yarn's capacity scheduler.

Chapter 5

Speeding up datacenter applications at runtime

Datacenter applications execute over 10s to 1000s of machines and consist of a multitude of logical components, with complex input-output dependencies forming a general directed acyclic graph. At such scale and complexity, random runtime events which cause significant delays in any component of the application can lead to increased end-to-end application latency. In fact, our analysis of production traces from several user-facing services at Microsoft Bing reveals that the end-to-end response latency is quite variable. Despite significant developer effort, we found over 30% of the examined services have 95th (and 99th) percentile of latency 3X (and 5X) their median latency. The causes for high and variable latency can include slow servers, network anomalies, complex queries, congestion due to improper load balance or unpredictable events, and software artifacts such as buffering. The sheer number of components involved ensures that each request has a non-trivial likelihood of encountering an anomaly.

Several techniques have been proposed to deal with such runtime issues, albeit at a stage-level [32, 65, 75, 104]. For example, running a duplicate or *reissuing* a part of an application or a whole request is a commonly used technique to reduce latency. In the context of data analytics applications, this involves running duplicate copies of tasks, which are determined to be *laggards*, in a particular stage of the DAG based on progress indicators or timeouts [22, 32]. For web services, a duplicate of the original request in a particular stage is reissued at a copy of the stage [65].

However, to reduce the end-to-end latency of datacenter applications, such techniques need to be applied in an end-to-end manner. This can be challenging for several reasons. First, different stages benefit differently from different techniques. For example, request reissues work best for stages with low mean and high variance of latency. Second, end-to-end effects of local actions depend on topology of the DAG; reducing latency of stages usually off the critical path may not improve end-to-end latency. Finally, many such latency reduction techniques have an associated overhead,

e.g., increased resource usage when reissuing a request.

Addressing this problem, we developed *Kwiken* [12], a framework which allows us to take an end-to-end view of request latency and optimizes for it. It decomposes the problem of minimizing latency, at runtime, over an application DAG into a manageable optimization over its individual components or stages. Kwiken uses the characteristics of each component in the DAG to determine which latency reduction technique best works for that component. It ensures that the cost incurred when using these techniques, in an end-to-end manner, is within a given bound.

Apart from showing how to use reissues in an end-to-end manner, we present two new latency reduction techniques: (a) a new timeout policy to *trade off partial answers for latency*, and (b) *catching-up* for laggard queries. The basic ideas behind these strategies are simple. First, many DAGs can still provide a useful end-to-end answer even when individual stages return partial answers. So, at stages that are many-way parallel, Kwiken provides an early termination method that improves query latency given a constraint on the amount of acceptable loss on answer quality. Second, Kwiken preferentially treats laggard queries at later stages in the DAG, either by giving them a higher service rate (more threads), being more aggressive about reissuing them or by giving them access to a higher priority queue in network switches. Kwiken incorporates these techniques into the optimization framework to minimize the end-to-end latency while keeping the total additional cost within a given budget.

While this dissertation focuses on applying Kwiken in the context of web services, it can be used more generally and can incorporate a large class of latency reduction techniques (e.g., the network latency reduction techniques described in [105]). It also applies to many applications where work is distributed among disjoint components and dependencies can be structured as a DAG. This includes page loading in web browsers [106] and mobile phone applications [107], and data analytics applications (discussed in Section 6.1).

We evaluate Kwiken using production web service DAGs at Microsoft Bing and show that Kwiken improves 99th percentile of latency by an average of 29% with just 5% extra resources and by about 50% when partial answers are allowed for just 0.1% of the queries. Further, we show that reissues and partial answers provide complementary benefits; allowing partial answers for 0.1% queries lets a reissue budget of 1% provide more gains than could be achieved by increasing the reissue budget to 10%.

The rest of this chapter is organized as follows. We first explore the causes of increased latencies at Microsoft Bing (Section 5.1). We next provide a high-level

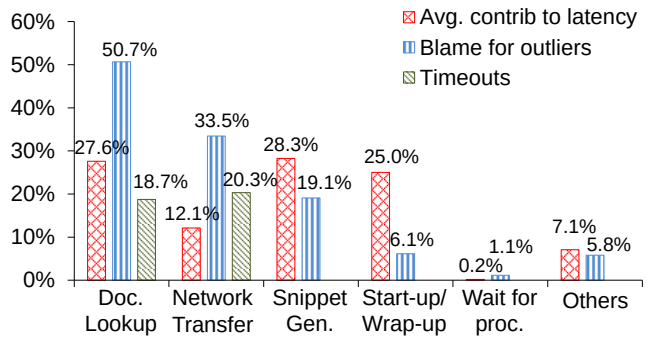


Figure 5.1: Stages that receive blame for the slowest 5% queries in the web search DAG.

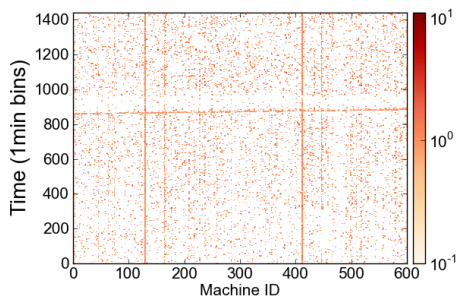


Figure 5.2: Heatmap showing how the latency varies across machines and time (for queries to the web service in Figure 2.1).

overview of Kwiken (Section 5.2), and then describe the techniques in Kwiken in detail (Section 5.3). We show the benefits of using Kwiken on production workloads (Section 5.4), discuss the limitations of the techniques in Kwiken (Section 5.5) and finally, describe the related work (Section 5.6) and conclude (Section 5.7).

5.1 Causes for high latencies in web service applications

To understand the causes of high latencies in web services, this section studies how latencies in the web search DAG at Microsoft Bing are affected by different stages that make up the service. This web search DAG is shown in Figure 2.1.

To this end, we collect a trace of request executions over several days and investigate in detail the 5% slowest queries in the trace. For these queries, we assign blame to a stage when its contribution to that query's latency is more than $\mu + 2\sigma$, where μ, σ are the mean and stdev of its contribution over all queries. If a stage takes too long for a query, it is timed-out. In such cases, the blame is still assigned to the stage,

citing timeout as the reason. Figure 5.1 depicts, for each stage of the web service DAG, its average contribution to latency along with the fraction of delayed responses for which it has timed-out or is blamed (includes timeouts). Since more than one stage can be blamed for a delayed response, the blame fractions add up to more than one.

We see that the document lookup and the network transfer stages receive the most blame (50.7% and 33.5% each). In particular, these stages take so long for some queries that the scheduler times them out in 18.7% and 20.3% of cases respectively. Network transfer receives blame for many more outliers than would be expected given its typical contribution to latency (just 12.1%). We also see that though the start-up/ wrap-up stage contributes sizable average latency, it is highly predictable and rarely leads to outliers. Further, the servers are provisioned such that the time spent waiting in queues for processing at both the doc lookup and the snippet generation stages is quite small.

Why would stages take longer than typical? To examine the document lookup stage further, we correlate the query latency with wall-clock time and the identity of the machine in the document lookup tier that was the last to respond. Figure 5.2 plots the average query latency per machine per second of wall time. The darkness of a point reflects the average latency on log scale. We see evidence of flaky machines in the document lookup tier (dark vertical lines); queries correlated with these machines consistently result in higher latencies. We conjecture that this is due to hardware trouble at the server. We also see evidence for time-dependent events, i.e., periods when groups of machines slow down. Some are rolling upgrades through the cluster (horizontal sloping dark line), others (not shown) are congestion epochs at shared components such as switches. We also found cases when only machines containing a specific part of the index slowed down, likely due to trouble in parsing some documents in that part of the index.

To examine the network transfer stage further, we correlate the latency of the network transfer stage with packet-level events and the lag introduced in the network stack at either end. We collected several hours of packet traces in production beds for the network transfer stage in the web search DAG (Figure 2.1). To compare, we also collect packet traces from production map-reduce clusters that use the same server and switch hardware but carry traffic that is dominated by large flows. The results of this analysis is shown in Table 5.1. We see that the request-response traffic has 10X higher loss rate than in the map-reduce cluster. Further, the losses are bursty with

Parameter	Value Percentiles		
	50th	90th	99th
at server, network load due to request-response traffic	895pps, .62Mbps	2242pps, 1.84Mbps	2730pps, 2.3Mbps
Lag to retransmit	67.2ms	113.3ms	168.7ms
Packet loss prob. of req-response traffic	.00443		
To compare: packet loss prob. of map-reduce	.0004336		
Fraction of losses recovered by RTO	.987		

Table 5.1: Network Characteristics

a coefficient of variation ($\frac{\sigma}{\mu}$) of 2.4536. The increased loss rate is likely due to the scatter-gather pattern, i.e., responses collide on the link from switch to aggregator. Most of the losses (over 98%) are recovered only by a retransmission timeout because there are not enough acks for TCP's fast retransmission due to the small size of the responses. Surprisingly, the RTO for these TCP connections was quite large, in spite of RTO_min being set to 20ms; we are still investigating the cause. We conclude that TCP's inability to recover from burst losses for small messages is the reason behind the network contributing so many outliers.

5.2 Key ideas in Kwiken

The goal of Kwiken is to improve the latency of datacenter application DAGs, especially on the higher percentiles. We pick the variance of latency as the metric to minimize because doing so will speed-up all of the tail requests; in that sense, it is more robust than minimizing a particular quantile[1].

Our framework optimizes the end-to-end latency at both the stage and DAG levels. At the stage/local level, it selects a policy that minimizes the variance of the stage latency. At the DAG/global level, it combines these local policies to minimize the end-to-end latency. We employ three core per-stage techniques for latency reduction —

[1]Delaying responses such that all queries finish with the slowest has a variance of 0, but is not useful. An implicit requirement in addition to minimizing variance, which Kwiken satisfies, is for the mean to not increase.

reissue laggards at replicas, skip laggards to return timely albeit (possibly) incomplete answers and *catch-up*, which involves speeding up requests based on their overall progress in the application DAG.

