Change Management Systems for Seamless Evolution in Data Centers

Omid Alipourfard
Yale University Graduate School of Arts and Sciences, h@omid.io

Abstract

Change Management Systems for Seamless Evolution in Data Centers

Omid Alipourfard

2021

Revenue for data centers today is highly dependent on the satisfaction of their enterprise customers. These customers often require various features to migrate their businesses and operations to the cloud. Thus, clouds today introduce new features at a swift pace to onboard new customers and to meet the needs of existing ones. This pace of innovation continues to grow, e.g., Amazon deployed 1400 new features in 2017 alone.

However, such a rapid pace of evolution adds challenges for both clouds and users. Clouds struggle to keep up with the deployment speed, and users struggle to learn which features they need and how to use them. Three contributions are needed to advance the state of the art, (1) clouds need systematic techniques, instead of rules of thumb, to manage the deployment of new features; and (2) customers need systematic techniques to identify features they need and how to use them. (3) we need adaptable measurement systems that keep up with the pace of innovation.

This dissertation makes original contributions to address the need. In particular, this dissertation introduces Janus to address the first need, and Cherrypick to address the second. Together, they contribute to fundamental techniques to enable continued cloud innovations.

Janus helps data center operators roll out new changes to the data center network. It automatically adapts to the data center topology, routing, traffic, and failure settings. The system reduces the risk of new deployments for network operators as they can now pick deployment strategies which are less likely to impact users' performance.

Cherrypick addresses challenges for users to effectively configure cloud resources

for key cloud usage (i.e., data analytics). It helps users to address a key challenge, how to search through new machine types that clouds are constantly introducing. Being able to adapt to new big-data frameworks and applications, Cheerypick computes cloud configurations that meet users' budget constraints and achieve near-optimal performance.

Our study of measurement algorithms shows that today's measurement algorithms can readily adapt to the pace of innovation. Specifically, today's workloads map well to current and future hardware architectuers. We find that for a wide range of settings simple hash tables often outperform more sophisticated measurement algorithms such as counting sketches.

As the pace of cloud innovations increases, it is critical to have tools that allow operators to deploy new changes as well as those that would enable users to adapt to achieve good performance at low cost. The tools and algorithms discussed in this thesis help accomplish these goals.

Change Management Systems for Seamless Evolution in Data Centers

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by
Omid Alipourfard

Dissertation Director: Yale University

June, 2021

Contents

List of Figures

List of Tables

Preface

I have always tried to be conscious of the decisions that I make in my life. Most often, I have tried to explore the sides that people tell me not to, just to give myself the opportunity to explore things that are not the norm and to feel that I am making conscious decisions for myself. Starting a PhD was a conscious decision for me. As opposed to many people in oppressed countries, I wasn't trying to run away to build a better life for myself. I genuinely was interested in the science and the impact that I could have in the bigger picture—as what I hope is typical with many students that take on this journey.

However, a few years along this path, I forgot myself. I lost consciousness. I lost mindfulness. And I became a slave to what was asked of me. I forgot what "I" was looking for, that killed my passion, and my program became my prison. This happens far too often, especially with young researchers that haven't given life enough chances to prepare them for it.

I am writing this not as a preface to this thesis. But as a preface to all the PhD students that want to take on this journey. If you are reading this, remember that it is far too easy to lose yourself on this journey (and in life). And it is too hard to actively remind yourself that you are bigger than what you believe, than what you are told, and than what you perceive.

Make your PhD a journey of not learning to think but a journey of learning to be conscious of your needs and emotions, a journey about being present, and a journey about being mindful. Frankly, if you get accepted to a PhD program, chances are that you already know how to think and you just need to be taught how to advertise your thinking. But that's the easy part. The hard part, and the part that no one tells you, is how to be there for yourself.

To my mother and brother ...

Chapter 1

Introduction

For the past decade or so, data centers have grown rapidly to keep up with the customer demands. Today, a typical cloud hosts upwards of a hundred thousand machines and offers hundreds of services [2, 3, 4] to its users. Such tight requirements require fast innovations. As an example, every major cloud has built its own dedicated wide-area backbone [5, 6, 7] and has gone through multiple generations of networking fabric [8].

Keeping up with this pace of growth is challenging for both operators and their users. Operators have to deploy new services and devices without disrupting their users. And users are on the clock to adapt to new services to save cost and get better performance.

The motivation for this thesis is the lack of tools that (1) enable safe, fast, and secure deployments for the operators; and (2) tools that enable users to adopt new technologies and services deployed in the cloud. Today, both operators and users rely on processes that are manual, are hard to adapt, and costly. We hope that by automating parts of these processes, we allow operators and users to have an easier time adopting innovations.

Ideally, such tools should (1) be able to adapt to the constantly changing environment of a data center. (2) They need to be accurate so operators and users can rely on them (3) and they should be cheap to use and maintain.

Adaptivity: If tools were tightly coupled to the applications or infrastructure of the data center, they would go obsolete quickly. For example, a tool that suggests cloud configurations (e.g., the type of the virtual machine and the number of such machines) for a SQL database in the cloud is free to use the knowledge of the inner workings of the SQL query planner, e.g., to decide on the RAM size. However, by doing so, the tool becomes tightly coupled to the architecture of the underlying SQL database. That is, if clouds introduced a new noSQL database or newer generations of big data analytics frameworks that heavily rely on fast discs, the tool would not be as effective.

High accuracy: A cloud configuration recommender tool suggesting suboptimal configurations leaves users unhappy. However, achieving both high accuracy and high adaptivity is difficult. Adaptivity needs abstractions and generalizations. Accuracy embraces details. For example, a cloud configuration recommender that knows how the query planner of a SQL database works is likely to make more accurate recommendations than one that does not. However, such a tool will have a harder time adapting to new types of databases.

Low cost: Finally, cloud tools should be cheap to use while being adaptive and accurate. Operators and users define cost in terms of the amount of time and money that they have to spend. For example, in the case of a cloud configuration optimizer, it is possible to run an exhaustive search across all possible configurations (that is accurate and adaptive). However, such a strategy is costly and the search space is not always intractable.

In this thesis, we look into building these tools from three viewpoints. In Cherrypick, we delve into the problem of adapting to the constantly changing landscape of hardware/software stack in the big data analytics world. In Janus, we look into deploying new products in data center networks. And finally, in our study of measurement algorithms for packet processing, we show that some algorithms automatically adapt to the changing environment because the available resources grow with the demand.

1.1 Innovations: the tussle for customers

Users need to adapt to newly introduced devices and services—new systems often solve problems in cheaper and more efficient ways [9]. The trend for introducing new features has rapidly grown in the past years. Amazon has gone from releasing 98 features per year in 2009 to 1400 features in 2017.

It is challenging for the users to keep up with this pace of innovation. For example, finding the optimal configuration to run a recurring big analytics job among 1000s of configurations is challenging and if users are not careful with their configuration, they may end up paying a lot more [10].

However, automatically adapting to the changing environment is a rocky road for two reasons. First, clouds introduce tens of new instances and frameworks every year. Building specialized tools for the cartesian product of all the possible configurations does not scale. Second, cloud customers have different usage patterns and software stacks. This makes it hard to rely on the stability of users' workloads or applications. Ideally, our tools should be able to build and learn a model within a few runs of the users' workload on the cloud.

1.2 Deployments: the tussle for clouds

Today, operators guarantee certain levels of availability and performance [11] for their customers. Amazon EC2 promises 99.95% availability for their virtual machines which translates to 20 minutes a month of downtime budget. The difficulty is that operators have to uphold these guarantees as they roll out new features and services. Any error in rollouts could show as performance degradations or even loss of connectivity for users resulting in revenue and reputation loss for the cloud operator.

New deployments are difficult because of the higher rate of failures and unexpected events. Typically, operators perform new deployments with additional support in terms of software (e.g., additional monitoring), hardware (e.g., additional spare capacity), humans (e.g., operators that are oncall), and time (e.g., slow rollout). Thus, it may take a long time to accrue the support needed to roll out a new feature—often even longer than the time it takes to design the feature.

Further, studying deployment risks is not straightforward and depends on the practices that cloud operators follow. Some operators prefer to buy more infrastructure upfront, so there is enough spare capacity to deal with failures. Others prefer to spend more on software-systems that are more reliable. These variations even occur even within the same cloud: private clouds are managed and built differently than public ones and there are regional differences across data centers due to geopolitical constraints. So ideally, we need tools that let operators pick their operating point where we optimize their settings.

1.3 Building tools that adapt

It is straightforward to build an algorithm that adapts to a changing landscape: A brute force search looking at all the possibilities can always come up with the optimal answer. The problem with this approach is that for almost any interesting problem, a brute force search is intractable (resource or cost-wise). For example, to find a cloud configuration to run a recurring analytical job, we can run the job on all the possible cloud configurations and choose the best one, however, the number of such configurations, typically in the order of thousands, makes a brute force search prohibitively expensive. Similarly, in a data center, to find the best deployment plan for a new product (e.g., a new type of networking switch), we can simulate and measure the risk of every deployment plan, but the number of plans is almost always exponential. In this thesis, we look into ways to make such search problems interactable.

In Cherrypick, we show that by combining modeling and searching, we find cloud configurations that are as good as a brute force search. Cherrypick relies on a feedback loop where modeling moves the focal point of the searching algorithm and the searching output updates the model. This coupling lets us find near-optimal cloud configurations even without having a lot of information about the underlying workloads, configurations, or the behavior of the cloud machines.

In Janus, we focus on the problem of finding a deployment plan for new products in datacenters. Datacenter networks are often super-symmetric (architecturally). Such a symmetric architecture has many benefits from easing the software development to easing the management and monitoring. Janus leverages the symmetry in datacenter topology to reduce the search space exponentially to the point that a brute force search algorithm becomes tractable.

Finally, in our study of measurement algorithms for software, we look into finding a measurement algorithm that adapts to different workloads and traffic distributions. Such an algorithm is crucial in a data center so that it adapts to the workloads of the users and the growth and change in data center traffic demands.

Chapter 2

Cherrypick: Searching for optimal cloud configurations for customer workloads

Picking the right cloud configuration for recurring big data analytics jobs running in clouds is hard, because there can be tens of possible VM instance types and even more cluster sizes to pick from. Choosing poorly can significantly degrade performance and increase the cost to run a job by 2-3x on average, and as much as 12x in the worst-case. However, it is challenging to automatically identify the best configuration for a broad spectrum of applications and cloud configurations with low search cost. To make matters worse, clouds introduce new products every year making it challenging to build a system that adapts to the changing landscape of the cloud. CherryPick is a system that leverages Bayesian Optimization to dynamically build and update performance models for various applications. In Cherrypick, the models are just accurate enough to let a searching algorithm distinguish the best or close-to-the-best configuration from the rest with only a few test runs. Our experiments on five analytic applications in AWS EC2 show that CherryPick has a 45-90% chance to find optimal configurations, otherwise near-optimal, saving up to 75% search cost compared to existing solutions.

2.1 Introduction

Big data analytics running on clouds are growing rapidly and have become critical for almost every industry. To support a wide variety of use cases, a number of evolving techniques are used for data processing, such as Map-Reduce, SQL-like languages, Deep Learning, and in-memory analytics. The execution environments of such big data analytic applications are structurally similar: a cluster of virtual machines (VMs). However, since different analytic jobs have diverse behaviors and resource requirements (CPU, memory, disk, network), their *cloud configurations* – the types of VM instances and the numbers of VMs – cannot simply be unified.

Choosing the right cloud configuration for an application is essential to service quality and commercial competitiveness. For instance, a bad cloud configuration can result in up to 12 times more cost for the same performance target. The saving from a proper cloud configuration is even more significant for *recurring* jobs [12, 13] in which similar workloads are executed repeatedly. Nonetheless, selecting the best cloud configuration, e.g., the cheapest or the fastest, is difficult due to the complexity of simultaneously achieving high accuracy, low overhead, and adaptivity for different applications and workloads.

Accuracy The running time and cost of an application have complex relations to the resources of the cloud instances, the input workload, internal workflows, and configuration of the application. It is difficult to use straightforward methods to model such relations. Moreover, cloud dynamics such as network congestions and stragglers introduce substantial noise [14, 15].

Overhead Brute-force search for the best cloud configuration is expensive. Developers for analytic applications often face a wide range of cloud configuration choices. For example, Amazon EC2 and Microsoft Azure offer over 40 VM instance types with a variety of CPU, memory, disk, and network options. Google provides 18 types and also allows customizing VMs' memory and the number of CPU cores [16]. Addition-

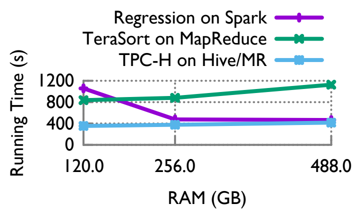


Figure 2.1: Regression and TeraSort with varying RAM size (64 cores)

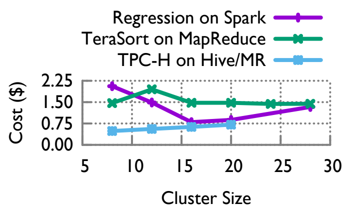


Figure 2.2: Regression and TeraSort cost with varying cluster size (M4)

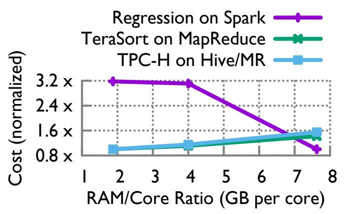


Figure 2.3: Regression and TeraSort cost with varying VM type (32 cores)

ally, developers also need to choose the right cluster size.

Adaptivity Big data applications have diverse internal architectures and dependencies within their data processing pipelines. Manually learning to build the internal structures of individual applications' performance model is not scalable.

Existing solutions do not fully address all of the preceding challenges. For example, Ernest [17] trains a performance model for machine learning applications with a small number of samples but since its performance model is tightly bound to the particular structure of machine learning jobs, it does not work well for applications such as SQL queries (poor adaptivity).
Further, Ernest can only select VM sizes within a given instance family, and performance models need to be retrained for each instance family.

In this chapter, we present *CherryPick*—a system that unearths the optimal or near-optimal cloud configurations that minimize cloud usage cost, guarantee application performance and limit the search overhead for recurring big data analytic jobs. Each configuration is represented as the number of VMs, CPU count, CPU speed per core, RAM per core, disk count, disk speed, and network capacity of the VM.

The key idea of *CherryPick* is to build a performance model that is *just accurate enough* to allow us to distinguish near-optimal configurations from the rest. Tolerating the inaccuracy of the model enables us to achieve both low overhead and adaptivity: only a few samples are needed and there is no need to embed application-specific insights into the modeling.

CherryPick leverages Bayesian Optimization (BO) [1, 18, 19], a method for optimizing black-box functions. Since it is non-parametric, it does not have any predefined format for the performance model. BO estimates a *confidence interval* (the range that the actual value should fall in with high probability) of the cost and running time under each candidate cloud configuration. The confidence interval is improved (narrowed) as more samples become available. *CherryPick* can judge which cloud configuration should be sampled next to best reduce the current uncertainty in modeling and get closer to go the optimal. *CherryPick* uses the confidence interval to decide when to stop the search. Section 2.3 provides more details on how BO works and why we chose BO out of other alternatives.

To integrate BO in *CherryPick* we needed to perform several customizations (Section 2.3.5): i) selecting features of cloud configurations to minimize the search steps; ii) handling noise in the sampled data caused by cloud internal dynamics; iii) selecting initial samples; and iv) defining the stopping criteria.

We evaluate *CherryPick* on five popular analytical jobs with 66 configurations on Amazon EC2. *CherryPick* has a high chance (45%-90%) to pick the optimal configuration and otherwise can find a near-optimal solution (within 5% at the median), while alternative solutions such as coordinate descent and random search can take up to 75% more running time and 45% more search cost. We also compare *CherryPick* with Ernest [17] and show how *CherryPick* can improve search time by 90% and search cost by 75% for SQL queries.

2.2 Background and Motivation

In this section, we show the benefits and challenges of choosing the best cloud configurations.

We also present two strawman solutions to solve this problem.

Application	**Avg/min**	**Max/min**
TPC-DS	3.4	9.6
TPC-H	2.9	12
Regression (SparkML)	2.6	5.2
TeraSort	1.6	3.0

Table 2.1: Comparing the maximum, average, and minimum cost of configurations for various applications.

2.2.1 Benefits

A good cloud configuration can reduce the cost of analytic jobs by a large amount. Table 2.1 shows the arithmetic mean and maximum running cost of configurations compared to the configuration with minimum running cost for four applications across 66 candidate configurations. The details of these applications and their cloud configurations are described in Section 2.5. For example, for the big data benchmark, TPC-DS, the average configuration costs 3.4 times compared to the configuration with minimum cost; if users happen to choose the worst configuration, they would spend 9.6 times more.

Picking a good cloud configuration is even more important for recurring jobs where similar workloads are executed repeatedly, e.g. daily log parsing. Recent studies report that up to 40% of analytics jobs are recurring [12, 13]. Our approach only works for repeating jobs, where the cost of a configuration search can be amortized across many subsequent runs.

2.2.2 Challenges

There are several challenges for picking the best cloud configurations for big data analytics jobs.

Complex performance model: In addition, performance under a cloud configuration is not deterministic. In cloud environments, which is shared among many tenants, stragglers can happen. We measured the running time of TeraSort-30GB on 22 different cloud configurations on AWS EC2 five times. We then computed the coefficient of variation (CV) of the five runs. Our results show that the median of the CV is about 10% and the 90 percentile is above 20%. This variation is not new [13].

Cost model: The cloud charges users based on the amount of time the VMs are up. Using configurations with a lot of resources could minimize the running time, but it may cost a lot more money. Thus, to minimize cost, we have to find the right balance between resource prices and the running time. Figure 2.2 shows the cost of running Regression on SparkML on different cluster sizes where each VM comes with 15 GBs of RAM and 4 cores in AWS EC2. We can see that the cost does not monotonically increase or decrease when we add more resources into the cluster. This is because adding resources may accelerate the computation but also raises the price per unit of running time.

Large searching space: Clouds offer a large number of instance types for users. For example, Amazon EC2 offers over 40 VM instance types. The instances are grouped into instance families such as general purpose (M4), compute optimized (C4), memory optimized (R3), and storage optimized (I2) [2]. Within each instance family, different instances have different amount of virtual CPUs (1-40 cores), memory (0.5-244 GB), network bandwidth (20Mbps-10Gbps), and disks (different amount of local disk and remote Elastic Block Storage [20]). Cloud providers also add new instance types every year [2]. In addition to instance types, users have to pick the right cluster size that minimizes the cost while ensuring a short running time.

Large searching space: Clouds provide a large number of instance types to users. Amazon, for example, offers 40 VM types grouped into general purpose (M4), compute optimized (C4), memory optimized (R3), and disk-optimized (I2). The resource variation across these families together with the newly added instance types every year, making brute force searching the right cloud configuration time consuming and costly.

The heterogeneity of applications: Figure 2.3 shows different shapes for TPC-DS and Regression on Spark and how they relate to instance types. For TeraSort, a low memory instance (8 core and 15 GBs of RAM) performs the best because CPU is a more critical resource. On the other hand for Regression, the same cluster has 2.4 times more running time than the best candidate due to the lack of RAM.

Moreover, the best choice often depends on the application configurations, e.g., the number of map and reduce tasks in YARN. Our work on identifying the best cloud configurations is complementary to other works on identifying the best application configurations (e.g., [21, 22]). *CherryPick* can work with any (even not optimal) application configurations.

2.2.3 Strawman solutions

The two strawman solutions for predicting a near optimal cloud configuration are modeling and searching.

Accurate modeling of application performance. One way is to model application performance and then pick the best configuration based on this model. However, this methodology has poor adaptivity. Building a model that works for a variety of applications and cloud configurations can be difficult because the knowledge of the internal structure of specific applications is needed to make the model effective. Moreover, building a model through human intervention for every new application can be tedious.

Static searching for the best cloud configuration. Another way is to exhaustively search for the best cloud configuration without relying on an accurate performance model. However, this methodology has high overhead. With 40 instance types at Amazon EC2 and tens of cluster sizes for an application, if not careful, one could end up needing tens if not hundreds of runs to identify the best instance. In addition, trying each cloud configuration multiple times to get around the dynamics in the

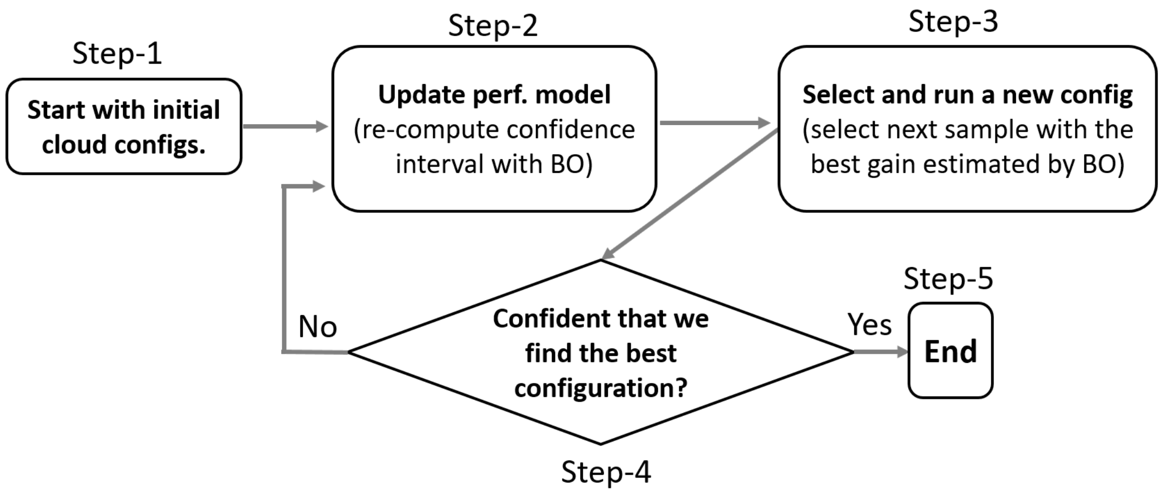

Step-1
Start with initial cloud configs.
Step-2
Update perf. model
(re-compute confidence interval with BO)
Step-3
Select and run a new config
(select next sample with the best gain estimated by BO)
No
Confident that we find the best configuration?
Yes
Step-5
End
Step-4

Figure 2.4: *CherryPick* workflow

cloud (due to resource multiplexing and stragglers) would exacerbate the problem even further.

To reduce the search time and cost, one could use *coordinate descent* and search one dimension at a time. Coordinate descent could start with searching for the optimal CPU/RAM ratio, then the CPU count per machine, then cluster size, and finally disk type. For each dimension, we could fix the other dimensions and search for the cheapest configuration possible. This could lead to suboptimal decisions if for example, because of bad application configuration a dimension is not fully explored or there are local minima in the problem space.

2.3 CherryPick Design

2.3.1 Overview

CherryPick follows a general principle in statistical learning theory [23]: "If you possess a restricted amount of information for solving some problem, try to solve the problem directly and never solve a more general problem as an intermediate step."

In our problem, the ultimate objective is to find the best configuration. We also have a very restricted amount of information, due to the limited runs of cloud configurations we can afford. Therefore, the model does not have enough information to be an accurate performance predictor, but this information is sufficient to find a good configuration within a few steps.

Rather than accurately predicting application performance, we just need a model that is accurate enough for us to separate the best configuration from the rest.

Compared to static searching solutions, we dynamically adapt our searching scheme based on the current understanding and confidence interval of the performance model. We can dynamically pick the next configuration that can best distinguish performance across configurations and eliminate unnecessary trials. The performance model can also help us understand when to stop searching earlier once we have a small enough confidence interval. Thus, we can reach the best configuration faster than static approaches.

Figure 2.4 shows the joint process of performance modeling and configuration searching. We start with a few initial cloud configurations (e.g., three), run them, and input the configuration details and job completion time into the performance model. We then *dynamically* pick the next cloud configuration to run based on the performance model and feed the result back to the performance model. We stop when we have enough confidence that we have found a good configuration.

2.3.2 Problem formulation

For a given application and workload, our goal is to find the optimal or a near-optimal cloud configuration that satisfies a performance requirement and minimizes the total execution cost. Formally, we use $T(\vec{x})$ to denote the running time function for an application and its input workloads. The running time depends on the cloud configuration vector $\vec{x}$, which includes instance family types, CPU, RAM, and other resource configurations.

