Infrastructure Support for Datacenter Applications

Michael Chang

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2021-244
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-244.html

December 1, 2021

Infrastructure Support for Datacenter Applications

by

Michael Alan Chang

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Scott Shenker, Chair
Professor Sylvia Ratnasamy
Professor Thomas M. Philip

Spring 2021

Infrastructure Support for Datacenter Applications

Abstract

Infrastructure Support for Datacenter Applications

by

Michael Alan Chang

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Scott Shenker, Chair

Cloud computing enables operators to spin up resources on-demand, and orchestrators enables the automatic deployment of complex distributed applications. As deployment of applications in datacenter environments grows increasingly automated and accessible, the logical next step is to consider how the infrastructure can support the operator in critical configuration tasks.

In this dissertation, we build and evaluate a number of infrastructure-based approaches that automate a few of these critical configuration tasks. First, we present AutoTune, an orchestrator-based tool that tunes an application's resource allocation in order to improve end-to-end performance and resource efficiency. Next, we describe Privoxy, a system which automatically checks that SQL queries issued by web applications are compliant with operator-defined privacy policies. Finally, we build a trace-driven packet simulator that evaluates how an architectural change to the datacenter networking fabric impacts the training time of convolutional neural networks for image recognition.

To Diane, Jack, and Max

To all my friends and my communities

Contents

List of Figures

List of Tables

Acknowledgments

To Scott: I have always felt that Scott has become invested in whatever has been most on my mind. Whether it was a networks or systems research question, an evaluation concern, questions about startups, personal and relationship issues, group culture, or topics of race and representation – Scott always approached it with curiosity, a willingness to inquire more if he didn't understand, wisdom, and, unfailingly, a stream of guffaws, profanity, and encouragement. Everything always felt wonderfully authentic. In these five years of personal growth and exploration, I am eternally grateful to have Scott in my corner. Thank you Scott.

To Panda: Panda was my first friend in Netsys, my first grad school research collaborator, the person who introduced me to the epicurean delights of the East Bay, and someone who I hope to have as a lifelong friend and collaborator. Your sheer knowledge of everything and anything, and your ability to process multiple threads of information simultaneously, will never cease to amaze me. I can't wait to tell my grandkids about the amazing humble Panda that I got to sit in 419 with all those years ago.

To Sylvia: Sylvia has been a pillar of Netsys since I joined. I have been in group meetings with Sylvia for five years now and have always appreciated how she pushed me during my talks, causing me to critically examine implicit, taken-for-granted assumptions in my own work. I will probably never forget the semester that I worked with her on CS168 in Spring 2020. I would never wish that on another TA, but I found great fulfillment in working with Sylvia to run the best possible course, given the circumstances.

To Thomas: I met Thomas in the end of my fourth year, where he encouraged me to consider the narratives that constrain the possibilities of what I can do with my life. Regardless of where I end up, Thomas will always be the one who opened my mind to possibilities outside of technology. Moreso than ever, I feel that I am on a path that finally aligns my personal values with my day-to-day work, and for that, I owe significant gratitude to Thomas.

To my all my labmates (and honorary labmates): Aisha Mushtaq, Ethan Jackson, Justine Sherry, Murphy McCauley, Radhika Mittal, Peter Xiang Gao, Chang Lan, Amin Tootoonchian, Sangjin Han, Kay Ousterhout, Silvery Fu, Wen Zhang, Emmanuel Amaro, Lloyd Brown, Vivian Fang, Eric Sheng, Anwar Hithnawi, Amy Ousterhout, Zhihong Liu, Chris Branner-Augmon, Shivaram Venkataraman, Ed Oakes, Wenting Zheng, Narek Galstyan, Sarah Mcclure: My greatest regret in graduate school is that I spent the last 1.5 years away from my labmates. The time that we did have together – eating, debating, shaking hands, hugging, drinking – will stay with me forever. I truly felt like this was a space where we absolutely and unequivocally wanted the best for each other, even – at times – at expense of our own well-being. I will never forget you all.

To my EECS friends: Stephanie Wang, Rohan Padhye, Michael Dennis, Stephanie Wang, Tyler Westenbroek, Sagar Karandikar: In my darkest and brightest days of graduate school, you all were my go-to people. We ate, we drank, we played board games, and laughed at Rohan's encyclopedic knowledge of random things, and his supernatural ability to guess the time. I will miss seeing each of you wave adorably at me through the 419 glass window as you make your way to an event at the Woz.

To my Undergraduate Collaborators: Will Wang, Eric Sheng, Anson Tsai, Rahul Balakrishnan, Lisa Jian, Dom Bottini, Pranay Kumar, Emerson Hsieh, Michelle Hwang: Where would I be without my undergraduate collaborators? Thank you all for taking a chance on me when I was a mere first year grad student, committing time and energy amidst your incredibly busy schedules. I will miss our URANAS meetings!

To my undergraduate advisors: Jennifer Rexford, Laurent Vanbever, Theo Benson: Thank you for taking a chance on me, an undergraduate with nothing except the most basic introductory CS courses under my belt. In the cold dreary days of New Jersey, I always felt supported as I stressed about my research projects.

To Netsys Staff: Boban Zarkovich, Kattt Atchley, Jon Kuroda, Dave Schonenberg, Carlyn Chinen, Jae Tabuada: Thank you all for being on top of everything for us at Netsys over the years. I imagine that at time we may appear grouchy, ungrateful, and frustrated to you all but I will be eternally grateful for the way you all supported me through my degree. I would be extremely fortunate to meet people with your organizational abilities and competence down the road.

To EECS Administrative Staff: Jean Nguyen, Angela Waxman, Shirley Salanio: I truly believe that I have filed more petitions to the department than any other CS graduate student. For that, I owe the greatest thanks to all of you.

To my Hillegass Parker housemates: I didn't know what it truly meant to live in a community of people until I joined the house. Thank you for nourishing me your kindness, sharing your experiences and life stories – the good and the bad – and ripping open assumptions I carried about how I should live my life. My Hillegass Parker Housemates have truly opened my eyes to nuance about life experiences, and to never think about things in terms of black and white, good and bad. I think this is a life lesson, and I hope it will stay with me forever more.

To Adrianne Zhong: You are my closest friend and dearest confidant. Our brutal honesty towards each other will keep both of us on the right path.

To Holly Flores: My beautiful decolonizer and community builder. You have been pillar in my graduate school career, and together, we shall grow from these hearty roots.

To my Family: Jack, Diane, Max: I wouldn't be here without all of you. Your support and love give me strength and determination to pursue my dreams. I appreciate how much all of you have grown and changed over the years. Your encouragement for me to follow my heart and do what I want to with my life, this is the greatest gift that anyone can afford me. Thank you for all you do for me: this confused succulent.

Chapter 1

Introduction

The barrier to entry of deploying applications in datacenter environments has been greatly reduced over the last decade. With the advent of cloud computing, application developers could access compute and storage on-demand, thus relieving the burdensome and expensive process of buying physical servers and manually setting up local clusters. When the cloud providers could no longer compete based on the cost of commodity servers, they began to develop vast ecosystems of cloud computing services, including data stores, event queues, monitoring, cost tracking, and countless more. Ubiquitous application services – that was previously re-implemented by each organization that needed it – became easily accessible as long as applications were already deployed on the cloud. Cloud computing enabled developers to focus on developing application logic, and provided many of the other requisite *pieces* to deploy applications in a datacenter environment.

Even as cloud computing resources became cheaper and the service ecosystem grew, the applications themselves continued to evolve. From web applications to data analytics pipelines, applications deployed in datacenters become increasingly complex, heterogenous, and decentralized [109, 141, 63, 122]. As these applications grew more complex, so too did it's deployment. The emergence of these heterogeneous applications directly led to the development of container orchestrators like Kubernetes [120]. In addition to deploying the resources necessary to run these applications, container orchestrators critically included a management layer that automated certain intricacies of deploying heterogeneous applications. These included previously vexing issues like networking, service discovery, and load balancing. Industry has relentlessly pursued systems that automate the deployment of even the most complex applications.

To get to a point where one can truly run applications with ease, they must look past the solved problem of automating *deployment* and now must turn their attention towards automating the *configuration* of these applications so they meet major operational requirements. In this dissertation, I seek to develop infrastructure-level systems and tools that automate the configuration of critical operational issues for datacenter applications.

1.1 Motivating Trends

For datacenter applications, what are some of the design trends – both from industry and academia – that inform the difficulty of configuring datacenter applications? Below, we discuss some of the trends, and emergent questions that motivate our research.

Shift from Monolithic Applications to Microservice Applications

Datacenter applications were previously architected as a monolith, comprised of a front-end, a single binary as the server backend, and a database [146]. This tightly coupled approach to software design slowed a company's ability to efficiently develop and ship new features, a priority for many competitive organizations. Consequently, many companies have shifted to a *microservice* paradigm, where the single software unit is deconstructed into 10s-100s of software components ("microservices") connected over a virtual network [141, 109]. Agreed-upon APIs dictate how the microservices interact with each other. This enables teams within a company to independently push and ship code for each individual microservice, so long as they comply with the agreed-upon API. The microservice paradigm enabled companies to develop and ship new features much more quickly.

However, the decentralized pattern of development came at the expense of having a centralized understanding of overall application behavior. Our conversations with industry practitioners indicated that many organizations did not even retain organization-wide diagrams showing connections between various microservices, let alone *how* these microservice interact. Fundamentally, any configuration task that requires the operator to reason over the complex interactions *between services* introduces significant challenges.

Operators must contend with this disorganization, especially when they configure operational goals that span multiple services. Nowhere is this more true than in performance tuning, which is a critical task dictated by contractual end-to-end Service Level Agreements (SLA) [102]. *Given that organizations possess virtually no understanding of the inter-microservice interactions that significantly impact end-to-end performance, how could a tool help an operator improve performance and resource efficiency?* This is the question that motivates the design of the first chapter of this dissertation, AutoTune.

Top-Down Policy Compliance

Recent data privacy acts (DPAs) like GDPR and CCPA limit who can access data. Not only do they impose general restrictions on data access, they impose financial penalties on scenarios where companies fail to comply with their own internal privacy policies [130]. These add to an already complex landscape of data permissions, where certain classes of data were already protected under acts like HIPPA and FERPA. All these policies enact severe financial penalties for misuses of a user's personal data [130]. This complex set of privacy policies are typically interpreted or developed by a privacy team, and frequently change based on customer feedback.

Ensuring that code complies with these changing privacy policies presents significant challenges. For operators, the process of enacting compliant policies in code is both time-consuming and error-prone. Our conversations with a chief privacy officer (CPO) at a large

startup indicated that even small changes to privacy policies (e.g., changing permissions a email addresses collected from users) took several weeks to actualize; the CPO needed to meet with every engineering team that touched the data attribute in their code. Moreover, compliance with these policies is error prone, and has resulted in production bugs in many examples [135, 82, 81]. To ensure that user data is appropriately protected, automation is necessary to assist the operator in ensuring policy compliance.

How can we automatically check that application-issued SQL queries are compliant with a set of operator-defined privacy policies? This is the central question of the second chapter of this dissertation, and the impetus for our creation of the Privoxy system.