Using latency reduction techniques incurs cost — for example, extra resources are used to serve reissued requests. So, we have to reason about apportioning a shared global cost budget across stages to minimize the end-to-end latency. For example, reissues have higher impact in stages with high variance. Similarly, speeding up stages that lie on the critical path of the application DAG is more helpful than those that lie off the critical path. Also, as shown in Figure 2.5, variance of some stages reduces quickly even with a few reissues, while other stages require more reissues to achieve the same benefits. Finally, the cost of reissuing the same amount of requests could be orders of magnitude higher in stages that are many-way parallel, a factor that has to be incorporated into the overall optimization.

To reason about how local changes impact overall latency, our basic idea is to decompose the variance of the application DAG's latency into the variance of individual stages' latency. If the random variable L_s denotes the latency at stage s, then the latency of DAG w is given by

$$\mathcal{L}_w(L_1, \ldots, L_N) = \max_p \sum_{s \in p} L_s, \tag{5.1}$$

where p stands for a *path*, namely an acyclic sequence of stages through the application DAG (from input to output). Ideally, we would use the variance of $\mathcal{L}_w$ as our objective function, and minimize it through allocating budget across stages. Unfortunately, however, the variance of $\mathcal{L}_w$ does not have a closed form as a function of the individual stages' statistics (e.g., their first or second moments). Instead, we resort to minimizing an upper bound of that variance. Recall from Section 2.1.2 that the different L_s can be roughly treated as independent random variables. Using this approximation together with (5.1) leads to the following decomposition:

$$\mathrm{Var}(\mathcal{L}_w) \leq \sum_{s \in w} \mathrm{Var}(L_s), \tag{5.2}$$

where $\mathrm{Var}(\cdot)$ denotes the variance of a random variable.

Proof. For each random variable L_s we introduce a new independent random variable L'_s which has the same distribution as L_s. Let $\mathbf{L} = (L_1, \ldots, L_N)$ and $\mathbf{L}^{(s)} = (L_1, \ldots, L_{s-1}, L'_s, L_{s+1}, \ldots, L_N)$. Then, using the Efron-Stein inequality [108], we have

$\mathrm{Var}(\mathcal{L}_w(\mathbf{L})) \leq \frac{1}{2}\sum_s E\big[(\mathcal{L}_w(\mathbf{L}) - \mathcal{L}_w(\mathbf{L}^{(s)}))^2\big] \leq \frac{1}{2}\sum_s E\big[(L_s - L'_s)^2\big] = \sum_s \mathrm{Var}(L_s)$.

The above bound is always tight for sequential DAGs, as stage variances add up. It can also be shown that (5.2) is the best general bound for parallel DAGs.

Using Chebyshev's inequality, (5.2) immediately implies that $Pr(|\mathcal{L}_w - E\mathcal{L}_w| > \delta) \leq \frac{(\sum_s \mathrm{Var}(L_s))^2}{\delta^2}$. The bound indicates that minimizing the sum of variances is closely related to minimizing the probability of large latency deviations, or latency percentiles. Better concentration bounds (e.g., Bernstein [108]) can also be obtained. We emphasize that we do not claim tightness of the bounds, but rather use them as a theoretical insight for motivating sum of variances minimization. As we elaborate below, the above decomposition to sum of variances leads to a tractable optimization problem, unlike other approaches for solving it.

Alongside the latency goal, we need to take into account the overall cost from applying local changes. Here, we describe the framework using reissues. Formally, let r_s be the fraction of requests that are reissued at stage s and let c_s be the (average) normalized resource cost per request at stage s, i.e., $\sum_s c_s = 1$. Then, the overall normalized cost from reissues is $C(\mathbf{r}) = \sum_s c_s r_s$, and the problem of apportioning resources becomes:

$$\begin{aligned} \text{minimize} \quad & \sum_s \mathrm{Var}\big(L_s(r_s)\big) \\ \text{subject to} \quad & \sum_s c_s r_s \leq B, \end{aligned} \tag{5.3}$$

where B represents the overall budget constraint for the application DAG[2] and $L_s(r_s)$ is the latency of stage s under a policy that reissues all laggards after a timeout which is chosen such that only an r_s fraction of requests are reissued. Since $\{c_s\}$ are normalized, B can be viewed as the fraction of additional resources used for latency reduction. The formulation in (5.3) can be generalized to accommodate multiple speedup techniques (see Section 5.3.4). This optimization problem is the basis for our algorithm.

We alternatively used a weighted version of the cost function, $\sum_s w_s \mathrm{Var}\big(L_s(r_s)\big)$, where the weights w_s can be chosen to incorporate *global* considerations. For example, we can set w_s based on (i) total number of paths crossing stage s, or (ii) mean latency of s. However, our evaluation showed no significant advantages compared to the

[2]Considerations on how to choose the budget for each DAG, which we assume as given by an exogenous policy, are outside the scope of this dissertation.

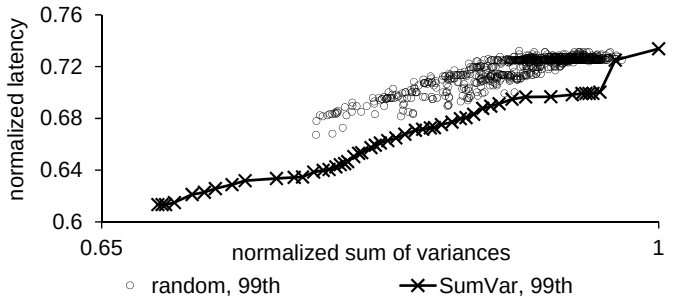


Figure 5.3: This plot shows the results of a random search in the reissue budget space (circles) and using SumVar on the same DAG (line with crosses). It illustrates that as sum of variances (x-axis, normalized) decreases, so does the 99th percentile of latency (y-axis, normalized).

unweighted version.

We solve (5.3) by first constructing per-stage *models* (or *variance-response functions*), denoted $V_s(r_s)$ that estimate $\text{Var}(L_s(r_s))$ for each stage s. Due to the complexity of these curves (see Figure 2.5), we represent them as empirical functions, and optimize (5.3) using an iterative approach based on gradient descent; see details in Section 5.3. Due to the non-convexity of (5.3), our algorithm has no performance guarantees. However, in practice, our algorithm already achieves adequate precision when the number of iterations is $O(N)$, where N is the number of stages in the DAG.

So far, we have considered "local" improvements, where latency reduction policy inside each stage is independent of the rest of the DAG. Our *catch-up* policies use the execution state of the entire request to make speed-up decisions. These are described in more detail in Section 5.3.3.

Finally, as described earlier, burst losses in the network are responsible for a significant fraction of high latencies. We recommend lowering RTO_min to 10ms and using a burst-avoidance technique such as ICTCP [109] at the application level. While not a perfect solution, it addresses the current problem and is applicable today.

5.3 Design of Kwiken

In this section, we provide the details of applying our framework to three different techniques — reissues, incompleteness, and catch-up.

5.3.1 Adaptive reissues

Per-stage reissue policies. A typical use of reissues is to start a second copy of the original request in stage s at a pre-determined time T_s if there is no response before T_s, and use the faster of the two responses [65]. Given the reissue budget r_s for each stage s in an application DAG, we use the techniques described in Section 3.2.2 to determine its variance-response function $V_s(r_s)$.

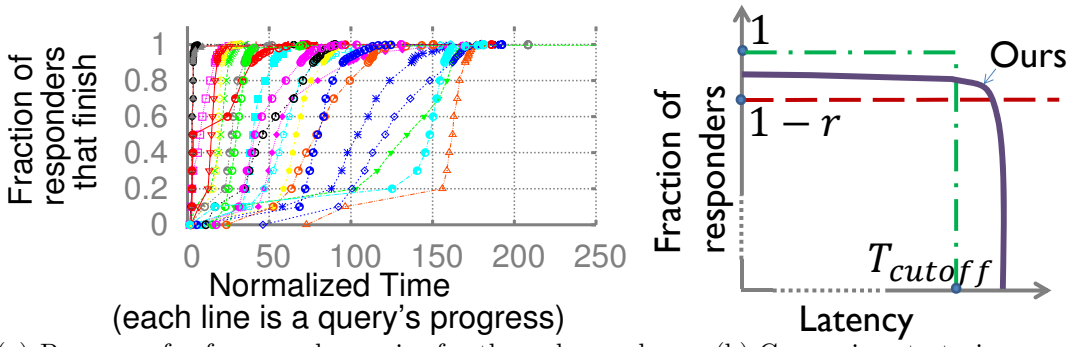
Note that we can compute $V_s(r_s)$ for different reissue policies and pick the best one for each r_s (e.g., launching two reissues instead of just one after a timeout or reissuing certain fraction of requests right away, i.e., timeout of zero). Using $V_s(r_s)$, we note that our framework helps abstract away the specifics of the per-stage latency improvements from the end-to-end optimization. Further, $V_s(r_s)$ needs to be computed only once per stage unless there are significant changes to the application; for example, a major code change or change in the DAG structure.

Apportioning budget across stages. Equipped with per-stage reissue policies captured in $V_s(r_s)$, we apportion budget across stages by solving (5.3) with $V_s(r_s)$ replacing $\mathrm{Var}\big(L_s(r_s)\big)$ for every s.