Let $P(\vec{x})$ be the price per unit time for all VMs in cloud configuration $\vec{x}$. We formulate the problem as follows:

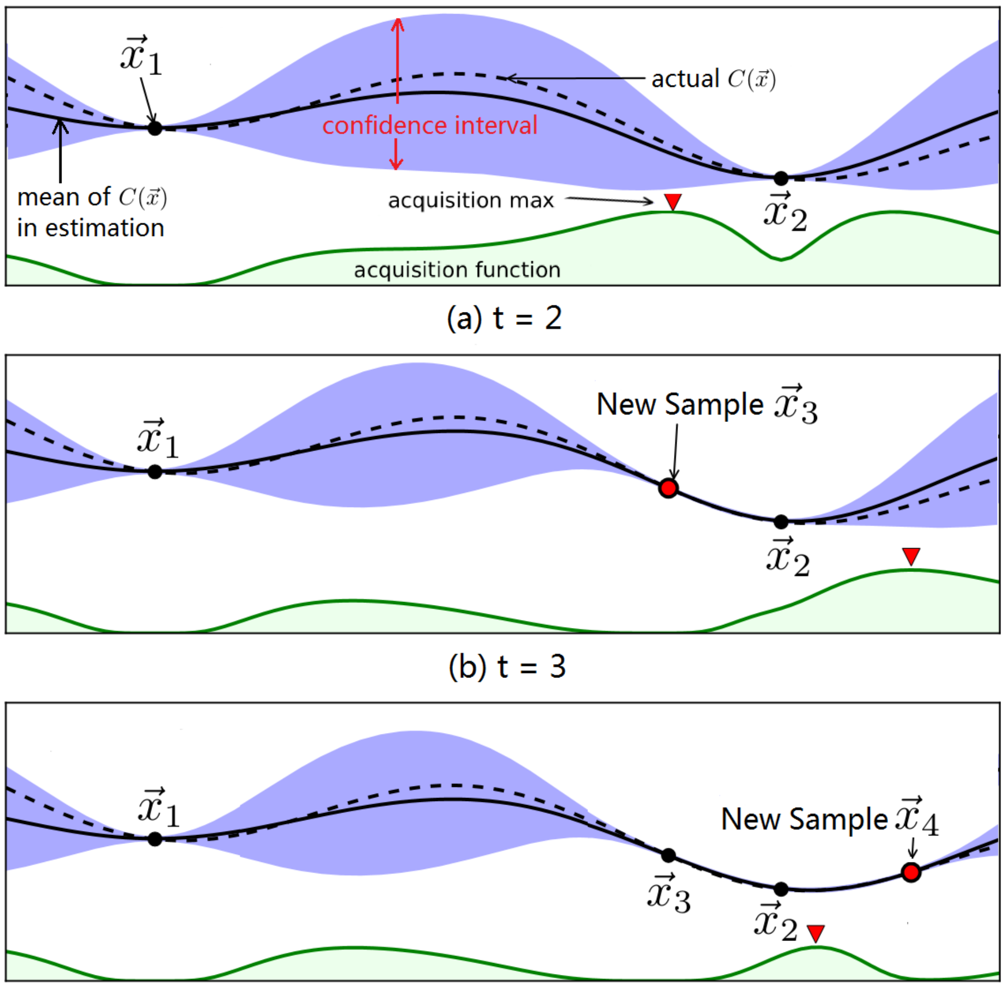
$$\begin{aligned} \underset{\vec{x}}{\text{minimize}} \quad & C(\vec{x}) = P(\vec{x}) \times T(\vec{x}) \\ \text{subject to} \quad & T(\vec{x}) \leqslant \mathcal{T}_{max} \end{aligned} \tag{2.1}$$

where $C(\vec{x})$ is the total cost of cloud configuration $\vec{x}$ and $\mathcal{T}_{max}$ is the maximum tolerated running time[1]. Knowing $T(\vec{x})$ under all candidate cloud configurations would make it straightforward to solve Eqn (2.1), but it is expensive because all candidate configurations need to be tried. Instead, we use BO (with Gaussian Process Priors, see Section 2.6) to directly search for an approximate solution of Eqn (2.1) with significantly smaller cost.

2.3.3 Solution with Bayesian Optimization

Bayesian Optimization (BO) [1, 18, 19] is a framework to solve optimization problem like Eqn. (2.1) where the objective function $C(\vec{x})$ is unknown beforehand but can be observed through experiments. By modeling $C(\vec{x})$ as a stochastic process, e.g. a Gaussian Process [24], BO can compute the *confidence interval* of $C(\vec{x})$ according to one or more samples taken from $C(\vec{x})$. A confidence interval is an area that the curve of $C(\vec{x})$ is most likely (e.g. with 95% probability) passing through. For example, in Figure 2.5(a), the dashed line is the actual function $C(\vec{x})$. With two samples at $\vec{x}_1$ and $\vec{x}_2$, BO computes a confidence interval that is marked with a blue shadowed area. The black solid line shows the expected value of $C(\vec{x})$ and the value of $C(\vec{x})$ at each input point $\vec{x}$ falls in the confidence interval with 95% probability. The confidence interval is updated (posterior distribution in Bayesian Theorem) after new samples are taken at $\vec{x}_3$ (Figure 2.5(b)) and $\vec{x}_4$ (Figure 2.5(c)), and the estimate of $C(\vec{x})$ improves as the area of the confidence interval decreases.

[1] $C(\vec{x})$ assumes a fixed number of identical VMs.



(a) t = 2

(b) t = 3

(c) t = 4

Figure 2.5: An example of BO's working process (derived from Figure 1 in [1]).

BO can smartly decide the next point to sample using a pre-defined *acquisition function* that also gets updated with the confidence interval. As shown in Figure 2.5, $\vec{x_3}$ ($\vec{x_4}$) is chosen because the acquisition function at $t = 2$ ($t = 3$) indicates that it has the most potential gain. There are many designs of acquisition functions in the literature, and we will discuss how we chose among them in Section 2.3.5.

BO is embedded into *CherryPick* as shown in Figure 2.4. At Step 2, *CherryPick* leverages BO to update the confidence interval of $C(\vec{x})$. After that, at Step 3, *CherryPick* relies on BO's acquisition function to choose the best configuration to run next. Also, at Step 4, *CherryPick* decides whether to stop the search according to the confidence interval of $C(\vec{x})$ provided by BO (details shown in Section 2.3.5).

Another useful property of BO is that it can accommodate observation noise in the computation of confidence interval of the objective function. Suppose in practice, given an input point $\vec{x}$, we have no direct access to $C(\vec{x})$ but can only observe $C(\vec{x})'$ that is:

$$C(\vec{x})' = C(\vec{x}) + \epsilon \tag{2.2}$$

where ϵ is a Gaussian noise with zero mean, that is $\epsilon \sim \mathcal{N}(0, \sigma_\epsilon^2)$. Because $C(\vec{x})'$ is also Gaussian, BO is able to infer the confidence interval of $C(\vec{x})$ according to the samples of $C(\vec{x})'$ and ϵ [1]. Note that in our scenario, the observation noise on $C(\vec{x})$ is negligible because the measurement on running time and price model is accurate enough. However, the ability to handle the additive noise of BO is essential for us to handle the uncertainty in clouds (details in Section 2.3.6).

In summary, by integrating BO, *CherryPick* has the ability to learn the objective function quickly and only take samples in the areas that most likely contain the minimum point. For example, in Figure 2.5(c) both $\vec{x_3}$ and $\vec{x_4}$ are close to the minimum point of the actual $C(\vec{x})$, leaving the interval between $\vec{x_1}$ and $\vec{x_4}$ unexplored without any impact on the final result.

2.3.4 Why do we use Bayesian Optimization?

BO is effective in finding optimal cloud configurations for Big Data analytics for three reasons.

First, BO does not limit the function to be of any pre-defined format, as it is non-parametric. This property makes *CherryPick* useful for a variety of applications and cloud configurations.

Second, BO typically needs a small number of samples to find a near-optimal solution because BO focuses its search on areas that have the largest expected improvements.

Third, BO can tolerate uncertainty. *CherryPick* faces two main sources of uncertainty: (i) because of the small number of samples, *CherryPick*'s performance models are imperfect and usually have substantial prediction errors; (ii) the cloud may not report a stable running time even for the same application due to resource multiplexing across applications, stragglers, etc. BO can quantitatively define the uncertainty region of the performance model. The confidence interval it computes can be used to guide the searching decisions even in face of model inaccuracy. In Section 2.3.6, we leverage this property of BO to handle the uncertainty from cloud dynamics.

One limitation of BO is that its computation complexity is $O(N^4)$, where N is the number of data samples. However, this is perfectly fine because our data set is small (our target is typically less than 10 to 20 samples out of hundreds of candidate cloud configurations).

Alternatives Alternative solutions often miss one of the above benefits: (1) linear regression and linear reinforcement learning are not generic to all applications because they do not work for non-linear models; (2) techniques that try to model a function (e.g., linear regression, support vector regression, boosting tree, etc.) do not consider minimizing the number of sample points. Deep neural networks [25], table-based modeling [26], and Covariance matrix adaptation evolution strategy (CMA-ES) [27] can potentially be used for black-box optimization but require a large number of samples. (3) It is difficult to adapt reinforcement learning [25, 28] to handle uncertainty and minimize the number of samples while BO models the uncertainty so as to accelerate the search.

2.3.5 Design options and decisions

To leverage Bayesian Optimization to find a good cloud configuration, we need to make several design decisions based on system constraint and requirements.

Prior function As most BO frameworks do, we choose to use Gaussian Process as the prior function. It means that we assume the final model function is a sample from Gaussian Process. We will discussion this choice in more details in Section 2.6.

We describe $C(\vec{x})$ with a mean function $\mu(\cdot)$ and covariance kernel function $k(\cdot,\cdot)$. For any pairs of input points $\vec{x}_1, \vec{x}_2$, we have:

$$\mu(\vec{x}_1) = \mathbb{E}[C(\vec{x}_1)];\ \mu(\vec{x}_2) = \mathbb{E}[C(\vec{x}_2)]$$

$$k(\vec{x}_1, \vec{x}_2) = \mathbb{E}[(C(\vec{x}_1) - \mu(\vec{x}_1))(C(\vec{x}_2) - \mu(\vec{x}_2))]$$

Intuitively, we know that if two cloud configurations, $\vec{x}_1$ and $\vec{x}_2$ are similar to each other, $C(\vec{x}_1)$ and $C(\vec{x}_2)$ should have large covariance, and otherwise, they should have small covariance. To express this intuition, people have designed numerous formats of the covariance functions between inputs $\vec{x}_1$ and $\vec{x}_2$ which decrease when $||\vec{x}_1 - \vec{x}_2||$ grow. We choose *Matern5/2* [29] because it does not require strong smoothness and is preferred to model practical functions [19].

Acquisition function There are three main strategies to design an acquisition function [19]: (i) Probability of Improvement (PI) – picking the point which can maximize the probability of improving the current best; (ii) Expected Improvement (EI) – picking the point which can maximize the expected improvement over the current best; and (iii) Gaussian Process Upper Confidence Bound (GP-UCB) – picking the point whose certainty region has the smallest lower bound(when we minimize a function). In *CherryPick* we choose EI [1] as it has been shown to be better-behaved than PI, and unlike the method of GP-UCB, it does not require its own tuning parameter [19].

Jones et al. [30] derive an easy-to-compute closed form for the EI acquisition function. Let X_t be the collection of all cloud configurations whose function values have been observed by round t, and $m = \min_{\vec{x}}\{C(\vec{x}) | \vec{x} \in X_t\}$ as the minimum function value observed so far. For each input $\vec{x}$ which is not observed yet, we can evaluate its expected improvement if it is picked as the next point to observe with the following equation:

$$EI(\vec{x}) = \begin{cases} (m - \mu(\vec{x}))\Phi(Z) + \sigma(\vec{x})\phi(Z), & \text{if } \sigma(\vec{x}) > 0 \\ 0, & \text{if } \sigma(\vec{x}) = 0 \end{cases} \tag{2.3}$$

where $\sigma(\vec{x}) = \sqrt{k(\vec{x}, \vec{x})}$, $Z = \frac{m - \mu(\vec{x})}{\sigma(\vec{x})}$, and Φ and ϕ are standard normal cumulative distribution function and the standard normal probability density function respectively.

The acquisition function shown in Eqn (2.3) is designed to minimize $C(\vec{x})$ without further constraints. Nonetheless, from Eqn 2.1 we know that we still have a performance constraint $T(\vec{x}) \leqslant \mathcal{T}_{max}$ to consider. It means that when we choose the next cloud configuration to evaluate, we should have a bias towards one that is likely to satisfy the performance constraint. To achieve this goal, we first build the model of running time function $T(\vec{x})$ from $\frac{C(\vec{x})}{P(\vec{x})}$. Then, as suggested in [31], we modify the EI acquisition function as:

$$EI(\vec{x})' = P[T(\vec{x}) \leqslant \mathcal{T}_{max}] \times EI(\vec{x}) \tag{2.4}$$

Stopping condition We define the stopping condition in *CherryPick* as follows: when the expected improvement in Eqn.(2.4) is less than a threshold (e.g. 10%) and at least N (e.g. $N = 6$) cloud configurations have been observed. This ensures that *CherryPick* does not stop the search too soon and it prevents *CherryPick* from struggling to make small improvements.

Starting points Our choice of starting points should give BO an estimate about the shape of the cost model. For that, we sample a few points (e.g., three) from the sample space using a quasi-random sequence [32]. Quasi-random numbers cover the sample space more uniformly and help the prior function avoid making wrong assumptions about the sample space.

Encoding cloud configurations We encode the following features into $\vec{x}$ to represent a cloud configuration: the number of VMs, the number of cores, CPU speed per core, average RAM per core, disk count, disk speed and network capacity of a VM.

To reduce the search space of the Bayesian Optimization, we normalize and discretized most of the features. For instance, for disk speed, we only define fast and slow to distinguish SSD and magnetic disks. Similarly, for CPU, we use fast and slow to distinguish high-end and common CPUs. Such discretization significantly reduces the space of several features without losing the key information brought by the features and it also helps to reduce the number of invalid cloud configurations. For example, we can discretize the space so that the CPUs greater (smaller) than 2.2GHz are fast (slow) and the disks with bandwidth greater (smaller) than 600MB/s are fast (slow). Then, if we suggest a (fast, fast) combination for (CPU, Disk), we could choose a 2.5Ghz and 700MBs instance (or any other one satisfying the boundary requirements). Or in place of a (slow, slow) configuration we could pick an instance with 2Ghz of speed and 400MB/s of IO bandwidth. If no such configurations exist, we can either remove that point from the candidate space that BO searches or return a large value, so that BO avoids searching in that space.

2.3.6 Handling uncertainties in clouds

So far we assumed that the relation between cloud configurations and cost (or running time) is deterministic. However, in practice, this assumption can be broken due to uncertainties within any shared environment. The resources of clouds are shared by multiple users so that different users' workload could possibly have interference with each other.

Moreover, failures and resource overloading, although potentially rare, can impact the completion time of a job. Therefore, even if we run the same workload on the same cloud with the same configuration for multiple times, the running time and cost we get may not be the same.

Due to such uncertainties in clouds, the running time we can observe from an actual run on configuration $\vec{x}$ is $\tilde{T}(\vec{x})$ and the cost is $\tilde{C}(\vec{x})$. If we let $T(\vec{x}) = \mathbb{E}[\tilde{T}(\vec{x})]$ and $C(\vec{x}) = \mathbb{E}[\tilde{C}(\vec{x})]$, we have:

$$\tilde{T}(\vec{x}) = T(\vec{x})(1 + \epsilon_c) \tag{2.5}$$

$$\tilde{C}(\vec{x}) = C(\vec{x})(1 + \epsilon_c) \tag{2.6}$$

where ϵ_c is a multiplicative noise introduced by the uncertainties in clouds. We model ϵ_c as normally distributed: $\epsilon_c \sim \mathcal{N}(0, \sigma_{\epsilon_c}^2)$.

Therefore, Eqn (2.1) becomes minimizing the expected cost with the expected performance satisfying the constraint.

BO cannot infer the confidence interval of $C(\vec{x})$ from the observation of $\tilde{C}(\vec{x})$ because the latter is not normally distributed given that BO assumes $C(\vec{x})$ is Gaussian and so is $(1 + \epsilon_c)$. One straightforward way to solve this problem is to take multiple samples at the same configuration $\vec{x}$, so that $C(\vec{x})$ can be obtained from the average of the multiple $\tilde{C}(\vec{x})$. Evidently, this method will result in a big overhead in search cost.

Our key idea to solve this problem (so that we only take one sample at each input) is to transform Eqn. (2.1) to the following equivalent format:

$$\begin{aligned} \underset{\vec{x}}{\text{minimize}} \quad & \log C(\vec{x}) = \log P(\vec{x}) + \log T(\vec{x}) \\ \text{subject to} \quad & \log T(\vec{x}) \leqslant \log \mathcal{T}_{max} \end{aligned} \tag{2.7}$$

We use BO to minimize $\log C(\vec{x})$ instead of $C(\vec{x})$ since:

$$\log \tilde{C}(\vec{x}) = \log C(\vec{x}) + \log(1 + \epsilon_c) \tag{2.8}$$

Assuming that ϵ_c is less than one (e.g. $\epsilon_c < 1$), $\log(1 + \epsilon_c)$ can be estimated by ϵ_c, so that $\log(1 + \epsilon_c)$ can be viewed as an observation noise with a normal distribution, and $\log \tilde{C}(\vec{x})$ can be treated as the observed value of $\log C(\vec{x})$ with observation noise. Eqn.(2.8) can be solved similar to Eqn.(2.2).

In the implementation of *CherryPick*, we use Eqn. (2.7) instead of Eqn. (2.1) as the problem formulation.

2.4 Implementation

In this section, we discuss the implementation details of *CherryPick* as shown in Figure 2.6. It has four modules.

1. Search Controller: Search Controller orchestrates the entire cloud configuration selection process. To use *CherryPick*, users supply a representative workload (see Section 2.6) of the application, the objective (e.g. minimizing cost or running time), and the constraints (e.g. cost budget, maximum running time, preferred instance types, maximum/minimum cluster size, etc.). Based on these inputs, the search controller

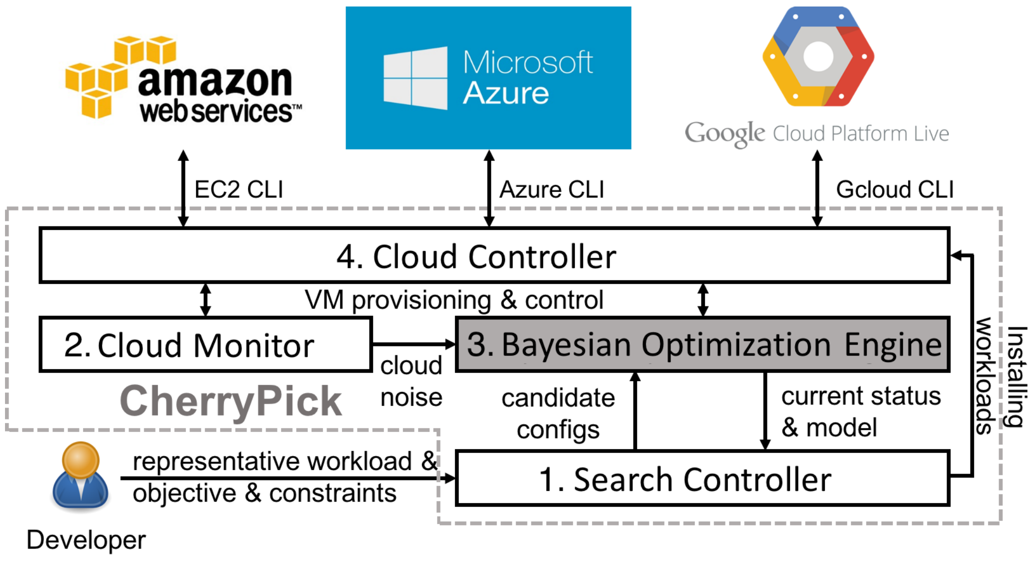


Figure 2.6: Architecture of *CherryPick*'s implementation.

obtains a list of candidate cloud configurations and passes it to the Bayesian Optimization Engine. At the same time, Search Controller installs the representative workload to clouds via Cloud Controller. This process includes creating VMs in each cloud, installing the workload (applications and input data), and capturing a customized VM image which contains the workload. Search Controller also monitors the current status and model on the Bayesian Optimization engine and decides whether to finish the search according to the stopping condition discussed in Section 2.3.5.

2. Cloud Monitor: Cloud Monitor runs benchmarking workloads of Big Data defined by *CherryPick* on different clouds. It repeats running numerous categories of benchmark workloads on each cloud to measure the upper-bound (or high percentile) of the cloud noise [2]. The result is offered to Bayesian Optimization engine as the ϵ_c in Eqn. (2.8). This monitoring is lightweight; we only need to run this system every few hours with a handful of instances.

3. Bayesian Optimization Engine: Bayesian Optimization Engine is built on top of Spearmint [33] which is an implementation of BO in Python. Besides the standard BO, it also has realized our acquisition function in Eqn (2.3) and the performance

[2]Over-estimating ϵ_c means more search cost.

constraint in Eqn (2.4). However, Spearmint's implementation of Eqn (2.4) is not efficient for our scenario because it assumes $C(\vec{x})$ and $T(\vec{x})$ are independent and trains the models of them separately. We modified this part so that $T(\vec{x})$ is directly derived from $\frac{C(\vec{x})}{P(\vec{x})}$ after we get the model of $C(\vec{x})$. Our implementation of this module focuses on the interfaces and communications between this module and others. For taking a sample of a selected cloud configuration, the BO engine submits a cluster creation request and a start workload request via the Cloud Controller.

4. Cloud Controller: Cloud Controller is an adaptation layer which handles the heterogeneity to control the clouds. Each cloud has its own APIs and semantics to do the operations such as create/delete VMs, create/delete virtual networks, capturing images from VMs, and list the available instance types. Cloud Controller defines a uniform API for the other modules in *CherryPick* to perform these operations. In addition, the API also includes sending commands directly to VMs in clouds via SSH, which facilitates the control of the running workload in the clouds.

The entire *CherryPick* system is written in Python with about 5,000 lines of code, excluding the legacy part of Spearmint.

2.5 Evaluation

We evaluate *CherryPick* with 5 types of big data analytics applications on 66 cloud configurations. Our evaluations show that *CherryPick* can pick the optimal configuration with a high chance (45-90%) or find a near-optimal configuration (within 5% of the optimal at the median) with low search cost and time, while alternative solutions such as coordinate descent and random search can reach up to 75% more running time and up to 45% more search time than *CherryPick*. We also compare *CherryPick* with Ernest [17] and show how *CherryPick* can reduce the search time by 90% and search cost by 75% for SQL queries. We discuss insights on why *CherryPick* works well and show how *CherryPick* adapt to changing workloads and various performance constraints.

2.5.1 Experiment setup

Applications: We chose benchmark applications on Spark [34] and Hadoop [35] to exercise different CPU/Disk/RAM/Network resources: *(1) TPC-DS* [36] is a recent benchmark for big data systems that models a decision support workload. We run TPC-DS benchmark on Spark SQL with a scale factor of 20. *(2) TPC-H* [37] is another SQL benchmark that contains a number of ad-hoc decision support queries that process large amounts of data. We run TPC-H on Hadoop with a scale factor of 100. Note that our trace runs 20 queries concurrently. While it may be possible to model each query's performance, it is hard to model the interactions of these queries together. *(3) TeraSort* [38] is a common benchmarking application for big data analytics frameworks [39, 40], and requires a balance between high IO bandwidth and CPU speed. We run TeraSort on Hadoop with 300 GB of data, which is large enough to exercise disks and CPUs together. *(4) The SparkReg* [41] benchmark consists of machine learning workloads implemented on top of Spark. We ran the regression workload in SparkML with 250k examples, 10k features, and 5 iterations. This workload heavily depends on memory space for caching data and has minimal use for disk IO. *(5) SparkKm* is another SparkML benchmark [41]. It is a clustering algorithm that partitions a space into k clusters with each observation assigned to the cluster with the closest mean. We use 250k observations with 10k features. Similar to SparkReg, this workload is dependent on memory space and has less stringent requirements for CPU and disk IO.

Cloud configurations: We choose four families in Amazon EC2: M4 (general

Instance Size	Number of instances					
large	16	24	32	40	48	56
xlarge	8	12	16	20	24	28
2xlarge	4	6	8	10	12	14
Number of Cores	32	48	64	80	96	112

Table 2.2: Configurations for one instance family.

purpose), C4 (compute optimized), R3 (memory optimized), I2 (disk optimized) instances.

Objectives: We define the objective as minimizing the cost of executing the application under running time constraints. By default, we set a loose constraint for running time so *CherryPick* searches through a wider set of configurations. We evaluate tighter constraints in Section 2.5.4. Note that minimizing running time with no cost constraint always leads to larger clusters, and therefore, is rather simple. On the other hand, minimizing the cost depends on the right balance between cluster size and cluster utilization.

***CherryPick* settings:** By default, we use EI= 10%, $N = 6$, and 3 initial samples. In our experiments, we found that EI= 10% gives a good trade-off between search cost and accuracy. We also tested other EI values in one experiment.

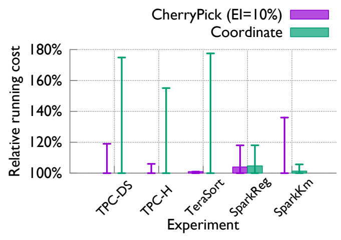
Alternative solutions: We compare *CherryPick* with the following strategies: *(1) Exhaustive search*, which finds the best configuration by running all the configurations; *(2) Coordinate descent*, which searches one coordinate – in order of CPU/RAM ratio (which specifies the instance family type), CPU count, cluster size, disk type – at a time (Section 2.2.3) from a randomly chosen starting point. The ordering of dimensions can also impact the result. It is unclear whether a combination of dimensions and ordering exists that works best across all applications. Similar approaches have been used for tuning configurations for Map Reduce jobs and web servers [42, 43].

(3) Random search with a budget, which randomly picks a number of configurations given a search budget. Random search is used by previous configuration tuning works

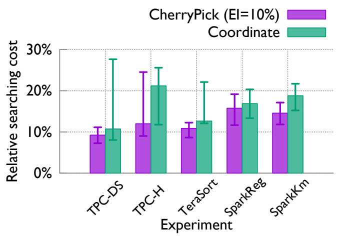
[44, 45].

Metrics: We compare *CherryPick* with alternative solutions using two metrics: (i) the running cost of the configuration: the expense to run a job with the selected configuration; (ii) the search cost: the expense to run all the sampled configurations. All the reported numbers are normalized by the exhaustive search cost and running cost across the clusters in Table 2.2.

We run *CherryPick* and random search 20 times with different seeds for starting points. For the coordinate descent, we start from all the 66 possible starting configurations. We then show the 10th, median, and 90th percentile of the search cost and running cost of *CherryPick* normalized by the optimal configuration reported by exhaustive search.



(a) Running cost



(b) Search cost

Figure 2.7: Comparing *CherryPick* with coordinate descent. The bars show 10th and 90th percentile.

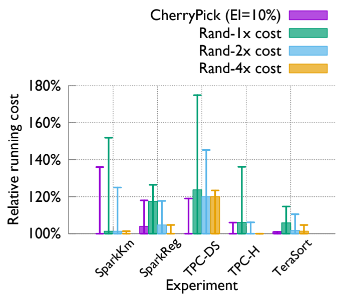


Figure 2.8: Running cost by *CherryPick* and random search. The bars show 10th and 90th percentile.