Hardware Advances in Networking

Operators must also make decisions about how their applications benefit from changes to the datacenter architecture, particularly as new hardware emerges. One such emergent hardware is the programmable switch, which has triggered a large number of novel proposals. While these programmable switches initially were introduced to test and deploy new protocols [19], recent proposals have suggested that these programmable switches could be used for more far-reaching tasks such as in-network computation. These applications include distributed DNN training, graph analytics, encryption, among others [125, 126, 19, 115].

Integrating programmable switches into a datacenter is a major architectural decision and task for any operator. Since the network fabric is typically abstracted away from the application, incorporating computation in the network fabric represents a significant deviation from the norm. Given the high effort and monetary cost required to deploy programmable switches for the purpose, operators must closely examine whether it is truly necessary to even undertake this task. In particular with the use of programmable switches for in-network computation, operators would be wise to consider all existing options that do not require ripping out all the existing network fabric.

How can operators estimate the operational improvements that come with significant architectural changes to the network fabric? The third chapter of this dissertation explores this question specifically in the context of training convolutional neural networks for image recognition.

1.2 Contributions

This thesis investigates how datacenter infrastructure can support operators in making complex configuration decisions for their distributed applications. The contributions of this thesis include the following.

- The design and implementation of AutoTune, a tool that improves end-to-end performance and resource efficiency for heterogeneous microservice applications. AutoTune treats the entire application as a black box, and requires no code changes. The operator only needs to provide a representative workload and a performance metric they want to optimize. We demonstrate over a series of microservice applications that Auto-

Tune can reduce the number of servers by up to 6.6x, while simultaneously improving performance by 20%.

- The design and implementation of Privoxy, a system that enforces data access policies that can be used with legacy applications and preserve application semantics. Our approach verifies compliance in a proxy that interposes on the connection between the web application and the database. Our verifier leverages SMT solvers and, to achieve adequate performance, generalizes and caches the results of previous compliance decisions. We have implemented this approach in a system called Privoxy, and our evaluations show that Privoxy supports legacy web applications while adding only modest overheads.

- A preliminary study on whether in-network changes are the most effective way to improve the performance of the distrbuted training of convolutional neural networks for image recognition. We developed a trace-drive packet simulator, which we used to demonstrate that changes to the fabric (i.e., in-network computation) had lower benefits than using host-based mechanisms that abandon the use of the parameter server in favor of other reduce strategies .

1.3 Dissertation Plan

The thesis proceeds as follows. Chapter 2 demonstrates how we use AutoTune to determine resource allocations and placements that improve performance and resource efficiency. Chapter 3 describes Privoxy, a system that automatically ensures that web applications issue SQL queries compliant with user-specified privacy policies. Chapter 4 discusses whether architectural changes to the networking fabric are needed to accelerate distributed training of convolutional neural networks. Finally, Chapter 5 concludes and discusses some opportunity for future work.

Chapter 2

AutoTune

2.1 Introduction

Motivation

Most modern web-scale applications – including those offered by Google [23], Uber [141], and Netflix [109] – are comprised of multiple (and sometimes many) microservices, such as web servers, caches, load balancers, and data management systems. Operators rely on *microservice orchestrators* such as Kubernetes [120] to connect and manage these microservices. Operators provide the orchestrator with an application specification – listing the required set of microservices, their resource requirements, and any placement constraints – and a cluster specification that describes the set of servers on which the application can be launched. Microservice orchestrators provide additional APIs that provide fine-grained control over an application's resource allocation and placement (which controls which components are colocated).

In theory, operators can use these APIs (along with application APIs to change various configuration parameters) to achieve good application performance and resource efficiency. However, this is not an easy task, because the end-to-end performance of microservice-based applications is an unknown and complex function of the resources (such as memory and cores) allocated to each microservice and, to a lesser degree, the microservice placement. There has been little progress in deriving analytical expressions for real-world application performance as a function of resources and placement. Faced with this unresolved complexity, many operators merely overprovision with resources [42, 23, 45, 79] and use simple affinity rules for placement [136, 117, 120]. As we will show, this results in good performance but an inefficient use of resources. To rectify this, we have designed a tool called AutoTune for *automatically* reducing resource consumption while preserving adequate performance.

Considerations

AutoTune's design is strongly influenced by three basic considerations. First, we want a solution that can be used by non-expert operators, and is applicable across a wide variety of systems and use-cases. We therefore treat microservices as blackboxes, rather than assuming that detailed models of their performance are available. Similarly, we directly measure application performance rather than attempt to derive it from performance measurements

on individual microservices. This allows the operator to select the performance objective most relevant to them (makespan, tail latency, etc.).

Second, we assume that operators can produce (based on previous experience running the application) an initial deployment (i.e., a specification of the placement and the resource allocations for each microservice) of their application that achieves acceptable performance. We take the performance of this initial deployment as the baseline performance expectation. Our goal is to find more efficient deployments that use fewer resources, yet achieves similar or better performance.

Third, in order to not interfere with the production environment, we intend AutoTune to be run on a testbed where various configurations can be evaluated; such testbed experiments can easily be carried out in public clouds. In order for these testbed runs to be useful, we assume that the operator has access to a representative workload (or set of workloads) where achieving good performance and resource efficiency on these test workloads results in similar benefits in production use. This is clearly true for recurring batch jobs, which have similar workloads across invocations. We later discuss how AutoTune can apply to more varying workloads, which we then evaluate using a production trace.

Approach

Our goal in designing AutoTune is not to achieve *provably optimal* performance or *provably optimal* efficiency, but to provide a widely-applicable and easy-to-use tool that can provide *as good or better* performance than the operator-provided baseline while typically reducing the number of servers required for deployment. Our approach to finding efficient deployments is comprised of two separate phases.

In phase 1, which we call *resource clampdown*, we try to identify overprovisioned resources where we can reduce the resource dedicated to an individual microservice without impacting overall application performance. Once we have eliminated all overprovisioning, we then try to place microservices on the minimal number of servers while respecting these resource allocations. We then verify that the resulting deployment matches the performance of the initial deployment, and make adjustments if necessary to make this hold.

In phase 2, which we call *performance improvement*, we take the deployment from phase 1 and test whether performance can be improved by assigning leftover resources and shifting resources on a server between microservices. The result is a locally optimal allocation of resources that equals or exceeds the original performance goal.

We demonstrate the efficacy of our approach by using AutoTune on three representative microservice applications exhibiting a variety of microservice design patterns with widely different microservices (both third-party and custom). We demonstrate that, for these use-cases, Privoxy can reduce the number of servers by up to 6.6×, while simultaneously improving performance by up to 20%.

Privoxy's resource allocation and placement are computed for a specific workload, but in practice workloads change over time. In §2.5 we demonstrate how Privoxy can be used to overprovision resource so it can handle workload changes. We find that overprovisioning resources by 30% is sufficient to handle a workload that has 1.6×. Additionally, we analyzed

a a four-week long production trace from Datadog [40] and found that in that trace a workload increase of this magnitude occurs approximately every 51 hours. Thus, our analysis demonstrates that slight overprovisioning, and infrequently running Privoxy is sufficient for handling dynamic workloads.

2.2 Background

We begin by describing current approaches to automatically scaling microservice applications – autoscaling and Bayesian optimization – and observe that they will not find efficient deployments for many microservice applications. This lack of applicability motivates the need for Privoxy. However, note that these approaches solve a subtly different problem than AutoTune in that they typically start with some configuration which is underperforming and then add resources until the performance is adequate; AutoTune starts with a configuration where the performance is adequate and then reduces (and rearranges) resources while maintaining (or improving upon) acceptable performance. The two approaches are complementary, and in AutoTune when there is a sudden load spike we can fall back on autoscaling to cope.

Autoscaling

Some orchestrators (including Kubernetes) and most cloud providers (including Amazon [7], Azure [100], and Google [56]) provide autoscaling services that observe each microservice's performance or utilization, and use this information to decide when to add additional instances of that microservice. These services accept as input a pre-defined threshold (*e.g.,* an acceptable response latency or utilization level) and increase the number of instances for a microservice whenever its performance drops below the threshold or its utilization exceeds the threshold.

Regardless of what metric is used to trigger scaling, existing horizontal and vertical autoscaling solutions focus on *local optimizations*: they consider the performance or utilization of an individual microservice, without regard to overall application performance. Additionally, when triggered they scale only the *triggering microservice*, without regard to how this might impact overall performance. This focus on local optimization has two shortcomings.

First, using performance metrics requires the applications operator to decompose an application level performance requirement into performance targets for each individual microservice. As has previously been observed [85, 76], the amount of time processing a request at different services depends not just on the application, but on various factors such as the request itself (which might dictate whether processing is done on the fast path or slow path) and the history of previous requests (which dictate whether a microservice might experience pauses due to garbage collection or other effects). Finding an appropriate decomposition for complicated real-world applications is thus quite challenging.

Second, the use of local optimization (whether performance or utilization driven) can result in suboptimal resource allocations, i.e., in situations where an application is allocated

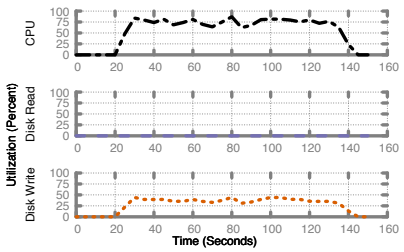


Figure 2.1: Observed utilization for frontend microservice.

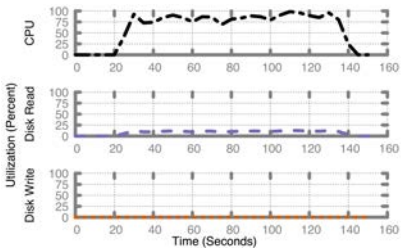


Figure 2.2: Observed utilization for storage microservice

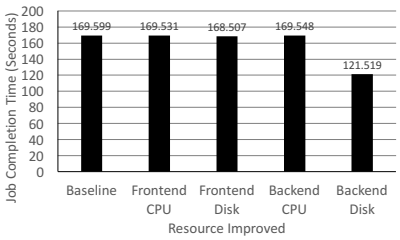


Figure 2.3: Actual Job Completion Time after adding resources to the multiservice microbenchmark.

more resources than required to meet a particular performance goal. We now illustrate this for utilization-triggered autoscaling.

Consider an application comprised of two microservices: a frontend microservice and a storage microservice. The frontend microservice receives client requests and performs a blocking remote read from the storage microservice, which on receiving a read request, reads and processes the request file before returning the processed results. We collected resource utilization and job-completion times for each microservice during a baseline run, which we show in Figure 2.1 and Figure 2.2. Utilization seems to indicate that bottleneck resource is either (a) the CPU on the frontend microservice; (b) the CPU on the storage microservice; or (c) the *disk* on the frontend microservice (used for logging). We then attempt to empirically validate these findings by increasing each of these resources and measuring their impact on the application's performance (Figure 2.3). We find, contrary to what the utilization measurements suggest, that none of them has any impact on application performance. In fact, the highest performance improvement comes from increasing *disk bandwidth* on the storage microservice, a resource that was not highly utilized! This is because the frontend is blocked while waiting for the storage service to read data, and utilization is not sufficient to capture this dependency.

While a microservice's performance (*e.g.,* a microservice's response latency) is a more relevant measure than its utilization, autoscaling based on such per-microservice performance metrics suffers from the same flaw of not capturing dependencies, and can thus result in inefficient provisioning.