Kwiken uses a greedy algorithm, SumVar, to solve (5.3) which is inspired by gradient descent. SumVar starts from an empty allocation ($r_s = 0$ for every stage)[3]. In each iteration, SumVar increases $r_{s'}$ of one stage s' by a small amount where s' is chosen so that the decrease in (5.3) is maximal. More formally, SumVar assigns resources of cost $\delta > 0$ per iteration (δ can be viewed as the step-size of the algorithm). For each stage s, δ additional resources implies an increase in r_s by δ/c_s (since $c_s r_s =$ resource cost) which reduces variance by the amount $(V_s(r_s) - V_s(r_s + \delta/c_s))$. Hence, SumVar assigns δ resources to stage $s' \in \mathrm{argmax}_s(V_s(r_s) - V_s(r_s + \delta/c_s))$; ties are broken arbitrarily.

We demonstrate our algorithm on a production DAG with 28 stages and significant sequential and parallel components. First, we generate 1000 random reissue allocation and for each, plot the achieved sum of variances and 99th percentile of end-to-end latency (circles in Figure 5.3). Notice that as sum of variances decreases, so does the latency percentile. This illustrates that the intuition behind our approach is correct; even in complex DAGs, sum of variances is a good metric to minimize in order to improve tail latencies. Second, we plot the progress of our algorithm on the same DAG (line with crosses in Figure 5.3, from right to left). It shows that the gradient descent approach can achieve lower latency than a random search.

[3]In our experiments, we tried other initial allocations, but they did not improve performance.



(a) Progress of a few sample queries for the web search DAG

(b) Comparing strategies

Figure 5.4: Trading off incompleteness for latency.

Our experiments show that it suffices to divide the budget into γN chunks of equal size, where $\gamma \in [2, 4]$ and N is the number of stages. Consequently, the number of iterations is linear in the number of stages. Each iteration requires $O(\log N)$ for recomputing a single gradient component and inserting it into a heap structure. Consequently, the overall complexity of our algorithm is $O(N \log N)$. Importantly, it is independent of the topology of the DAG.

5.3.2 Trading off completeness for latency

Our goal is to minimize response latency given a constraint on utility loss. To be able to use (in)completeness as a tool in our optimization framework, we treat utility loss as a "resource" with budget constraint (average quality loss) and decide how to apportion that resource across stages and across queries within a stage so as to minimize overall latency variance. We emphasize that this formulation is consistent with the requirements at Microsoft Bing; both reduction in latency and higher quality answers improve user satisfaction and can be used interchangeably in optimization.

Discussions with practitioners at Microsoft Bing reveal that enormous developer time goes into increasing answer relevance by a few percentage points. While imprecise answers are acceptable elsewhere (e.g., in approximate data analytics), web service DAGs can only afford a small amount of utility loss to improve latency. Hence Kwiken works within a strict utility loss rate of .001 (or 0.1%), i.e., at most 1 out of 1000 requests can have incomplete answers.

Using incompleteness within a stage. The basic setup here is of a single stage,

latency distribution	//'ism (n)	utility loss (r)	Latency Reduction (% over baseline) 95th	99th
Normal mean=1, sd=10	10000	.001	25.33%	29.22%
		.01	44.29%	47.67%
LogNormal meanlog=1, sdlog=2	10000	.001	90.34%	93.83%
		.01	98.22%	98.96%
LogNormal meanlog=1, sdlog=2	1000	.001	59.93%	64.71%
		.01	93.30%	96.12%
Web	1000s	.001	4.1%	4.0%
		.01	43.1%	77.7%
Image	100s	.001	0%	0%
		.01	42.6%	81.2%
Video	100s	.001	0%	0%
		.01	31.2%	51.3%

Table 5.2: Given utility loss rate (r), the improvement in latency from stopping when the first $\lceil n(1-r) \rceil$ responders finish.

with a constraint on the maximal (expected) quality loss rate, denoted r. Consider a simple strategy: let each query run until $\lceil n(1-r) \rceil$ of its n responders return their answer. Then if the best document's location is uniform across responders, the expected utility loss is r. To appreciate why it reduces latency, consider a stage with n responders whose latencies are $X_1 \ldots X_n$. This strategy lowers query latency to the $\lceil n(1-r) \rceil$'th largest value in $X_1, \ldots X_n$ as opposed to the maximum X_i.

Table 5.2 shows the latency reductions for a few synthetic distributions and for the web, image, and video search stages where we replay the execution traces from tens of thousands of production queries at Microsoft Bing. First, we note that even small amounts of incompleteness yield disproportionately large benefits. For a normal distribution with mean 1 and stdev 10, we see that the 95th percentile latency reduces by about 25% if only 0.1% of the responses are allowed to be incomplete. This is because the slowest responder(s) can be much slower than the others. Second, all other things equal, latency gains are higher when the underlying latency distribution has more variance, or the degree of parallelism within the stage is large (as shown in Table 5.2). LogNormal, a particularly skewed distribution, has 3.5X larger gains than Normal but only 2.2X larger when the number of parallel responders drops to 10^3. However, we find that the gains are considerably smaller for our empirical distributions. Partly, this is because these distributions have bounded tails, as the user or the system times-out queries after some time.

To understand why the simple strategy above does not help in practice, Figure 5.4 (a) plots the progress of example queries from the web-search stage. Each line corresponds to a query and shows the fraction of responders vs. elapsed time since the query began. We see significant variety in progress — some queries have consistently quick or slow responders (vertical lines on the left and right), others have a long tail (slopy top, some unfinished at the right edge of graph) and still others have a few quick responders but many slow ones (steps). To decide which queries to terminate early (subject to overall quality constraint), one has to therefore take into account *both* the progress (in terms of responders that finished) and the elapsed time of the individual query. For example, there are no substantial latency gains from early termination of queries with consistently quick responders, as even waiting for the last responder may not impact tail latency. On the other hand, a slow query may be worth terminating even before the bulk of responders complete.

Building up on the above intuition, Kwiken employs dynamic control based on the progress of the query. Specifically, Kwiken terminates a query when either of these two conditions hold: *i*) the query has been running for T_d time after p fraction of its components have responded, *ii*) the query runs for longer than some cut-off time T_c. The former check allows Kwiken to terminate a query based on its progress, but not terminate too early. The latter check ensures that the slowest queries will terminate at a fixed time regardless of however many responders are pending at that time. Figure 5.4 (b) visually depicts when queries will terminate for the various strategies.

Kwiken chooses these three parameters empirically based on earlier execution traces. For a given utility loss budget r, Kwiken parameter sweeps across the (T_c, T_d, p) vectors that meet the quality constraint with equality, and computes the variance of stage latency. Then, Kwiken picks the triplet with the smallest variance. Repeating this procedure for different values of r yields the variance-response curve $V(r)$. Note that the approach for obtaining $V(r)$ is data driven. In particular, the choice of parameters will vary if the service software is rewritten or the cluster hardware changes. From analyzing data over an extended period of time, we see that parameter choices are stable over periods of hours to days (we show results with disjoint test and training sets in Section 5.4.3).

Composing incompleteness across stages. Any stage that aggregates responses from parallel components benefits from trading off completeness for latency. When a DAG has multiple such stages, we want to apportion utility loss budget across them so as to minimize the end-to-end latency. The approach is similar in large part to the

case of reissues — the variance-response curves computed for each stage help split the overall optimization to the stage-level.

Unlike reissue cost, which adds up in terms of compute and other resources, utility loss is harder to compose, especially in general DAGs where partial results are still useful. Modeling such scenarios fully is beyond the scope of this dissertation. Nevertheless, we show two common scenarios below, where the budget constraint can be written as a weighted sum over the loss rates at individual stages. First, consider a sequential DAG with N stages where an answer is useful only if every stage executes completely. If r_i is the loss budget of stage i, the overall utility loss is given by $r_1+(1-r_1)r_2+\cdots+\left(\prod_{s=1}^{N-1}(1-r_s)\right)r_N$. which is upper bounded by $\sum_i r_i$. Hence, the budget constraint is $\sum_i r_i \leq B$, i.e., the "cost" of loss c_s is one at all stages. Second, consider stages that are independent i.e., the usefulness of a stage's answer does not depend on any other stage (e.g., images and videos returned for a query). Here, the overall utility loss can be written as $\frac{\sum r_s u_s}{\sum u_s}$ where u_s is the relative contribution from each stage's answer. Then, the budget constraint is given by $\sum_s c_s r_s \leq B$, where $c_s = \frac{u_s}{\sum_{s'} u_{s'}}$.

5.3.3 Catch-up

The framework described so far reduces end-to-end latency by making local decisions in each stage. Instead, the main idea behind catch-up is to speed-up a request based on its progress in the DAG as a whole. For example, when some of the initial stages are slow, we can reissue a request more aggressively in the subsequent stages. We consider the following techniques for catch-up: (1) allocate more threads to process the request; given multi-threaded implementation of many stages at Microsoft Bing, we find that allocating more threads to a request reduces its latency. (2) Use high-priority network packets on network switches for lagging requests to protect them from burst losses. And (3), reissue requests more aggressively based on the total time spent in the DAG — we call this *global reissues* to distinguish it from local reissues (discussed in Section 5.3.1), where the reissue is based on time spent within a stage.

Each of these techniques uses extra resources and could affect other requests if not constrained. To ensure catch-up does not overload the system, Kwiken works within a catch-up budget per application DAG. Given per-stage budget, Kwiken estimates a threshold execution latency, T_z, and speeds the parts of a query that remain after T_z using the techniques above. As the decisions of a catch-up policy

depend on request execution in previous stages, allocating catchup budget across stages cannot be formulated as a separable optimization problem, unlike the case of local techniques (Section 5.3.1). We therefore use simple rules of thumb here. For example, for global reissues, we allocate catch-up budget proportionally to the budget allocation for local reissues. Intuitively, a stage that benefits from local reissues, can also speed-up lagging requests. We evaluate the catch-up policies in Section 5.4.4.