2.5.2 Effectiveness of *CherryPick*

***CherryPick* finds the optimal configuration in a high chance (45-90%) or a near-optimal configuration with low search cost and time:** Figure 2.7a shows the median, 10th percentile, and 90th percentile of running time for the configuration picked by *CherryPick* for each of the five workloads. *CherryPick* finds the exact optimal configuration with 45-90% chance, and finds a configuration within 5% of the optimal configuration at the median. However, using exhaustive search requires 6-9 times more search cost and 5-9.5 times more search time compared with *CherryPick*. On AWS, which charges on an hourly basis, after running TeraSort 100 times, exhaustive search costs \$581 with \$49 for the remainder of the runs. While *CherryPick* uses \$73 for searching and \$122 for the rest of the runs saving a total of \$435.

In terms of accuracy, we find that that *CherryPick* has good accuracy across applications. On median, *CherryPick* finds an optimal configuration within 5% of the optimal configuration. For TPC-DS, *CherryPick* finds a configuration within 20% of the optimal in the 90th percentile; For TPC-H, the 90th percentile is 7% worse than optimal configuration; Finally, for TeraSort, SparkReg, and SparkKm *CherryPick*'s 90th percentile configuration is 0%, 18%, 38% worse than the optimal respectively. It is possible to change the EI of *CherryPick* to find even better configurations.

***CherryPick* is more stable in picking near-optimal configurations and has less search cost than coordinate descent.** Across applications, the median configuration suggested by coordinate descent is within 7% of the optimal configuration. On the other hand, the tail of the configuration suggested by coordinate descent can be far from optimal. For TPC-DS, TPC-H, and TeraSort, the tail configuration is 76%, 56%, and 78% worse than optimal, while using comparable or more search cost. This is because coordinate descent can be misled by the result of the run. For example, for TPC-DS, C4 family type has the best performance. In our experiment, if coordinate descent starts its search from a configuration with a large number of machines, the C4 family fails to finish the job successfully due to the scheduler failing. Therefore, the C4 family is never considered in the later iterations of coordinate descent runs. This leads coordinate descent to a suboptimal point that can be much worse than the optimal configuration.

In contrast, *CherryPick* has stronger ability to navigate around these problems because even when a run fails to finish on a candidate configuration, it uses Gaussian process to model the global behavior of the function from the sampled configurations.

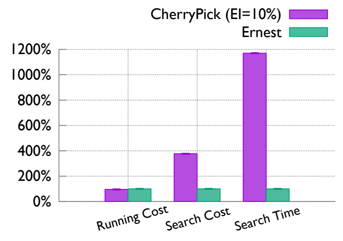


Figure 2.9: Comparing Ernest to cherrypick (TPC-DS).

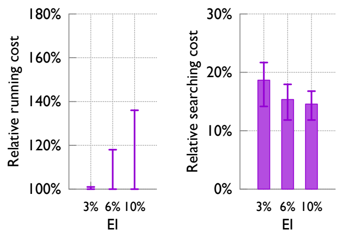


Figure 2.10: Search cost and running cost of SparkKm with different EI values.

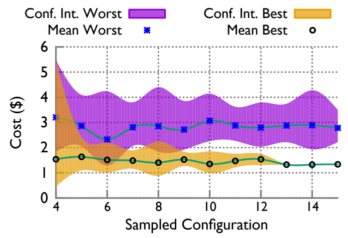
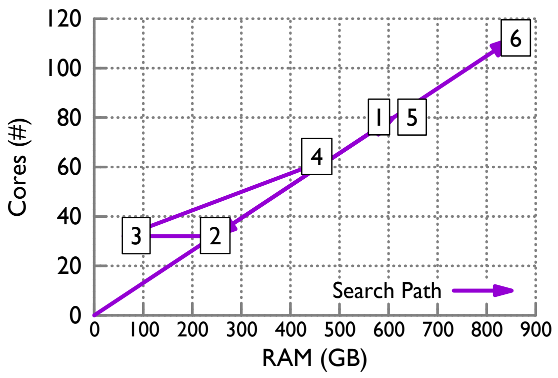


Figure 2.11: Bayesian opt. process for the best/worst configuration (TeraSort).

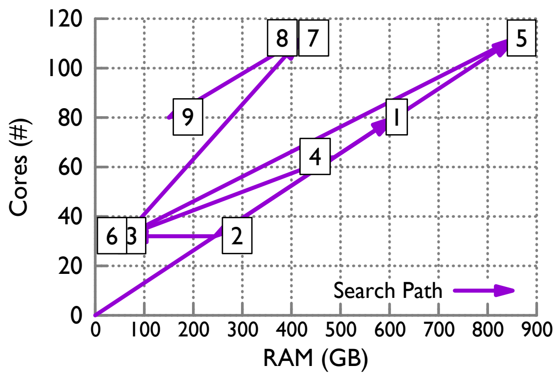
***CherryPick* reaches better configurations with more stability compared with random search with similar budget:** Figure 2.8 compares the running cost of configurations suggested by *CherryPick* and random search with equal/2x/4x search cost. With the same search cost, random search performs up to 25% worse compared to *CherryPick* on the median and 45% on the tail.

With 4x cost, random search can find similar configurations to *CherryPick* on the median. Although *CherryPick* may end up with different configurations with different starting points, it consistently has a much higher stability of the running cost compared to random search. *CherryPick* has a comparable stability to random search with 4x budget, since random search with a 4x budget almost visits all the configurations at least once.

***CherryPick* reaches configurations with similar running cost compared with Ernest [17], but with lower search cost and time:** It is hard to extend Ernest to work with a variety of applications because it requires using a small representative dataset to build the model. For example, TPC-DS contains 99 queries on 24 tables, where each query touches a different set of tables. This makes it difficult to determine which set of tables should be sampled to build a representative small-scale experiment. To overcome this we use the TPC-DS data generator and generate a dataset with scale factor 2 (10% of target data size) and use that for training. We then use Ernest to predict the best configuration for the target data size. Finally, we



(a) SparkReg



(b) TPC-DS

	SparkReg		TPC-DS	
Step	**VM Type**	**# VMs**	**VM Type**	**# VMs**
1	r3.2xlarge	10	r3.2xlarge	10
2	r3.2xlarge	4	r3.2xlarge	4
3	c4.xlarge	8	c4.xlarge	8
4	r3.large	32	r3.large	32
5	i2.2xlarge	10	r3.2xlarge	14
6	r3.2xlarge	14	c4.2xlarge	4
7			m4.xlarge	28
8			m4.2xlarge	14
9			c4.2xlarge	10

(c) Search path for TPC-DS and SparkReg

Figure 2.12: Search path for TPC-DS and SparkReg

note that since Ernest builds a separate model for each instance type we repeat the above process 11 times, once for each instance type.

Figure 2.9 shows that Ernest picks the best configuration for TPC-DS, the same as *CherryPick*, but takes 11 times the search time and 3.8 times the search cost. Although Ernest identifies the best configuration, its predicted running time is up to 5 times of the actual running time. This is because, unlike iterative ML workloads, the TPC-DS performance model has a complex scaling behavior with input scale and this is not captured by the linear model used in Ernest. Thus, once we set a tighter performance constraint, Ernest suggests a configuration that is 2 times more expensive than *CherryPick* with 2.8 times more search cost.

***CherryPick* can tune EI to trade-off between search cost and accuracy:** The error of the tail configuration for SparkKm as shown in Figure 2.7a can be as high as 38%. To get around this problem, the users can use lower values of EI to find better configurations. Figure 2.10 shows the running cost and search cost for different values of EI. At $EI < 6\%$, *CherryPick* has much better accuracy, finding configurations that at 90th percentile are within 18% of the optimal configuration. If we set $EI < 3\%$, *CherryPick* suggests configurations that are within 1% of the optimal configuration at 90th percentile resulting in a 26% increase in search cost.

This can be a knob where users of *CherryPick* can trade-off optimality for search cost. For example, if users of *CherryPick* predict that the recurring job will be popular, setting a low EI value can force *CherryPick* to look for better configurations more carefully. This may result in larger savings over the lifetime of the job.

2.5.3 Why *CherryPick* works?

Previous performance prediction solutions require many training samples to improve prediction accuracy. *CherryPick* spends the budget to improve the prediction accuracy of those configurations that are closer to the best. Figure 2.11 shows the means and confidence intervals of the running cost for the best and worst configurations, and how the numbers change during the process of Bayesian optimization. Initially, both configurations have large confidence intervals. As the search progresses, the confidence interval for the best configuration narrows. In contrast, the estimated cost for the worst configuration has a larger confidence interval and remains large. This is because *CherryPick* focuses on improving the estimation for configurations that are closer to the optimal.

Figure 2.13 shows *CherryPick*'s final estimation of the running time versus cluster size. The real curve follows Amdahl's law: (1) adding more VMs reduces the running time; (2) at some point, adding more machines has diminishing returns due to the sequential portion of the application. The real running time falls within the confidence interval of *CherryPick*. Moreover, *CherryPick* has smaller confidence intervals for the more promising region where the best configurations (those with more VMs) are

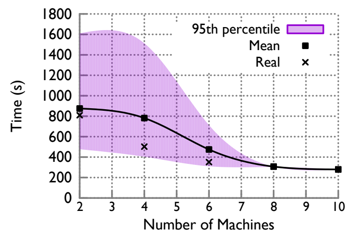


Figure 2.13: *CherryPick* learns diminishing returns of larger clusters (TPC-H, c4.2xlarge VMs).

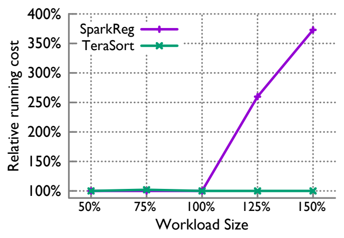


Figure 2.14: Sensitivity to workload size

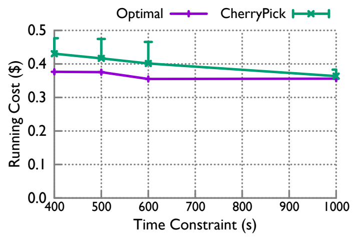


Figure 2.15: *CherryPick* works with time constraints (TPC-H).

located. It does not bother to improve the estimation for configurations with fewer VMs.

Even though *CherryPick* has minimal information about the application, it adapts the search towards the features that are more important to the application. Figure 2.12 shows example search paths for TPC-DS and SparkReg from the same three starting configurations. For SparkReg, *CherryPick* quickly identifies that clusters with larger RAM (R3 instances) have better performance and redirects the search towards such instances. In contrast, for TPC-DS, the last few steps suggest that *CherryPick* has identified that CPU is more important, and therefore the exploration is directed towards VMs with better CPUs (C4 instances). Figure 2.12 shows that *CherryPick* directly searches more configurations with larger #cores for TPC-DS than for SparkReg.

2.5.4 Handling workload changes

CherryPick depends on representative workloads. Thus, one concern is *CherryPick*'s sensitivity to the variation of input workloads. In Figure 2.14, we keep the best configuration for the original workload (100% input size) C_{100} and test the running cost of the $C_{100\%}$ on workloads with 50% to 150% of the original input size. For TeraSort, we can continue to use $C_{100\%}$ to achieve the optimal cost with different input sizes. For SparkReg, $C_{100\%}$ remains effective for smaller workloads. However, when the workload is increased by 25%, $C_{100\%}$ can get to 260% the running cost of the new best configuration ($C_{125\%}$). This is because $C_{100\%}$ does not have enough RAM for SparkReg, which leads to more disk accesses.

Since input workloads usually vary in practice, *CherryPick* needs a good selection of representative workloads For example, for SparkReg, we should choose a relatively larger workload as the representative workload (e.g., choosing 125% gives you more stability than choosing 100%). We will discuss more on how to select representative workloads in Section 2.6.

When the difference between *CherryPick*'s estimation of the running cost and the actual running cost is above a threshold, the user can rerun *CherryPick*. For example, in Figure 2.14, suppose the user trains *CherryPick* with a 100% workload for SparkReg. With a new workload at size 125%, when he sees the running cost becomes 2x higher than expected, he can rerun *CherryPick* to build a new model for the 125% workload.

2.5.5 Handling performance constraints

We also evaluate *CherryPick* with tighter performance constraints on the running time (400 seconds to 1000 seconds) for TPC-H, as shown in Figure 2.15.

CherryPick consistently identifies near-optimal configuration (2-14% difference with the optimal) with similar search cost to the version without constraints.

2.6 Discussion

Representative workloads: *CherryPick* relies on representative workloads to learn and suggest a good cloud configuration for similar workloads. Two workloads are similar if they operate on data with similar structures and sizes, and the computations on the data are similar. For example, for recurring jobs like parsing daily logs or summarizing daily user data with the same SQL queries, we can select one day's workload to represent the following week or month, if in this period the user data and the queries are not changing dramatically. Many previous works were built on top of the similarity in recurring jobs [12, 13]. Picking a representative workload for non-recurring jobs hard, and for now, *CherryPick* relies on human intuitions. An automatic way to select representative workload is an interesting avenue for future work.

The workload for recurring jobs can also change with time over a longer term. *CherryPick* detects the need to recompute the cloud configuration when it finds large gaps between estimated performance and real performance under the current configuration.

Larger search space: With the customizable virtual machines [16] and containers, the number of configurations that users can run their applications on becomes even larger. In theory, the large candidate number should not impact on the complexity of *CherryPick* because the computation time is only related with the number of samples rather than the number of candidates (BO works even in continuous input space). However, in practice, it might impact the speed of computing the maximum point of the acquisition function in BO because we cannot simply enumerate all of the candidates then. More efficient methods, e.g. Monte Carlo simulations as used in [33], are needed to find the maximum point of the acquisition function in an input-agnostic way. Moreover, the computations of acquisition functions can be parallelized. Hence, customized VM only has small impacts on the feasibility and scalability of *CherryPick*.

Choice of prior model: By choosing Gaussian Process as a prior, we assume that the final function is a sample from Gaussian Process. Since Gaussian Process is non-parametric, it is flexible enough to approach the actual function given enough data samples. The closer the actual function is to a Gaussian Process, the fewer the data samples and searching we need. We admit that a better prior might be found given some domain knowledge of specific applications, but it also means losing the automatic adaptivity to a set of broader applications.

Although any *conjugate distribution* can be used as a prior in BO [46], we chose Gaussian Process because it is widely accepted as a good surrogate model for BO [19]. In addition, when the problem scale becomes large, Gaussian Process is the only choice which is computationally tractable as known so far.

2.7 Related Work

Current practices in selecting cloud configurations Today, developers have to select cloud configurations based on their own expertise and tuning. Cloud providers only make high-level suggestions such as recommending I2 instances in EC2 for IO intensive applications, e.g., Hadoop MapReduce. However, these suggestions are not always accurate for all workloads. For example, for our TPC-H and TeraSort applications on Hadoop MapReduce, I2 is not always the best instance family to choose.

Google provides recommendation services [47] based on the monitoring of average resource usage. It is useful for saving cost but is not clear how to adjust the resource allocation (e.g. scaling down VMs vs. reducing the cluster size) to guarantee the application performance.

Selecting cloud configurations for specific applications The closest work to us is Ernest [17], which we have already compared in Section 2.1. We also have discussed previous works and strawman solutions in Section 2.2 that mostly focus on predicting application performance [21, 48, 17]. Bodik *et al.* [49] proposed a framework that learns performance models of web applications with lightweight data collection from a production environment. It is not clear how to use such data collection technique for modeling big data analytics jobs, but it is an interesting direction we want to explore in the future.

Previous works [26, 50] leverage table based models to predict performance of applications on storage devices. The key idea is to build tables based on input parameters and use interpolation between tables for prediction. However, building such tables requires a large amount of data. While such data is available to data center operators, it is out of reach for normal users. *CherryPick* works with a restricted amount of data to get around this problem.

Tuning application configurations: There are several recent projects that have looked at tuning application configurations within fixed cloud environments. Some of them [21, 22, 51] propose to monitor resource usage in Hadoop framework and adjust Hadoop configurations to improve the application performance. Others search for the best configurations using random search [21] or local search [42, 43]. Compared to Hadoop configuration, cloud configurations have a smaller search space but a higher cost of trying out a configuration (both the expense and the time to start a new cluster). Thus we find Bayesian optimization a better fit for our problem. *CherryPick* is complementary to these works and can work with any application configurations.

Online scheduler of applications: Paragon [52] and Quasar [53] are online schedulers that leverage historical performance data from scheduled applications to quickly classify any new incoming application, assign the application proper resources in a datacenter, and reduce interferences among different applications. They also rely on online adjustments of resource allocations to correct mistakes in the modeling phase. The methodology cannot be directly used in *CherryPick*'s scenarios because usually, users do not have historical data, and online adjustment (e.g., changing VM types and cluster sizes) is slow and disruptive to big data analytics. Containers allow online adjustment of system resources, so it might be worth revisiting these Approaches.

Parameter tuning with BO: Bayesian Optimization is also used in searching optimal Deep Neural Network configurations for specific Deep Learning workloads [54, 19] and tuning system parameters [55]. *CherryPick* is a parallel work which searches cloud configurations for big data analytics.

2.8 Conclusion

We present *CherryPick*, a service that selects near-optimal cloud configurations with high accuracy and low overhead. *CherryPick* adaptively and automatically builds performance models for specific applications and cloud configurations that are *just accurate enough* to distinguish the optimal or a near-optimal configuration from the rest. Our experiments on Amazon EC2 with 5 widely used benchmark workloads show that *CherryPick* selects optimal or near-optimal configurations with much lower search cost than existing solutions.

Chapter 3

Janus: Searching for optimal deployment plans in data centers

Data center networks evolve as they serve customer traffic. When applying network changes, operators risk impacting customer traffic because the network operates at reduced capacity and is more vulnerable to failures and traffic variations. The impact on customer traffic ultimately translates to operator cost (e.g., refunds to customers). However, planning a network change while minimizing the risks is challenging as we need to adapt to a variety of traffic dynamics, cost functions, and operator and geo-related constraints while scaling to large networks and large changes. Today, operators often use plans that maximize the residual capacity (MRC), which often incurs a high cost under different traffic dynamics. Instead, we propose Janus, which searches the large planning space by leveraging the high degree of symmetry in data center networks. Our evaluation on large Clos networks and Facebook traffic traces shows that Janus generates plans in real-time only needing 33~71% of the cost of MRC planners while adapting to a variety of settings.

3.1 Introduction

Data center networks are evolving fast to keep up with traffic doubling every year [8, 56] and frequent rollouts of new applications. They continuously change both hardware and software to scale out and add new features. These changes include *repairs* such as firmware security patches and *upgrades* such as addition of new features to switch hardware or software. Such changes are even more common in recent years with the adoption of software-defined networking [57, 58, 59] and programmable switches [60, 61, 62].

Changes come with an inherent risk of impacting customers and their traffic: operators have to apply network changes while upholding high availability and good performance—draining the entire data center before applying changes is too costly (typically measured through SLAs). When a change is taking place, the network operates at reduced capacity and has less headroom for handling traffic variations and failures [63, 64]. Google reports that 68% of failures occur during the network changes [63]. There are also other risks due to delayed changes and bugs in the change itself (§3.2.1).

A risk is the likelihood of any event impacting customer traffic. These events result in a violation of service-level objectives (SLOs) and hurt operator income. For example, Amazon refunds 30% of credits to customers experiencing less than 90% uptime. Thus, reducing risk is critical for all operators, but it also requires investment and is not cheap. Operators can reduce risk by overprovisioning the network [64]: with enough capacity, the network has headroom to absorb traffic variations and failures during network changes seamlessly, but this comes at a high CAPEX and OPEX cost.

There is a fundamental tradeoff between risk tolerance and cost: operators can choose to pay more, upfront, by overprovisioning to keep network utilization and risk low; or run the network at high utilization and accept a moderate risk of impacting customer traffic. Each data center operator can choose its operating point based on their budget for network resources and penalties associated with SLO violations.

Given an operating point (i.e., the level of capacity overprovisioning), operators have to make decisions on when and how to apply changes in a way that minimizes the *expected cost* of risks. However, planning a network change is challenging because it has to meet two goals:

Adaptivity: The best change plan depends on (1) *Temporal and spatial traffic dynamics* influence the expected risk of a plan. A safe plan *now* could be unsafe one hour later when traffic volumes are high (temporal dynamics). Similarly, whether we can apply a change to a core switch depends on the intra-pod and inter-pod traffic (spatial dynamics). (2) *Cost functions* which are the penalties operators incur when customers' traffic is affected in the network. The penalty depends on the customer/cloud agreements, and it is often defined based on the type of service [65, 66] (see examples in §3.2.1). (3) *Other factors* also need to be taken into account, e.g., failures, the topology, and routing (see §3.2.2).

Today, most operators use capacity-based planning. For example, Google [8, 63] divides the switches involved in a change into equal-sized sets and applies the change sequentially to maximize the residual capacity during the change. This approach is simple and scalable, but it does not adapt to traffic dynamics or failures, and it often results in higher penalties such as increased cost (see §3.2.2). Therefore, new solutions are needed that can adapt to such changes.

Scalability: Finding a plan for a change is not easy: the space of possible plans is super-exponentially large (§3.2.1). For example, there are 3.4×10^{1213} plans for upgrading 500 switches. Brute force search of the entire space is not scalable. We often need to plan changes in real-time (as plans become obsolete after long durations due to changes in traffic variations and failures) and therefore, there is a need for a system that can search the space of possible plans efficiently to find the best possible plan. We build *Janus* to do exactly that.

Janus is a change planner that leverages the inherent symmetry of data center networks to search for the best plan in a large planning space. *Janus* has the following key ideas:

Find blocks of equivalent switches: Given topology and routing, *Janus* identifies blocks of switches that connect to the same set of other switches (i.e., switches in one block are interchangeable). Within a block, we do not need to decide which individual switches to change at any given time, but rather how many switches to change (§3.3.1).

Find equivalent subplans: Some subplans include switches in different blocks but have the same impact on customer traffic under all traffic settings (§3.3.2). We leverage graph automorphism to identify these equivalent subplans.

Scale cost estimation: We run flow-level Monte-Carlo simulations to estimate the impact of each subplan on customer traffic (for various risk factors) and compute its cost. To speed up simulations, we build *quotiented* network graphs, a compressed representation of a data center network while ensuring its estimation accuracy (§3.3.3).

Account for failures: Data centers have frequent failures that lower network capacity and impact customers. It is challenging to estimate the impact of a change due to the sheer number of failure scenarios that need to be taken into account. We introduce the notion of equivalence failure classes similar to equivalent subplans (§3.3.4).

We evaluate *Janus* on large-scale Clos topologies [8] and Facebook traffic traces [56]. Our evaluation shows that *Janus* only needs 33~71% of the cost compared to current best practice approaches and can adjust to a variety of network change policies such as different cost functions and different deadlines. *Janus* generates plans in real-time: it only takes 8.75 seconds on 20 cores to plan a change on 864 switches in a Jupiter-size [8] network (61K hosts and 2400 switches).

3.2 Challenges and key ideas

In this section, we formulate the network change planning problem. We use examples to discuss strawmans (maximum residual capacity planners) and their limitations. We then summarize *Janus*'s design addressing these limitations.

3.2.1 Risk assessment for network changes

We focus on *planned network changes* (such as upgrading switch firmware or replacing faulty links and switches) where operators can reliably prepare ahead of time. Such changes are typically at a larger scale and require more time than unplanned changes—ones that are in reaction to unexpected failures (e.g., mitigating a fault).

Risk assessment is critical for planning such changes: these changes reduce the residual network capacity and leave less headroom for dealing with unexpected events—such as traffic variations, concurrent failures, and failed changes.

To plan a network change, we consider operator specified risks and probabilities and estimate their impact on customers and the corresponding penalty to operators (i.e., cost). We choose a plan that minimizes the expected cost —operators can choose to minimize other metrics such as 99th percentile to be more resilient to the worst-case events. The steps involved are as follows:

Operator specifies risks and probabilities: *Janus* relies on operators to provide the types of risks and their probabilities. Some risks are easier to estimate than others; for example, operators keep historical failures of devices, which makes it easy to determine the risk of failures for network devices [63, 67, 68]. Operators also keep historical traffic matrices [69], which we can use to estimate the risk of traffic variations. However, there are other risks which are harder to measure, for example, the risk of losing customers during downtimes, the impact of downtimes during high profile events such as Black Friday, or the risk of delaying a pushing a security upgrade.

We posit that even though we cannot measure the impact of these risks accurately, allowing operators to express such types of risks (with estimates or best guesses) allows for better planning decisions.

We refer readers to site reliability engineering (SRE) books, blog posts, and talks [70, 71, 72] for more detail on techniques to estimate risks. Improving these techniques is a research topic in and of itself and is out of scope for this chapter.

Estimating the impact on customer traffic: We next estimate the impact of these risks on customer traffic during network changes. We measure impact by counting the percentage of ToR pairs experiencing packet loss (similar to prior work [69, 73]). We consider ToR pairs (as opposed to host pairs) to reduce the traffic matrix size while preserving the traffic dynamics inside the network [69]. We use packet loss as our measure of impact as it is an important customer experience indicator [73]. Our solution can be extended to support cost functions defined on throughput and latency.

The impact is a random variable that depends on the probabilities of traffic matrices and risks. We run Monte Carlo simulations to estimate the impact under various traffic matrices and risks. For example, we model concurrent failures by enumerating failure scenarios and their probabilities. Under each failure scenario, the network has a lower capacity (removing all the switches that fail in this scenario). We simulate and measure the impact on customer traffic.

Assessing the cost to operators: Customer impact ultimately translates to operator cost because cloud providers have to refund customers for missing any service-level agreements (SLAs). These functions are often staged: For example, Amazon uses a staged function for refunding credits for availability violations: it provides 10% refund between 99.99% and 99.0% uptime, 30% refund for anything below 99.0% uptime [66]. Similarly, Azure provides its own version: 10% refund between 99.99%-99.0% uptime, 25% refund for 95%-99%, and 100% for anything below [65]. These functions may differ depending on the type of service [65, 66] and customer settings (e.g., enterprise agreements [11]). For example, operators may want to assign a higher penalty when interrupting critical systems, such as lock services that many other systems depend on, than interrupting background jobs (e.g., log analysis systems). Similar to customer impact, the cost is also a random variable given various risk probabilities.