Bayesian Optimization

In recent work [6], Bayesian optimization techniques were used to improve the performance of an application consisting of a single service (Spark executors) with a single microservice per server. Optimizing this deployment merely requires determining resource allocations for that single microservice; this results in a relatively low-dimensional search space on which Bayesian optimization performs well. We seek a tool that can apply to applications with a heterogeneous set of microservices which interact with each other in a variety of

ways. Such a setting results in a high-dimensional search space, making it impractical to apply Bayesian optimization. Furthermore, when dealing with a heterogeneous set of microservices, the mechanisms controlling scaling and placement decisions must consider resource and operational constraints, and these are difficult to incorporate in such Bayesian optimization algorithms. In contrast, AutoTune uses a less sophisticated search algorithm than Bayesian optimization, but can handle a high-dimensional search space and complicated operational constraints.

2.3 The Design of AutoTune

Having reviewed existing approaches to optimizing application performance and resource efficiency, we now describe Privoxy's design. We begin by defining terminology used through the rest of the paper, then describe the types of deployments we target, and finally present Privoxy's design.

Terminology

Privoxy's inputs are the application A and performance metric P to be optimized, an initial deployment D_0, and a representative workload W. The initial deployment consists of a resource allocation R_0 that specifies what resources are allocated to each microservice in A, and a set of placement decisions that specifies what microservices are colocated with each other. Given these inputs, Privoxy uses W to compute an initial performance P_0, and Privoxy's optimization process ensures that any deployments perform at least as well or better than P_0.

Privoxy considers four resources: memory, CPU cores (both full cores and fractional assignments), disk bandwidth, and network bandwidth. We use the term microservice resource (MR) to refer to the allocation of one of these resources to a particular microservice on a particular server. AutoTune adjusts microservice resource allocations and microservice placements in order to minimize the number of servers required and then produces a new deployment at the end of this process.

Target Deployments

We envision that Privoxy is deployed in a test cluster, and the resulting deployment is then used in production. Privoxy's computation relies on changing resource allocations, and observing the performance impact of such changes. Running this in a test cluster ensures that Privoxy does not negatively impact performance in production. Additionally, Privoxy is designed to be used in clusters where resources are allocated in the form of servers or VMs (we refer to both as servers below), and the orchestrator can place zero or more microservices in each VM. As a result, we measure resource efficiency in terms of servers, and Privoxy's optimization objective aims to minimize the number of servers, rather than merely minimizing total allocated resources.

Phase 1: Resource Clampdown

Privoxy's first phase focuses on identifying over-provisioned microservice resources in the initial deployment, and then eliminating over-provisioning.

Privoxy identifies over-provisioned resource by *stressing* individual MRs; i.e., reducing the amount of the resource allocated to the microservice and then measuring the end-to-end application performance. Privoxy uses the results from stressing to classify each MR as either an *impacted microservice resource* (IMR) – a MR whose reduction hurts application performance – or a *non-impacted microservice resources* (NIMR) – a MR whose reduction has no negative impact on application performance. Whenever a NIMR is identified, Privoxy reduces the resource allocation by the same amount used in the stressing step. Privoxy repeats this process until no more NIMRs are identified. The resulting resource allocations are *tight*, in that any resource reduction will result in performance degradation.

While the previous step yields a tight resource allocation, it does not necessarily free up any servers, since it does not change placement. The next part of phase 1 thus produces a new placement that packs all of the microservices into fewer servers. While doing so, Privoxy attempts to avoid colocating any pair of microservices that might affect each other's performance. In order to do so we rely on the ability to order the impact of different microservice resources. We define a microsevice resource r as being more impacted than MR r' if reducing r's allocation has a greater impact on application performance than r'. Given this ordering, we can also identify the set of most impacted MR's.

Observe that the stressing results above already provide us with sufficient information to identify the most impacted resource for each microservice. Given this information, and the number of available servers, the Privoxy placement algorithm begins by placing microservices so that any two colocated microservices differ in their most impacted MR. When this placement strategy is no longer feasible, Privoxy switches to a round-robin placement strategy. The resulting placement is guaranteed to fit within only as many servers as are needed for the tight resource allocation identified previously; however, this change in placement might have resulted in worse performance.

To protect against this, Privoxy checks application performance with the computed placement. If the new performance is worse than the initial performance P_0, then Privoxy tries other variants (by changing the order in which microservices are placed) to identify a placement that performs as well or better than P_0. If no such placement can be found, Privoxy increases the number of servers and reruns the placement strategy. This process necessarily terminates, since the initial placement had performance P_0.

At the end of phase 1, Privoxy produces a new deployment where no MR is overprovisioned, and that potentially uses fewer servers without negatively impacting application performance.

Phase 2: Performance Improvement

The deployment produced in phase 1 might not utilize all server resources. In the second phase Privoxy allocates these resources so as to improve application performance. This stage

begins with Privoxy assigning unallocated resources on each server to the microservice most likely to benefit from additional resources. To do so, for each type of resource available on a server, Privoxy identifies the microservice on that server that is most impacted by that resource type, and allocates the microservice all available resources of that type. Privoxy reuses measurements from phase 1 to determine impacted microservices on a server.

While the previous step allocates all available resources, in many cases performance can be further improved by changing resource allocations. This is because in cases where two microservices α and β on the same server are impacted by the same resource, then it is possible that moving impacted resources from one service (*e.g.,* β) to the other (*e.g.,* α) results in a net improvement in application performance. We identify opportunities for transferring resources using a gradient descent algorithm which iteratively matches highly sensitive MRs with less sensitive MRs of the same resource type that are colocated on the same machine.

2.4 Additional Issues

Next we discuss some of the additional challenges addressed by our design.

Reducing Search Space

Because we have four resources, stressing microservice resources one at a time would require that Privoxy evaluate performance for $4M$ different deployments (where M is the number of microservices). Privoxy reduces the number of deployments that have to be tested by randomly partitioning all MRs into p partitions. Privoxy then measures performance after reducing allocation for *all MRs* in a partition. This measured performance is used differently by the two phases:

Phase 1: Partitions in Resource Clampdown If application performance does not degrade when a partition is stressed, Privoxy infers that none of the MRs in the partition are impacted. If on the other hand a performance degrades, Privoxy splits the partition itself into p random partitions, and uses the same process to select those subpartitions which degrade performance. This process continues recursively until all impacted MRs are found.

Phase 2: Partitions in Performance Improvement When using gradient descent, AutoTune first identifies the most-impacted partition. Within that partition, AutoTune applies the original gradient descent method, stressing each MR individually. Thus, in each iteration of the the gradient descent process, AutoTune with the pruning method with p partitions only needs to explore $\frac{N \cdot k}{p} + p$ MRs per iteration, rather than of having to explore $N \cdot k$ MRs.

Observe that for partitioning to be effective in either phase we need most partitions to not impact application performance. In our tests we have found that empirically this is indeed the case, and the set of impacted MRs is relatively small. In our experiments we found that the use of partitioning reduced Privoxy's runtime by $2-3\times$ on average. However, even when there are many impacted MRs the use of partitioning has no impact on correctness, nor does it significantly impact Privoxy's efficiency.

Minimizing Placement Changes

Changing the placement of a microservice takes longer than merely changing its resource allocation. This is because changes to placement necessitate launching a new copy of the microservice, while resource changes can often be done without needing to terminate an existing copy. This observation influences our design in two ways: first, we minimize the number of times placement is changed (only at the end of phase 1 and the beginning of phase 2); second, we stress (i.e., reduce allocated resources) rather than increase resources to measure a MR's impact on application performance. Stressing a resource is guaranteed not require placement changes, while increasing resource allocation might in cases where a microservice is running on a fully utilized server.

Discrete Gradient Descent

Analytically, gradient descent uses derivatives in order to choose the direction for change. Here, we must implement a discrete derivative (when stressing the application); in order to evaluate the derivative, we compare the application performance with the original MR with one that differs by an amount δ, and the question is how do we choose δ. AutoTune sets δ proportional to the server's resource capacity. However, rather than use a fixed percentage, we start at 30% and then, at each iteration of the algorithm, we decrease this percentage by a factor of 1.2. This allows us to take finer and finer tests of discrete changes. We also implement a binary search feature when evaluating resource transfers. When we increase one MR by δ and decrease another by the same amount and find no change in application performance, we then execute a binary search between the original and tested allocations to see if we overstepped a local optima.

Placement

Our experience suggests that placement is typically not a crucial factor in application performance, aside from the special case of affinity. However, because containers (and indeed most other software isolation mechanisms) do not completely isolate performance, interference between colocated microservices can negatively impact application performance. Discovering this would require exhaustively searching through all placement possibilities, which would be too time-consuming. Instead, we address this issue by preferring, when when searching for optimized deployments, those that colocate microservices that were colocated in the initial deployment and enforcing any administrator provided placement constraints. This allows AutoTune to incorporate any insights the operator might have about possible performance interference.

2.5 Dynamic Workloads

Thus far our design has focused on applications with relatively predictable load patterns where we could assume that the operator supplied a representative workload which could be used to evaluate performance. Since almost all workloads have some natural variation, we further assume that this supplied workload represents the high-end of anticipated workloads (and the degree of such overestimation is up to the operator based on their tradeoff

between performance and efficiency). We also assumed that the operator provided an initial deployment on which this workload has acceptable performance. These are reasonable assumptions for batch workloads and relatively stable real-time workloads. However, we cannot make these assumptions for real-time workloads with significant variation (which we assume is the common case for real-time workloads).

For these real-time workloads, the common practice is to significantly overprovision (we have heard some operators provision their deployment to handle up to three times the average workload), and then rely on autoscaling (as described in Section 2.2, this independently scales the microservices based on local measurements) to handle any further overloading. We propose an alternate approach, based on AutoTune, to increase efficiency. In this case, the operator provides several different workloads (based on historical usage patterns) and their corresponding deployments which achieve satisfactory performance. AutoTune is run on each of these workload/deployment pairs, producing more efficient deployments which achieve the same or better performance than the given deployment for each workload. In production, the operator uses an automated mechanism that measures the current load and picks the most appropriate deployment for the current load.

For this approach to be feasible, we must ensure that the deployment changes are relatively rare, which requires a degree of overprovisioning and using some amount of hysteresis so that the deployment is not changed based on short-term fluctuations. To investigate how this might work in practice, we study a production Datadog trace that contains the rate of API calls in 10 second intervals over a four week time period.

We apply this trace to a web application developed using the MEAN Stack (described in §2.7), and implement the following scheme. We pick some degree of overprovisioning (i.e., pick a deployment designed for a load level higher than the average by some amount) and stick with the degree of overprovisioning until we observe five minutes of load that are significantly higher or lower than the previously observed average. The five minute period is unlikely to cause major performance disruptions because, as we show in §2.8, AutoTune-generated deployments hold performance even when the workload is 10-20% greater than experimental workload, and then performance degrades gracefully that.

When the aberrant load is sustained for more than five minutes, the system switches to a deployment that can handle the new load. This new deployment is similarly overprovisioned, so it can handle increased load in the future. Since we assume that Privoxy only produces a finite number of deployment configurations, we pick the deployment generated using the workload with the lowest average request rate that is still higher (by the same overprovisioning factor) than the production workload's recently observed average request rate.