5.3.4 Putting it all together

To conclude, we briefly highlight how to combine different techniques into a unified optimization framework. Using superscripts for the technique *type*, let $k = 1, \ldots, K$ be the collection of techniques. Then, our optimization framework (5.3) extends as follows to multiple dimensions:

$$\begin{aligned} \text{minimize} \quad & \sum_s \mathrm{Var}\big(L_s(r_s^1, \ldots, r_s^K)\big) \\ \text{subject to} \quad & \sum_s c_s^k r_s^k \leq B^k, \quad k = 1, \ldots, K. \end{aligned} \tag{5.4}$$

As before, $\mathrm{Var}\big(L_s(r_s^1, \ldots, r_s^K)\big)$ are the per-stage variance-response curves. These models abstract away the internal optimization given $(r_s^1, \ldots, r_s^K)$. Greedy gradient-like algorithms (such as SumVar) can be extended to solve (5.4), however, the extension is not straightforward. The main complexity in (5.4) is hidden in the computation of the variance-response curves — as opposed to a scan over one dimension in (5.3), variance-response curves in (5.4) requires a scan over the k-dimensional space, $(r_s^1, \ldots, r_s^K)$. In practice, we note that the optimization often decouples into simpler problems. For example, assume $K = 2$ with reissues and partial responses as the two techniques for reducing latency. Partial responses are mostly useful in many-way parallel services which have a high cost for reissues. Hence, we can use the utility loss budget only for the services with high fan-out and the reissue budget for the rest of the services. Finding a general low complexity algorithm to solve (5.4) is left to future work.

5.4 Evaluation

In this section, we evaluate the individual techniques in Kwiken by comparing them to other benchmarks (Section 5.4.2 - Section 5.4.4), followed by using all Kwiken techniques together (Section 5.4.5). Using execution traces and DAGs from Microsoft Bing, we show that:

- With a reissue budget of just 5%, Kwiken reduces the 99th percentile of latency by an average of 29% across DAGs. This is over half the gains possible from reissuing every request (i.e., budget=100%). Kwiken's apportioning of budget across stages is key to achieving these gains.
- In stages that aggregate responses from many responders, Kwiken improves the 99th percentile of latency by more than 50% with a utility loss of at most 0.1%.
- Using simple catch-up techniques, Kwiken improves the 99th percentile latency by up to 44% by using just 6.3% more threads and prioritizing 1.5% network traffic.
- By combining reissues with utility trade-off we see that Kwiken can do much better than when using either technique by itself; for example, a reissue budget of 1% combined with a utility loss of 0.1% achieves lower latency than just using reissues of up to 10% and just trading off utility loss of up to 1%.
- Kwiken's data-driven parameter choices are stable.

5.4.1 Methodology

Traces from production. To evaluate Kwiken, we extract the following data from production traces for the 45 most frequently accessed DAGs at Microsoft Bing: the details of the DAG, request latencies at each stage as well as the end-to-end latency, the cost of reissues at each stage and the usefulness of responses (e.g., ranks of documents) when available. To properly measure latencies on the tail, we collected data for at least $10,000$ requests for each DAG and stage. The datasets span several days during Oct-Dec 2012. We ignore requests served from cache at any stage in their DAG; such requests account for a sizable fraction of all requests but do not represent the tail.

We conducted operator interviews to estimate the cost of reissue at each stage. Our estimates are based on the resources expended per request. For stages that process the request in a single thread, we use the mean latency in that stage as an approximation

to the amount of computation and other resources used by the request. For more complex stages, we use the sum of all the time spent across parallel servers to execute this stage. Kwiken only relies on the relative costs across stages when apportioning budget.

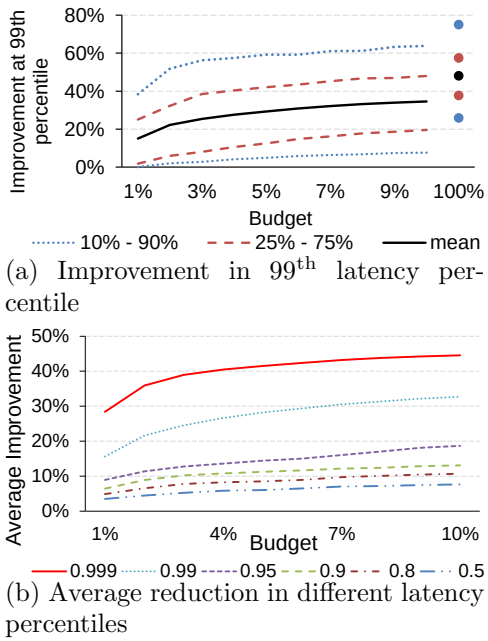
Simulator. We built a trace-driven simulator, that mimics the application DAG controller used in production at Microsoft Bing, to test the various techniques in Kwiken. The latency of a request at each stage and that of its reissue (when needed) are sampled independently from the distribution of all request latencies at that stage. We verified that this is reasonable: controlled experiments on a subset of DAGs where we issued the same request twice back-to-back showed very small correlation; also, the time spent by a request in different stages in its DAG had small correlation (see Section 2.1).

Estimating policy parameters. The parameters of the Kwiken policies (such as per-stage reissue timeouts) are trained based on traces from prior executions. While we estimate the parameters on a *training data set*, we report performance of all polices on a *test data set* collected at a different period of time. In all cases, both training and test data sets contain traces from several thousands to tens of thousands of queries.

5.4.2 Reissues

We first evaluate the effect of using per-stage reissues within Kwiken's framework. Figure 5.5a plots Kwiken's improvements on the end-to-end latency due to reissues, using the SumVar algorithm described in Section 5.3.1. The x-axis depicts the fraction of additional resources provided to reissues and the y-axis shows the fractional reduction at the 99th percentile. The solid line shows the mean improvement over the 45 most frequent DAGs at Microsoft Bing; the dashed lines represent the spread containing the top 25% and bottom 75% of DAGs and the dotted lines show the improvements for the top 10% and bottom 90% of DAGs (sorted with respect to percentage improvement). The circles on the right edge depict latency improvements with a budget of 100%.

We see that Kwiken improves 99th percentile of latency by about 29%, on average, given a reissue budget of 5%. This is almost half the gain that would be achieved if all requests at all stages were reissued, i.e., a budget of 100%. This indicates that Kwiken ekes out most of the possible gains, i.e., identifies laggards and tries to replace them with faster reissues, with just a small amount of budget. Most DAGs see gains



(a) Improvement in 99th latency percentile

(b) Average reduction in different latency percentiles

Figure 5.5: Reduction in latency using per-stage reissues in Kwiken.

but some see a lot more than the others; the top 10% of DAGs improve by 55% while the top 75% of DAGs see at least 10% improvement each. The variation is because different DAGs have different amounts of inherent variance.

Figure 5.5b plots the average gains at several other latency percentiles. As before, we see that small budgets lead to sizable gains and the marginal increase from additional budget is small. This is because some stages with high variance and low cost can be reissued at substantial fraction (e.g., 50%), yet consume only a fraction of total budget. It is interesting though that just a small amount of budget (say 3%) leads to some gains at the median. Observe that higher percentiles exhibit larger improvements, which is consistent with theory (Section 5.3). We note that Kwiken scales efficiently to large DAGs. Computing per-stage models takes about 2 seconds per stage and is parallelizable. Running SumVar takes less than one second for most of the DAGs.

Comparing SumVar to other benchmarks. First, we compare against the current reissue strategy used in Microsoft Bing. The actual reissue timeout values used in Microsoft Bing are very conservative. The additional cost is only 0.2% and reduces 99th percentile of latency only by 3%. The timeouts are so conservative because without an end-to-end framework such as Kwiken, it is hard to reason about how much of the overall capacity should be used for reissues at each stage.

Second, we compared with several straw man policies. One policy would assign each stage the same reissue fraction $r_i = r$. However such policy has clear shortcomings; for example, if a single stage has high cost c_i it will absorb most of the budget. If that stage has low variance, then the resulting end to end improvement will be negligible. Other policies like allocating equal budget to each stage exhibit similar drawbacks.

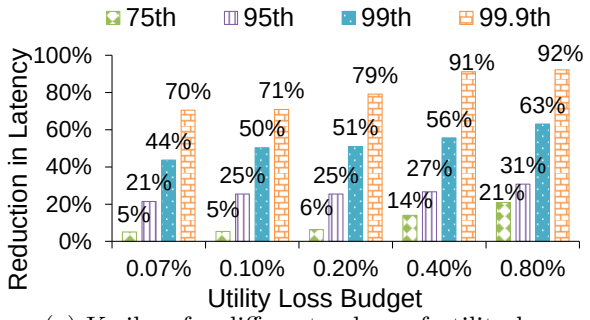
Finally, lacking an optimal algorithm (recall that even (5.3) has a non-convex objective), we compare with two brute-force approaches. For a subset of nine smaller DAGs and budgets from 1% to 10%, we pick the best timeouts out of $10,000$ random budget allocations. Compared to training Kwiken on the same data, this algorithm was about 4 orders of magnitude slower. Hence, we did not attempt it on larger DAGs. Here, Kwiken's results were better on average by 2%; in 95% of the cases (i.e., {DAG, budget} pairs), Kwiken's latency reduction was at least 94% of that achieved by this method. The second approach uses gradient-descent to directly minimize the 99th percentile of the end-to-end latency using a simulator (i.e., avoiding the sum of variances approximation). This method was equally slow and performed no better. Hence, we conclude that Kwiken's method to apportion budget across stages is not only useful but also perhaps nearly as effective as an ideal (impractical) method.

We also evaluated two weighted forms of Kwiken that more directly consider the structure of the DAG (Section 5.2): weighting each stage by its average latency and by its likelihood to occur on a critical path. While both performed well, they were not much better than the unweighted form for the examined DAGs.