The change planning problem: We define a network *change* as a *set* of operations on switches or links. When applying each operation, we move traffic away from the associated switch or link (drain), apply the operation, and move traffic back (undrain). *A plan of execution* is a partitioning of changes into subsets where changes in each subset run concurrently. We refer to each subset of changes in a plan of execution as a *subplan*. Given a plan, we compute the cost as the sum of the cost of all the subplans (i.e., steps).

Janus searches for the best plan that minimizes the expected cost[1] given an operator-specified deadline. Operators set deadlines to ensure bug fixes and feature updates are done in a timely fashion. Operators may also set other planning constraints (e.g., plan cable replacement according to the technician's work hours) and tie-breaker policies for plans with equal cost (e.g., select the plan that finishes faster when multiple plans have equal cost).

Janus tunes the plan in response to traffic variations, failures, and other sources of risks. When the risk of continuing a change is too high, operators can opt to rollback the change.

3.2.2 Challenges

Given a deadline for applying a change, operators follow rules of thumb that guides them to devise a plan. For example, Google [8, 63] uses a capacity-based planner

[1]We can also minimize other statistics such as 90^{th} percentile.

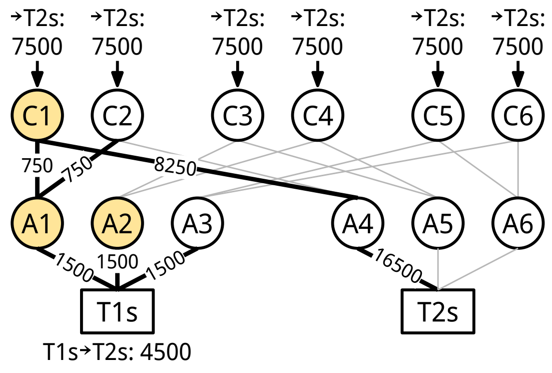


Figure 3.1: No upgrades.

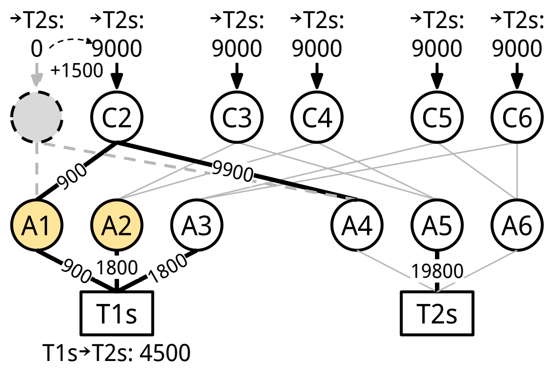


Figure 3.2: Upgrading C1 (no congestion)

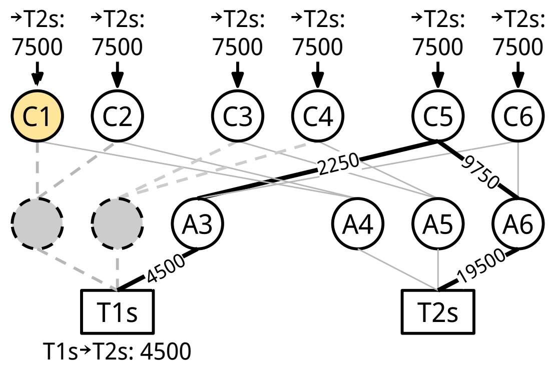


Figure 3.3: Upgrading A1, A2 (no cong.)

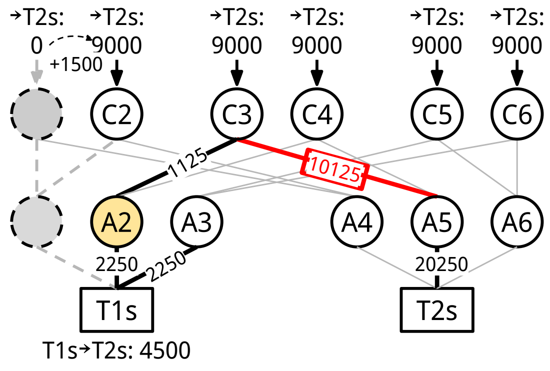


Figure 3.4: Upgrading A1, C1 (congestion)

Figure 3.5: The impact of different subplans. ToR to aggregate links are 40Gbps and aggregate to core links are 10Gbps. The traffic from T1s to T2s is 4500 Mbps; other traffic to T2s are 6*7500= 45000 Mbps. The change task is to upgrade A1, A2, and C1 (yellow circles); Grey circles are switches under changes. The network runs ECMP: numbers on each link indicates the traffic on the link.

that at every step changes an equal number of aggregate switches in each pod and an equal number of core switches, which leaves an equal amount of residual capacity at each step. Such rules of thumb typically aim to maximize the minimum residual capacity during the change on the operator's network.

Without having additional information about when, where, or how badly traffic variations and failures happen, planners that maximize the minimum residual capacity (MRC planners) are the best planners for absorbing the impact of worst-case events in the network.

However, in data centers, operators continuously monitor traffic variations and failures [8, 56]. This means we have an opportunity to do much better than MRC if we consider these factors when planning network changes. We use a few examples to discuss MRC's limitations and where there is an opportunity to improve.

Say we want to upgrade switches A1, A2, C1 in Fig. 3.5. Given a deadline of 2 steps, an MRC planner may upgrade switches A1, C1 and then A2. This plan ensures the minimum ToR capacity to any other ToR is $\frac{2}{3}$ of its original capacity.[2] However, we show that this plan has more cost than alternative plans under some traffic settings and cost functions.

Plan choices depend on spatial traffic distribution. Consider that traffic from T1s and T2s is 4.5 Gbps, and traffic from the rest of the network to the T2s is 45 Gbps. The MRC plan of upgrading A1 and C1 in the first step causes congestion at links between C3/C4/C5/C6 and A4/A5/A6 (Fig. 3.5(d)). Instead, if we upgrade two aggregate switches (A1, A2) and then C1, there is no congestion (Fig. 3.5(c)).

Plan choices depend on temporal traffic dynamics. Let us consider a different scenario where the *steady-state* traffic between the T1s to T2s is for the majority of the time around 10 Gbps and the rest of the network to T2s is on average 45 Gbps. Say when we start the upgrade task, the *current* traffic between the T1s to T2s becomes 4.5 Gbps. MRC, which upgrades A1 and C1, still causes congestion. However, if we know about the temporal traffic changes (i.e., the steady-state is 10Gbps), we can choose to upgrade C1 now and upgrade A1 and A2 after. Delaying upgrading C1 to later means we may never have the chance to upgrade it safely later because of steady-state traffic dynamics.

Plan choices depend on cost functions. The cost function further complicates change planning. Suppose based on the current traffic dynamics, the probabilities of impacting traffic for different subplans are summarized in Table 3.3. Inspired by

[2]Other plans leave less residual capacity: Upgrading A2 and C1 first (and then A1) reduces the capacity of ToRs in the first pod (T1s) to ToRs in the second pod (T2) by 50%. Similarly, upgrading A1 and A2 reduces the network capacity for ToRs in pod one to one-third.

	uptime	refund	uptime	refund	uptime	refund
Staged-1	<95%	100%	<99%	25%	<99.95%	10%
Staged-2	<95%	50%	<99%	25%	<99.99%	10%
Staged-3	<95%	100%	<99%	30%	<99.99%	10%

Table 3.1: Example staged cost functions from cloud providers

Cost function	Formula	Cost at 99.95%	99.90%	99.75%
log	$C(100 \ln(637x+1))$	20	40	90
linear	$C(50000x))$	20	50	100
quad	$C((4200x)^2))$	0	10	100
exp	$C(100(e^{277x}-1))$	10	30	100

Table 3.2: Cost functions for purely mathematical functions where C clamps the output between 0 and 100.

Subplan	C1	A1	A2	C1, A1	C1, A2	A1, A2	C1, A1, A2
%ToR pairs	0.1%	0.1%	0.1%	0.2%	2%	2%	4%

Table 3.3: An example of different subplans impacting different percentage of ToR pairs.

clouds today, we define three types of staged cost functions in Table 3.1. For Staged-3 function, the optimal plan choice is to upgrade A1, A2, and C1 in three steps and sequentially (cost of 10+10+10 = 30). However, for Staged-1, the optimal plan choice is to upgrade A1, A2, C1 concurrently (cost of 25). If Staged-1 returned 35 instead of 25, then the plan choice would have been the same as Staged-3. Alternatively, if we were upgrading 4 switches (instead of 3), each upgrade incurring 10 units of cost, then the best plan would be to upgrade all switches concurrently.

Other factors that impact the plan choice. The best plan also depends on other factors: topology, failures, and routing. In a Fat-tree topology, we need to be careful about the aggregate core connectivity [74] but not in a Clos topology (where each aggregate has the same set of connections). The best plan also depends on failures: if switches in a given pod have higher failure rates than other pods (e.g., because they are from different vendors), we have to apply their changes more carefully. Finally, different data centers employ different routing algorithms which react to failures and traffic variations differently [75, 8].

Key challenge: In summary, the plan choice depends on factors such as spatiotemporal traffic dynamics, cost functions, topology, routing, and failures. Such diversity makes it challenging to find a heuristic that works for all cases.

We could search for all possible plans, but there are many possible plans for upgrading n switches: the number of possible k-step plans is the number of ways we can divide n switches into k subsets (i.e., Stirling number S(n,k)) where $1 \leqslant k \leqslant n$). Therefore, the number of plans grows super-exponentially ($\sum_{k=1}^{n} k!S(n,k) \approx \mathcal{O}(\frac{n!}{\log_e^{n+1} 2})$). For example, for a change involving 500 switches, we have more than 3.4×10^{1213} plans. Even by exploiting the high degree of symmetry in data center topologies, the number of plans still remains prohibitively large. The same upgrade task (for 500 switches) in Jupiter topology [8] has more than 2^{120} plan realizations—this is true even after we eliminate plans that violate operator specified constraints.

The problem is exacerbated when we consider traffic dynamics and failures, forcing us to make planning decisions in real-time and in response to in-network events. The planning decisions should be faster than the operation time (or by the time we can apply the plans they are obsolete). Since many network operations, especially ones on switches, take minutes [76], the planning time itself should be in seconds.

In summary, *Janus* has to be *adaptive* and support a variety of constraints, *scalable* and work with the largest data centers, and *fast* so that it can select plans in real-time.

3.2.3 *Janus*'s key ideas

Given a set of switches (or links) involved in a network change, the plan navigator builds a repository of candidate plans (i.e., an ordered set of subplans) based on operator specified constraints. *Janus* continuously monitors traffic dynamics, evaluates the cost of plans using a simulator, and selects a plan with the minimum cost. After each subplan (step) finishes, *Janus* adjusts the plans for the remainder of the change based on traffic changes and failures.

Our key idea is to leverage the high degree of symmetry in data centers to navigate the large planning space in real-time. We show how we use network automorphism using an example topology in Fig. 3.6:

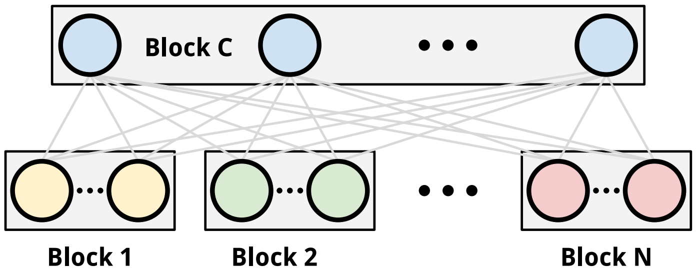


Figure 3.6: *Janus* decomposes network graphs into blocks

Identifying blocks of equivalent switches: We first identify switches that have the same connectivity and routing tables and group them into *blocks*. Switches in each block are, for all traffic purposes, indistinguishable (§3.3.2). Therefore, a subplan operating on a block needs to only care about the *number* of switches and not *which* switches it is changing. Fig. 3.6 shows several core and aggregate blocks. Given n blocks, we can describe a subplan as a tuple of n numbers $< b_1, ...b_n >$ where the i_{th} index is the number of steps for upgrading the i_{th} block. Operators can further reduce this space by taking similar actions on different blocks, i.e., merge two blocks to build superblocks (§3.3.1).

Identifying equivalent subplans using graph automorphism: For most data center networks, the number of blocks is large and so is the number of subplans. However, many subplans, even on different blocks, have the same impact on customers. For example, in Fig. 3.5, upgrading A1, C1 is equivalent to upgrading A2, C3 even though A1 and A2 are in different blocks. *Network automorphisms* can identify such equivalent subplans. Equivalent subplans speed up planning by confining risk simulations to unique subplans (§3.3.1).

Estimating the cost of subplans with scalable Monte Carlo simulations: We run Monte Carlo simulations on all possible traffic matrices during the network change. We discuss how we predict future possible traffic matrices and handle prediction errors

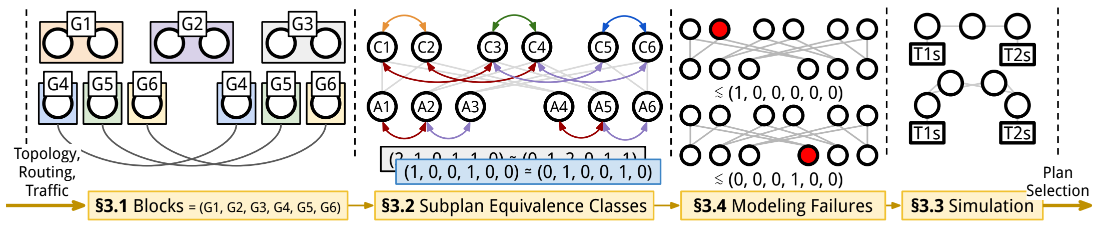
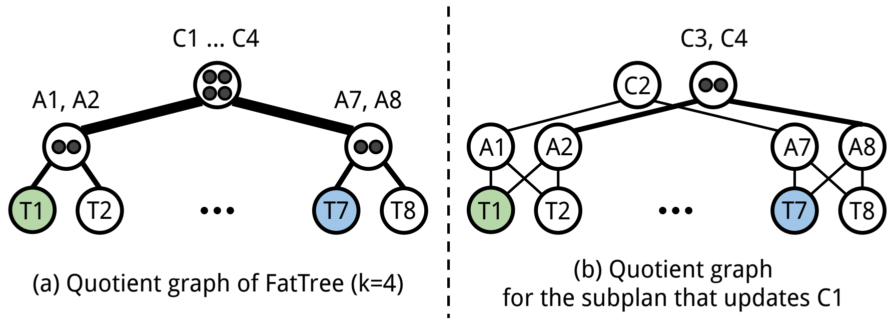


Figure 3.7: *Janus*'s Design



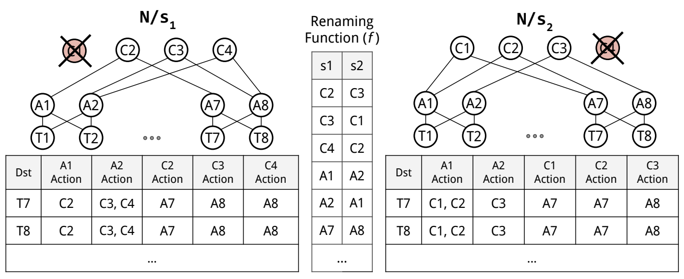
C1 ... C4
A1, A2
A7, A8
(a) Quotient graph of FatTree (k=4)
C3, C4
(b) Quotient graph for the subplan that updates C1

Figure 3.8: Example of quotient graphs for FatTree topology.

in §3.3.3. For each traffic matrix and its probabilities, we run flow-level simulations to estimate its risk of impacting customer traffic and the corresponding costs. We then compute the expected cost under all scenarios.

Monte Carlo simulation on many different TMs take a long time, e.g., the simulation for a single TM takes minutes even for a modest size data center with 600 switches (a relatively small data center) on a single core (§3.5.3). To reduce simulation time, we leverage network symmetry to simulate flows on a *quotient graph* instead of the original topology but ensure the estimated risk remains the same (See §3.3.3).

Failure equivalence: To estimate the cost of a large number of failure scenarios, we introduce *failure equivalence classes* similar to equivalent subplans. Data centers typically use a fail-stop model to deal with failures. This makes failures similar to subplans as they both bring down a set of switches, links, or line cards. We thus can model failures as subplans taking down the failed elements for a change task.



Dst	A1 Action	A2 Action	C2 Action	C3 Action	C4 Action
T7	C2	C3, C4	A7	A8	A8
T8	C2	C3, C4	A7	A8	A8
...					

s1	s2
C2	C3
C3	C1
C4	C2
A1	A2
A2	A1
A7	A8
...	

Dst	A1 Action	A2 Action	C1 Action	C2 Action	C3 Action
T7	C1, C2	C3	A7	A7	A8
T8	C1, C2	C3	A7	A7	A8
...					

Figure 3.9: Example of equivalent subplans. The actions show forwarding decisions at each switch for a rule matching destination (Dst).

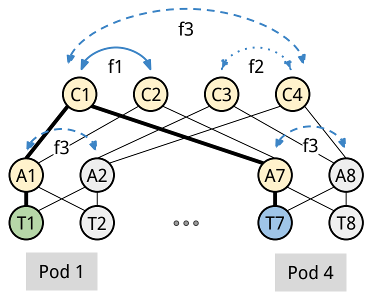


Figure 3.10: Renaming functions for finding equivalent subplans

3.3 *Janus* Design

Janus has to adapt to a variety of conditions (e.g., traffic dynamics, failures, and cost functions) and scale to large networks and large changes. For that, *Janus* leverages the high degree of symmetry in data center topologies to search the large planning space. Fig. 3.7 shows the four key components in *Janus*: (1) Given the topology and routing information, *Janus* starts by identifying blocks of equivalent switches; (2) *Janus* then identifies equivalent subplans across blocks; (3) *Janus* runs Monte-Carlo simulations using quotient networks to estimate the impact and cost of each subplan and selects plans accordingly; (4) To estimate the impact of failures, we identify equivalent classes of failures in the same way as equivalent subplans.

3.3.1 Identifying blocks of equivalent switches

Given topology and routing information, we group switches connecting to the same hosts and have the same routing table into *blocks*. There are many such blocks in data centers today (Fig. 3.6). A block is fully specified by two values: a switch and the number of such switches in that block. Operators can then granulate the number of steps it takes to upgrade switches in a block, e.g., $\frac{i}{k}$ with $0 \leqslant i \leqslant k$ of switches in each block. Blocks are a good representation because they are high level enough for operators to understand and are succinct enough for planning purposes—we only need to know a switch in that block and the number of such switches.

Operators can further make the planning space coarser by collapsing multiple blocks into one and using the same steps to upgrade them. We call these groupings *superblocks*. The intuition behind superblocks is that in large data center networks, there is enough path diversity and redundancy that many *close* plans have a similar impact on traffic. For example, for two pods with 20 aggregate switches, upgrading 3 switches in pod 1 and 4 switches in pod 2 versus upgrading 4 switches in both pods are practically similar from the residual capacity standpoint. Therefore, instead of searching in the exact planning space, we can search in a coarse-grained planning space with superblocks.

There are many ways to group blocks into superblocks. For example, they can build superblocks based on communication patterns, so that they upgrade two blocks talking with each other as one entity; or by spreading blocks with high traffic across different super blocks—so that two blocks with high traffic have the opportunity of being upgraded separately; or in its simplest form group blocks based on the type of switches, e.g., upgrade all aggregate blocks together and upgrade all core blocks together.

3.3.2 Finding equivalent subplans

The most computationally intensive part of planning is estimating the impact of subplans on large scale topologies. The saving grace here is that many subplans have an equal impact under all settings. If we had an automated way of identifying such subplans, we then would only need to simulate for each unique class of subplans. However, checking the equivalence of two subplans is not straightforward because of topological and routing complexities (§3.2.2). Here, we formalize the notion of subplan equivalence and discuss how we can efficiently find such subplans.

Definition 3.3.1 (**Subplan Equivalence**)**.** We define two subplans s_1 and s_2 to be equivalent in a network N, when a renaming function f exists that satisfies three properties:

1. *P1: Equivalent topologies.* f maps switches in G_{N/s_1} (i.e., the topology after removing switches in the subplan s_1) and G_{N/s_2}, where for each link (A, B), for switches A and B in G_{N/s_1}, there exists a matching link, ($f(A)$,$f(B)$), in G_{N/s_2} with the same capacity.
2. *P2: Equivalent traffic matrices.* The traffic volume between ToRs (A, B) in s_1 is the same as the traffic volume between $(f(A), f(B))$ in s_2.
3. *P3: Equivalent routing.* For a routing algorithm that makes forwarding decisions based on the topology in P1 and the traffic matrix in P2, all the forwarding tables in N/s_1 and N/s_2 are equivalent. That is, for switch $S \in N/s_1$ and $f(S) \in N/s_2$ we have: for the i_{th} rule on switch S of the form (src, dst, action) there exists an i_{th} rule (f(src), f(dst), f_A(action)) on switch $f(A)$ in N/s_2, where action is a set of (nexthop, weight) tuples, and $f_A(action) = \{(f(\text{nexthop}), \text{weight}) | (\text{nexthop}, \text{weight}) \in action\}$.

For example, in Fig. 3.9, using this definition, a subplan, s_1, that updates C1 and a subplan, s_2, that updates $C4$ are equivalent. To show this, consider the renaming function, f, shown in the table of the same figure. Using this function, the topologies in N/s_1 after removing $C1$ and N/s_2 after removing $C4$ are equivalent (i.e., isomorphic), because we can map {C2 → C3, C3 → C1, C4 → C2, A1 → A2, A2 → A1, A7 → A8, A8 → A7, ...}. Similarly, since the traffic sources T1, T2, ..., T7, T8 map to themselves, their flows remain intact and the flow volumes between the pairs remain the same. If we use a routing algorithm that makes forwarding decisions based on P1 and P2, then P3 is also satisfied.

Many routing algorithms are equivalent, i.e., they match P3. For example, ECMP shortest path routing only uses topology information to devise multiple shortest paths between pairs of hosts. Similarly, WCMP matches P3 because its routing decisions only depend on the topology. It is possible to extend this definition to other routing algorithms that rely on switch configurations, such as BGP, by defining an equivalence between the switch configurations.

Theorem 3.3.1. If traffic forwarding only uses the topology, traffic, and routing as defined in Definition 3.3.1, two equivalent subplans have the same impact under all traffic scenarios.

Proof sketch: P1, P2, and P3 guarantee that traffic between two ToRs traverses in the same exact manner throughout the network and thus sees the same impact during the execution of the two equivalent subplans: we can find a bisimulation between the two subplan networks (see §3.3.2).

Subplan equivalence with graph automorphism. A naive approach to finding equivalent subplans may enumerate all the subplans and do a pairwise equivalence check. However, this takes too much time. Instead, we focus on finding equivalence classes of subplans: if we find a renaming function for the network that preserves P1, P2, and P3 before applying a subplan, we could rename the network first. The subplan lacks enough information to tell the difference between the original and the renamed network. Thus, we can apply the subplan on the renamed network, and in the process make it change a different set of switches. For example, in Fig. 3.9, if we rename C1 to C4 and C4 to C1, a subplan that operated on C1 now can also operate on the renaming of C1, that is C4. Concretely:

Theorem 3.3.2 (Network Automorphism)**.** For a subplan, s, and a renaming function, f, that maps network N onto itself, if f preserves properties P1, P2, and P3, the two subplans s and $f \cdot s$ (the subplan after applying the renaming function to its elements) are equivalent.

Proof sketch: By finding a renaming f that preserves P1, P2, and P3 for the network, N, we guarantee we can find a renaming function between N/s and $(f \cdot N)/s$.

Similarly, we can also prove that a renaming function between $N/(f \cdot s)$ and $(f \cdot N)/s$ exists. Therefore, $N/(f \cdot s)$ and N/s are equivalent (see appendix for proof §3.3.2).

For example, consider the renaming function f in Fig. 3.9 where f, maps {C1 → C4, C2 → C3, C3 → C2, C4 → C1, A1 → A2, ..., A7 → A8, T1 → T1, ..., T8 → T8}. The two subplans $(f \cdot N)/s$ and N/s are equivalent under the renaming function f, because they preserve P1, P2, P3. Similarly, the two subplans $(f \cdot N)/s$ and $N/(f \cdot s)$ are equivalent under the identity function, which indeed preserves P1, P2, and P3. Therefore, $N/(f \cdot s) \equiv N/s$.

The theorem shows that using the set of renaming functions for a network N, we can generate many subplans equivalent to any other given subplan.

Given a set of renaming functions and a set of subplans, we can use the renaming functions to partition the subplans into equivalence classes. We observe the set of renaming functions forms a permutation group (it has identity, inverse, associativity, and closure properties). Using this group, we define a group action on our subplans: $G \cdot s = \{\{f \cdot v | v \in s\} | f \in G\}$ where G is the group of renaming functions, s is a subplan, v is a switch in the subplan, and f is a renaming function. This action preserves the basic properties of group actions: compatibility and identity. Group actions partition the set they act on—by using the group action, we can partition the subplan set to find equivalence classes of subplans.

For example, Fig. 3.10 shows three renaming functions for a k=4 FatTree. The three functions are: {f1: (C1 C2), f2: (C3 C4), f3: (C1 C3)(C2 C4)(A1 A2)(A3 A4)}. We can use the three renaming functions f1, f2, and f3, subplan {C1} is equivalent to subplan f1·{C1} = {f1·C1} = {(C1 C2)·C1} = {C2}, f3·{C1} = {C3}, and f3f2·{C1} = {C4}. Similarly, a subplan s2 = {A1, C2} is equivalent to f1·s2 = {A1, C1}, f3·s2 = {A2, C3} and f3f2·s2 = {A2, C4} but not to {A2, C1}. This is because no possible combination of generators renames A1 to A2 only.

Encoding for graph automorphism engines. We can use a graph automorphism engine to find the renaming group that preserves P1, P2, and P3. Graph automorphism engines typically find automorphism groups of vertex-colored graphs—a vertex-colored graph is a graph where a coloring function, C, assigns colors to nodes. The automorphism engine guarantees the permutation of the nodes respects the coloring: we can only permute nodes that have the same color. We can define colors in a way that two nodes have the same color when they satisfy properties P1, P2, P3.