Table 2.1 shows how often the deployment would need to change, for different levels of over-estimating workload. If the operator overprovisions by 10%, the system would have to transition to a new deployment once every 45 minutes. At the other extreme, if the operator overprovisions by a factor of three, the system would only have to transition to a new deployment once every week.

Where in between these two extremes an operator would choose to operate would depend

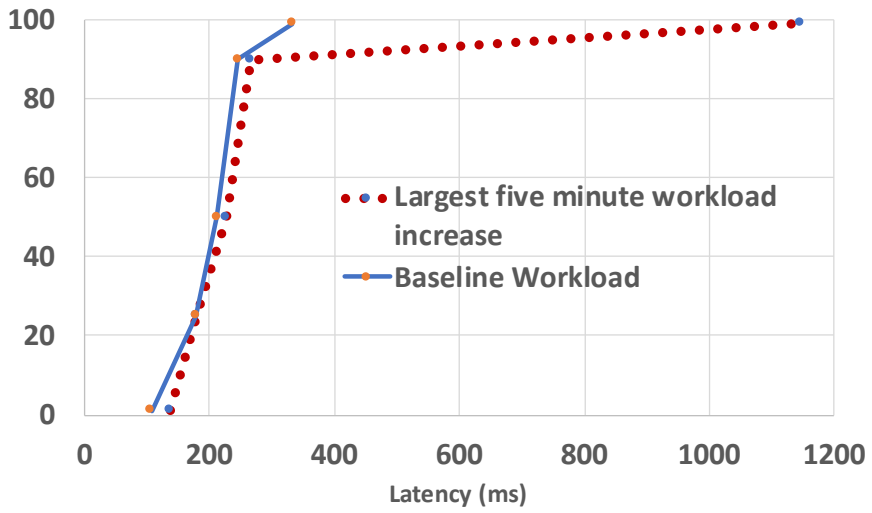


Figure 2.4: CDF of response latency for a 30% overprovisioned deployment with the expected request rate (baseline) and a request rate that is 1.6× higher (Largest five minute increase).

in part on how expensive (in terms of resources) it is to overprovision for higher loads. These results are also shown in Table 2.1. Note that overprovisioning for a factor of three costs roughly 54.5% extra in resources, and doing so for a factor of 2 costs roughly 34% (and requires deployment changes roughly once every three days).

Finally, we measure the impact of response latency to changes in workload in an overprovisioned setting, since too large an increase in response latency would pose an impediment to the use of Privoxy in dynamic settings. For this evaluation we overprovisioned the cluster with 30% additional resources, and then considered a scenario where request rates instantaneously increase to 1.6× the original request rate. The 1.6× increase represents the largest consistent increase (i.e., sustained for 5 minutes) we observed in the Datadog trace. The request rates and distributions were taken from the trace. We show a CDF of response latencies for the original request rate (baseline) and the increased request rate in Figure 2.4. We observe that median and 90^{th} percentile response latencies remain unchanged. While we do observe a nearly 6× increase in maximum latency, this is likely due to the additional request queuing. We can thus conclude that most requests see reasonable response latencies, and hypothesize that additional overprovisioning might be necessary in scenarios where low tail latencies are crucial.

Thus, assuming a production deployment similar to the one captured by the Datadog trace, one can use AutoTune for dynamic workloads without significant overprovisioning and infrequent deployment changes.

Workload Increase	Percent Overprovisioned	Time between change
1.2×	0.01	1.49 hours
1.4×	27.5	1.95 hours
1.6×	29.86	51.3 hours
1.8×	30.2	59.62 hours
2×	34.0	77.88 hours
3×	54.5	166.1 hours

Table 2.1: The impact of changes in workload: by how much do resources need to be overprovisioned to handle workload increases; and, on average, how much time (measured in hours) elapses in our trace before overprovisioned resources are no longer adequate.

2.6 Implementation

Next we describe Privoxy's prototype implementation. Privoxy is designed so it can be integrated with any container orchestrator. Our prototype implementation integrates with two orchestrators: Kuberentes and Kelda [75]. Below we detail some of the interesting aspects of our prototype implementation.

Workload Generation

Similar to prior approaches for resource allocation [6], Privoxy requires that the operator provide a representative workload, which it then uses for optimization. In our evaluation (§2.8) we use Apache Bench [1] as a workload generator, and this approach might be usable by many applications. In addition, operators can also use tools such as Kraken [144], GoReplay [57], etc. to record and replay production traffic when using Privoxy.

Additionally, many organization rely on canary deployments for testing and profiling. Live production traffic is often mirrored to these canary deployments, thus allowing profiling and debugging on live traffic. AutoTune can be used in canary environments. While this might increase the time for convergence due to traffic changes between experiments, this does not impact correctness. Moreover, recent work [97] has shown that one can use transfer learning to extend performance models learned in more constrained environments (*e.g.,* in Canary deployments) to more general environment (*e.g.,* production clusters). In future work we plan to investigate the use of similar techniques in order to improve Privoxy's robustness.

Reconfiguring Applications

The configuration for some microservies, *e.g.,* Nginx or Spark, depend on the number of resources allocated to them, and thus must be changed by Privoxy. For such microservices, we require that operators provide scripts that Privoxy can invoke to update microservice configuration whenever resource allocations are changed. Programmatic APIs to change configuration are increasingly common, andother tools including Kubernetes Helm [72] rely on similar mechanisms.

Stressing

As we described previously in §3.2, Privoxy stresses microservice resources in order to identify impacted MRs. Our current implementation considers four resources: CPU core allocation, CPU quota per core, network bandwidth, disk bandwidth, and memory allocation. AutoTune can be easily extended to consider other resources. Below we briefly describe how we stress each of these resources:

CPU Quotas: Privoxy relies on `cgroups` to control CPU allocations, and allocates microservices both timeslices on a single core and whole cores. Privoxy relies on Linux's completely fair scheduler (CFS) to allocate timeslices on a core by specifiying a microservice's container period and quota. The current implementation allocates a quota relative to a fixed period (chosen at the start of the experiment) such that the maximal stressing never exceeds the minimal scheduling time quantum in the underlying operating system.

CPU Cores: Privoxy allocates CPU cores to a microservice by increasing its CPU quota sufficiently so that no other microservice can be colocated. More precisely, given a microservice with *quota* aggregate CPU quota and c cores, throttling down by one core would mean provisioning a quota of $\frac{quota}{c} \cdot (c-1)$.

Memory stressing: AutoTune uses `cgroup`'s memory quotas to limit the amount of physical memory and swap space allocated to a microservice. In order to prevent out-of-memory errors, Privoxy monitors each microservice's memory utilization and ensures that the application has sufficient memory and swap space allocated.

Disk Bandwidth: AutoTune stresses the disk using `blkio` [118], and sets hard limits on read and write bandwidth from block devices. AutoTune uses both joint and individual throttling of read and write bandwidth.

Network Bandwidth: AutoTune stresses the network by limiting link bandwidth Linux `tc`[137]. To do this we first measure the maximum attainable inter-VM network bandwidth, and then impose $k\%$ stress by limiting the container's bandwidth to $(100-k)\%$ of this maximum. `tc` uses hierarchical token bucket (HTB) to implement this rate limit, and scheduling network traffic using HTB imposes some CPU overhead. In our experience this additional overhead did not noticeably affect our results.

2.7 Microservice Application Overview

For our evaluation we use microservice applications that resemble those running in production, and used by enterprises and moderate sized deployments, rather than services such as S3 and Google. We first describe some common microservice design patterns that we observed, and then we present three end-to-end microservice applications that we used to evaluate Privoxy.

We surveyed a number of industry practitioners to determine (i) which open source microservices were commonly used together, e.g., ELK (Elasticsearch, Logstash, Kibana) stack, Spark/MySQL, etc. and (ii) how these services are architected, e.g., chain or aggregation patterns [63] [122]. In response, we selected three applications: two (slightly modified) open source microservice applications and one of our own.

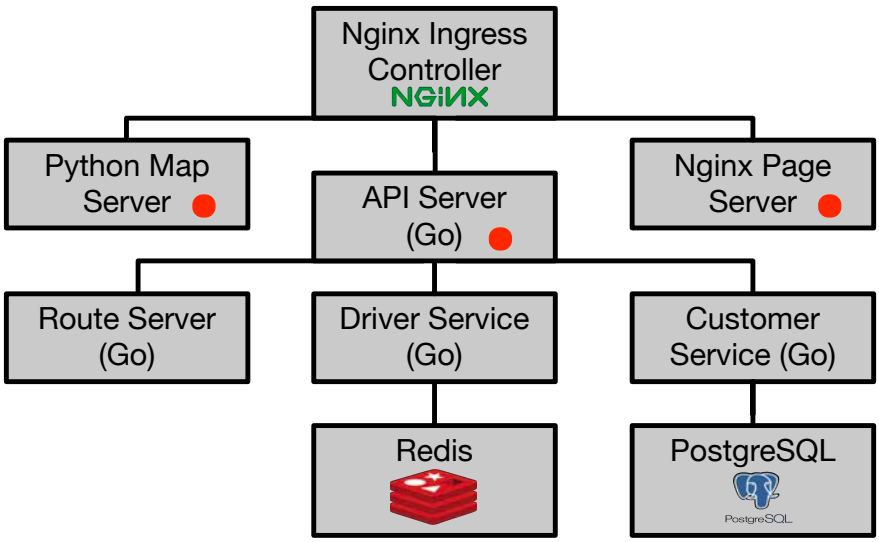


Figure 2.5: *HotROD* Application, modified from Uber. Red circles indicate endpoints.

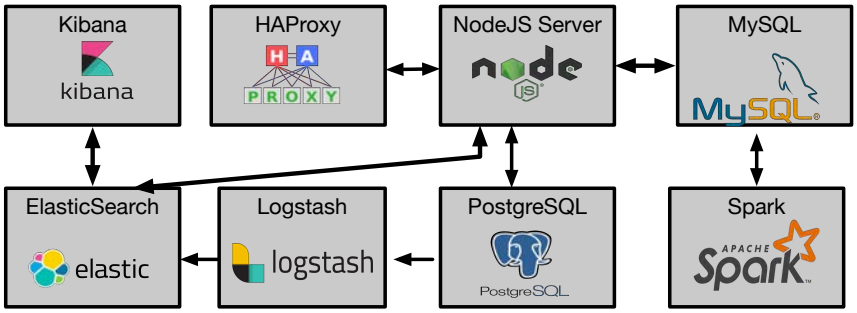


Figure 2.6: Apartment Rental Multi-service Application

HotROD Application [132] (30 MR) (Figure 2.5) is a modified distributed tracing application created by Uber to demonstrate its open source tool Jaegertracing [132]. We extended HotROD by adding two additional features: First, we split the frontend to be a separate API service and Nginx service to serve static pages. We also added a mapping service to pull graph data from S3 which is plotted using Networkx [66] and returns a JSON file which is rendered on the webpage. We also add an Nginx ingress and Haproxy load balancers to distribute load across all replicas of the mapping and API service. These extensions increase the number of MRs Privoxy need consider when applied to HotROD.