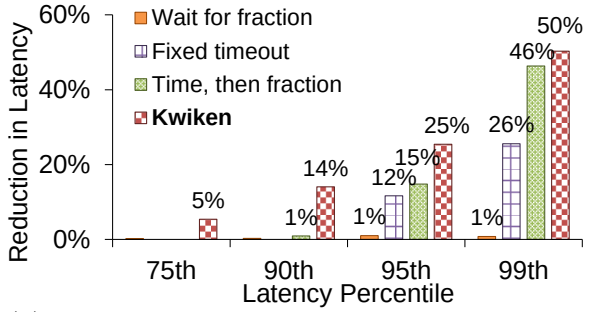
5.4.3 Trading off completeness for latency

Next, we evaluate the improvements in latency when using Kwiken to return partial answers. Figure 5.6a plots the improvement due to trading off completeness for latency for different values of utility loss. Recall that our target is to be complete enough that the best result is returned for over 99.9% of the queries, i.e., a utility loss of 0.1%. With that budget, we see that Kwiken improves the 99th (95th) percentile by around 50% (25%). The plotted values are averages over the web, image and video stages. Recall that these stages issue many requests in parallel and aggregate the responses (see Figure 2.1).

Figure 5.6b compares the performance of Kwiken with a few benchmarks for utility loss budget of 0.1%: *wait-for-fraction* terminates a query when b fraction of its responders return, *fixed-timeout* terminates queries at T_{cutoff}, and *time-then-fraction*



(a) Kwiken for different values of utility loss



(b) Kwiken vs. other variants for utility loss = 0.1%

Figure 5.6: Latency reductions achieved by Kwiken and variants when trading off completeness for latency.

terminates queries when both these conditions hold: a constant T' time has elapsed and at least α fraction of responders finish.

We see that Kwiken performs significantly better. *Wait-for-fraction* spends significant part of budget on queries which get the required fraction relatively fast, hence slower queries that lie on the tail do not improve enough. *Fixed-timeout* is better since it allows slower queries to terminate when many more of their responders are pending but it does not help at all with the quick queries– no change below the 90th percentile. Even among the slower queries, it does not distinguish between queries that have many more pending responders and hence a larger probability of losing utility versus those that have only a few pending responders. *Time-then-fraction* is better for exactly this reason; it never terminates queries unless a minimal fraction of responders are done. However, Kwiken does even better; by waiting for extra time after a fraction of responders are done it provides gains for both the quicker queries and variable amounts of waiting for the slower queries. Also, it beats *time-then-fraction* on the slowest queries by stopping at a fixed time.

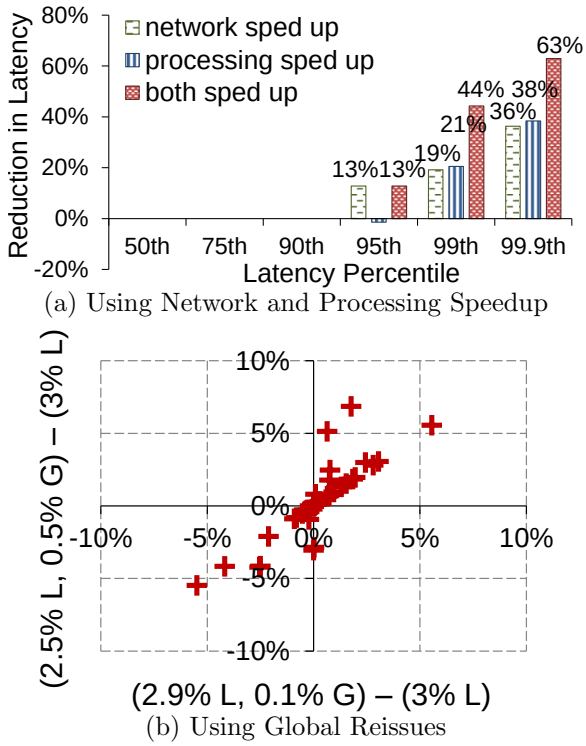


Figure 5.7: Latency improvements from using different catch-up techniques.

5.4.4 Catch-up

Here, we estimate the gains from the three types of catch-up mechanisms discussed earlier. Figure 5.7a shows the gains of using multi-threaded execution and network prioritization on the web search DAG (Figure 2.1), relative to the baseline where no latency reduction techniques are used. We note that the speedup due to multi-threading is not linear with the number of threads due to synchronization costs and using 3 threads yields roughly a 2X speed up. We see that speeding up both stages offers much more gains than speeding up just one of the stages; the 99th percentile latency improves by up to 44% with only small increases in additional load – about 6.3% more threads needed and about 1.5% of the network load moves into higher priority queues.

Next, we evaluate the usefulness of using global reissues on DAGs. Using a total reissue budget of 3%, Figure 5.7b plots the marginal improvements (relative to using the entire budget for local reissues) from assigning $\frac{1}{30}$th (x-axis) vs. assigning $\frac{1}{6}$th of the budget to global reissues (y-axis), for the 45 DAGs we analyze. The average reduction in 99th percentile latency is about 3% in both cases, though, assigning $\frac{1}{6}$ of budget leads to higher improvements in some cases. Overall, 37 out of the 45 DAGs

see gains in latency. We end by noting that this experiment only shows one way to assign global reissues; better allocation techniques may yield larger gains.

5.4.5 Putting it all together

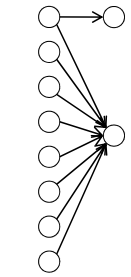
To illustrate the advantage of using multiple latency reduction techniques in the Kwiken framework together, we analyze in detail its application to a major DAG in Microsoft Bing that has 150 stages. A simplified version of the DAG with only the ten highest-variance stages is shown in Figure 5.8a. In three of the stages, we use utility loss to improve latency and use reissues on all stages.

We compare Kwiken with other alternatives in Figure 5.8b; on left, we fix utility loss budget at 0.1% and vary reissues, on right, we vary utility loss and fix reissues at 3%. We see complementary advantage from using reissues and utility loss together. In the left graph, using Kwiken with reissues only at 10% performs worse than using both reissues at 1% and 0.1% utility loss. Also, using both together is about 20% better than using just utility loss. Graph on the right shows that, with reissue budget at 3%, increasing utility loss has very little improvements beyond .1%. We observe that the larger the reissue budget, the larger the amount of utility loss that can be gainfully used (not shown). Further, how we use the budget also matters; consider *K for reissues; wait-for-fraction* on the left. For the same amount of utility loss, Kwiken achieves much greater latency reduction.

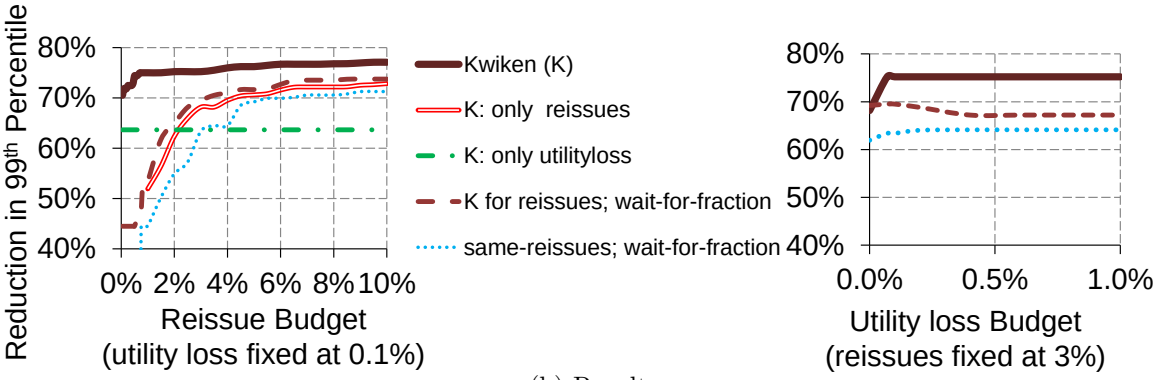
So what does Kwiken do to get these gains? Figure 5.8c shows for each of the ten stages, the latency variance (as a fraction of all variance in the DAG) and the amount of allocated budget (in log scale). We see that the budget needs to be apportioned to many different stages and not simply based on their variance, but also based on the variance-response curves and the per-stage cost of request reissue. Without Kwiken, it would be hard to reach the correct assignment.

5.4.6 Robustness of parameter choices

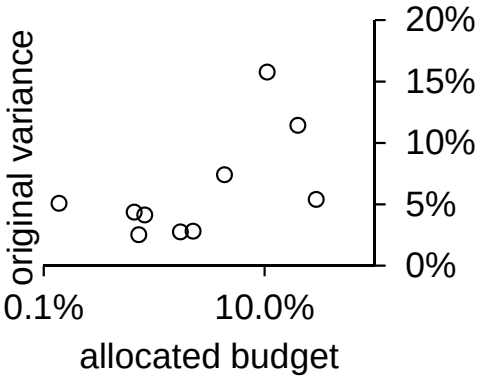
Recall that Kwiken chooses its parameters based on traces from prior execution. A concern here is that due to temporal variations in our system, the chosen parameters might not yield the gains that are expected from our optimization or may violate resource budgets.