We define a label tuple for each node with one label per property in Theorem 3.3.1. Two nodes are permutable, if their labels exactly match, i.e., all the properties of Theorem 3.3.1 hold. To build the labels:

For P1, take the topology as an input to the graph automorphism engine. To encode each links' bandwidth, we assign a unique label per unique link capacity to each edge, e.g., if the data center topology uses 40G and 100G links, we use two unique labels to describe each link.

For P2, we assign a unique label to each traffic source. This coloring ensures that for every pair of traffic source, (A, B), there exists a pair, (f(A), f(B)), in the renamed network—the number of unique colored pairs matches the number of cells in the traffic matrix. If two traffic sources see similar traffic, we can allow the coloring to rename them by using the same labels. This ensures that each pair in the network has a unique traffic label assigned to it.

As P3 depends on P1 and P2, and we already label those properties, the same labels can be used for P3.

After labeling, we assign a unique color to each unique label. The number of unique colors is equal to the number of unique label tuples in the network. It is true: no polynomial algorithms are known for the general case of graph automorphism, but many polynomial-time algorithms exist for special cases of this problem [77]. In particular, we found Nauty [78] can find the automorphism groups of a large data center with 2,400 switches in 6.25 seconds (§3.5.3), which matches the real-time requirements of planning—we observed similar computation times for expanders [79], fat-tree [75], and bCube [80] topologies.

Proof for the subplan equivalence theory:

Here we show a formal proof of the subplan equivalence theory. We define a model to capture the states for a flow level simulation of the network.This allows us to refine the operational semantics of other algorithms on top of the network state for various purposes. For example, the semantics could use proportional fairness or max-min fairness.

Definition *Network*: We define a network as a tuple (G, R, S) where:

P1) $G = (V, E)$ is a graph specifying the network topology. V is the set of nodes and $E : V \times V \to \mathbb{R}$ is a function specifying which nodes are connected together and what is the capacity of the edge.

P2) R is a function assigning rules to nodes:

$$\begin{aligned} R : V \to \mathbb{R}^* \text{ where } \mathbb{R} = \{(src, dst, t, action) \mid in, out \in V, \\ t \text{ is packet specific test condition}\} \end{aligned}$$

We refer to the i_{th} rule as $R_{v,i}$; t describes packet testing conditions not captured in the form of source or destination nodes, e.g., protocol or port; and *action* is one of **drop** or **fwd** P where $P \subset (V \times \mathbb{R})$. P specifies the portion of traffic that goes through a specific port.

P3) S is a partial function specifying the traffic sent from the end-hosts: $S : V \times V \rightharpoonup \mathbb{T}$. Where u actively generates traffic towards v, $S(u, v)$ describes that traffic in terms of a model-specific encoding $\mathbb{T}$.

Definition *Network Isomorphism*: We say two networks are equivalent up to isomorphism if there is a vertex renaming function (bijection) that permutes the nodes between the two networks while preserving the G, R, S relations. More concretely, two networks, $N \simeq N'$ are isomorphic if $\exists \pi_V : V \longleftrightarrow V'$ where $G \simeq_\pi G', R \simeq_\pi R', S \simeq_\pi S'$. For a renaming function π_V:

1) G and G' are isomorphic when:

$$E(v1, v2) = E'(\pi_V(v1), \pi_V(v2))$$

2) R and R' are isomorphic when:

$$\begin{aligned} R_i = (v, t) \Leftrightarrow R'_i &= (\pi_V(v), \pi_T(t))\text{where:} \\ \pi_T(t) &= (\pi_V(v1), \pi_V(v2), t, \pi_A(a)) \\ \pi_A(a) &= \begin{cases} \textbf{drop}, & \text{if } a = \textbf{drop} \\ \textbf{fwd } \{\pi_V(v) | \forall v \in \text{ports}\}, & \text{if } a = \textbf{fwd} \text{ ports} \end{cases} \end{aligned}$$

3) S and S' are isomorphic when: $\forall u, v \in V : S(u, v) = \pi_T(S'(\pi_V(u), \pi_V(v)))$ where π_T permutes the nodes encoded in the traffic using π_V.

Definition *Isomorphic network function*: A network-isomorphic-invariant function $F : N \to T$ is a function that does not use identifying information for the nodes. That is, F is invariant under network isomorphisms if $N \simeq N' \Rightarrow F(N) = F(N')$ for all networks N, N'.

Theorem A.1: A network-isomorphic-invariant function, F, outputs the same value for two isomorphic networks, N, N', that is: $F(N) = F(N')$.

Proof: The proof is given by the definition of F.

Theorem A.2: Max-min fairness is agnostic under network isomorphism.

Proof: Max-min fairness is solving the following equation:

$$\begin{aligned} &\text{maximize} && \sum_i U(x_i) && \\ &\text{s.t.} && \sum_i R_{li} x_i \leqslant c_l && \text{variables} x_i \geqslant 0 \end{aligned}$$

Where, x_i is the rate allocation between two nodes. c_l is the capacity of the link l. And R_{li} is the routing on the links. R_{li} is one when flow i goes through link l and zero otherwise.

By using P1 and P2, we guarantee that the set of equations that we write for max-min fairness are the same between the two networks. We know that the output of max-min fairness is unique. Therefore, an arbitrary renaming of the variables names does not impact the optimization result. Therefore, since the two sets of equations between the two networks are only different in the name of the variables and since the result is unique, we can conclude that max-min fairness is network-isomorphic invariant.

Network Automorphism

Theorem 3.3.3 (Network Automorphism)**.** For a subplan, s, and a renaming function, f, that maps network N onto itself, if f preserves properties P1, P2, and P3, the two subplans s and $f \cdot s$ (the subplan after applying the renaming function to its elements) are equivalent.

Proof: To prove this it is enough to show that a renaming function between the two networks exist. We prove this in two parts: First, we prove that $(f \cdot N)/s$ is equivalent to $N/(f \cdot s)$. Second, We then prove that $(f \cdot N)/s$ is equivalent to N/s. Finally, we conclude that $N/(f \cdot s) \equiv N/s$.

To prove the first part, we use the identity function as the renaming function.

First, it is easy to verify that the two graphs are isomorphic, that is, for every switch $A \in (f \cdot N)/s$ there exists a switch with the same name $A \in N/(f \cdot s)$. Similarly, for every link, between two switches in one network, we can find a similar link in the other network. Therefore, P1 is true.

For P2, since a subplan does not impact traffic sources, we know that for every traffic source in one graph, there exists a traffic source in the other graph. And therefore, the traffic between the two has not changes.

P3 is true given that we have proved P1 and P2.

To prove the second part, we already know by definition that $f \cdot N$ and N are equivalent, therefore, we can replace N in place of $f \cdot N$.

3.3.3 Estimating cost with Monte Carlo simulations

We measure the number of ToR pairs experiencing packet losses using flow-level Monte-Carlo simulations under various traffic matrices and translate the number based on cost functions. Since we search the entire planning space, we can support various cost functions. We can also extend *Janus* to support multiple tenants, each with their cost function.

We have to model congestion in the network, that is, how competing ToRs divide (the scarce) bandwidth among themselves. For that, we run max-min fairness to decide how much bandwidth each ToR gets (similar to [57]). This objective matches that of TCP. We also consider the network's routing tables, which is important as the network reacts to failures and traffic variations through routing changes. This is in contrast to previous work that used multi-commodity flow (MCF) for simulating data center traffic [81, 82]: while MCF is a reasonable estimation of the bisection bandwidth, it ignores routing algorithms and fairness objectives.

Our simulation relies on knowing possible traffic matrices during the change. Today, data center operators continuously collect traffic matrices (TMs). We use the current TM to represent what happens in the next planning interval and use the past TMs to predict the TMs for the remainder of the change. Previous work [69] use a similar approach and find that the current TM is a good estimation of the future (e.g., the next step of the plan)—the intuition is that the TM does not (typically) change dramatically in such a short time. When traffic is unpredictable, and our predictions are not a good representative sample of future TMs, *Janus* may lose some of its temporal benefits—because *Janus*'s view of the future was incorrect. However, *Janus* still gains spatial benefits due to more accurate short-term predictions.

With max-min fairness as our objective, we have to find ways to speed up simulations. This is especially important for larger data centers where simulations may take much longer to complete. We use the inherent symmetries in the data center network to achieve faster simulations: we build a quotient graph per subplan by merging switches with the same forwarding rules (e.g., all the ECMP paths). Fig. 3.8(a) shows the quotient graph for a k=4 FatTree. Fig. 3.8(b) shows the quotient graph for a subplan upgrading C1.

We build quotient graphs by using network automorphism (see §3.3) to identify equivalent sets of switches under P1, P2, P3. We run the group action G on individual switches (instead of subplans) and build an equivalence relation on the switches. We can merge equivalent switches because they have similar, per-link, traffic patterns. For all switches in the same equivalence class, we build a super-switch and have one virtual forwarding table across all original switches—we can merge the forwarding tables if they are the same, e.g., all the core switches have similar forwarding tables in a Fat-tree network. To add links between super switches, we only have to ensure the link capacity between super-switches is the same as the original network. Since the new topology has far fewer links/paths, we can simulate the network much faster.

3.3.4 Handling failures

Failures are a common risk source when planning network changes. Google reports that nearly 68% of failures occur when a change is in progress [63]. *Janus* models failures as capacity reductions—a failure on a set of switches remove these switches from the network graph (fail-stop), which increases the risks of impacting customer traffic.

Operators can input failure scenarios and probabilities based on their logging of historical failure events for each vendor [67, 63]. Given failure scenarios and probabilities, we can run simulations to measure impact and estimate the expected cost for each network change plan.

However, the size of failure space is exponential in the number of switches, e.g., to model independent switch failures for 2400 switches, we have 2^{2400} possible scenarios. Instead, we model the most likely failure scenarios that cover P (e.g., 99%) of the most probable failures, i.e., $\Pr[\text{Failures}] \geqslant P$. For example, if switches have 0.1% failure rate in a topology of 2400 switches, we only need to simulate up to 7 concurrent failures (binomial distribution) to cover 99% of failures.

To further reduce the number of failure scenarios, we introduce *failure equivalence classes*, i.e., failures that result in isomorphic network graphs. We can view a failure scenario as a subplan bringing down switches in the failure set (or links/line-cards). Thus, to simulate failures during a change, *Janus* considers a bigger change task involving both failed switches and change switches. We can then apply the same techniques above to estimate cost under failures.

3.4 Implementation

Janus has 7.2k lines of C code. It operates in three steps:

Operators specify the change, the cost function, and the risks. Operators can input arbitrary change requests into *Janus*. For each change, operators specify the length of its operations and a deadline for the change. Operators then define a cost function where the input is the percentage of ToR pairs impacted during a change interval, and the output is the associated cost. Operators can also specify time-based cost functions—to model time constraints during planning, e.g., to emphasize the risk of delaying a critical bug fix. In its current state, *Janus* can model concurrent failure of switches in the data center where failures are independent. We chose to implement this failure model following the example of previous work [83]. For more complex failure models, e.g., correlated failures, we rely on previous work and use their proposed sampling techniques [84] to cover the failure space.

Simulation. We assume the data center network upholding Max-min fairness for the traffic it routes through its network. Max-min fairness is also commonly used [85, 86, 57] to model how TCP flows affect each other during congestion. To model Max-min fairness, we simulate the network while respecting the routing, topology, and link constraints. Since our simulation uses P1, P2, and P3 (in Definition 3.3.1), it satisfies the conditions of Theorem 3.3.1. Therefore, we can rely on Theorem 3.3.1 to reduce the subplan search space. For each setting, we convert the network into a quotient network, then run our network simulator on the quotient network.

Janus uses the current TM as a prediction of the traffic for the upcoming subplan and the 10 previously observed TMs as a prediction of traffic for the rest of the plan. In practice, data center operators may have better traffic predictors and are free to use their own.

Estimate cost in real-time and adjust the plan. At runtime, *Janus* goes through all the subplans, applies the failure model on each subplan, and uses the TM predictions to estimate the impact of each choice. It then measures the impact of each plan and chooses a plan with the lowest expected cost. If there are multiple candidate plans, *Janus* picks the plan according to operator-specified tiebreakers. Other termination conditions are also possible; for example, ones that return the best plan within a deadline.

Scalability. Monte-Carlo simulations are easily parallelizable: we can run each scenario (i.e., subplan and traffic matrix) independently from others and on different machines/cores. We can then merge the results of all scenarios to build the cost random variable of each subplan.

Plan ports and links changes. *Janus* supports port and link changes by modeling them as *virtual switches*. To plan changes for links, we replace each link in the network graph with a passthrough virtual switch that sends the incoming traffic on each of its port to its other port. Any operation on links can thus be modeled as an operation on virtual switches. The virtual switch abstraction allows us to use the previous theorems for scaling. Similarly, to handle ports, we model each as a passthrough virtual switch similar to links.

Janus supports line card changes (e.g., replacements). A line card is a collection of N ports. We can substitute a virtual switch with $N + N$ ports in place of a line card. We connect the first N virtual switch ports to the links and the second N ports to the switch where the line card belongs. The routing table of the virtual switch is, again, a passthrough table where the first port is directly connected to port $N + 1$, the second port to port $N + 2$ and so on.

Rollbacks. It is possible that due to unexpected events, a change task becomes costly, e.g., because there are no suitable plans or simply because the change is faulty. In that case, operators would want to rollback the upgrade. *Janus* generates rollback plans instantly: A rollback plan is a change plan for a subset of original change tasks.

Failed instructions. Operators may fail to follow *Janus*'s instructions accurately. In such cases, operators can accommodate by adding these failed instructions back into the change. For example, if during the execution of a change, *Janus* issues an impossible instruction, e.g., because switches are physically too far apart, operators can mark these instructions as incomplete so that *Janus* schedules them in the upcoming intervals.

***Janus* offline.** There are cases where operators cannot spare the computational cost of real-time planning, e.g., if they lack good traffic predictors (so they have to model many TMs) or when using complex failure models or simulators that prohibit real-time planning. Under such circumstances, operators can use *Janus* in what we call the offline-mode.

In offline-mode, operators feed a large number of traffic matrices (possibly from previous days) and historical failures into *Janus*. For example, operators could use historical traffic of recent days to predict future days [69, 87, 88, 89]. *Janus* then finds a *static* plan for the change that will highly likely minimize the expected cost under the provided traffic and failure settings. Operators may also want to change the objective of *Janus* to, for example, minimizing the 99th percentile of the risk, so that the plans that *Janus* suggests are resilient to worst-case scenarios. This mode is very similar to MRC, as both planners find static plans. However, *Janus* still enjoys the spatial benefits, and it also respects operators planning constraints such as deadlines and cost functions.

3.5 Evaluation

Here, we demonstrate the cost reduction, scalability, and generality of *Janus* using large-scale data center topologies, network change tasks, and realistic traffic traces. Our evaluation shows that *Janus* only needs 33~71% of MRC cost and can adjust to a variety of network change policies such as different cost functions and different deadlines. *Janus* generates plans in real-time: it only takes 8.75 seconds on 20 cores to plan a change on 864 switches in a Jupiter-size [8] network (61K hosts and 2400 switches).

Topology	# switches in the DC			# hosts	# upgrades (cores, aggs)
	# pods	# cores, aggs, ToRs	# switches		
Scale-1	8	8, 64, 96	168	3840	72 (8, 64)
Scale-4	16	24, 192, 384	600	15360	216 (24, 192)
Scale-9	24	54, 432, 864	1350	34560	486 (54, 432)
Scale-16	32	96, 768, 1536	2400	61440	864 (96,768)

Table 3.4: Configurations and change task for each topology. We upgrade all core and aggregate switches in all the pods.

3.5.1 Evaluation settings

Topology. We evaluate *Janus* on Clos topologies (Table 3.4). We use four different scales ranging from the default *Scale-1* which updates 8 pods (3.8K hosts and 168 switches) to a scale comparable to the size of Google's Jupiter topology [8] (61K hosts and 2400 switches).

Traffic. We generate a cloud-like trace using Google job traces to model the size and arrivals of tenants [90] and Facebook traffic traces [56] to model the traffic for each tenant. Specifically, for each tenant, we decide its arrival and leaving times and the number of ToRs it runs on based on the Google job trace. We then randomly select its traffic type: either Hadoop or web traffic, and select the corresponding trace from Facebook. We generate 400 such traffic matrices at a 5-minute interval—Minute-level TMs map to the granularity that operators use to measure SLOs in data centers today. By default, our traffic has an average maximum link utilization (MLU) of 80% (the median link utilization is 17%). We use average MLUs ranging from 65% to 95%.

Network change tasks. We evaluate *Janus* on a large change so that it has to explore a large planning space. Concretely, we upgrade all core and aggregate switches in the data center. Table 3.4 shows the details for each upgrade task. We assume each upgrade takes one timeslot (5 minutes), i.e., one traffic matrix, matching the length of firmware upgrades of today's switches [76]. Each upgrade is repeated 50 times across different hours. We report the average and standard deviation of this cost. We set deadlines of 2, 4, or 8 steps for finishing the change and choose 4 as default—this means that MRC leaves 50%, 75%, 87.5% of residual capacity in the network at each step.

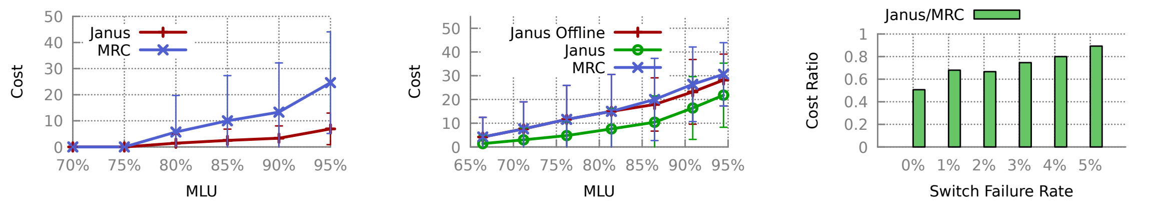
Cost functions. We define three types of staged cost functions following the shape of the refund functions of major cloud providers such as Azure, Amazon, and GCloud (Table 3.1)[3]. To test the generality of *Janus* under various cost functions, we also evaluate a range of synthetic functions, namely, logarithmic, linear, quadratic, and exponential, where the input is the number of ToR pairs experiencing packet loss and the output is a cost value between 0 and 100. The details of these functions are in Table 3.2.

We use the Staged-1 function by default. One should only interpret the relative cost differences across approaches and settings, not the absolute values because despite using cloud cost functions (that operators use today in practice), it is difficult to gauge whether the combination of our choices of cost functions, topology, and traffic matrices represent what operators experience in practice.

Planners. We evaluate two planners: (1) *Janus* which uses the last 10 and the current traffic matrices to plan the change; *Janus* adjusts the plan based on traffic changes (§3.4). (2) *Janus Offline* which uses history traffic to choose a fixed plan that does not change during execution. (3) *MRC*: a planner that maximizes the residual capacity at each step of the plan §3.2.2, similar to the state-of-the-art solutions used in data centers today [8].

Evaluation metrics. We report the expected cost of applying a network change while meeting each change's deadline. For each data point, we run 50 experiments and take the average.

[3]Even though we use these functions differently than the clouds today, we suspect that the shape and nature of cost functions will be the same.



(a) Avg./std. of plan costs (static traffic) (b) Avg./std. of plan cost (dynamic traffic) (c) Avg. plan cost ratio under switch failure

Figure 3.11: Comparing *Janus* with MRC under various settings.

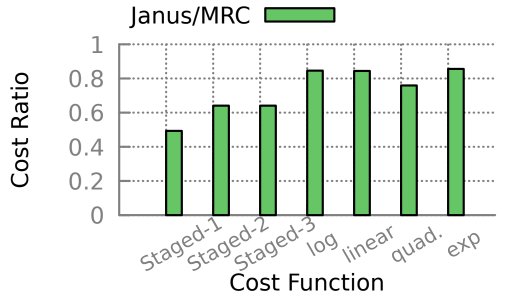
3.5.2 Cost savings over MRC

Spatial benefits: We start our evaluation with a simple scenario of static traffic (using a randomly chosen TM). Because the traffic does not change, *Janus* online is the same as *Janus* offline. *Janus* achieves lower or equal cost to MRC under all MLU settings (Fig. 3.11a). At 85% MLU, *Janus* takes only 25% of the cost of MRC (2.5 units of cost vs. 10 units). When MLU is low (e.g., $\leqslant$ 75%), there is enough capacity in the network so both *Janus* and MRC can pick plans that apply the change with zero cost. *Janus* picks plans that upgrade more switches initially and fewer switches later on and only for busy pods. In contrast, MRC equally allocates the switches at each step. When MLU is high (e.g., $\geqslant$ 80%), every step of the plan is likely to impact ToR pairs. *Janus* automatically changes its goal to choose plans with a fewer number of steps to minimize the duration of traffic disruption.

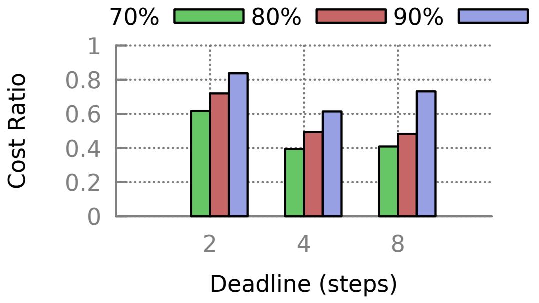
Temporal benefits: Next, we evaluate *Janus* with tenant and traffic dynamics as discussed in our evaluation settings. Fig. 3.11b shows that both *Janus* and *Janus* offline have a lower cost than MRC under all MLUs. On average, *Janus* has 33~71% of the cost of MRC. At 85% MLU, *Janus* takes only 52% of the cost comparing to MRC. This is because *Janus* can change more switches under a low traffic load and fewer switches under a higher load.

Janus offline does not consider traffic dynamics and thus performs worse than *Janus*, but still better than MRC. At 85% MLU, *Janus* offline takes 90% of the cost comparing to MRC. This is because of the spatial benefits mentioned above. In our setting, the spatial benefit is smaller than the temporal benefit because, with tenant dynamics, the traffic shifts across ToRs fast, so there is not as much spatial skewness.

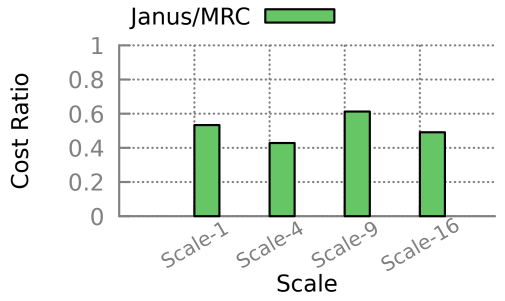
MRC also has higher variance than *Janus* because it chooses a fixed plan which sometimes performs very poorly. Such lack of predictability makes it difficult to understand the potential impacts of MRC plans on customers. In contrast, both *Janus* and *Janus* offline identify the best plan based on operators' policies (including plan deadlines, other constraints, and tiebreakers).



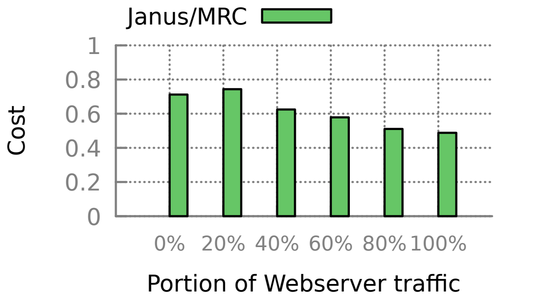
(a) *Janus* has benefits for all cost functions.



(b) Different deadlines. Each color is a different MLU setting.



(c) *Janus* has benefits across all data center scales.



(d) Impact of Hadoop/Webserver user split on *Janus* plans.

Figure 3.12: *Janus* adjusts to operators constraints and cost functions and has universal benefits across all settings. The bars show the average cost of the plans by *Janus* compared to the MRC planner.

Predictability of traffic. *Janus* lowers the planning cost even when the traffic is hard to predict. Here, we try 5 different traffic traces where we change the ratio of Hadoop (unpredictable: all to all communication patterns that exhibit on-off chatters) to Web servers (predictable: spatially stable and constant chatter of Web servers to cache servers) users in our trace while keeping the MLU fixed. Fig. 3.12d shows that

Janus saves cost under all settings. As we increase the proportion of Web server to Hadoop users, *Janus* costs decrease from 74% of MRC-plan to 51%. As traffic becomes less predictable, *Janus*'s temporal benefits disappear, but *Janus* gains benefit because of spatial patterns.

Concurrent failures. *Janus* also considers the probabilities of failures when it plans network changes. Here we mode independent switch failures using Bernoulli random variables, that is a switch either fails or does not with 1-5% failure rate at every step (i.e., every 5 minutes). Typically, failure rates are lower in data centers, e.g., Gill et al. [91] report 2.7% failure rate for aggregate switches over a year. However, we choose high failure rates to ensure there is a non-zero chance of concurrent switch failures in Scale-1: at 5% failure rate, we expect 4 concurrent switch failures in the space of 80 switches. High failure rate stress tests *Janus* as it requires the simulation of a much larger failure space: to model 99% of possible failures at 5% failure rate across 80 switches (binomial distribution), we have to consider more than 2.6 trillion failure scenarios: $\sum_{x=1}^{10} \binom{80}{x} \geqslant 2.6$trillion.

Fig. 3.11c shows that *Janus* online has 52% to 85% of the cost of MRC. As we increase the failure rates, *Janus* becomes more conservative in preparing for potential failures and thus requires a higher cost. MRC does not consider failure rates, and its cost remains the same for all failure rates. With a higher failure rate, *Janus* gets closer to MRC. This is because, as discussed in §3.2.2, MRC is a good option when we have little information about failures. Interestingly, as we increase failure rates, we are indirectly reducing our knowledge of failures by increasing the number of failure scenarios that we have to consider. More concretely, to cover 99% of probable failures for 80 switches, we only need to simulate 85k different scenarios at 1% failure rate, whereas that number explodes to 2.6 trillion at 5% failure rate.