MEAN Stack [110] (12 MR) is an application commonly used in tutorials introducing programmers to the MEAN stack, a popular server-side Javascript stack. The application uses three different types of microservices: MongoDB (database service), NodeJs (web server), and HAProxy (load balancer).

Apartment Rental Application (48 MR) (Figure 2.6) is a web-application we developed to test this project, that is comprised of eight microservices. We use two different workload

when testing this application: a write heavy workload (referred to as *Apartment App Write*) and a workload consisting of a mix of reads and writes (referred to as *Apartment App Mix*).

2.8 Evaluation

We evaluated Privoxy using the applications described in §2.7, and running them on clusters of *m4.xlarge* AWS EC2 instances, each of which ran Ubuntu 17.04. Unless otherwise specified, we use 99^{th} percentile response latency as our performance metric. We have tested on a broader range of metrics, and observed similar results. Below we first present end-to-end evaluation results showing Privoxy's efficacy, and then present microbenchmarks.

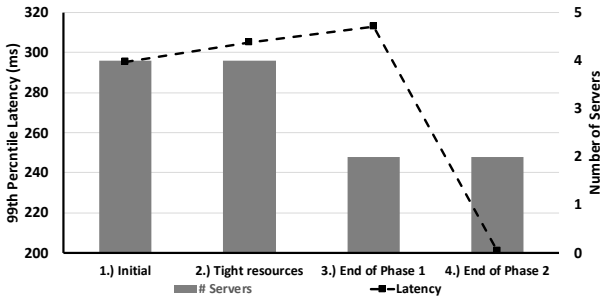
End-to-end Experiments

We start by applying Privoxy to the applications in §2.7. We show results for the MEAN stack in Figure 2.7a, the apartment application with mixed workload in Figure 2.7b, and the HotROD application in Figure 2.7c. In these graphs, we show how performance (99th percentile latency) and server usage changes as Privoxy runs. We show performance (line) and number of servers (bar) at four points: (1) the initial overprovisioned deployment, (2) after stressing and reducing each microservices resources (but before changing placement), (3) at the end of phase 1 (§2.3) after microservices have been packed onto fewer servers, and (4) at the end of the performance improvement phase (§2.3). We use the performance of the initial deployment as our baseline, this is because Privoxy's primary goal is to achieve the same performance with fewer resources. Additionally, finding an optimal deployment (i.e., the global minima for latency) would require an exhaustive search which is infeasible for the applications we consider here.

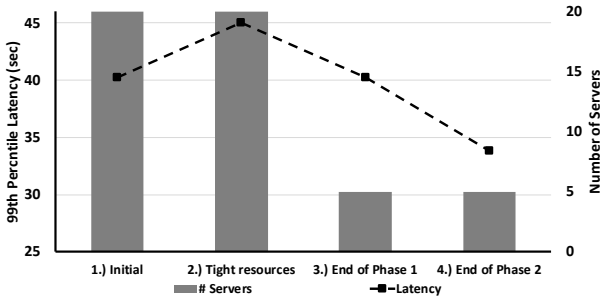
In the three applications we evaluated, the initial deployment split resources evenly between the three microservice instances. After running Privoxy, the MEAN stack achieved 33% better performance with half the number of servers, the apartment application achieved 20% better performance with 2.5× fewer servers, and the HotROD application used half the servers with roughly the same performance level. In order to determine these final deployments, AutoTune required roughly 13, 18, and 18 hours for MEAN stack, Apartment App, and HotRod App, respectively.[1] While the exact improvements vary based on application, we see significant improvements in all for resource efficiency, and performance that is *at least* as good as the initial deployment – in a reasonable amount of time.

The graphs also show that the MEAN stack and the write heavy workload of the apartment application experience slight performance degradation in the initials stages of the process, but show overall improvements after phase 2. This is because the available network bandwidth between microservices increases when they are packed onto fewer machines, and this shifts the bottleneck (in this case to the CPU). The Performance Improvement Phase can hence correct for this problem through resource transfer. While AutoTune was able to improve the HotROD application's resource efficiency, observe that performance does not improve in the Performance Improvement Phase. The HotROD application contains 3

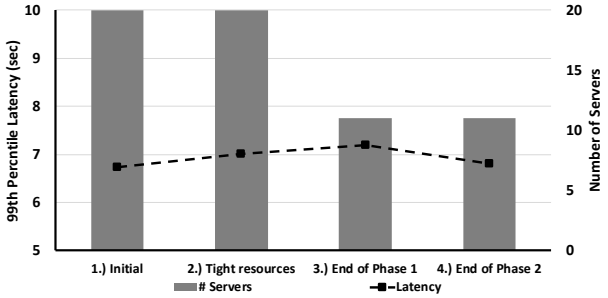
[1]These times might seem long, but we run every trial up to 25 times in order to reduce fluctuations in the results.



(a) MEAN Stack



(b) Apartment Application



(c) HotRod Application

Figure 2.7: Changes in number of servers (bars) and 99 percentile response latency (lines) over the course of running AutoTune end-to-end. We show both values at (1) the initial configuration; (2) after overprovisioned MRs have been reduced; (3) at the end of phase 1 which includes changes to placement; (4) after phase 2.

separate microservices that are all compute bottlenecked; consequently, in the clampdown deployment, there was very little free resources for impacted MRs to grow into. Nevertheless, AutoTune was able to maintain the performance of the application. This is in contrast to the Apartment Application where all three resources are bottlenecks for different microservices (CPU, Disk, and network); this diversity enabled performance improvements in the

Performance Improvement Phase.

Privoxy Robustness

Next we evaluate Privoxy's robustness to variations in user input and deployment conditions.

Robustness to Initial Deployment

Recall that AutoTune identifies locally optimal resource allocations; thus the operator-provided initial deployment can have a significant impact on the outcome. In the end-to-end experiments shown above (2.8), the initial deployments placed three microservice instances on each server, where each microservice instance has an equal split of the server's total resources. We look at three different sets of initial deployments. 1.) deployments where each microservice instance has been provisioned the maximal amount of resources for the instance type 2.) *randomly generated* deployments and 3.) deployments that result from horizontal autoscaling.

Maximally Provisioned Initial Deployment: In this initial deployment, each microservice instance was provisioned an entire EC2 VM instance. In doing so, AutoTune continued to demonstrate effectiveness in reducing server usage. AutoTune was able to reduce server usage by 75% for the Apartment App and the HotRod App by 40%. However, AutoTune did not register performance improvements because IMRs were already maximally provisioned; during the performance improvement phase, there were no resources to transfer because the impacted MRs were already maximally provisioned. Compared to the initial deployments described in the end-to-end experiments (with 3 per machine), AutoTune identifies a deployment that is more performant but also more expensive. For instance, for the HotRod application, starting from a maximally provisioned initial deployment results in a final deployment that requires 2.5x more servers, but also achieves 2.2x better performance.

Randomized Initial Deployment: Table 2.2 demonstrates four such random deployments with the Apartment App. Note that in all cases, AutoTune was able to cut resource usage by 57% – all while either improving or maintaining performance. Interestingly, regardless of the initial performance, 3 out of 4 of the random initial deployments converged to the same local optima. In the one auspicious scenario where the initial random deployment resulted in improved performance, AutoTune was able to converge back to that same performance. For brevity, we elide results from other applications.

Random Dep. #	# Server Reduction	Init. Perf. (s)	Fin. Perf. (s)	% Improve
1	57%	68.3	45.2	33.8
2	57%	38.2	38.7	-0.01
3	57%	63.4	47.0	25.9
4	57%	58.5	47.9	18.1

Table 2.2: End-to-end runs of Apartment Application App with randomly generated initial deployments

Application	Scaling Thres	w/o AutoTune		w/ AutoTune	
		Servers	Perf	Servers	Perf
MEAN	90%	11	1668	6	1630
Apartment App	90%	21	7919	13	7615
HotRod	90%	22	20781	18	21302
MEAN	10%	29	1141	14	1188
Apartment App	10%	25	7686	22	6158
HotRod	10%	–	–	–	–

Table 2.3: Server Usage and Performance resulting from horizontal autoscaling, compared against use of AutoTune. Horizontal Autoscaler scales when avg. utilization exceeds *scaling thres*

Horizontal Autoscaling: In Section 2.4, we discussed how AutoTune could be used jointly with dynamic workloads. In this section, we presented two possible scenarios. In this section, we step through the AutoTune-based solution where increased workload results in performance degradation.

Upon a workload change, resource allocations revert to being overprovisioned (i.e., the initial deployment) and will horizontally scale until some threshold in the scaling policy is met. Suppose the operator decides that this (now stabilized) workload level occurs regularly; the operator can optionally offline deploy AutoTune using that new workload level. Later, when that workload level is encountered again, AutoTune can alter the resource allocations without any operator intervention. What is the performance and resource efficiency advantage of this approach over horizontal autoscaling? To this end, we deployed all three of our microservice applications on Kubernetes, and enabled horizontal autoscaling (HPA). HPA in Kubernetes implements a control loop that checks the average percentage utilization across all pods, and scales up if that threshold is exceeded. We ran our experiments across two separate utilization thresholds: 10% and 90%.

To simulate a sudden burst in load, we increased the workload by a factor of 10x. Upon observing the performance degradation as a result of the increased workload, the horizontal autoscaler would first take over, switching to an overprovisioned resource allocation. Rather than falling back to this overprovisioned deployment everytime, the operator could record the increased workload and subsequently run AutoTune on that higher workload.

The comparison between the horizontal autoscaling approach and AutoTune are in Table 2.3. Performance ranged from very slight decreases to 20% increases, while the number of servers decreased between 12% and 52%. We omit the result of HotRod with 10% scaling because it scaled aggressively beyond the capacity of our resources.

Robustness to Noise

As we discussed in §2.4, a microservice's performance can be influenced by other colocated programs, including programs running in other VMs belonging to other tenants, impacting

Application	% Server Reduction	% Performance Improvement
Apartment	57	14
Apartment (w/ noise)	57	10
HotRod	42	4.6
HotRod (w/ noise)	42	1
MEAN	50	4
MEAN (w/ noise)	50	17

Table 2.4: Comparing AutoTune's efficacy towards server reduction and performance increase with and without noise.

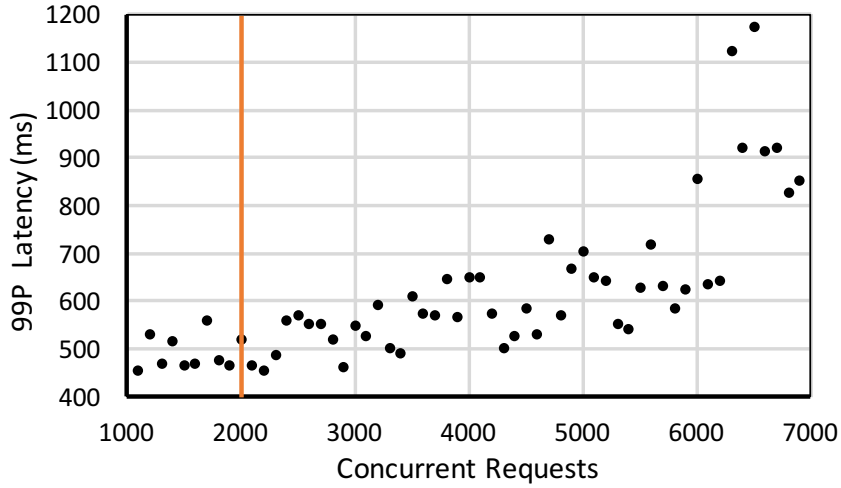


Figure 2.8: Robustness of MEAN App performance with additional concurrent requests. AutoTune used a workload of 2000 requests, indicated by the dashed vertical line.

the measurements that Privoxy relies on. We evaluate the impact of this by deploying our application containers alongside containers running computation at random intervals. This emulates the effect of a multitenant environment. The noisy neighbors run workloads that use all resources i.e., CPU, disk, memory and network. To emulate multitenant environments, these neighboring containers are pinned to a different core from the application under test.