(a) Web search DAG

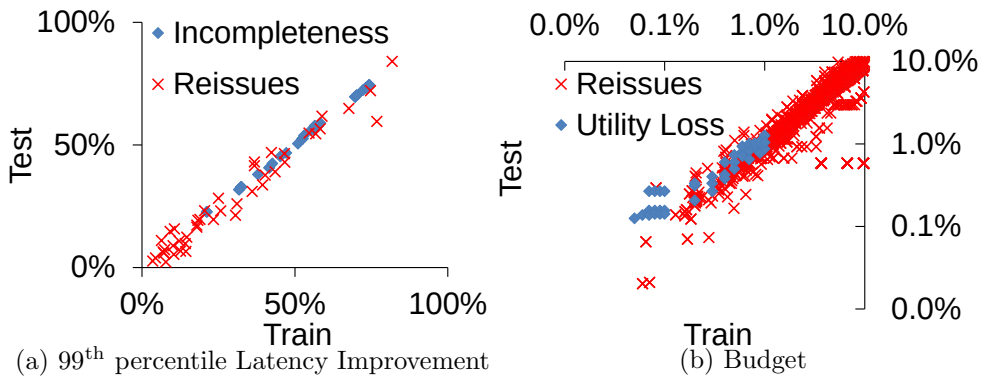


(b) Results



(c) Allocation

Figure 5.8: A detailed example to illustrate how Kwiken works.



(a) 99th percentile Latency Improvement (b) Budget

Figure 5.9: For 45 DAGs, this figure compares the 99th percentile latency improvement and budget values of the training and test datasets.

To understand the stability of the parameter choices over time, we compare the improvements for 99th percentile latency and the budget values obtained for the "training" dataset, to those obtained for the "test" datasets. The test datasets were collected from the same production cluster on three different days within the same week. Figure 5.9 shows that the latency improvements on the test datasets are within a few percentage points off that on the training datasets. The utility loss on the test dataset is slightly larger but predictably so, which allows us to explicitly account for it by training with a tighter budget. In all, we conclude that Kwiken's parameter choices are stable. Also, reallocating budget is fast and can be done periodically whenever the parameters change.

5.5 Discussion

Kwiken uses three different techniques to speed up web services at runtime – reissues, incomplete responses and catch-up. Each of these techniques rely on certain assumptions about how web services are architected:

(a) The technique of reissues assumes that multiple replicas of a particular stage exist and uses them to start duplicate copies of the original request. We note that most modern web services consist of such replicas in order to support traffic bursts and to tolerate failures.

(b) Incomplete responses are useful only for stages where not all responses are necessary to create a "useful" response to the user. This assumption generally holds for

many-way parallel stages such as search, where multiple responses are aggregated and only a certain fraction of them are actually sent back to the user. Kwiken applies this technique only to such stages and the quality loss can be bounded by specifying the necessary quality loss budget.

(c) The technique of catch-up assumes that it is possible to speed up requests by providing additional resources to them or changing their priority. This can be achieved in stages with multi-threaded implementations or which maintain (application-level or network-level) queues. We found that these assumptions were true for a variety of stages that make up the web services at Microsoft Bing.

We note that the end-to-end optimization framework in Kwiken is not limited to the above techniques. It can be extended to a variety of per-stage latency reduction techniques (e.g., [105]) applied to DAG-structured applications, to limit the cost incurred by them to within a cost budget while optimizing for the end-to-end tail latencies.

5.6 Related work

Improving latency of datacenter applications has attracted much recent interest from both academia and industry. Most work in this area [20, 21, 110–112] focuses on developing transport protocols to ensure network flows meet specified deadlines. Approaches like Chronos [113] modify end-hosts to reduce operating system overheads. Kwiken is complementary to these mechanisms, which reduce the latency of individual stages, because it focuses on the end-to-end latency of distributed applications.

Some recent work [16, 32, 114] reduces the job latency in (MapReduce-like) batch processing frameworks [22] by adaptively reissuing tasks or changing resource allocations. Other prior work [102] explores how to (statically) schedule jobs, that are modeled as a DAG of tasks to minimize completion time. Neither of these apply directly to the context of Kwiken which targets large DAGs that can finish within a few hundreds of milliseconds (in the case of web services) and may involve thousands of servers. Static scheduling is relatively easy here and there is too little time to monitor detailed aspects at runtime (e.g., task progress) which was possible in the batch case. Some recent work concludes that latency variability in cloud environments arises from contention with co-located services [115] and provides workload placement strategies

to avoid interference [116]. However, unlike Kwiken, such techniques do not apply to private datacenters.

Some of the techniques used by Kwiken have been explored earlier. Reissuing requests has been used in many distributed systems [65, 117] and networking [105, 118, 119] scenarios. Kwiken's contribution lies in strategically apportioning reissues across the stages of a DAG to reduce end-to-end latency whereas earlier approaches consider each stage independently. Partial execution has been used in AI [120] and programming languages [121, 122]. The proposed policies, however, do not translate to the distributed services domain. Closer to us is Zeta [123], which devises an application-specific scheduler that runs beside the query to estimate expected utility and to choose when to terminate. In contrast, Kwiken relies only on opaque indicators of utility and hence the timeout policies are more generally applicable.

5.7 Conclusion

We propose and evaluate Kwiken, a framework for optimizing the end-to-end runtime latency in datacenter application DAGs. Kwiken takes a holistic approach by considering end-to-end costs and benefits of applying various latency reduction techniques and decomposes the complex optimization problem into a much simpler optimization over individual stages. We also propose novel policies that trade off utility loss and latency reduction. Overall, using detailed simulations based on traces from Microsoft Bing, we show that Kwiken improves the 99th latency percentile by over 75% when just 0.1% of the responses are allowed to have partial results and 3% extra resources are used for reissues.

Chapter 6

Conclusions and future work

In this dissertation, we proposed Concorde, a scheduling framework which uses coordination between different stages and sub-systems in datacenter applications to improve their end-to-end latencies. To this end, Concorde uses the abstraction of latency-response functions to model how application latency varies based on the resources allocated to it. These response functions are determined using application characteristics, which are derived from application execution history or runtime profiling. Using these response functions and application requirements (e.g., completion time goals, resource constraints etc.), Corral, the planning component in Concorde, determines how many resources should be allocated to each application and where in the cluster these resources should be allocated. Corral uses coordination between the storage and compute sub-systems to improve application end-to-end latency.

The cluster scheduler in Concorde uses the output of Corral to determine how should resources be allocated to the applications. Next, the online component in Concorde, Kwiken, uses various latency reduction techniques to speed up the applications at runtime. In particular, we explore the use of (a) request reissues, where a duplicate of the original request is started at a replica of the stage processing it, (b) partial responses, where many-way parallel stages are terminated before all the responses are collected, and (c) catch-up, where laggard requests is given extra resources (e.g., more threads) in later stages of the application. Each of these techniques has a cost associated with them. Kwiken uses an end-to-end optimization framework which coordinates across all stages in the application to limit this cost within a given budget, while optimizing for the end-to-end tail latencies.

With large-scale simulations and deployments on a cluster consisting of 210 machines, we use production traces from Microsoft Bing and Microsoft Cosmos to show that Concorde can result in significant performance improvements compared to the state-of-the-art. This validates our thesis that coordination and joint scheduling across different stages and sub-systems in datacenter applications is required to optimize for

their goals on end-to-end latency.

In the rest of this chapter, we describe various extensions to our work and the future directions this dissertation opens up.

6.1 Future work

Coordinating compute with network flow-level scheduling. While the techniques in Concorde use coordination between the storage and compute sub-systems of datacenter applications, they are agnostic to the way network flows are scheduled for such applications. Preliminary experiments (Section 4.5.7) show that using techniques in Concorde along with flow-level scheduling techniques results in significantly better performance than either of them used separately. We can potentially perform even better when the decisions in Concorde take the underlying flow-level schedule into account. For example, Varys [18] uses a *shortest-bottleneck-first* heuristic to schedule network flows. Scheduling jobs within a rack in the same order can result in better end-to-end job latencies as opposed to scheduling in other orders. The research challenge here is to ensure that extensions to Concorde are not specific to a particular flow-level scheduler but can work well with any scheduler.

Coordination across storage hierarchies. Concorde models datacenter storage as consisting of one or a few disks per machine in the cluster. However, in practice the storage system is hierarchical with several tiers [124] — a remote or long term storage infrastructure (e.g., [125]), multiple disks and SSDs per machine, and the memory. Coordinating the placement of data across this storage hierarchy can lead to significant benefits for datacenter applications. For example, systems like Spark [10], and RAMCloud [126] use memory for data storage resulting in significant application speed-up. However, the amount of memory available across machines in a datacenter is much smaller than the SSD or disk storage space. Hence, it is important to use this resource efficiently and determine which applications can actually benefit with data pre-loaded into the memory. For example, MapReduce jobs which are I/O bound can benefit from this optimization but not those which are CPU bound.

Thus, to achieve optimal performance from datacenter storage, it would be interesting to understand how to coordinate between the different storage hierarchies, determine which tier a particular data set or file should be stored on, and develop prefetching techniques to move data between tiers based on application characteristics

and requirements.

Pricing based on application requirements. Today, tenants in public clouds pay based on the amount of time they use compute resources. For instance, with today's setup, the cost for a tenant renting N VMs for T hours is $k_v \cdot NT$, where k_v is the hourly VM price. Such *resource-based pricing* can be extended to multiple resources. However, this results in a mismatch of tenant and provider interests. The cheapest resource tuple to achieve the tenant's goal may not concur with the provider's preferred resource combination. Further, resource based pricing entails tenants have to pay based on a job's *actual* completion time. Hence, from a pricing perspective, there is a disincentive for the provider to reduce the completion time.

Concorde uses application goals (e.g., completion time or deadline requirements) to determine the resources to be allocated to the application. This opens up interesting opportunities to investigate new pricing models. By decoupling the tenants from the underlying resources, Concorde offers the opportunity of moving away from resource based pricing. Instead, tenants could be charged based only on the characteristics of their job, the input data size and the desired completion time. Such *job-based pricing* can benefit both entities. Tenants specify *what* they desire and are charged accordingly; providers decide *how* to efficiently accommodate the tenant request based on job characteristics and current datacenter utilization. Further, as the final price does not depend on the completion time, providers now have an incentive to complete tenant jobs on time, possibly even *earlier* than the desired time (Section 4.4.4). Such a pricing model can enable a symbiotic tenant provider relationship where tenants benefit due to fixed costs upfront and better-than-desired performance while providers use the increased flexibility to improve goodput and, consequently, total revenue.