3.5.3 Scalability

Janus finds plans in real-time even for large topologies. We evaluate on Scale-1 to Scale-4 (61k hosts) topologies. The details of these topologies are shown in Table 3.4). *Janus* online plans cost 42% to 61% of MRC plans (Fig. 3.12c).

Janus spends the majority of its time (>99%) estimating the impact of subplans at every step, which depends on the number of subplans and the simulation time to estimate the impact of each subplan. In §3.3.2, we discussed how network automorphism allows *Janus* to reduce both the number of subplans using subplan equivalence and the simulation time through quotient network graphs. Another added benefit is that as subplans are completely independent of each other, we can parallelize *Janus* very easily by computing the impact of each subplan (or TM) on a different core/machine.

We measure the total running time of *Janus* across all the steps on one core and report it in Table 3.5. We also interpolate the time to 20 cores [4] to show that *Janus* can plan changes in real-time even for the largest data centers. With 1 core, it takes *Janus* 175 seconds to plan a change for upgrading 864 switches for Scale-16. With 20 cores, it takes 8.75 seconds.

We also compare the simulation time per traffic matrix for a four-step plan with and without the quotient graph optimization: The running time on one core of Scale-1 improves from 2.9s to 0.01s, a reduction of 290x. Similarly, the running time of Scale-4 improves from 184 seconds to 0.045, a reduction of 4100x–at Scale-4 topology finding a plan could take upwards of 12 hours on a single core. We could not run the flow simulations at Scale-9 and Scale-16 without quotient graphs because we ran out of memory.

[4]This is an artifact of the code running single-threaded.

3.5.4 Adaptivity

Janus is adaptive in selecting plans that have low expected cost for various planning constraints and metrics.

Different cost functions: Fig. 3.12a shows that *Janus* online and offline are consistently better than MRC under a variety of cost functions. *Janus* online's plans cost is 64% of MRC under Staged-2 and Staged-3 cost functions and *Janus* offline's plans cost is 86% of the MRC cost. The results are similar for the Staged-2 and Staged-3 functions as their cost functions are similar when the packet loss rate is low (10% credit for 99.99% ToR pair connectivity). The benefits under Staged-1's cost function is larger (49% of cost compared to MRC) because Azure's cost function has more room for losses (10% credit for 99.95% availability).
Janus online uses 75~85% cost compared to MRC for logarithmic, linear, quadratic, and exponential cost functions. *Janus* is uniformly better than MRC regardless of cost function as *Janus* exhaustively searches the entire plan space.

Different deadlines: Fig. 3.12b shows that *Janus* has a lower cost than MRC for all deadlines. The cost ratio of *Janus* follows a U-shape for all MLUs: when the deadline is small, there are fewer candidate plans and thus less room for *Janus* to reduce cost compared to MRC. When the deadline is far away, MRC touches fewer switches per step and incurs less cost. For deadlines in the middle (where the majority of settings are), *Janus* has the most gains over MRC. The actual deadline with the best gain depends on the MLU.

Rollback: We show a scenario where the cost estimates provided by *Janus* helps operators to make rollback decisions. As before, the change involves upgrading all the core and aggregate switches in the Scale-1 topology (72 switches). *Janus* initially selects an eight-step plan but continuously estimates the cost of other plans and rollback plans, as shown in Fig. 3.13. At step 5, *Janus* reports that the expected

Topology (Change size)	Planning time		simulation time per TM	
	1 core	20 cores	Without quotient	With quotient
Scale-1 (72)	2.5 s	0.125 s	2.9 s	0.01 s
Scale-4 (216)	10.06 s	0.503 s	184 s	0.045 s
Scale-9 (486)	35.9 s	1.795 s	Out of mem.	0.149 s
Scale-16 (864)	175.0 s	8.75 s	Out of mem.	0.8 s

Table 3.5: *Janus* planning time.

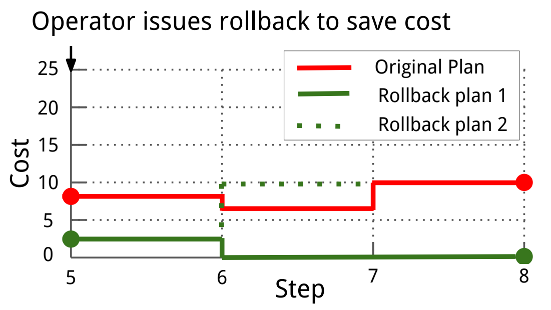


Figure 3.13: *Janus* suggests a rollback plan (Green line) that safely revert an on-going change.

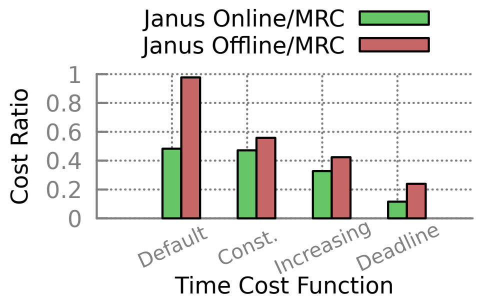


Figure 3.14: Different cost functions for delayed changes. MRC fails to factor time and incurs heavy cost.

cost of the remainder of the plan (42 switches left) is 9.901 units (red curve) and the cost of rollback of the initial bit of the plan (30 switches) is 3.354 (green curve). If operators consider the cost of 9.901 to be too high (e.g., because their budget is only 5 units), they may choose the rollback plan. After issuing the rollback, *Janus* can immediately select a plan for it. For example, Fig. 3.13 shows two rollback plans provided by *Janus*: Plan 1 upgrades 17, 12, 1 switches in 3 steps, and Plan 2 upgrades 17, 13 switches in 2 steps. At step 6, *Janus* picks Plan 1 as Plan 2 is too risky (cost of 10) due to traffic dynamics.

Delaying changes: In practice, operators may not have a strict deadline but instead, have to pay for a cost if a change takes a longer time. *Janus* can plan for such cases. We introduce three types of cost for delayed changes: (1) *Constant cost (labeled as Constant):* Each step of the plan has a fixed cost (4 units). For example, applying a change may require a fixed amount of engineering effort in each step. (2) *Increasing cost (labeled as Increasing):* We use a linear cost function where the n_{th}

step of the plan costs n units. This happens if, for example, we need to fix critical bugs quickly and the longer we wait, the more network remains vulnerable (i.e., more cost to operators). (3) *Cost after a deadline (labeled as Deadline):* We model this as a fixed cost of 30 units after the 6th step. This happens, for example, when an engineer relays the rest of the change to another engineer at the end of his shift (and increases the risk of making errors). (4) The *Default* bar is the original function with only customer impact cost. We minimize the total expected cost of customer impact and delayed changes.

Fig. 3.14 shows that *Janus* online only takes 11%-47% of the MRC cost; similarly, *Janus* offline has 24%-55% of the MRC cost. *Janus* adjusts the plan based on the cost function. However, MRC can only use a fixed-step plan (e.g., 8 steps in this case) independent of the cost function.

For *Constant* and *Increasing*, *Janus* selects a shorter plan (on average 2.82 and 2.84 steps) to reduce the cost of delayed changes at the expense of increasing the customer impact cost (from 8.6 in *default* to 12.4 and 11.96). In this way, *Janus* identifies the best tradeoff between the two types of cost. For *Deadline*, because there is a significant cost beyond 6 steps, *Janus* fits the plan within 6 steps to reduce the overall cost with the expense of slightly increasing the customer impact cost (from 8.6 to 9).

3.6 Related Work

Scheduling network updates. A few prior efforts focus on planning network updates (i.e., forwarding plane changes). Reitblatt et al. [92] introduce consistent switch rule updates to avoid loops or black-holes. zUpdate [93] plans traffic migrations (caused by network updates) with no packet loss during the worst-case traffic matrices. SWAN [58] and Dionysus [94] schedule forwarding plane updates for WAN by breaking the updates into stages with barriers in-between. While these low-level tools are useful in updating individual switch configurations, *Janus* plans upgrades for a (large) group of switches or links in data centers. Moreover, *Janus* adjusts plans based on traffic changes in real-time.

Failure mitigation. Autopilot [95] manages end-host updates and remedies failures at the end-hosts through reimaging or rebooting. Bodik et al. [96] discuss an optimization framework for increasing the resiliency of end-host applications to faults. *Janus* deals with the general problem of network upgrades and can provide scheduling support for these failure mitigation solutions.

Network symmetry. Beckett et al. [97] compress the control plane of large networks to test data plane properties, e.g., reachability and loop freedom. Plotkin et al. [98] scale up network verification for reachability properties by using symmetry. It is unclear how such techniques apply to network change planning under traffic dynamics. *Janus* builds a compressed data-plane to speed up simulations and uses subplan equivalence to prune the plan search space.

3.7 Conclusion

Fast network changes are critical for enabling quick evolutions of data centers today. *Janus* applies network changes by estimating the impact of various plans and dynamically adjusting the plans based on traffic variation and failures. *Janus* uses network automorphism to scale to a large number of plans. *Janus* plans in real-time even for the largest of data-centers and finishes upgrades with 33% to 71% of the cost of MRC planners.

Chapter 4

Revisiting measurement algorithms in software switches

Many network functions are moving from hardware to software to get better programmability and lower cost. Measurement is critical to most network functions because getting detailed information about traffic is often the first step to make control decisions and diagnose problems. The key challenge for measurement is how to keep a large number of counters while processing packets at line rate. Previous work on measurement algorithms mostly focuses on reducing memory usage while achieving high accuracy. However, software servers have plenty of memory but incur new challenges of achieving both high performance and high accuracy. In this chapter, we revisit the measurement algorithms and data structures under the new metrics of performance and accuracy. We show that saving memory through extra computation on these switches is not worthwhile. As a result, a linear hash table and count array outperform more complex data structures such as Cuckoo hashing, Count-Min sketches, and heaps in a variety of scenarios. We argue that this trend is to be expected granted that the memory, network speed, and CPU have grown at proportional speeds.

4.1 Introduction

To reduce the cost and management complexity of hardware switches and middleboxes, there is a growing need of moving network functions to software. For example, today, data centers often run load balancing and firewalls in software [99, 100], and ISPs have started to deploy virtualized network functions (VNFs) to replace their hardware boxes [101].

Measurement is a key component in many network functions: for detecting anomalies (e.g., heavy hitters, superspreaders), profiling traffic of applications, or inspecting individual packets (DPI). Other network functions such as load balancing and traffic engineering also rely on accurate measurement of traffic statistics [89]. Measurement tasks can run on bare-metal, e.g., a software switch [102], or inside containers either standalone or as part of another NFV, e.g., a load balancer container that detects and spreads heavy-hitter flows across all the backend servers [103].

To support measurement functions, we need to keep a large number of counters for individual packets and flows. Therefore, most measurement algorithms focus on how to store many counters with limited memory while retaining measurement accuracy, at the expense of more hash functions (e.g., Cuckoo hashing [104] and Count-Min sketch [105, 106]) or more computations (e.g., heaps). Even some recent proposals focusing on software measurement also target reducing memory usage [107, 108, 109, 110, 111, 112].

However, we argue that, in software, the key metric is not memory usage, but packet processing performance (i.e., throughput and latency). This is because modern servers have plenty of memory, an efficient caching hierarchy, and highly optimized compilers. Instead, the key challenge is to achieve high throughput and low latency. If we spend too many CPU cycles to fetch measurement data into cache and compute the right values and locations for counters, we may delay the packet processing and affect throughput. Note that the tail latency also matters because even if a few packets experience long delay, the queue size increases which causes packet drops.

In this chapter, we re-evaluate measurement algorithms in software with a focus on performance and accuracy metrics. We study three measurement tasks (heavy hitters, superspreaders, and change detection) on a variety of measurement algorithms (including hash tables, sketches, and heaps). Our key observations are:

1. We show that saving memory through extra computation is not worthwhile in achieving high performance and high accuracy for measurement in software. For example, using more hash functions in Cuckoo hashing or a Count-Min sketch provides worse performance than a linear hash table or a count array. Using more computationally intensive data structures (e.g., heaps) also hurts performance. Instead, to improve the accuracy, one can simply allocate larger memory to a simple linear hash table or a count array while still achieving better performance than the other data structures with more computation. (Section 4.3)
2. Our conclusion holds for heavy hitter detection and other measurement tasks with different memory access patterns (superspreader detection) and more complex computation (change detection). It also holds for measurements with different entry sizes, value sizes, and traffic skews. (Section 4.4)
3. In a multicore setting, it is a bad idea to save memory by sharing resources across cores. Instead, we should maintain separate data structures across cores to avoid synchronization and aggregate the results during the reporting time. (Section 4.5)

In addition to the above observations, we discuss possible ways to improve measurement algorithms in software in Section 4.6. We describe related works in Section 4.7 and conclude in Section 4.8.

4.2 Background and Motivation

To support various network functions, we need a variety of measurement tasks such as heavy hitter detection, traffic change detection, and flow size distribution estimation. We observe that most of these tasks are often implemented using three classes of algorithms (Table 4.1). In this section, we give some backgrounds on these measurement tasks and algorithms and their design principles. We then motivate why it is important to re-evaluate these algorithms in the software context.

4.2.1 Three classes of measurement algorithms

We consider three classes of measurement algorithms: hash tables, sketches, heap/tree-based solutions. To illustrate their design principles, we take heavy hitter detection as an example. We define a heavy hitter as a source and destination IP address pair that sends traffic volume more than a pre-specified threshold. Heavy hitters are very useful for many management tasks. For example, operators can collocate chatty VMs (source-destination pairs with heavy traffic) in the same server or rack to save network bandwidth in data centers.

Hash tables: Hash tables compute a hash function for each key and use the result to locate a bucket in the array to store the key and its value. To handle hash collisions, many hash table designs such as linear hashing, Cuckoo hashing [104], or hopscotch hashing [124] probe a set of additional buckets to identify an empty bucket to hold the key. When the hash table has a high occupancy rate (load factor), finding an empty bucket takes multiple probing rounds, which leads to high packet processing delay and delay variance. We compare Cuckoo hashing that is commonly used for software switches [125, 126] to the linear hash table.

For heavy hitter detection in the hash table, we use the source and destination IP pair as the key and count the number of packets for each pair. A pair is a heavy

Function	**Meaning**	**Sketch**	**Heap/tree-based**	**Hash table**
Heavy hitter	A traffic aggregate identified by a packet header field that exceeds a specified volume	NSDI'13[113] [106]	[105, 111], ANCS'11 [114]	SIGCOMM '02[115]
Super spreader	A source IP that communicates with a more than a threshold number of distinct destination IP/port pairs (Defined for destinations in a similar way.)	NSDI'13[113] [107]		IMC'10 [116], [110]
Flow size distribution	The distribution of sizes of flows distinguished by a set of packet header fields	[117]		IMC'10 [116]
Change detection	A drastic change of volume/# packets from a traffic aggregate compared to a prediction model	IMC'04 [108] [118]	[119]	IMC'10 [116]
Entropy estimation	Entropy (A measure of randomness/diversity) of volume/# packets from different flows	[120]		IMC'10 [116], SIGMETRICS'06 [121]
Quantiles	Dividing an ordered set of flows (e.g., based on source IP) into equal-weight subsets	[122]	SIGMOD'01 [109], SIGMOD'99 [123],[106]	

Table 4.1: A survey of proposed measurement solutions

hitter if its count is above a certain threshold. The implementation details of the hash table may affect the packet processing performance significantly [127]. To speed up the hash table, we applied several system optimizations such as cache prefetching, cache access alignment, and SIMD instructions to calculate the hash function.

Sketches: Sketches are summaries of streaming data to approximately answer a specific set of queries. For example, Count-Min sketch [106] is commonly used to find heavy hitters [128, 129, 130] [1]. A Count-Min sketch keeps a two-dimensional array of counters with d rows and w columns. It computes d hash functions per packet and updates the corresponding d positions in each row. To find the counter for a given IP pair, the minimum counter in the d locations is returned because it has minimum collisions. If the minimum counter is above the threshold, we add the IP pair to a set. Later at the report time, we report the set of IP pairs as heavy hitters. In contrast, a count array sketch computes one hash function per packet. When there are hash collisions, a count array simply adds up the counters for the collided keys.

Heaps and trees: Heaps reduces the memory usage by only keeping the most

[1]The conclusions of this chapter is easily extensible to other sketches.

important entries for the measurement query (e.g., big flows). For example, the SpaceSaving algorithm [105] finds heavy hitters by tracking the volume of traffic from IP pairs in a small hash table. When the hash table gets full, it finds the entry with the minimum volume, say v_{min}, replaces that with the new IP pair, and adds the packet volume to the original counter (v_{min} plus the size of the new packet). To find the minimum entry, we need to keep a heap data structure [105]. Thus for each entry in the hash table, there is a corresponding entry in the heap, and for each packet, the heap must be updated to maintain its property.

Trees are also used to store a hierarchical set of counters [114, 119, 128]. For example, to detect heavy hitters, we can build an IP prefix tree and dynamically zoom in and out the subtrees based on the monitored traffic counters to reduce the number of monitored prefixes.

4.2.2 Previous works on measurement algorithms

Many previous works on measurement algorithms [115, 116, 113, 114, 121, 131, 119, 118, 132, 106, 110] promote the sketch-based solutions which maintain approximate counters with compact memory by leveraging multiple hash functions. This idea fits hardware switches which typically have limited high-speed memory. However, in software with a memory hierarchy, the total memory usage does not matter, but the number of memory accesses at different levels of the cache hierarchy affects the packet processing latency and throughput. As a result, it is not worthwhile to reduce the total memory usage at the expense of more instructions for calculating additional hash functions and more time to access extra entries. In fact, we will show in our evaluation that if we can reduce the number of hash functions and memory accesses, we can still achieve low latency and high throughput with a large total memory.

Unfortunately, even previous measurement works that target software environments [107, 108, 109, 110, 111, 112], only compare the different set of sketch and heap solutions and focus on the comparison of total memory usage. Some papers [129, 105] that compare hierarchical Count-Min sketch and heap-based solutions show that heap-based solutions can achieve better performance and accuracy. Other work claim to achieve reasonable performance without rigorous testing on modern servers and comparison with single hash-based solutions.

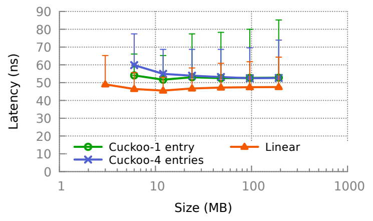
Instead, in this chapter, we focus on a systematic comparison of both the performance and accuracy of hash tables, sketches, and heaps through extensive evaluations. We conclude that simple is often the best. For example, the simplest implementations of hash tables and sketches (i.e., the linear hash table and the count array) achieve the best performance and accuracy for heavy hitter detection. We also extend the evaluation to other measurement tasks and over different traffic traces.

4.3 Evaluation of measurement algorithms in software

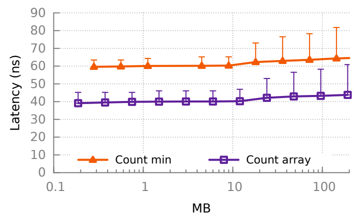
Our key observation is that saving memory through extra computation is not worthwhile in achieving high performance and high accuracy for measurement in software. This is because packet batching and memory prefetching can mask the memory access latency. On the other hand, the latency due to extra computation cannot be masked as easily—superscalar processors and compilers already perform efficient interleaving of instructions and utilize the computation resources as much as possible.

We noticed two common approaches that use more computation to save memory: more hashes and complex data structures: (1) Computing multiple hashes to save memory degrades performance. For example, a count-array that uses a single hash function and large memory beats a Count-Min sketch that uses a smaller memory but makes up for accuracy loss by using multiple hashes. Also, the linear hash table has lower average and tail latency than the Cuckoo hash table that saves memory using multiple hashes. (2) It is possible to achieve the accuracy of more computationally intensive data structures by allocating more memory to simpler data structures while achieving better performance: we compare data structures based on sketch, hash table, and heap.

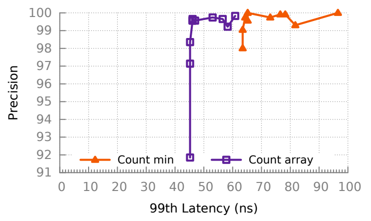
We start by evaluating measurement algorithms for heavy hitter detection in a single-core setting, and then we extend the result to other measurement tasks in Section 4 and multicore settings in Section 5.



(a) Linear vs Cuckoo hash table



(b) Average and tail latency of count array and Count-Min sketch



(c) Precision vs. tail latency of count array and Count-Min sketch

Figure 4.1: Comparing a single hash function with multiple ones

4.3.1 Evaluation settings

Testbed: We use a Xeon E5-2650 v3 processor with 10 cores, 256 KB of L2 cache per core, 25 MB of shared L3 cache, and a 10G network interface card. On this processor, the L1 access time is 1.6 ns, L2 access time is 5 ns, L3 is 15 ns, and main memory is 69 ns [133]. Typically, the access time of L1, L2, L3, and main memory follows a similar trend across the latest CPU architectures [134].

Traffic traces: We use a one-minute trace from Equinix data center at Chicago from CAIDA [135] with 27 million packets and around 1 million unique flows. The CAIDA trace has a skew of Z=1.1 (which means that the most frequent entry has 10 times more packets than the 8^{th} most frequent one [136, 137]). To generate traffic traces with different skews, we build a pool of source and destination IPs from the base CAIDA trace and sample from this pool using a Zipfian distribution.

In all experiments, we use the smallest TCP packet size, i.e., 64 bytes, to stress-test the measurement tasks under the highest possible per packet rate.

Measurement tasks: We focus on heavy hitter detection. We define heavy hitters as the source and destination IP pairs that have more than 0.1% of the total traffic in an epoch. We report in epochs of 2 million packets, which translates to a 130ms time window on a 10Gbps network interface card with 64-byte packets. We evaluate the generality of our observations for tasks that save more information per flow by evaluating heavy hitter detection with a variety of value sizes.

Measurement algorithm implementation: We evaluate three types of measurement algorithms: hash tables including linear hash tables and Cuckoo hash tables, sketches including count arrays and count-min sketches, and heaps (§4.2.1). By default, we keep the keys as source and destination IP pairs and the values as 12 byte counters. For each algorithm, we do not implement unnecessary features (e.g., for heavy hitter detection, we do not need to perform bookkeeping or have a decrement operator). This decision lets us save as many cycles as possible for each algorithm. We now describe our algorithm implementation in detail:

Hash tables: Our implementation of linear hash table holds one item per bucket and performs linear search on collisions. There are also other collision resolution techniques, e.g., Hopscotch [124] or Robin Hood hashing [138]. We opted not to use them, because as the size of the data structure increases the number of collisions decrease, which hides the impact of collision resolution strategy for packet processing. For Cuckoo hash table, we followed DPDK implementation [139] but removed the bookkeeping (required for deletion) to improve the performance.

Sketches: Count-array implementation is similar to the linear hash table, but instead the collision resolution strategy overwrites previous values. Our Count-Min sketch uses three count-arrays with pairwise independent hash functions.

Heaps and trees: We use a binary min heap as a representative tree like data structure for packet processing that is actively used across many algorithms, e.g., change detection [119], heavy hitter detection [105, 111]. We optimized the implementation by ensuring that we only heapify-down when updating values because the flow metrics, e.g., volume or packet count, can only increase.

We perform extensive system optimizations to make the measurement system as efficient as possible. For example, we use DPDK [139] to read packets from the NIC and send them as a batch to the application. Batching packets has several benefits: (a) it gives the compiler more freedom to optimize the code, e.g., through data-flow analysis [140], (b) it enables the instruction level parallelism across packets in the same batch; and (c) the compiler and the programmer can use prefetching and Single Instruction Multiple Data (SIMD) instructions to hide the latency of memory and CPU operations [141, 142].

Evaluation metrics: We consider two metrics: *(1) Performance:* We measure the average and tail latency (i.e., 99th percentile latency). We measure the latency from fetching packets from the NIC to sending the packets out of the measurement module and maintain the histogram. The average latency dictates the packet processing throughput. The tail latency indicates the variance of packet processing time. A larger tail latency causes more packet drops because the NIC needs to maintain a longer queue. Note that this can happen even when the average latency per packet is low. *(2) Accuracy:* We measure the precision and recall for each measurement task. For example, to measure the precision of heavy hitter detection, we count the fraction of selected flows that are true heavy hitters; similarly, the recall is the fraction of true heavy hitters that are detected. The recall and precision of other tasks, e.g., superspreader or change detection, follow the same definition.

Evaluation settings: We run a warm up trace right before each experiment to ensure that the software switch code is cached. We perform zero-packet-loss performance benchmark: for each experiment, we replay the trace at the highest throughput where packet loss is zero.

We process packets in batches of 64. To compute the average and tail latency, as it is too expensive to record the delay per packet, we measure the number of cycles to process each batch and add the corresponding per packet cycle into a histogram. The histogram has 2k buckets with each bucket representing 2 cycles.

4.3.2 A single hash function is better than multiple

We compare data structures with a single hash function to those with multiple hash functions (linear hash table vs. Cuckoo hashing and count array vs. Count-Min sketch). We observe that using a single hash function achieves better performance on average and in tail than using more hash functions without losing accuracy.

The linear hash table has lower average and tail latency than Cuckoo hash table. Figure 4.1a shows the average and the 99th percentile latency for the linear and Cuckoo hash tables. For the Cuckoo hash table, we first consider an implementation with one entry per bucket. For each hash bucket, we store one key-value pair (i.e., one entry per bucket is labeled as *Cuckoo-1 entry*). The Cuckoo hash table has between 30% (40%) to 10% (13%) higher average (tail) latency than the linear hash table over the whole range. This is because, with lookup misses, Cuckoo hashing always needs two hash functions to verify the miss whereas the linear hash table always requires one. This also means that Cuckoo hashing needs to make two random memory accesses, whereas linear hash table only needs to probe the current entry. The locality and predictability of reference in linear hash table and the size of the cache sizes (64 bytes) further help to ensure the availability of next key in cache. This makes linear hash table have an overall better performance even with larger data structures and when the load factor is low.