Table 2.4 shows the server reductions across deployments with and without noise. Server reductions remain unchanged while performance improvements varied somewhat (in both directions). Note that the results in this table are deployed on a different instance type (with more resources to facilitate the noisy neighbors) and should not be compared with earlier end-to-end experiments.

Robustness to Workload Fluctuations

Privoxy's mission is simple: given an initial deployment and a representative workload, Privoxy tries to find a more efficient deployment (fewer servers) with at least as good performance. One might worry that after this process of reducing servers, the resulting deployment

would be so highly tuned that it would become very sensitive to workload fluctuations. This would be bad, since fluctuations are inherent in customer-driven applications. Fortunately, in our experience of running the applications we discuss here, the resulting performance is not highly sensitive to small workload fluctuations. We illustrate this with an example.

We took the MEAN application, and then altered the workload by increasing the rate of requests. Because the clients operate in closed request loops (request, then response, then new request), we merely increased the concurrency of this loop to issue several simultaneous requests. The original workload had a concurrency of 2000, and Figure 2.8 shows how MEAN application performance changes as this level of concurrency is varied, while using the deployment computed by Privoxy on the original workload. As shown in the graph, 99p latency grows gradually until the number of concurrent requests triples (from 2000 to 6000 concurrent requests), at which point the performance starts degrading far more rapidly. At this point, the operator would benefit from rerunning AutoTune with a workload consisting of 6000 concurrent requests or more.

This fits with our general experience with Privoxy that application performance does not suddenly degrade. One reason for this is that while the clampdown process produces a tight resource allocation, once these allocations are used to produce a new deployment, there are excess resources on each server because one typically cannot perfectly binpack the microservices into a minimal set of servers. In addition, applications are typically designed to degrade gracefully rather than fall off a cliff, and this might provide another reason for resilience.

Resource Clampdown Phase Illustration

Now that we have examined Privoxy's overall impact in terms of server reduction and application performance, we focus in on the two phases of AutoTune to see where these improvements come from. In this subsection, we demonstrate the resource efficiency gains and runtime of just Resource Clampdown Phase. Table 2.5 details the server savings as well as the Clampdown runtime. In keeping with the previous sections, we provide two initial deployment configurations: one container per machine, and three containers per machine. Only one initial deployment is shown for MEAN stack, since the MEAN stack only has four microservice instances. We find that in all cases resource utilization reduces at the end of this phase, and that across applications this phase took between 30 minutes and 7 hours. This time depends on several factors including the input deployment and the chosen workload, and can hence be reduced by having the operator make appropriate choices.

Performance Improvement Phase Illustration

In this subsection, we evaluate the second phase of AutoTune: Performance Improvement. Like the previous section, we start by providing an overview of its performance gains and runtime. Next, we go a step deeper into the two primary mechanisms that underlie the Performance Improvement Phase. First, we evaluate the effectiveness of resource stressing for IMR identification. We then provide an example of how resource transfer works.

Application	Initial Servers	Final Servers	Iterations	Clampdown Hours
Apartment App Write	21	3	3	1
Apartment App Write	7	3	4	1
Apartment App Mix	21	5	8	3.5
Apartment App Mix	7	4	4	1.75
HotRod	21	11	9	5
HotRod	7	4	6	4.4
MEAN Stack	4	2	4	1

Table 2.5: Running just Resource Clampdown on applications across two different initial deployment settings.

Application	% Improve	Time (hr)
Apartment App Mix,99p latency	26	11
Apartment App Mix, 50p latency	30.4	16
MEAN Stack, 99p latency	65.9	9
MEAN Stack, 50p latency	45	10
HotRod, 99p latency	6	12

Table 2.6: Overall Performance gains from gradient step in performance hill climbing.

Performance Improvement Phase: results

We start by providing overall results from applying Performance Hill Climbing to these applications, before evaluating the individual steps in this phase. Table 2.6 shows the performance improvements in this phase, showing that Performance Hill Climbing in these cases improves application performance by between 17% to 66%. While achieving this improvement takes several hours for some applications, this is determined by the performance metric and workload selected by the operator, and hence depends on the setting in which AutoTune is used.

Application	Single MR	% Improve	Most impacted?
Apartment App Write	Node, CPU	30.4	Yes
Apartment App Mixed	Node, CPU	16.1	Yes
HotROD App.	Api, CPU	16.1	Yes
MEAN Stack	Node, CPU	55.5	Yes

Table 2.7: Effect of Provisioning more resources to the Most Impacted MR on the first iteration of the Performance Hill Climbing Phase.

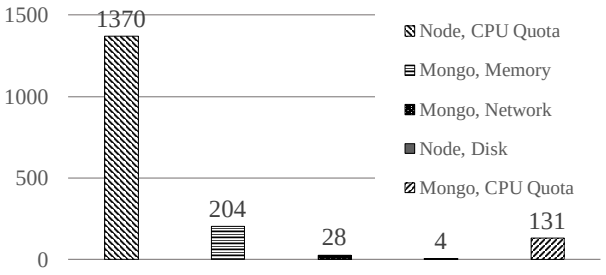


Figure 2.9: MEAN Stack Performance Degradation upon Single Resource Gradient (99^{th} percentile latency)

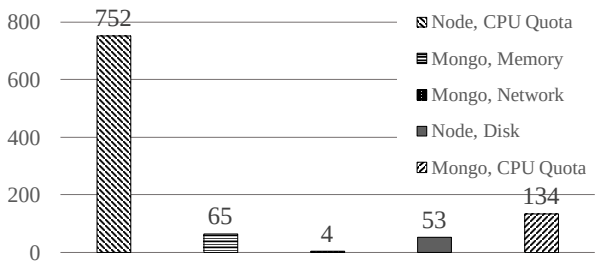


Figure 2.10: MEAN Stack Performance Improvement upon actual improvement resources (99^{th} percentile latency)

Performance Improvement Phase: Stressing Effectiveness

We dive a level deeper and explore the inner workings of AutoTune, starting with the fundamental mechanism of AutoTune: resource stressing. The ability to determine the relative sensitivity of an MR underlies AutoTune's effectiveness. Since AutoTune uses stressing to infer something about each MR, false positives are possible in the Performance Improvement Phase. While we empirically have found that AutoTune does not require an exact, complete ordering of MR sensitivity, the algorithm revolves around two basic tasks that are important to get right. The first is identifying the most impacted MR, as it is the most direct way of improving AutoTune performance in the Performance Improvement Phase. The second is to distinguish impacted MRs from non-impacted MRs, since we only want to transfer resources to impacted MRs. In this subsection, we illustrate how effectively AutoTune accomplishes these two goals for the applications we tested.

First, we demonstrate that resource stressing is an effective way of identifying the most impacted MR. Table 2.7 shows that in every application that we tested, AutoTune was able to identify the most impacted MR correctly. We confirmed that the MR was indeed the most impacted by exhaustively increasing each MR's resource allocation and measuring the resulting performance.[2]

Next, we look at AutoTune's ability to positively identify an impacted MR. We illustrate IMR identification with an example using the MEAN stack. Figure 2.9 shows the measured end-to-end performance resulting from stressing the a single MR. Note that we omit some MRs from the figure for clarity; the MRs we removed did not impact performance when resource allocation was decreased. Based on this result of this gradient, AutoTune would identifies following MRs as being IMRs: (Node, cpu), (Mongo, memory), and (Mongo, cpu). When compared to the the actual performance improvements for the same MRs (Figure 2.10), we see that provisioning more resources to those exact three MRs results in performance gains. The other MRs – correctly identified as NIMRs – did not result in

[2]Note that we can do this kind of resource increase for this special test, but when running Privoxy such increases are not possible because they typically violate the resource constraints on the server.

significant performance improvements when the resource allocation was increased. For the MEAN stack, the stressing mechanism positively identified all the IMRs. Additionally, note that provisioning more resources to the most impacted MR causes end-to-end performance to improve by 752 ms (which is nearly 30%). It is important to re-emphasize here that the Performance Improvement Phase makes no guarantees about the exact *magnitude* of the performance improvement upon the mitigation of the bottleneck.

We observed similar behavior among the Hotrod and Apartment Apps.

Performance Improvement Phase: Resource Transfer Example

Now that we have shown that AutoTune can positively identify IMRs, we discuss concretely how resource transfer works. The gradient phase improves performance by transferring resources between less impacted MRs and more impacted MRs. Transferring resources in this way should work well if everything was cleanly linear in behavior. However, this assumption does not hold in practice, and simultaneously taking two actions (*e.g.,* moving resources from an NIMR to an IMR) can yield unexpected behavior.

To evaluate how well this works in practice, we observed the ability of AutoTune to transfer resources in various setups. Within the Apartment App, we zoomed into the ELK stack, which consists of three microservices: Elasticsearch, Logstash, and Kibana. CPU-Quota was initially evenly distributed between Elasticsearch and Kibana. The baseline performance under this configuration was measured to be 31.49 seconds. In the first iteration, AutoTune identifies Elasticsearch CPU-Quota as the most impacted MR. Upon reducing the resource allocation of $(Kibana, cpu - quota)$ by 25% and increasing $(Elasticsearch, cpu - quota)$ by the same amount, performance improved to 28.48 seconds, equivalent to the performance exhibited simply by improving $(Elasticsearch, cpu - quota)$ by 25% whilst leaving $(Kibana, cpu - quota)$ to be constant. We observed similar behavior for the MEAN stack.

While this does not cover the space of all application behaviors (even within the applications that we explored), thus far we have not seen an application where transfering resources from a correctly identified non-impacted MR and provisioning it to a correctly identified most-impacted MR has resulted in undesirable behavior. In the event of this undesirable behavior, AutoTune is capable of detecting it. Once detected, the *backtrack step* would simply take over and ultimately undo the resource transfer.

Additional Issue: Effect of Interference

In this section, we dive into the issue of interference, which we discussed in Section 2.4 (Additional Issues). In particular, we illustrate how AutoTune copes with interference on the HotRod application, the only application we tested where placement contributed to the application performance. This is because the HotRod applications exhibits a significant number of compute-impacted MRs. AutoTune empirically detects interference by measuring the clamped down performance with the original placement and compares it with the performance on the new compacted placement. For the HotRod application, performance degraded by 12% – despite having the same resource allocations.