Accounting for WAN delays in executing user requests. Concorde uses coordination across multiple application components to improve the datacenter-side latency of applications like web services. However, the latency experienced by the end users of such applications also includes the time spent by the query over the Internet or the wide-area network (WAN) before reaching the datacenter. It is also important to consider this time while processing queries as it can account for a substantial part of the user-experienced latency [31].

Concorde can be extended to account for the WAN latencies by using techniques such as Timecard [31]. Timecard keeps track of the time elapsed since the user sends a request to when it reaches the datacenter and an estimate of the time the response would take to get back to the user. These latencies can be used as part of tech-

niques like catch-up (Section 5.3.3) to determine when to provide extra resources for a query. For example, queries whose WAN latency is greater than a certain threshold (e.g., 400ms) can be assigned multiple threads to reduce their datacenter-side latency. Thus, utilizing knowledge of external WAN latency to perform scheduling decisions within the datacenter is an interesting avenue for future research. This can lead to significantly lower user-experienced latencies.

Runtime techniques to speed up data analytics applications. In this dissertation, the focus of the latency reduction techniques developed in Kwiken has been web services. However, those techniques can also be applied to data analytics applications. Prior work has investigated the use of reissues (called *task cloning*) [104] for improving the latency of MapReduce workloads. The use of such techniques for general DAG-style data analytics workloads remains unexplored. In particular, the following research problem is interesting — given at most x% tasks in a data analytics DAG can be reissued (or cloned), how do we apportion x across the various stages or jobs in the DAG? The challenge here lies in determining which stages would benefit more from reissues and how to change the budget apportioning based on runtime issues such as data skew [61]. Further, when running multiple such DAGs in a cluster, it would be important to decide how would the overall reissue budget be partitioned across DAGs and if a slot is available in the cluster, would it be used to run the reissue of an already running task or a new task.

References

[1] "The Internet of Things: How the Next Evolution of the Internet is Changing Everything," http://www.cisco.com/web/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf.

[2] "Forecast for global shipments of tablets, laptops and desktop PCs from 2010 to 2019," http://www.statista.com/statistics/272595/global-shipments-forecast-for-tablets-laptops-and-desktop-pcs/.

[3] "Google Processing 20,000 Terabytes A Day, And Growing," http://tinyurl.com/26lb8p7.

[4] "Facebook data grows by over 500 TB daily," http://tinyurl.com/96d8oqj/.

[5] D. Duellmann, "Examples of future large scale scientific databases," in *Extremely Large Databases Workshop*, 2010.

[6] E. M. Marcotte and S. V. Date, "Exploiting big biology: Integrating large-scale biological data for function inference," *Briefings in Bioinformatics*, vol. 2, no. 4, pp. 363–374, 2001.

[7] L. A. Barroso and U. Hölzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*, 2009.

[8] "Apache Hadoop," http://hadoop.apache.org/.

[9] "Apache Hive," https://hive.apache.org/.

[10] M. Zaharia, N. M. M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-53, May 2010. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-53.html

[11] R. Chaiken, B. Jenkins, P. Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou, "SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets," in *VDLB*, 2008.

[12] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan, "Speeding up Distributed Request-Response Workflows," in *ACM SIGCOMM*, 2013.

[13] J. R. Dabrowski and E. V. Munson, "Is 100 Milliseconds Too Fast?" in *CHI*, 2001.

[14] E. Schurman and J. Brutlag, "The User and Business Impact of Server Delays, Additional Bytes, and Http Chunking in Web Search," http://velocityconf.com/velocity2009/public/schedule/detail/8523, 2009.

[15] J. Brutlag, "Speed Matters for Google Web Search," http://googleresearch.blogspot.com/2009/06/speed-matters.html, 2009.

[16] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca, "Jockey: Guaranteed Job Latency in Data Parallel Clusters," in *Eurosys*, 2012.

[17] P. Bodik, I. Menache, J. S. Naor, and J. Yaniv, "Brief Announcement: Deadline-Aware Scheduling of Big-Data Processing Jobs," in *SPAA*, June 2014.

[18] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient Coflow Scheduling with Varys," in *ACM SIGCOMM*, 2014.

[19] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron, "Decentralized Task-aware Scheduling for Data Center Networks," in *ACM SIGCOMM*, August 2014.

[20] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron, "Better Never than Late: Meeting Deadlines in Datacenter Networks," in *ACM SIGCOMM*, 2011.

[21] C. Y. Hong, M. Caesar, and P. B. Godfrey, "Finishing Flows Quickly with Preemptive Scheduling," in *ACM SIGCOMM*, 2012.

[22] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI*, 2004.

[23] "Facebook," https://www.facebook.com/.

[24] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive - A Warehousing Solution Over a Map-Reduce Framework," in *VLDB*, 2009.

[25] "Google," https://www.google.com/.

[26] "Analytics. Powered by the AWS Cloud." http://aws.amazon.com/solutions/case-studies/analytics/.

[27] "Hadoop Distributed Filesystem," http://hadoop.apache.org/hdfs.

[28] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig Latin: A Language for Data Processing," in *SIGMOD*, 2008.

[29] A. Singla, B. Chandrasekaran, P. B. Godfrey, and B. Maggs, "The Internet at the Speed of Light," in *Hotnets*, 2014.

[30] "Workshop on Reducing Internet Latency," http://www.internetsociety.org/latency2013.

[31] L. Ravindranath, J. Padhye, R. Mahajan, and H. Balakrishnan, "Timecard: Controlling User-perceived Delays in Server-based Mobile Applications," in *SOSP*, 2013.

[32] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the Outliers in MapReduce Clusters Using Mantri," in *OSDI*, 2010.

[33] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica, "PACMan: Coordinated Memory Caching for Parallel Jobs," in *NSDI*, 2012.

[34] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris, "Scarlett: Coping with Skewed Content Popularity in Mapreduce Clusters," in *EuroSys*, 2011.

[35] M. Chowdhury, S. Kandula, and I. Stoica, "Leveraging Endpoint Flexibility in Data-Intensive Clusters," in *ACM SIGCOMM*, 2013.

[36] V. Jalaparti, H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, "Bridging the Tenant-Provider Gap in Cloud Services," in *SOCC*, 2012.

[37] "Mumak: Map-Reduce Simulator," http://bit.ly/MoOax.

[38] G. Wang, A. R. Butt, P. Pandey, and K. Gupta, "A Simulation Approach to Evaluating Design Decisions in MapReduce Setups," in *MASCOTS*, 2009.

[39] A. Ganapathi, Y. Chen, A. Fox, R. H. Katz, and D. A. Patterson, "Statistics-Driven Workload Modeling for the Cloud," in *SMDB*, 2010.

[40] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar, "Network-Aware Scheduling for Data-Parallel Jobs: Plan When You Can," in *ACM SIGCOMM*, 2015.

[41] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou, "Reoptimizing Data Parallel Computing," in *NSDI*, 2012.

[42] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, "Towards Predictable Datacenter Networks," in *ACM SIGCOMM*, 2011.

[43] K. P. Belkhale and P. Banerjee, "An Approximate Algorithm for the Partitionable Independent Task Scheduling Problem," *Urbana*, vol. 51, p. 61801, 1990.

[44] J. Du and J. Y.-T. Leung, "Complexity of Scheduling Parallel Task Systems," *SIAM J. Discret. Math.*, 1989.

[45] J. Turek, J. L. Wolf, and P. S. Yu, "Approximate Algorithms Scheduling Parallelizable Tasks," in *SPAA*, 1992.

[46] "Hadoop YARN Project," http://tinyurl.com/bnadg9l.

[47] Y. Chen, A. Ganapathi, R. Griffith, and Y. Katz, "The Case for Evaluating MapReduce Performance Using Workload Suites," in *MASCOTS*, 2011.

[48] "TPC Benchmark H," http://www.tpc.org/tpch/.

[49] "Hadoop mapreduce next generation - capacity scheduler," http://hadoop.apache.org/docs/r2.2.0/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html.

[50] "Bing," http://www.bing.com/.

[51] S. Rao, R. Ramakrishnan, A. Silberstein, M. Ovsiannikov, and D. Reeves, "Sailfish: A framework for large scale data processing," in *SOCC*, 2012.

[52] K. Elmeleegy, "Piranha: Optimizing Short Jobs in Hadoop," *Proc. VLDB Endow.*, 2013.

[53] T. White, *Hadoop: The Definitive Guide*. O'Reilly, 2009.

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

[55] N. Joukov, A. Traeger, R. Iyer, C. P. Wright, and E. Zadok, "Operating System Profiling via Latency Analysis," in *OSDI*, 2006.

[56] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal, "Dynamic instrumentation of production systems," in *USENIX ATC*, 2004.

[57] M. Kremer and J. Gryz, "A Survey of Query Optimization in Parallel Databases," York University, Tech. Rep., 1999.

[58] "Apache Tez," http://hortonworks.com/hadoop/tez/.

[59] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "SkewTune: Mitigating Skew in Mapreduce Applications," in *ACM SIGMOD*, 2012.

[60] J. Lin, "The Curse of Zipf and Limits to Parallelization: A Look at the Stragglers Problem in MapReduce," in *Large-Scale Distributed Systems for Information Retrieval*, 2009.

[61] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "Skew-Resistant Parallel Processing of Feature-Extracting Scientific User-Defined Functions," in *SOCC*, 2010.