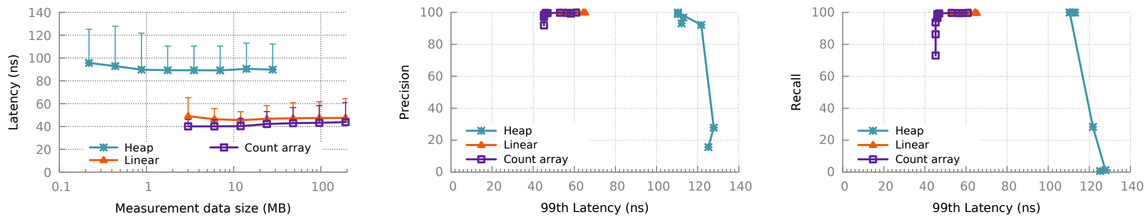
To increase the locality of reference, we may reduce the number of memory operations in the Cuckoo hashing by chaining, e.g., saving four entries per bucket (labeled as *Cuckoo-4 entries*)[143]. Thus, when collisions happen, we can save the entry in the same bucket with high probability without computing the second hash. Note that we chose four entries per bucket because the four entries fit in one cache line. Although Cuckoo-4 improves the tail latency of Cuckoo-1, it still has higher latency compared to linear hash table (Figure 4.1a). This is because with equal sized tables, there are fewer indices available in Cuckoo-4 than linear hash table, and thus, Cuckoo-4 can require multiple comparisons to find the key.

There is a large body of works on using Cuckoo hash tables for applications with high performance such as forwarding tables of switches [144] and for key-values stores [143]. Previous works choose Cuckoo hash tables because they focus on the load factor of the hash table, but in our context, we care less about the load factor since the number of records is much smaller than a table for a key-value store. In other words, Cuckoo hashing is not the fastest in our context because each lookup may require two hash computations and an insertion may require random shuffling of many entries in the hash table. Instead, we can use a large table—because the table size is only a fraction of the total memory size in modern software switches—and avoid computations that allow Cuckoo hash table to achieve a high load-factor.

The count array has lower average and tail latency than the Count-Min sketch. The count array with one hash function has lower average and tail latency with the same accuracy than Count-Min sketch, which uses three hashes, across all data structure sizes (Figure 4.1b). This is because the Count-Min sketch computes multiple hashes and needs multiple random memory accesses per packet, which defeats the purpose of smaller memory size for packet processing. The tradeoff between the performance (i.e., 99th percentile tail latency) and accuracy (i.e., precision[2]) is shown in Figure 4.1c. For example, the count array reaches 98% precision with 45 ns tail latency while the Count-Min sketch takes 64 ns for the same precision due to the

[2]Recall also has the same trend.

additional hash function computations. Even when count array memory does not fit in the CPU cache, most of its memory accesses are still served by the cache because of the packet batching, memory prefetching, and traffic skews, which is common in networks [145, 146].



(a) Average and tail latency (b) Precision vs tail latency (c) Recall vs tail latency

Figure 4.2: Performance and accuracy comparison of hashes, sketches, and heaps (traffic skew Z=1.1)

4.3.3 Use data structures with the simplest computation

We compare three classes of algorithms for heavy hitter detection: count arrays, linear hash tables, and heap-based algorithms. Among the three, count array has the least amount of computations, linear hash table is a bit more complex because of the collision resolution strategy, and the heap-based algorithm is the most computationally demanding but uses smaller memory. We show that using more computation to save memory does not improve the performance.

Count array has the lowest average latency compared to linear hash table and heap. We first compare the average latency of the three algorithms for heavy hitter detection for different sizes of data structures in Fig. 4.2a. The count array has 142% better performance than the heap implementation and 28% better performance than the linear hash table. Figure 4.2a shows that as the size of the data structure grows, the latency difference between the linear hash table and count array vanishes because collisions rarely happen.

The heap has the worst performance among the three algorithms as it takes multiple memory accesses to navigate and maintain the heap data structure. For example, updating a heap entry with a subtree of height three may require updating all the tree layers.

Note that heap is still faster than more complex algorithms such as *hierarchical* sketches [129, 105]. Hierarchical sketches use multiple sketches to extract the heavy hitter flow information from their counters as counters in sketches do not keep the flow information (e.g., IP). However, updating multiple sketches in software requires many hash computations and memory accesses [3].

With larger data sizes, the tail latency increases significantly for the count array and linear hash table, but decreases for heap. The error bars in Figure 4.2a show the 99th percentile tail latency. The tail latencies of count array and linear hash table increase significantly when the measurement data size is above the L3 cache of the CPU (25 MB). If the measurement data is larger than the L3 cache, the memory access latency affects the *tail* latency of the packet processing pipeline.

It is worth noting that small linear hash tables have higher latency than the larger ones. This is due to the high load factor of small tables that incurs additional collision resolution cost. For example, in our experiments, a linear hash table with 3 MB performs 22% more memory accesses than a linear hash table with 200 MB.

However, the average and tail latencies of min-heap decrease with more memory. This is because with larger heaps, more heavy hitters end up in the leaves (versus nodes inside the heap), which makes heapify operation cheap because it only touches the leaves.

To achieve 100% accuracy, we should use the linear hash table; if accuracy loss is acceptable, count array has the best performance. Figure 4.2b compares the tradeoff between accuracy (i.e., precision/recall) and the performance (i.e., latency/tail latency) of different measurement algorithms. Even though

[3]Our approach for using the count array is to simply keep heavy hitters in a set (i.e., add the flow to a set if it updates a counter above the threshold). Thus, we only need one sketch, and count array becomes a better choice than the heap for software.

heap works well with small memory space, it has the highest latency and the worst accuracy among the three algorithms and is never a good choice for measurement in software.

The linear hash table always achieves 100% precision and recall because it handles collisions. Its average latency is $46ns$ and its tail latency is $53ns$. However, count array achieves 99.5% precision and 96.54% recall with $40ns$ average latency and $46ns$ tail latency. Saving $6ns$ in average latency improves the throughput by 9% for the smallest packet size where we only have 67ns to process each packet ($\frac{6_{\text{ns}}}{67_{\text{ns}}}$). The reduction in tail latency also lowers the chance of packet drops in the NIC queue as the maximum queue length drops. Therefore, the count array is the best choice if the consumer of measurement data can tolerate some accuracy loss. For example, for traffic engineering, handling a few small flows as heavy hitters ($<$ 100% precision) or missing a few heavy hitters ($<$ 100% recall) does not have much impact, especially, because the size of false detected heavy hitters and missed heavy hitters is close to the threshold [130].

4.4 Generality to diverse measurement tasks

We discussed that for detecting heavy hitters, large and computationally lightweight data structures have better performance and comparable accuracy to small and complex data structures. Here, we generalize the result to a group of measurement tasks that keep per item state and update that state for every incoming packet. All the six measurements in Table 4.1 follow this model. For example, heavy hitter detection increments the per flow counters, superspreader detection updates a bloom filter per source IP. Such measurement tasks only rely on a data-structure that maps items to their state, i.e., a key-value store. We can implement a key-value store in software using (1) hash tables or (2) tree based algorithms.

Hash tables rely on hash functions and collision resolution strategies to find the location of an item; on the other hand, trees traverse a path from the root node and incur multiple memory accesses to find the location of the item. To compare the solutions, we need to compare the number of cycles used to find the location of a key.

Under no collisions, a hash table requires a single hash function to locate a key-value pair in the table. There are many well designed uniformly random hash function implementations [147], e.g., Metrohash, Cityhash, Murmur3, which typically take between 40~60 cycles for 16 bytes (>5 tuples) of data to execute. In comparison, L2 and L3 accesses take 10 and 40 cycles respectively. Thus, a hash table with no collisions takes between 50~100 cycles to locate the value of a key. On the other hand, a tree based solution requires multiple memory accesses (typically in the $\mathcal{O}(log(n))$ memory accesses and comparisons) to find the location of a key. Assuming the same memory access latency numbers for L2 and L3, a tree that is completely cached in L2 memory can only have between 63~2047 entries—ignoring any computational overhead and branch mispredictions—for a comparable performance to a hash table, which can be much larger. This means that a hash table with no collisions has a much better performance than tree based solutions.

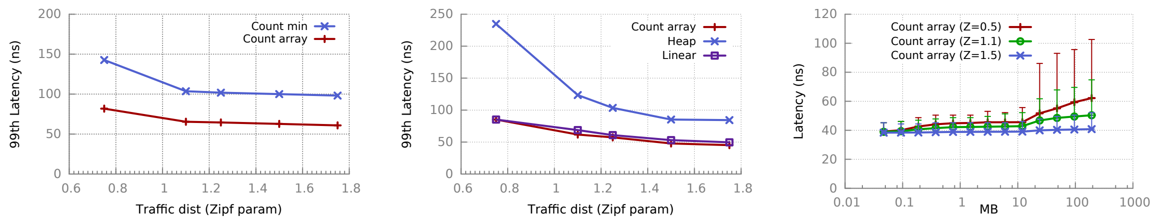
The unique opportunity for network measurement tasks is that they can avoid collisions in hash tables using large tables. This is because the data of measurement algorithms is a fraction of the software memory hierarchy (e.g., 10s of MBs compared to 10s of GBs available on modern software switches). Thus, we can make the hash tables large enough that collisions become rare. Furthermore, we can mask the memory access latency through packet batching and prefetching. In contrast, the database and hardware switch community [143, 148], where most streaming algorithms come from, do not have the luxury of serving most queries from cache and thus have to rely on trading off computation and accuracy for memory size.

Finally, different measurement tasks have different strategies for updating the values associated with the keys. For example, when using count array for heavy hitter detection, values that map to the same bucket overwrite each other, whereas a linear hash table would resolve collisions through probing, and a heap would move the items around to preserve the heap property. Later in this Section, we discuss how the general result, use simple but large data structures, also apply to superspreader, which has complex memory access procedure for updates, and change detection, which is computationally complex.

4.4.1 Impact of traffic skew, data structure size, and value size

The efficiency of memory hierarchy in software switches depends on the location of a state associated with a packet. This is because when the state is in upper layers of the memory hierarchy, the access latency becomes multiplicatively slower. For example, on our test server, the access latency of memory is 4.6 times slower than L3. There are two factors that dictate the location of a packet state in the memory hierarchy: (1) Traffic skew. With a skewed traffic, the packet processing pipeline serves a larger fraction of packets from the cache, leading to overall lower latency per packet. In contrast, a more uniform traffic distributes the state associated with a packet across all the layers, leading to higher latency per packet. (2) The data structure size. Whereas the data of a small data structure may fit in L1-L3 cache, a large data structure might still need to access memory to locate its data, leading to overall higher latency per packet. Here we discuss the impact of entry size, traffic skew, and data-structure size on the performance of packet processing pipelines.

Traffic skew. We study the impact of skew on measurement tasks by fixing the measurement task to heavy hitter detection and the data structure size to 32 MB. This size ensures that the measurement task does not fit in the L3 cache in our test server.



(a) Hash based algorithms (32MB) (b) Measurement algorithms (32MB) (c) Measurement data sizes

Figure 4.3: Effect of traffic skews on measurement algorithms and sizes

We first compare the impact of the skew across implementations with varying number of hash functions and memory accesses. Typically, as the traffic skew decreases, access patterns distribute more evenly across the memory hierarchy. Thus, measurement tasks with lower number of memory accesses per packet are less affected by the skew. Fig. 4.3a shows the tail latency of count array and Count-Min sketch for heavy hitter detection. Since the Count-Min sketch makes 3 memory access per packet as opposed to only one for count array, the jump from skew 1.1 to 0.75 is larger for the Count-Min than the count array—even though the amount of computation per packet does not change with the skew.

Then, we compare the impact of skew across the heap, count array sketch, and linear hash table. Fig. 4.3b shows the 99th percentile per packet processing latency of these implementations across varying skews. Since the data structure is large (32 MB), the collisions are rare, and thus, the performance of linear hash table is only slightly worse than the count array sketch. Thus, under our settings, operators may prefer the linear hash table because it provides a guaranteed 100% accuracy with negligible impact on latency versus the count array.

However, the effect of skew on heap is more prominent: the skew not only affects the number of memory accesses but also the amount of computation that the heap performs. This is because under low skew the heap is more likely to move an item across multiple levels than when the skew is high. Fig. 4.3b shows this where the heap latency grows by 45ns from skew 1.5 to 1.1, but by more than 100ns when going from skew 1.1 to 0.75.

Data structure size. To study the effect of skew changes together with the size of data structures, we fix the measurement task algorithm to count array and measure the packet latency when the traffic skew changes. Here, the skew dictates the working set of the data structure in L1, L2, and L3 cache of CPU. Traffic with higher skew is more likely to access the lower layer cache for packet data than the traffic with higher latency. Figure 4.3c shows the average and tail latency of the count array when we change its size from 48 KB to 200 MB over different traffic skews.

The two jumps at 200kB and 32MB indicate the size of the L2 and L3 cache. When the data structure is small enough to fit in L2 cache, no matter how the access pattern looks like, it will always get served from the L1 and L2 cache. As the data structure size increases, data gets distributed across other layers of the memory hierarchy. With less skewed traffic, we are more likely to access upper layer memories, and thus the latency gets affected more. This is visible in the figure by the separation of latency for different traffic skews once we pass the L3 cache size.

Entry size. A key factor for the difference in performance of measurement algorithms is the size of the stored state. The size of the entry dictates the percentage of the entries that are available in the lower layer of the memory hierarchy. Fewer number of larger entries fit in lower layer cache as opposed to smaller entries.

An entry contains both the key and the value. Typically, the key size depends on the flow granularity, e.g., whether we keep one IP address (4 bytes), source and destination IP addresses (8 bytes), or 5 tuples (13 bytes). For the value field, we keep a 4 byte counter together with the first few bytes of the latest packet to fill out the remaining space for that entry. We fix the key size to avoid incurring additional memory comparisons and hash function computation overhead and only keep the source and destination IPs (8 byte keys).

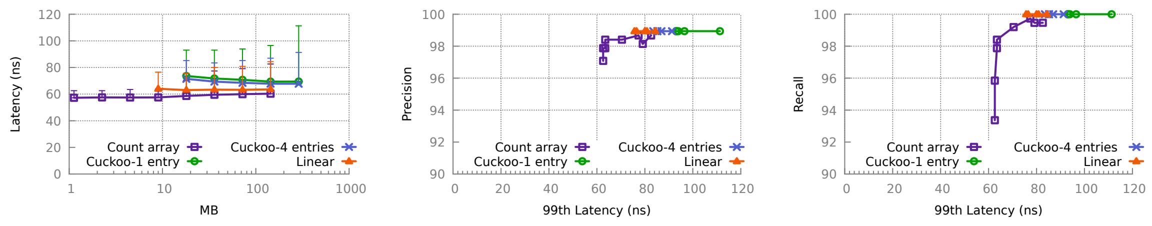
We implementat heavy hitter detection algorithms as discussed in the previous section. Measurement tasks may keep additional information, e.g., timestamp per flow (with a total value size of 12 bytes) or keep a list of destinations that the flow has contacted (e.g., 20 bytes on average). But that should not affect the generality of the impact of entry size on the performance.

Fig. 4.6 shows that the tail latency of count array, linear hash table, and heap increases as the entry size grows. This is because we are more likely to access the upper layers of the memory hierarchy to locate our data. The jump for the heap here is linear and smaller than the jump shown in Fig. 4.3b because here the number of memory accesses or the amount of computation of the heap does not change with varying entry size—we still use the packet counter to reorder the heap.

4.4.2 Impact of measurement tasks and storage of key-values

To cover the impact of memory and computational aspect of measurement task, we study two tasks: (1) superspreader detection, which updates a large memory portion per value, and (2) change detection, which is computationally more intensive than heavy hitter detection. We show that our results from the previous section still hold even on the two extremes of memory and computation complexity. Finally, we study the impact of value size on the performance, and suggest a strategy to decide whether the key and values should be colocated in the hash table or not.

Superspreader detection. Superspreaders are the sources that chat with a large number of distinct destinations. They can identify distributed denial of service attacks (DDoS) or sudden changes in traffic pattern. We implement the superspreader module to report all the source IPs that send traffic to more than 128 different destinations in every epoch (2 mil packets). For every source IP, we keep a distinct Bloom filter counter per entry [149] with three hash functions and 1024 bits of data to identify new destinations. Due to the Bloom filter, superspreader has a more complex *update* procedure than heavy hitter detection.

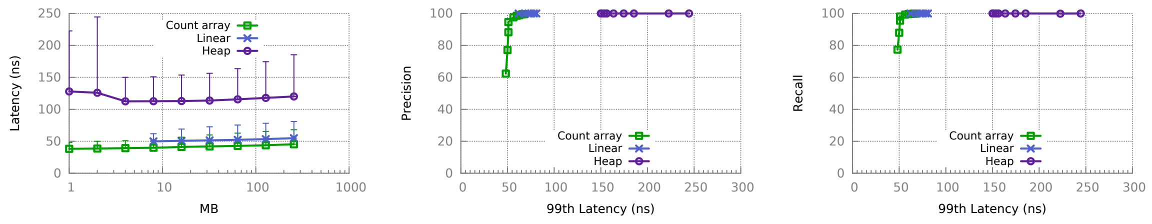


(a) Latency/Size (b) Precision/99th Latency (c) Recall/99th Latency

Figure 4.4: Performance and accuracy of superspreader detection

Figure 4.4b shows the average and tail latency of different implementations of superspreader detection. Count array still has the lowest latency among the algorithms while reaching 97% precision (Figure 4.4b). The hash tables all have a precision of 99% and recall of 100% (The accuracy is less than 100% because of the Bloom filter error in distinct counting) with linear hash table being the fastest. The conclusion here follows the result for heavy hitter detection algorithms.

Change detection. Change detection identifies anomalies in packet streams, e.g., when the traffic pattern of a host suddenly changes or when the traffic volume changes too rapidly. Operators can use change detection for detecting compromised hosts, or as a signal to a control framework, e.g., load balancing, when sudden changes happen. For evaluation, we use an EWMA model to predict the traffic of each flow and report the flows that are outside the predicted value. Due to the prediction model, change detection is more computationally intensive than heavy hitter detection in updating per flow state.



(a) Latency/Size (b) Precision/99th Latency (c) Recall/99th Latency

Figure 4.5: Performance and accuracy of change detection

Figure 4.5b shows the average and tail latency for different implementations of change detection. Count array still has the lowest latency among all the algorithms while reaching 99% precision (Figure 4.5b). The linear hash table has a precision of 100% and recall of 100%. Similarly, heap also has a precision and recall of 100% but with 80-100 ns higher latency.

Direct and indirect key-value storage. For measurement tasks with large values, it is better to store the values separately and only store a pointer in the hash table. We can then keep a contiguous list of keys to increase the locality of memory accesses for lookups when collisions happen. However, when the value size is small, it is more beneficial to keep the key and values together so that they share the cache line. To understand this tradeoffs, we implement two versions of linear hash tables: *Linear* which store keys and values together and *LinearPtr* which stores the keys with a pointer to the values.

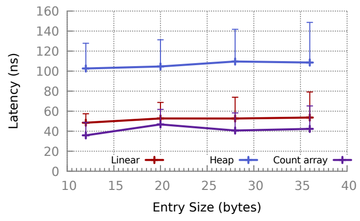
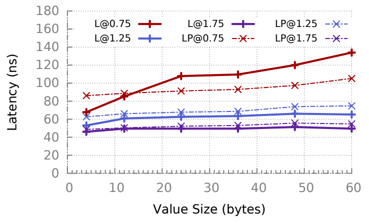
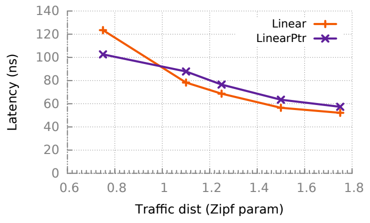


Figure 4.6: Latency of measurement algorithms for tasks with various entry sizes (traffic skew Z=1.1)



(a) Impact of value sizes



(b) Impact of traffic skews

Figure 4.7: Comparing linear hash tables with and without pointers (key size=48 bytes)

Figure 4.7a shows the tail latency of both solutions with different traffic skews. For the lowest skew ($Z = 0.75$), the working set does not fit in cache and entries come in and go out of the cache. Each pointer is 8 bytes so keeping values smaller than 8 bytes only incurs additional delay. However as the value becomes larger, using a pointer becomes more beneficial. For example, for value size of 60 bytes, using a value pointer (LinearPtr) decreases the tail latency by 30%. This is because with large values lookups and insertions in a linear hash table are more likely to traverse multiple cache lines. Instead, value pointers promote key locality, which improve insertions and lookups by lowering cache lines that we go through.

For higher skew traffic ($Z = 1.75$ and $Z = 1.25$), the working set is small enough to fit in the CPU cache while the additional memory accesses due to the separation of keys and values has negligible overhead (about 5ns).

4.5 Measurement algorithms on multiple cores

Measurement tasks never run in a standalone fashion. With a pipeline of network functions, it becomes harder for a single core to sustain the line rate packet processing. To get around this, we can load balance the incoming traffic across multiple cores based on a hash of the flows [150]; each core then runs the pipeline for a subset of flows [151]. Although, this leaves us with isolated measurement functions on each core and requires state synchronization across the cores. In this section, we will first investigate how to share the measurement data across cores running only measurement tasks. We will then study the impact of sharing resources with other applications.

4.5.1 Sharing states across multiple cores

When a measurement function runs over multiple cores, we need to synchronize states across cores. Maintaining locks on the shared state for consistency has a huge overhead, especially when the cache line that holds the lock is passed between the cores [152]. To get around this, we can either use (a) shared lockless data structures or (b) separated data structure for each core.

Shared lockless data structures. The linear hash table and the count array are easy to implement in a lockless fashion. For example, we can use compare-and-swap (or similar atomic operations) to update a counter atomically in a multithreaded environment. However, it is harder to implement lockless access for more complex data structures such as a heap.

Separated data structures. Each core maintains its own copy of the data structure. When we need to report the overall measurement results, we can merge the state/results from each data structure accordingly. Typically, merging the measurement results from multiple cores has little overhead if the reporting frequency is a few orders of magnitude greater than the packet processing time (e.g., >10ms reporting frequency vs. 67ns processing per packet). This is because a separate core can merge the data with low memory bandwidth usage. For example, with a measurement interval of 100ms and a 5MB data structure per core, a reporting core merging measurements of 10 cores only requires 500MB/s memory bandwidth, which is less than 1% of the total available memory bandwidth of our Xeon processor.

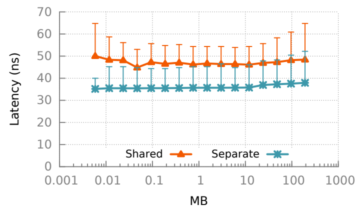
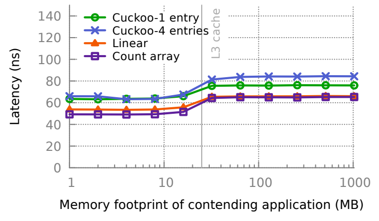
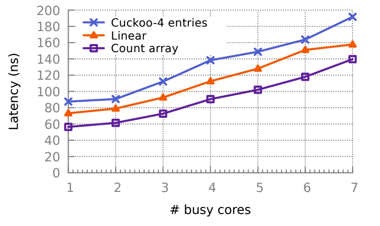


Figure 4.8: Latency of shared vs. separate count array across two cores



(a) Impact of L3 cache contention



(b) Impact of memory controller contention

Figure 4.9: Impact of resource sharing across applications

Separated option has lower latency than shared option. Figure 4.8 compares the latency of heavy hitter detection for the two options using a count array of different sizes. The average and tail latency of the separate approach are consistently lower than the shared one (The accuracy is not shown because it is the same). For example, when the size of count array is 32 KB, the tail latency of the shared count array is 12 ns higher than the separated count arrays. This is because of the overhead of running the compare and swap operations for maintaining the consistency of the shared data structure. Moreover, because the L3 cache is shared, the cache-coherency protocol will perform additional operations when a cache line in a core is read by a different core. On modern CPUs, this additional overhead can be as large as 40 cycles [153]. In contrast, the overhead of merging separate data structures is lower because we only need to pay the overhead at the reporting time rather than on a per-packet basis. Thus, saving memory also decreases the performance on multiple cores because it requires some sort of synchronization and wastes CPU cycle.

The shared count array latency is initially high and then decreases by 10% and remains almost constant until 20MB. For smaller count-arrays there is a higher chance that the two cores access the same entry (and cache line) in the count array, leading to extra latency due to compare-and-swap and the cache coherency protocol.

4.5.2 Sharing resources with other applications

The measurement tasks may run in conjunction with other network functions or applications on the same machine. Because all of these applications share resources, e.g., the cache and memory bandwidth, they end up affecting each other. For example, previous works have shown that cache-hungry applications can degrade the performance of other network functions [151]. To understand the impact of sharing resources on measurement algorithms, in addition to running the heavy hitter detection algorithm, we run two types of concurrent applications: (a) We run a single L3 aggressive application on a separate core that accesses random memory locations to show the impact of the contention on the L3 cache; (b) We run multiple of such applications on different cores that aggressively read and write memory to show the impact of the contention at the memory controller.

Impact of the L3 cache contention. We run a memory aggressive application on a core in the same NUMA domain as our measurement task. The application uses a hash function to access a random memory address and increment the value there. To guarantee that this application has higher priority for using the L3 cache than our measurement pipeline, we lowered the traffic rate so that the measurement task accesses the L3 at a much slower pace than the application core. We then measure the latency of the measurement task as the memory footprint of this application increases.

Figure 4.9a shows that the latency of the measurement task remains almost constant up to the L3 cache size. After, the memory aggressive application starves L3 cache and leaves no room for the measurement task, which cause the measurement task to access the main memory, leading to the sudden jump. Increasing the memory footprint of cache aggressive applications further does not increase the latency. This is because the next bottleneck is the memory bandwidth and our bandwidth usage is less than 10% of the available bandwidth of a NUMA domain (1.1GB/s out of 17GB/s) [154].