Recall that AutoTune reduces the likelihood of new interference by 1.) preserving colocated microservice instances from the initial deployment and 2.) exclusively assigning core(s) to those microservices. Upon applying affinity-based placements and core pinning, performance degradation upon clampdown decreased to 2%. With this mechanism, more compute resources were utilized, but the number of packed nodes remained constant. Ultimately, for this version of the HotRod application, this simple mechanism improves performance after the clampdown phase at no additional cost to the operator. There are clear limitations to this approach, especially when all compute MRs are *similarly* impacted. Our solution here does not attempt to exhaustively eliminate container interference; other projects have addressed this problem extensively [91, 140]. Nevertheless, in the only case where our applications encountered interference, the combination of core pinning and invariant placement sufficiently mitigated the effect of placement on performance.

2.9 Related Work

Recent work to determine resource allocations has largely focused on provisioning resources for *individual microservices*, rather than considering complex dependencies in end-to-end heterogeneous microservice applications. These projects can take either a data-driven or domain specific approach. Examples of work that adopt the data-driven approach include Paragon [44] and Quasar [45] that rely on microarchitectural details to reduce colocation overheds. While these works do not make any assumptions about the application, and can account for behaviours such as noisy neighbors (which are ignored by AutoTune), they cannot scale to applications containing several microservices. Domain specific approaches include work such as Ernest [145], CherryPick [6] and Paris [153] which make strong assumptions about application structure (*e.g.,* assuming map-reduce like data parallel frameworks) or that the application is comprised of homogeneous services. Other approaches to the resource allocation problem have relied on the use of instrumentation and custom tooling to infer application dependencies. This approach adds performance overheads and increases system complexity complexity. Sieve [138] considers performance across multiple microservices, but requires instrumentation like *sysdig* or *dtrace*, with overhead of up to 100% in the worst case [48]. Additionally, these tools suffer from causation ambiguity and leads to high rates of false positives [86, 77]. Other proposed systems allow operators to infer performance behaviors of various systems, but they largely require significant modifications to the hypervisor or to the application under test [64, 65] or only explore performance improvements along a single resource dimension (e.g., network) [65, 111, 22]. Other works have suggested improving application performance through profiling programs and optimizing code [39, 101, 24]; these provide a different set of knobs from impacted MR allocations, and thus can be jointly used with AutoTune to improve application performance. Past proposals on resource scheduling [50, 96, 59] assume that the administrator provides resource requirements as input; thus we view these work as complimentary.

2.10 Conclusion

It is important to consider Privoxy relative to its goals. Privoxy was not intended to provide optimal performance, nor optimal efficiency, nor provable guarantees, nor to work well with arbitrarily varying workloads. Instead, it was designed to be easy-to-use (operators need only provide an initial deployment, a representative workload, and a performance metric) and generally applicable (it does not make any assumptions about the nature of the application) tool that reduces the number of servers needed to deploy applications while maintaining (or improving) performance. We believe Privoxy achieves this goal, and we are not aware of any other tool that does so. However, only widespread use of Privoxy will allow us to fully understand its limitations, which hopefully will lead to further improvements.

Chapter 3

Privacy Policy Enforcement for Web Applications

3.1 Introduction

Many modern web applications use relational databases to store and retrieve data. This data is usually subject to policies that limit who can access the data. These policies can arise from several sources. For example, it is now common for web applications to support privacy settings that allow users to limit visibility of their data to a small subset of other users. In addition, there are general legal regulations such as GDPR and CCPA that limit how data can be shared, and certain kinds of data are subject to more restrictive policies such as HIPAA and FERPA in the U.S. Moreover, some institutions impose their own policies for internal security or privacy reasons.

At present, web developers implement application specific logic to enforce data privacy policies. In most cases an application's permission checking logic relies on session context (*e.g.,* to identify what user is logged in) and metadata retrieved from the database (*e.g.,* to identify what they can access) in order to determine whether or not a request complies with data protection policies. Getting this logic correct is challenging, and has previously been a source of bugs in production websites [135, 82, 81, 8, 98, 60].

As a result, there has been significant research focused on building tools, frameworks, and languages to simplify the enforcement of data protection policies. While we delay a discussion of these approaches to §3.8, these approaches either (i) only support programs written using specific languages and APIs and so cannot be applied to existing applications, or (ii) rewrite query results to remove data that would violate data protection policies, which could result in unexpected application behavior (*e.g.,* the user has no idea that there are missing results).

In this paper, we propose an alternate approach to enforcing data protection policies that meets four goals:

- **Policy expressiveness:** The approach should be able to enforce a wide range of realistic data protection policies.

- **Backwards compatibility:** The approach should apply to applications built using common web frameworks without requiring significant code modification.
- **Semantic transparency:** The approach should not impact application semantics. It should instead fully answer queries that comply with protection policies, and completely block any queries that violate these policies (rather than providing partial, and perhaps misleading, results for non-compliant queries).
- **Low overhead:** The approach should have limited impact on the application's page load time.

We build a system called Privoxy that meets these four requirements. Privoxy interposes a proxy between a web server and its backend database, dynamically verifies that queries issued by the web server comply with data protection policies, and rejects non-compliant queries. In Privoxy, policies are expressed as a set of *authentication views*, but queries are issued against *base tables* (rather than the views). Compliant queries are forwarded to the database, while non-compliant queries result in an error that the application needs to handle. We assume that in production environments non-compliant queries are rare (having been mostly eliminated during testing), and focus on the efficiency of handling compliant queries.

Privoxy (Figure 3.1) uses SMT solvers to check query compliance. This requires access to not just the query, but also a *trace* of all previous queries issued by the request and their results because what data is accessible to a user is often determined by additional metadata stored in the database, which could have been accessed by a previous query. The basic technique employed by Privoxy is to convert the query to be checked, the trace of past queries and results, and policies into first-order logic formulas and use an SMT solver to check the satisfiability of the resulting formulas. As we explain later, the SMT solver returns an unsatisifiability proof when queries comply with policies, and a test demonstrating violations in the other case.

While correct, this basic approach is impractical since it can add significantly to page-load times. We address this problem by first generalizing and then caching query compliance checking results. We use the unsat-core [129] generated by the SMT solver for compliant queries to identify a class of other queries that are guaranteed to also be compliant. The resulting generalization is cached, and the cached result is used whenever the application issues a query belonging to this general class. Note that this approach does not allow us to cache results for non-compliant queries (because there is no unsat-core).

Implementing SMT-based compliance checking, generalization, and caching required addressing several challenges:

- Extending existing approaches for translating queries into first-order logic to cover most queries that appear in practical web applications. To do this, we developed techniques to rewrite complex SQL queries into basic queries that can be converted into first-order logic (§3.4).

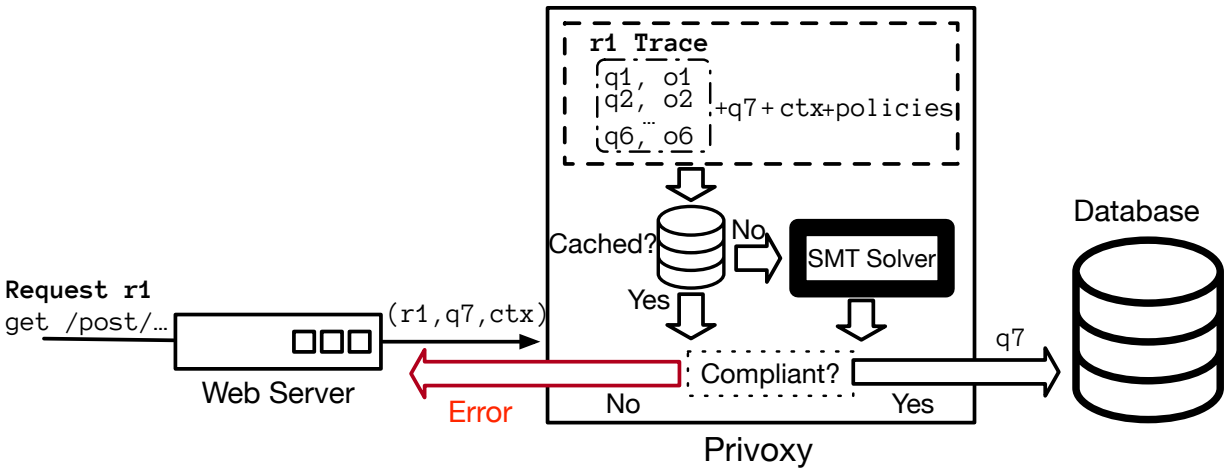


Figure 3.1: An overview of Privoxy

- Efficiently checking query compliance using SMT solvers. This requires invoking multiple SMT solvers in parallel (§3.2) and finding an over-approximation that improves checking performance for non-compliant queries (§3.4).
- Generating unsat-cores that improve cache hit rates (§3.5).

We have implemented Privoxy and evaluated its efficiency using diaspora* [46], a social networking application. We found that our approach imposes modest page load overheads of 7–11% when queries are cached.

Privoxy has some important limitations. Specifically, it assumes that the application obtains all of its information from the database through SQL queries that are visible to the proxy. If an application can obtain additional information through other means, policies could be violated. Furthermore, Privoxy only supports a subset of SQL, as described in §3.4, and is at the mercy of solver performance and unsat-core size.

3.2 System Design

Application Assumptions and Threat Model

Privoxy targets web applications that store persistent data in a relational database and interact with the database using SQL. We assume that a user is logged in (i.e., we do not model authentication) and that the current user's identifier is stored in a *request context*. The application has access to the database and the request context when serving a web request; each request is handled independently from others. We assume that the application authenticates the user correctly, and that the correct request context is sent to the enforcement system (§3.2).

A *privacy policy* dictates, for a given request context, what information stored in the database is *accessible* to the user and what information is *inaccessible*. Any information

outside the database is assumed to be public knowledge. We assume that the user only interacts with the application by sending web requests and inspecting the responses, and does not observe any side channels such as running time.

Finally, as a simplifying assumption, we only focus on privacy enforcement for database *reads* as done in prior work [**lefevre04:hippocratic**, **stonebraker74:acl**, 4, 15, 16, 20, 67, 123, 131, 150] since reads are more likely subject to fine-grained privacy policies. We briefly discuss how to extend Privoxy to handle database updates in §3.9.

System Overview

As a privacy enforcement system, the most basic requirement of Privoxy is *soundness*: It must prevent the application from leaking information that is inaccessible to the user. However, as it is challenging to achieve both soundness and completeness, we accept that Privoxy may reject certain application behavior even when this does not violate data privacy policies. We discuss a main source of such false positives in §3.2.

Privoxy is a proxy that sits between the application and the database (Figure 3.1). It monitors SQL queries issued by the application and rejects those that could expose information that should be inaccessible to the user. It takes (1) the database schema and (2) the privacy policy specified as database *views* (§3.3) as configuration, and checks privacy compliance for each web request handled by the application separately.

At the start of serving a web request, the application informs Privoxy of the request context (e.g., the user ID), which will dictate, in conjunction with the privacy policy, what information is accessible to this request. Then, every SQL query issued by the application traverses Privoxy, which attempts to verify that the query can only reveal information accessible to the current user; if this is the case, we call the query *compliant* (formally defined in §3.3), and pass it on to the database without modification. If verification fails, the query is blocked.