[62] S. Ibrahim, H. Jin, L. Lu, S. Wu, B. He, and L. Qi, "LEEN: Locality/Fairness-Aware Key Partitioning for MapReduce in the Cloud," in *CLOUDCOM*, 2010.

[63] E. Lazowska, J. Zahorjan, S. Graham, and K. Sevcik, *Quantitative system performance: computer system analysis using queuing network models*, 1984.

[64] E. Krevat, J. Tucek, and G. R. Ganger, "Disks Are Like Snowflakes: No Two Are Alike," in *HotOS*, 2011.

[65] J. Dean and L. A. Barroso, "The Tail at Scale," *Commun. ACM*, vol. 56, no. 2, pp. 74–80, Feb. 2013.

[66] "Cloudera: Hadoop and Big Data," http://www.cloudera.com/content/cloudera/en/about/hadoop-and-big-data.html.

[67] H. Herodotou, F. Dong, and S. Babu, "No One (Cluster) Size Fits All: Automatic Cluster Sizing for Data-intensive Analytics," in *ACM SOCC*, 2011.

[68] K. Morton, M. Balazinska, and D. Grossman, "ParaTimer: A Progress Indicator for MapReduce DAGs," in *SIGMOD*, 2010.

[69] F. Tian and K. Chen, "Towards Optimal Resource Provisioning for Running MapReduce programs in Public Cloud," in *CLOUD*, 2011.

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

[71] A. Verma, L. Cherkasova, and R. Campbell, "ARIA: Automatic Resource Inference and Allocation for Mapreduce Environments," *ICAC*, 2011.

[72] A. Li, X. Zong, S. Kandula, X. Yand, and M. Zhang, "CloudProphet: Towards Application Performance Prediction in Cloud," in *SIGCOMM (Poster)*, 2011.

[73] D. Tertilt and H. Krcmar, "Generic Performance Prediction for ERP and SOA Applications," in *ECIS*, 2011.

[74] 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 *EuroSys*, 2010.

[75] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: Fair Scheduling for Distributed Computing Clusters," in *SOSP*, 2009.

[76] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar, "ShuffleWatcher: Shuffle-aware Scheduling in Multi-tenant MapReduce Clusters," in *USENIX ATC*, 2014.

[77] D. Xie, N. Ding, Y. C. Hu, and R. Kompella, "The Only Constant is Change: Incorporating Time-varying Network Reservations in Data Centers," in *ACM SIGCOMM*, 2012.

[78] A. Li, X. Zong, S. Kandula, X. Yang, and M. Zhang, "CloudProphet: Towards Application Performance Prediction in Cloud," in *ACM SIGCOMM*, 2011.

[79] C. Stewart and K. Shen, "Performance Modeling and System Management for Multi-component Online Services," in *NSDI*, 2005.

[80] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi, "An Analytical Model for Multi-tier Internet Services and Its Applications," in *ACM SIGMETRICS*, 2005.

[81] N. R. Devanur, K. Jain, B. Sivan, and C. A. Wilkens, "Near Optimal Online Algorithms and Fast Approximation Algorithms for Resource Allocation Problems," in *EC*, 2011.

[82] A. Kamath, O. Palmon, and S. Plotkin, "Routing and Admission Control in General Topology Networks with Poisson Arrivals," in *ACM-SIAM SODA*, 1996.

[83] A. Borodin and R. El-Yaniv, *Online Computation and Competitive Analysis*. Cambridge University Press, 2005.

[84] S. Lee, R. Panigrahy, V. Prabhakaran, V. Ramasubramanian, K. Talwar, L. Uyeda, and U. Wieder, "Validating Heuristics for Virtual Machines Consolidation," MSR, Tech. Rep. MSR-TR-2011-9, 2011.

[85] R. L. Graham, "Bounds on Multiprocessing Timing Anomalies," *SIAM Journal on Applied Mathematics*, vol. 17, no. 2, pp. 416–429, 1969.

[86] "Tom's Hardware Blog," http://bit.ly/rkjJwX.

[87] A. Shieh, S. Kandula, A. Greenberg, and C. Kim, "Sharing the Datacenter Network," in *NSDI*, 2011.

[88] "Amazon's EC2 Generating 220M+ Annually," http://bit.ly/8rZdu.

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

[90] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly, "The Quantcast File System," *Proc. VLDB Endow.*

[91] "ORC File Format," http://docs.hortonworks.com/HDPDocuments/HDP2/HDP-2.0.0.2/ds_Hive/orcfile.html.

[92] H. Herodotou, F. Dong, and S. Babu, "No One (Cluster) Size Fits All: Automatic Cluster Sizing for Data-intensive Analytics," in *SOCC*, 2011.

[93] R. Agarwal, A. Khandelwal, and I. Stoica, "Succinct: Enabling queries on compressed data," in *NSDI*, 2015.

[94] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource Packing for Cluster Schedulers," in *ACM SIGCOMM*, 2014.

[95] B. Cho, M. Rahman, T. Chajed, I. Gupta, C. Abad, N. Roberts, and P. Lin, "Natjam: Design and Evaluation of Eviction Policies for Supporting Priorities and Deadlines in Mapreduce Clusters," in *SOCC*, 2013.

[96] M. Y. Eltabakh, Y. Tian, F. Özcan, R. Gemulla, A. Krettek, and J. McPherson, "CoHadoop: Flexible Data Placement and Its Exploitation in Hadoop," *Proc. VLDB Endow.*, 2011.

[97] R. T. Kaushik, M. Bhandarkar, and K. Nahrstedt, "Evaluation and Analysis of GreenHDFS: A Self-Adaptive, Energy-Conserving Variant of the Hadoop Distributed File System," in *IEEE Second International Conference on Cloud Computing Technology and Science*, 2010.

[98] B. Palanisamy, A. Singh, L. Liu, and B. Jain, "Purlieus: Locality-aware Resource Allocation for MapReduce in a Cloud," in *SC*, 2011.

[99] M. Li, D. Subhraveti, A. R. Butt, A. Khasymski, and P. Sarkar, "CAM: A Topology Aware Minimum Cost Flow Based Resource Manager for MapReduce Applications in the Cloud," in *HPDC*, 2012.

[100] H. Alkaff, I. Gupta, and L. M. Leslie, "Cross-Layer Scheduling in Cloud Systems," in *IEEE International Conference on Cloud Engineering, IC2E*, 2015.

[101] K. LaCurts, S. Deng, A. Goyal, and H. Balakrishnan, "Choreo: Network-aware Task Placement for Cloud Applications," in *IMC*, 2013.

[102] Y. Kwok and I. Ahmad, "Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors," *ACM Computing Surveys (CSUR)*, 1999.

[103] R. Lepère, D. Trystram, and G. J. Woeginger, "Approximation Algorithms for Scheduling Malleable Tasks Under Precedence Constraints," *International Journal of Foundations of Computer Science*, vol. 13, no. 04, pp. 613–627, 2002.

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

[105] A. Vulimiri, O. Michel, P. B. Godfrey, and S. Shenker, "More is Less: Reducing Latency via Redundancy," in *HotNets*, 2012.

[106] X. S. Wang et al., "Demystifying Page Load Performance with WProf," in *NSDI*, 2013.

[107] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh, "AppInsight: Mobile App Performance Monitoring in the Wild," in *OSDI*, 2012.

[108] S. Boucheron, G. Lugosi, and O. Bousquet, "Concentration Inequalities," *Advanced Lectures on Machine Learning*, 2004.

[109] H. Wu, Z. Feng, C. Guo, and Y. Zhang, "ICTCP: Incast Congestion Control for TCP in Data Center Networks," in *ACM CONEXT*, 2010.

[110] M. Alizadeh, A. Greenberg, D. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data Center TCP (DCTCP)," in *SIGCOMM*, 2010.

[111] M. Alizadeh, S. Yang, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "Deconstructing Datacenter Packet Transport," in *Hotnets*, 2012.

[112] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. H. Katz, "DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks," in *ACM SIGCOMM*, 2012.

[113] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat, "Chronos: Predictable Low Latency for Data Center Applications," in *SOCC*, 2012.

[114] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, I. Stoica, "Improving MapReduce Performance in Heterogeneous Environments," in *OSDI*, 2008.

[115] "Amazon Web Services," http://aws.amazon.com/.

[116] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey, "Bobtail: Avoiding Long Tails in the Cloud," in *NSDI*, 2013.

[117] G. Decandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo : Amazon's Highly Available Key-value Store," 2007.

[118] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. N. Rao, "Improving web availability for clients with MONET," in *NSDI*, 2005.

[119] D. Han, A. Anand, A. Akella, and S. Seshan, "RPT: Re-architecting Loss Protection for Content-Aware Networks," in *NSDI*, 2012.

[120] S. Zilberstein, "Using Anytime Algorithms in Intelligent Systems," *AI Magazine*, vol. 17, no. 3, pp. 73–83, 1996. [Online]. Available: http://rbr.cs.umass.edu/shlomo/papers/Zaimag96.html

[121] W. Baek and T. M. Chilimbi, "Green: A Framework for Supporting Energy-Conscious Programming using Controlled Approximation," in *PLDI*, 2010.

[122] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard, "Dynamic Knobs for Responsive Power-Aware Computing," in *ASPLOS*, 2011.

[123] Y. He, S. Elnikety, J. Larus, and C. Yan, "Zeta: Scheduling Interactive Services with Partial Execution," in *SOCC*, 2012.

[124] A. Elnably, H. Wang, A. Gulati, and P. Varman, "Efficient QoS for Multi-tiered Storage Systems," in *HotStorage*, 2012.

[125] "Amazon S3," https://aws.amazon.com/s3/.

[126] "RAMCloud," https://ramcloud.stanford.edu/.