Impact of memory controller contention. Today, many big data analytics frameworks rely on the large memory available on modern servers to improve their performance. For example, Spark [155] keeps most of the intermediate data in memory for later usage; Hadoop [35] keeps portions of the files in memory for faster successive accesses. These applications can quickly drain the available memory bandwidth. Previous studies [156] show that Spark on average uses 40% of the memory bandwidth can can burst up to 90%. This high memory bandwidth usage affects the performance of the measurement tasks running on the same server. To study this, we wrote an application that aggressively utilizes the memory bandwidth. A single instance of this application utilizes 12GB/s of the 17GB/s of memory bandwidth[4]. We run many instances of this application to increase the contention of the memory bandwidth.

Figure 4.9b shows that as the number of applications increases, the latency of the measurement task increases. This is because with more requests to the memory controller, it becomes harder for the measurement task to fetch the packet data from memory, and therefore, with 7 cores the average latency of the measurement task increases by a factor of 2.9 for the count array.

Note that in both cache and memory bandwidth contention scenarios, the differ-

[4]We found out that even by running multiple instances of this application, we cannot utilize more than 14.5GB/s of the bandwidth, which we attribute to the queuing effect and the CPU parameters.

ences between the measurement algorithms still hold. The count array always has the lowest latency in all settings.

4.6 Related Work

In addition to the related works covered in Section 4.2, our previous workshop paper [157] performed a preliminary evaluation of measurement algorithms. This chapter extends the workshop paper in the following aspects: (1) Implementation: Our previous study was on the Click modular router [158], which is limited in throughput as it did not let us use advanced techniques such as batching and packet data prefetching from the cache. In this chapter, we run all algorithms directly on DPDK and apply different techniques to reach the maximum packet rate. (2) Algorithms: Our previous study mainly focuses on count array, Count-Min sketch, and heap. In addition, this chapter investigates more in hash table implementation. It compares the linear hash table and the Cuckoo hashing and shows that the linear hash table is the fastest choice when we need 100% accuracy. (3) Measurement tasks: In addition to heavy hitter detection in [157] which identifies keys with heavy volume counters, we also tested superspreader detection which counts the number of distinct items and change detection which identifies anomalies in traffic. (4) Settings: We also evaluate these algorithms on a variety of scenarios including multiple cores, different traffic skews, and a variety of entry sizes.

In addition to the three classes of algorithms introduced in Section 4.2, there are other packet and flow sampling solutions [159, 115, 160, 116]. These solutions are orthogonal to our algorithms and can always be combined to reduce the measurement load.

Recent works on optimizing the performance of network function in software switches [151, 161] mostly focus on better management of the memory usage of different network functions. Our work can help improve the performance of network functions by guiding developers to design and select the best measurement algorithms. Dobrescu et al. [151] associate the degradation of the network functions performance with the number of L3 references that competing applications make. We give insights on how to improve the performance of measurement components in such settings.

4.7 Discussion

Theoretical model. While a theoretical model for estimating the latency of measurement pipeline helps in making design and optimization decisions, it is a challenging task as the performance of the packet processing pipeline depends on many factors, e.g., implementation of the algorithm (packet batching and/or prefetching, SIMD instructions), other resident applications, CPU properties (pipelining, speculative execution). Our previous work [157] shows a preliminary model for estimating the measurement algorithm latency. We incorporate the above factors into the model in the future.

4.8 Conclusion

With the trend of running network functions in software, keeping states inside these functions, and performing measurement to guide the deployment of these functions, it is important to understand which algorithms and data structures work the best in software. The key metrics in software are performance and accuracy rather than memory and accuracy in hardware. Our experiments and analysis show that simple is often the best. For measurement tasks that do not require perfect accuracy, a count array, which is general enough for a wide range of measurement tasks, has the lowest latency and the highest throughput. For tasks that require 100% accuracy, we recommend a linear hash table. We verified this conclusion for a variety of traffic settings, measurement tasks, and multiple core settings.

Chapter 5

Conclusions

In this dissertation, we explored how we can utilize effective searching algorithms that adapt to changes in the underlying systems while being accurate and cost-efficient. In Cherrypick, we looked at Bayesian optimization which adapts to variations in software, workload, and machine types in the cloud for big data analytic workloads. In Janus, we leveraged the symmetry of data center networks to replace a brute force search over an exponentially large space (deployment plans) with a brute force search over a much smaller space. Finally, in our study of measurement algorithms in software, we showed how a simple hash table can adapt to variations in traffic and workload while more complex solutions fail when traffic or workload changes.

The central theme to making adaptable tools is to focus our effort on building algorithms for a layer that changes slower than the problem we are solving. More concretely, in Cherrypick, we assume that most of the time, moving software or workload from one cloud configuration to another closely related cloud configuration does not result in radically different behavior. Said differently, there is a smoothness to the function that maps cloud configurations of a workload to performance and this is independent of the workload. This assumption gives the modeling and searching approach tremendous power over where to search. Similarly, in Janus, we note that the symmetry of a data center is an indivisible part of a data center—without it managing and debugging a data center becomes almost impossible—so we leverage this knowledge to reduce the search space while allowing each operator to encode their risk and expectations. And finally, in our study of software switches, todays' traffic workload follows a pattern that fits in the cache of any modern CPU, suggesting that an algorithm that does the least amount of computation has the best performance. Ultimately, a systematic study of searching algorithms and how we can use them to build adaptable tools is left to future work.

Bibliography

[1] Eric Brochu, Vlad M Cora, and Nando De Freitas. A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599*, 2010.

[2] Amazon EC2 Instance Types. `https://aws.amazon.com/ec2/instance-types/`.

[3] Microsoft Azure: Virtual Machine Pricing. `https://azure.microsoft.com/en-us/pricing/details/virtual-machines/`.

[4] Google Cloud Platform: Machine Types. `https://cloud.google.com/compute/docs/machine-types`.

[5] Leveraging AWS Global Backbone for Data Center Migration and Global Expansion, 2020.

[6] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven WAN. In *ACM SIGCOMM*, 2013.

[7] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a globally-deployed software defined WAN. *ACM SIGCOMM*, 2013.

[8] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. SIGCOMM, 2015.

[9] AWS: Previous Generation Instances, 2021.

[10] AI dungeon 2 costing over $10k/day to run on GCS/Colab, 2019.

[11] Enterprise Cloud Computing on AWS, 2019.

[12] Sameer Agarwal, Srikanth Kandula, Nicolas Bruno, Ming-Chuan Wu, Ion Stoica, and Jingren Zhou. Re-optimizing Data-parallel Computing. NSDI, 2012.

[13] Andrew D Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM European Conference on Computer Systems*, 2012.

[14] Ang Li, Xiaowei Yang, Srikanth Kandula, and Ming Zhang. CloudCmp: comparing public cloud providers. In *SIGCOMM*, 2010.

[15] G. Wang and T. Ng. The impact of virtualization on network performance of Amazon EC2 data center. In *INFOCOM*, 2010.

[16] Custom Machine Types - Google Cloud Platform. `https://cloud.google.com/custom-machine-types/`.

[17] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *NSDI*, 2016.

[18] Jonas Mockus. *Bayesian approach to global optimization: theory and applications.* Springer Science & Business Media, 2012.

[19] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical Bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems*, 2012.

[20] Amazon EC2 - Elastic Balance Storage. `https://aws.amazon.com/ebs/`.

[21] Herodotos Herodotou, Fei Dong, and Shivnath Babu. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. In *Proceedings of the 2nd ACM Symposium on Cloud Computing.* ACM, 2011.

[22] Abhishek Verma, Ludmila Cherkasova, and Roy H Campbell. ARIA: automatic resource inference and allocation for mapreduce environments. In *Proceedings of the 8th ACM International Conference on Autonomic Computing.* ACM, 2011.

[23] Vladimir N. Vapnik. *Statistical Learning Theory.* John Wiley & Sons, 1998.

[24] David JC MacKay. Introduction to Gaussian processes. *NATO ASI Series F Computer and Systems Sciences*, 1998.

[25] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 2015.

[26] Eric Anderson. HPL-SSP-2001-4: Simple table-based modeling of storage devices, 2001.

[27] Ilya Loshchilov and Frank Hutter. CMA-ES for Hyperparameter Optimization of Deep Neural Networks. *arXiv preprint arXiv:1604.07269*, 2016.

[28] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press Cambridge, 1998.

[29] Carl Edward Rasmussen. Gaussian processes for machine learning. MIT Press, 2006.

[30] Donald R Jones, Matthias Schonlau, and William J Welch. Efficient global optimization of expensive black-box functions. *Journal of Global optimization*, 1998.

[31] Jacob R Gardner, Matt J Kusner, Zhixiang Eddie Xu, Kilian Q Weinberger, and John Cunningham. Bayesian Optimization with Inequality Constraints. In *International Conference on Machine Learning*, 2014.

[32] Ilya M Sobol. On quasi-monte carlo integrations. *Mathematics and Computers in Simulation*, 1998.

[33] Spearmint. `https://github.com/HIPS/Spearmint`.

[34] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot Topics in Cloud Computing*, 2010.

[35] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2009.

[36] TPC Benchmark DS. `http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-ds_v2.3.0.pdf`.

[37] TPC Benchmark H. `http://www.tpc.org/tpc_documents_current_versions/pdf/tpch2.17.1.pdf`.

[38] Owen O'Malley. Terabyte sort on apache hadoop. *Yahoo, available online at: http://sortbenchmark.org/Yahoo-Hadoop.pdf*, 2008.

[39] Apache Spark the fastest open source engine for sorting a petabyte. `https://databricks.com/blog/2014/10/10/spark-petabyte-sort.html`.

[40] Alexander Rasmussen, George Porter, Michael Conley, Harsha V. Madhyastha, Radhika Niranjan Mysore, Alexander Pucher, and Amin Vahdat. TritonSort: A balanced large-scale sorting system. In *NSDI*, 2011.

[41] Performance tests for Spark. `https://github.com/databricks/spark-perf`.

[42] Min Li, Liangzhao Zeng, Shicong Meng, Jian Tan, Li Zhang, Ali R. Butt, and Nicholas Fuller. MrOnline: MapReduce online performance tuning. In *International Symposium on High-performance Parallel and Distributed Computing*, 2014.

[43] Bowei Xi, Zhen Liu, Mukund Raghavachari, Cathy H. Xia, and Li Zhang. A smart hill-climbing algorithm for application server configuration. In *International Conference on World Wide Web*, 2004.

[44] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. Starfish: A Self-tuning System for Big Data Analytics. In *Conference on Innovative Data Systems Research*, 2011.

[45] Tao Ye and Shivkumar Kalyanaraman. A recursive random search algorithm for large-scale network parameter configuration. *SIGMETRICS Perform. Eval. Rev.*, 2003.

[46] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando de Freitas. Taking the human out of the loop: a review of Bayesian optimization. *Proceedings of the IEEE*, 2016.

[47] Google VM rightsizing service. `https://cloud.google.com/compute/docs/instances/viewing-sizing-recommendations-for-\instances`.

[48] Yurong Jiang, Lenin Ravindranath, Suman Nath, and Ramesh Govindan. WebPerf: Evaluating what-if scenarios for cloud-hosted web applications. In *SIGCOMM*, 2016.

[49] Peter Bodik, Rean Griffith, Charles Sutton, Armando Fox, Michael I Jordan, and David A Patterson. Automatic exploration of datacenter performance regimes. In *Proceedings of the 1st workshop on Automated control for datacenters and clouds*. ACM, 2009.

[50] Mengzhi Wang, Kinman Au, Anastassia Ailamaki, Anthony Brockwell, Christos Faloutsos, and Gregory R Ganger. Storage device performance prediction with CART models. In *Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*. IEEE, 2004.

[51] Zhuoyao Zhang, Ludmila Cherkasova, Abhishek Verma, and Boon Thau Loo. Automated Profiling and Resource Management of Pig Programs for Meeting Service Level Objectives. In *International Conference on Autonomic Computing*, 2012.

[52] Christina Delimitrou and Christos Kozyrakis. QoS-Aware Scheduling in Heterogeneous Datacenters with Paragon. *ACM Trans. Comput. Syst.*, 2013.

[53] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and QoS-aware Cluster Management. ASPLOS. ACM, 2014.

[54] Whetlab. `http://www.whetlab.com/`.

[55] Valentin Dalibard. A framework to build bespoke auto-tuners with structured Bayesian optimisation. Technical report, University of Cambridge, Computer Laboratory, 2017.

[56] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the Social Network's (Datacenter) Network. In *SIGCOMM*, 2015.

[57] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a Globally-deployed Software Defined Wan. SIGCOMM, 2013.

[58] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving High Utilization with Software-driven WAN. SIGCOMM, 2013.

[59] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, et al. Onix: A distributed control platform for large-scale production networks. In *OSDI*, 2010.

[60] Software for Open Netwroking in the Cloud. `https://bit.ly/2WbFspS`, 2016.

[61] Sean Choi, Boris Burkov, Alex Eckert, Tian Fang, Saman Kazemkhani, Rob Sherwood, Ying Zhang, and Hongyi Zeng. Fboss: Building switch software at scale. In *SIGCOMM*, 2018.

[62] Amin Vahdat Jim Wanderer. Google Cloud using P4Runtime to build smart networks. `http://bit.ly/2HG2jG4`, 2018.

[63] Ramesh Govindan, Ina Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. Evolve or Die: High-Availability Design Principles Drawn from Googles Network Infrastructure. In *SIGCOMM*, 2016.

[64] Albert Greenberg, James Hamilton, David A. Maltz, and Parveen Patel. The

cost of a cloud: Research problems in data center networks. *SIGCOMM Computer Communication Review*, 2008.

[65] Azure Service Level Agreements. `https://bit.ly/1TnZwOn`, 2018.

[66] Amazon Compute Service Level Agreement. `https://amzn.to/2NMCHuJ`, 2018.

[67] Navendu Jain and Rahul Potharaju. When the network crumbles: An empirical study of cloud network failures and their impact on services. In *SOCC*, 2013.

[68] Why The Drain in the Bathtub Curve Matters, 2012.

[69] Xin Wu, Daniel Turner, Chao-Chih Chen, David A. Maltz, Xiaowei Yang, Lihua Yuan, and Ming Zhang. Netpilot: Automating datacenter network failure mitigation. SIGCOMM '12, 2012.

[70] B. Beyer, C. Jones, J. Petoff, and N.R. Murphy. *Site Reliability Engineering: How Google Runs Production Systems*. 2016.

[71] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz André Barroso. Failure trends in a large disk drive population. In *FAST*, 2007.

[72] Large Installation System Administration Conference, 2019.

[73] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Klaus-Tycho Förster, Arvind Krishnamurthy, and Thomas Anderson. Understanding and mitigating packet corruption in data center networks. In *SIGCOMM*, 2017.

[74] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas Anderson. F10: A fault-tolerant engineered network. NSDI, 2013.

[75] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, , Parantap Lahiri, Dave Maltz, and and. Vl2: A scalable and flexible data center network. In *SIGCOMM*, 2009.

[76] Arista warrior. `http://bit.ly/2DxBYoI`, 2012.

[77] Eugene M Luks. Isomorphism of graphs of bounded valence can be tested in polynomial time. *Journal of computer and system sciences*, 1982.

[78] Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, {II}. *Journal of Symbolic Computation*, 2014.

[79] Simon Kassing and Asaf Valadarsky and Gal Shahaf and Michael Schapira and Ankit Singla. Beyond Fat-Trees without Antennae, Mirrors, and Disco-Balls. SIGCOMM, 2017.

[80] Chuanxiong Guo and Guohan Lu and Dan Li and Haitao Wu and Xuan Zhang and Yunfeng Shi and Chen Tian and Yongguang Zhang and Songwu Lu and Guohan Lv. BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers. SIGCOMM, 2009.

[81] Brandon Schlinker, Radhika Niranjan Mysore, Sean Smith, Jeffrey C. Mogul, Amin Vahdat, Minlan Yu, Ethan Katz-Bassett, and Michael Rubin. Condor: Better Topologies Through Declarative Design. SIGCOMM, 2015.

[82] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P. Brighten Godfrey. Jellyfish: Networking Data Centers Randomly. NSDI'12, 2012.

[83] Hongqiang Harry Liu, Srikanth Kandula, Ratul Mahajan, Ming Zhang, and David Gelernter. Traffic engineering with forward fault correction. In *SIGCOMM*, 2014.

[84] Ennan Zhai and Ruichuan Chen and David Isaac Wolinsky and Bryan Ford. Heading Off Correlated Failures through Independence-as-a-Service. OSDI, 2014.

[85] Alok Kumar, Sushant Jain, Uday Naik, Anand Raghuraman, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C. Stephen Gunn, Jing Ai, Björn Carlin, Mihai Amarandei-Stavila, Mathieu Robin, Aspi Siganporia, Stephen Stuart, and Amin Vahdat. BwE: Flexible, Hierarchical Bandwidth Allocation for WAN Distributed Computing. SIGCOMM '15, 2015.

[86] Emilie Danna, Subhasree Mandal, and Arjun Singh. A practical algorithm for balancing the max-min fairness and throughput objectives in traffic engineering. INFOCOM, 2012.

[87] Masoud Moshref, Minlan Yu, Abhishek Sharma, and Ramesh Govindan. Scalable Rule Management for Data Centers. NSDI, 2013.

[88] Timothy Wood, Prashant Shenoy, Arun Venkataramani, and Mazin Yousif. Sandpiper: Black-box and Gray-box Resource Management for Virtual Machines. *Comput. Netw.: The International Journal of Computer and Telecommunications Networking*, 2009.

[89] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Microte: Fine grained traffic engineering for data centers. CoNEXT '11, 2011.

[90] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *EuroSys*, 2015.

[91] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. SIGCOMM, 2011.

[92] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for Network Update. SIGCOMM, 2012.

[93] Hongqiang Harry Liu, Xin Wu, Ming Zhang, Lihua Yuan, Roger Wattenhofer, and David Maltz. zUpdate: Updating Data Center Networks with Zero Loss. In *SIGCOMM*, 2013.

[94] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. Dynamic Scheduling of Network Updates. SIGCOMM, 2014.

[95] Michael Isard. Autopilot: Automatic Data Center Management. SIGOPS, 2007.

[96] Peter Bodik, Ishai Menache, Mosharaf Chowdhury, Pradeepkumar Mani, David A. Maltz, and Ion Stoica. Surviving Failures in Bandwidth-constrained Datacenters. SIGCOMM, 2012.

[97] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. Control Plane Compression. SIGCOMM, 2018.

[98] Gordon D. Plotkin, Nikolaj Bjørner, Nuno P. Lopes, Andrey Rybalchenko, and George Varghese. Scaling network verification using symmetry and surgery. POPL '16, 2016.

[99] Amin Vahdat. Enter the Andromeda zone - Google Cloud Platform's Latest Networking Stack, 2014.

[100] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. Ananta: Cloud Scale Load Balancing. In *SIGCOMM*, 2013.

[101] AT&T Domain 2.0 vision white paper, 2013.

[102] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J. Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Jonathan Stringer, Pravin Shelar, Keith

Amidon, and Martín Casado. The design and implementation of open vswitch. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, 2015.

[103] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. In *NSDI*, 2014.

[104] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *J. Algorithms*, 2004.

[105] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *ICDT*, 2005.

[106] Graham Cormode and S. Muthukrishnan. An Improved Data Stream Summary: The Count-Min Sketch and its Applications. *Journal of Algorithms*, 2005.

[107] G Cormode and S Muthukrishnan. Space Efficient Mining of Multigraph Streams. In *PODS*, 2005.

[108] Robert Schweller, Zhichun Li, Yan Chen, Yan Gao, Ashish Gupta, Yin Zhang, Peter A. Dinda, Ming-Yang Kao, and Gokhan Memik. Reversible Sketches: Enabling Monitoring and Analysis over High-speed Data Streams. *Transaction on Networking*, 2007.

[109] Michael Greenwald and Sanjeev Khanna. Space-efficient Online Computation of Quantile Summaries. In *SIGMOD*, 2001.

[110] Shobha Venkataraman, Dawn Song, Phillip B. Gibbons, and Avrim Blum. New Streaming Algorithms for Fast Detection of Superspreaders. In *NDSS*, 2005.

[111] Michael Mitzenmacher, Thomas Steinke, and Justin Thaler. Hierarchical heavy hitters with the space saving algorithm. *arXiv preprint arXiv:1102.5540*, 2011.

[112] Qun Huang, Xin Jin, Patrick P. C. Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. SketchVisor: Robust network measurement for software packet processing. In *SIGCOMM*, 2017.

[113] Minlan Yu, Lavanya Jose, and Rui Miao. Software Defined Traffic Measurement with OpenSketch. In *NSDI*, 2013.

[114] Faisal Khan, Nicholas Hosein, Chen-Nee Chuah, and Soheil Ghiasi. Streaming Solutions for Fine-Grained Network Traffic Measurements and Analysis. In *ANCS*, 2011.

[115] Cristian Estan and George Varghese. New Directions in Traffic Measurement and Accounting. In *SIGCOMM*, 2002.

[116] Vyas Sekar, Michael K. Reiter, and Hui Zhang. Revisiting the Case for a Minimalist Approach for Network Flow Monitoring. In *IMC*, 2010.

[117] Abhishek Kumar, Minho Sung, Jun (Jim) Xu, and Jia Wang. Data Streaming Algorithms for Efficient and Accurate Estimation of Flow Size Distribution. In *SIGMETRICS*, 2004.

[118] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding Frequent Items in Data Streams. In *Automata, Languages and Programming.* 2002.

[119] Ying Zhang. An Adaptive Flow Counting Method for Anomaly Detection in SDN. In *CoNEXT*, 2013.

[120] Ping Li and Cun-Hui Zhang. A New Algorithm for Compressed Counting with Applications in Shannon Entropy Estimation in Dynamic Data. In *COLT*, 2011.

[121] Ashwin Lall, Vyas Sekar, Mitsunori Ogihara, Jun Xu, and Hui Zhang. Data Streaming Algorithms for Estimating Entropy of Network Traffic. In *SIGMETRICS/Performance*, 2006.

[122] Richard Wang, Dana Butnariu, and Jennifer Rexford. OpenFlow-based Server Load Balancing Gone Wild. In *Hot-ICE*, 2011.

[123] Gurmeet Singh Manku, Sridhar Rajagopalan, and Bruce G. Lindsay. Random Sampling Techniques for Space Efficient Online Computation of Order Statistics of Large Datasets. In *SIGMOD*, 1999.

[124] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. Hopscotch hashing. In *Proceedings of the 22Nd International Symposium on Distributed Computing*, 2008.

[125] OVS hash map, 2017.

[126] Dong Zhou, Bin Fan, Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. Scalable, High Performance Ethernet Forwarding with CuckooSwitch. In *CoNEXT*, 2013.

[127] Writing a damn fast hash table with tiny memory footprints, 2017.

[128] Masoud Moshref, Minlan Yu, and Ramesh Govindan. Resource/Accuracy Tradeoffs in Software-Defined Measurement. In *HotSDN*, 2013.

[129] Graham Cormode and Marios Hadjieleftheriou. Finding Frequent Items in Data Streams. *VLDB Endowment*, 2008.

[130] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. SCREAM: Sketch Resource Allocation for Software-defined Measurement. In *CoNEXT*, 2015.

[131] Nick Duffield, Carsten Lund, and Mikkel Thorup. Estimating Flow Distributions from Sampled Flow Statistics. In *SIGCOMM*, 2003.

[132] Daniel Egloff and Markus Leippold. Quantile Estimation with Adaptive Imprtance Sampling. *The Annals of Statistics*, 2010.

[133] Intel haswell, 2013.

[134] 7-Zip LZMA Benchmark, 2018.

[135] CAIDA Anonymized Internet Traces 2012, 2012.

[136] Flip Korn, S Muthukrishnan, and Yihua Wu. Modeling Skew in Data Streams. In *SIGMOD*, 2006.

[137] Graham Cormode and S Muthukrishnan. Summarizing and Mining Skewed Data Streams. In *SIAM International Conference on Data Mining*, 2005.

[138] Pedro Celis, Per-Ake Larson, and J Ian Munro. Robin hood hashing. In *Foundations of Computer Science, 1985., 26th Annual Symposium on*, pages 281–288. IEEE, 1985.

[139] DPDK. `http://dpdk.org`, 2016.

[140] Martin Kong, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël Pouchet, and P. Sadayappan. When Polyhedral Transformations Meet SIMD Code Generation. PLDI, 2013.

[141] Luigi Rizzo. netmap: A novel framework for fast packet i/o. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012.

[142] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. Packetshader: A gpu-accelerated software router. In *Proceedings of the ACM SIGCOMM 2010 Conference*, 2010.

[143] Bin Fan, David G. Andersen, and Michael Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, 2013.

[144] Dong Zhou, Bin Fan, Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. Scalable, high performance ethernet forwarding with cuckooswitch. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, 2013.

[145] Theophilus Benson, Aditya Akella, and David A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *IMC*, 2010.

[146] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data Center TCP (DCTCP). *SIGCOMM computer communication review*, 2011.

[147] SMHasher, 2017.

[148] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. *ACM SIGCOMM*, 2013.

[149] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 1970.

[150] Scaling in the linux networking stack.

[151] Mihai Dobrescu, Katerina Argyraki, and Sylvia Ratnasamy. Toward Predictable Performance in Software Packet-Processing Platforms. In *NSDI*, 2012.

[152] Silas Boyd-Wickizer, M Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. Non-scalable Locks Are Dangerous. In *Linux Symposium*, 2012.

[153] David Levinthal. Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors, 2009.

[154] Intel performance counter monitor.

[155] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, 2010.

[156] Tao Jiang, Qianlong Zhang, Rui Hou, Lin Chai, Sally A Mckee, Zhen Jia, and Ninghui Sun. Understanding the behavior of in-memory computing workloads. In *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, 2014.

[157] Omid Alipourfard, Masoud Moshref, and Minlan Yu. Re-evaluating measurement algorithms in software. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, 2015.

[158] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click Modular Router. *Transaction on Computer Systems*, 2000.

[159] Benoit Claise. Cisco Systems NetFlow Services Export Version 9. RFC 3954, 2004.

[160] Nicolas Hohn and Darryl Veitch. Inverting Sampled Traffic. In *IMC*, 2003.

[161] Mihai Dobrescu, Katerina Argyraki, Gianluca Iannaccone, Maziar Manesh, and Sylvia Ratnasamy. Controlling Parallelism in a Multicore Software Router. In *PRESTO*, 2010.