Privoxy verifies query compliance by encoding the conditions for *non-compliance* into an SMT formula (§3.4) and checking its satisfiability by invoking an *ensemble* of SMT solvers in parallel (§3.6). As we found that no single SMT solver delivers the best performance on all queries, we adopt this ensemble approach to reap the benefit of the fastest solver for each query. If all solvers time out, Privoxy conservatively blocks the query.

One challenge in verifying compliance is that a query's compliance can *depend on the outputs of previous queries* issued in the same request. For example, a bulletin board application might issue a first query to check whether the user is authorized to view a post and, only if so, issue a second query to fetch the post content. The second query, while compliant in this context, would not have been compliant had there not been a previous query that determined the user is authorized.

To address this problem, Privoxy maintains a *trace* of all queries and responses it has seen so far in the current request, and checks query compliance *in the context* of this trace. As we assume the application serves different requests independently, Privoxy can safely clear the trace after a request ends.

Privoxy sits on the critical path of request handling and thus can impact application performance. Unfortunately, calling SMT solvers for every single query incurs significant overhead. To reduce this overhead, Privoxy *caches* compliance decisions rendered by SMT solvers so that it can bypass the solvers in the future (§3.5). As it is unlikely to encounter the *exact same* query (and trace) many times, we *generalize* a compliance decision to make it applicable to other queries of the same form (e.g., as issued when a different user visits the same URL with different parameters), thus improving cache hit rates. Our cache generalization technique relies on extracting *unsatisfiability cores* from SMT solvers indicating a subset of assumptions that are sufficient to make a query compliant.

Required Code Changes

Porting an application to Privoxy requires three modifications. First, the application must cleanly handle rejected queries, presumably by returning some error message. Second, it must send the request context (e.g., user ID) to the proxy at the start of each request. To explain the third requirement, we note that in HotCRP when a user attempts to view a paper submission, HotCRP directly fetches the requested paper from the database and then, after the fact in application code, checks whether the user is authorized to view the paper. Privoxy would block this pre-emptive fetch of the requested paper. Thus, the application must be modified to not request information that it does not plan on revealing to the user, so that it does not incur spurious blocked queries.

In §4.7, we report that porting an existing social networking app to Privoxy requires less than 10 lines of code change.

3.3 View-based Policy and Compliance

As mentioned in §3.2, Privoxy expects privacy policies to be specified using database *views*. Views are a well-established access control mechanism in databases, having been studied for at least three decades [103]. As views are defined in SQL, we expect it to be easy for programmers to specify policies without having to learn a separate policy language.

In this section, we introduce how to specify a privacy policy using views (§3.3) and what it means for a query to be *compliant* to such a policy (§3.3). While the *specification* scheme is borrowed from the database literature, a novel *compliance* formulation is required due to a crucial difference in our use case: While databases require queries to be written in terms of the views, our target applications, having no knowledge of the views, issue queries against the base tables only. We contrast Privoxy with databases in more detail in §3.8.

To help with the explanation, we will use as a running example a simple calendar web application backed by a relational database with the following schema:

$$\begin{gathered} \mathit{Users}(\underline{\mathit{UId}}, \mathit{Name}, \mathit{IsAdmin}), \\ \mathit{Events}(\underline{\mathit{EId}}, \mathit{Title}, \mathit{Duration}), \\ \mathit{Attendances}(\underline{\mathit{UId}, \mathit{EId}}, \mathit{ConfirmedAt}), \end{gathered}$$

where primary and unique keys are underlined. Separately, the developer also specifies a request context; in this case, it consists only of a single parameter *MyUId*, which denotes the *UId* of the currently logged-in user.

Specifying Policies as Views

In Privoxy, a privacy policy consists of a set of views, each of which is a SQL query that selects information that should be accessible to the request context (here, the logged-in user). A view definition can refer to parameters from the request context, allowing the policy to differ depending on the context. Each view can be thought of as a "rule" granting access to certain data, and the information exposed by the views taken together constitutes the entirety of accessible information.

We will illustrate this scheme with three examples views definitions, where `?MyUId` refers to the current user ID.

Example 3.3.1. The rule "*the current user can view everyone's user information*" is captured by view V_1:

```
SELECT * FROM Users
```

Example 3.3.2. The rule "*the current user can view all events they attend*" is captured by view V_2:[1]

```
SELECT * FROM Events e, Attendances a
WHERE e.EId = a.EId AND a.UId = ?MyUId
```

This view joins the *Events* and *Attendances* tables to find events attended by the current user, who is identified by the request context parameter *MyUId*.

Example 3.3.3. The rule "*the current user, if they are an admin, can view all events*" is captured by view V_3:[2]

```
SELECT e.*, a.*
FROM Events e, Attendances a, Users u
WHERE e.EId = a.EId AND
       u.UId = ?MyUId AND u.IsAdmin
```

If the current user is an admin, V_3 returns *all* event and attendee information; if not, V_3 returns empty as expected.

Using these three views, we can write down the privacy policy for this calendar application as $\mathcal{V} = \{V_1, V_2, V_3\}$, encompassing all information accessible to the current user. We call it "policy $\mathcal{V}$" for brevity.

[1]For simplicity, view V_2 does not reveal the *other* attendees of a meeting. If desired, this information can be included using a self join on *Attendances*.

[2]For simplicity, we assume that each event has at least one attendee.

Compliance to View-based Policy

Suppose the application, while handling a request, has issued queries $Q_1, \ldots, Q_{n-1}$ and received a response for each query. It now issues query Q_n. Intuitively, we say that Q_n is *compliant* to a policy $\mathcal{V}$ if the query, executed on *any* database instance D, reveals no more information than the views in $\mathcal{V}$ do, assuming that (1) D conforms to the database schema, and (2) D is consistent with the outputs of previous queries $Q_1, \ldots, Q_{n-1}$.

Before formally defining compliance, we discuss three example queries and their compliance to the policy $\mathcal{V}$ from §3.3.

Example 3.3.4. The query Q_1:

```
SELECT UId, Name from Users
```

is compliant, since the information it reveals is a subset of V_1.

Example 3.3.5. The query Q_1':

```
SELECT Title FROM Events WHERE EId = 5
```

is *not* compliant—on a database D where the current user is not an attendee of Event #5 and not an admin, this query reveals information not covered by $\mathcal{V}$.

While the previous two examples feature one query only, our next example comprises a trace of two queries, where the second query's compliance depends on the first query.

Example 3.3.6. Suppose $MyUId = 2$. The application first issues a compliant query Q_1'':

```
SELECT * FROM Attendances
WHERE UId = 2 AND EId = 5
```

and gets back a row, indicating that User #2 attends Event #5. The application then issues a second query Q_2'':

```
SELECT * FROM Events WHERE EId = 5
```

This is the same query as the non-compliant query from Example 3.3.5, but it *is* compliant in this context because here we only consider databases D that are consistent with the output of Q_1''. In other words, given that Q_1'' establishes the current user's attendance at Event #5, query Q_2'' is compliant.

Definition 3.3.7 (Query compliance given trace)**.** We say that a query Q is *compliant* to a policy $\mathcal{V}$ given a trace of query–output pairs $\{(Q_i, O_i)\}_{i=1}^{n-1}$ if for every pair of databases D_1, D_2 that conform to the schema and satisfy:

$$\begin{aligned} Q_i(D_1) &= O_i, && (\forall 1 \le i \le n-1) \\ Q_i(D_2) &= O_i, && (\forall 1 \le i \le n-1) \\ V(D_1) &= V(D_2), && (\forall V \in \mathcal{V}) \end{aligned}$$

we have $Q(D_1) = Q(D_2)$.

This definition requires that any two databases that are consistent with the trace and agree under every view in $\mathcal{V}$ must also agree under Q. In other words, the output of Q is *uniquely determined* by $\mathcal{V}$ among the databases under consideration.

Open Question: Complexity of Compliance

Our definition of query compliance is a generalization of *query determinacy* [107, 128], which is equivalent to compliance given an empty trace. Query determinacy is undecidable even when views and queries are restricted to conjunctive queries [54, 53]. Although several decidable cases have been proposed [107, 3, 113], they are not expressive enough for our use case.[3]

However, given our success in deciding compliance using SMT solvers (§3.7), we hypothesize that there exists a natural class of views and queries, which captures the privacy use case, for which determinacy, and compliance, are decidable. Identifying such a class and showing the complexity of compliance checking is a promising avenue for future work.

3.4 Compliance Checking with SMT

In this section, we describe how we verify compliance using SMT solvers. Our approach is based on the correspondence between relational algebra, on which SQL is based, and first-order logic [37]. This correspondence allows us to formulate (the negation of) compliance as an SMT formula (§3.4).

Unfortunately, this formulation does not trivially extend to practical queries because SQL semantics are beyond that of basic relational algebra. In §3.4, we discuss how to *rewrite* practically relevant SQL queries into basic relational algebra, enabling the use of SMT solvers.

Even if we are able to encode the compliance property into first-order logic, checking satisfiability using a SMT solver can take unbounded time. Empirically, while the solvers can typically check *compliant* queries in a reasonable amount of time, they time out on *non-compliant* queries. To solve the latter case, in §3.4 we formulate an alternative "size-bounded" formula that allows the solvers to quickly find *small models* for non-compliance, which we demonstrate in §3.7 to be sufficient in many practical cases. Privoxy checks the size-bounded formula after the regular formula times out, an indication that the query is not compliant. We introduce two other optimizations for compliant queries in §3.4.

Translating Compliance to SMT

Given a schema, policy, trace, and query, we formulate the *negation* of query compliance (Definition 3.3.7) as an SMT formula F; the query is compliant if and only if F is unsatisfiable.

Our formulation is based on a fundamental result in database theory stating, informally, that relational algebra (RA) under set semantics has the same expressiveness as first-order

[3]Furthermore, every study of query determinacy we are aware of assumes set semantics only, and do not consider bag / SQL semantics (§3.4).

logic (FOL) [37]. Relational algebra consists of five basic operators—projection, selection, cross product, union, and difference—and relations (including tables and query results) are interpreted as *sets* of rows, i.e., no duplicates. Under this equivalence, tables correspond to relations in FOL, and the operators can be implemented using existential quantifiers, conjunctions, disjunctions, and negations. We illustrate this translation with examples.

Example 3.4.1. Using our running example (§3.4), we translate into FOL the following query Q executed on a database D:

```
SELECT e.EId, e.Title
FROM Events e, Attendances a
WHERE e.EId = a.EId AND a.UId = 2
```

Let $E^D(\cdot,\cdot,\cdot)$ and $A^D(\cdot,\cdot)$ be relations in FOL representing the *Events* and *Attendances* table in the database D, and write:

$$\begin{aligned}\mathsf{Q}^D(\mathsf{x}_e,\mathsf{x}_t) \coloneqq \exists x_d, x_u, x'_e, x_c.\, E^D(\mathsf{x}_e,\mathsf{x}_t,x_d) \wedge A^D(x_u,x'_e,x_c)\\ \wedge \mathsf{x}_e = x'_e \wedge x_u = 2.\end{aligned}$$

$\mathsf{Q}^D(\mathsf{x}_e,\mathsf{x}_t)$ corresponds to the statement that $(\mathsf{x}_e,\mathsf{x}_t) \in Q(D)$, i.e., that the row (v_e,v_t) is returned by Q when executed on database D. Note that Q^D is not a symbol in FOL, but merely a shorthand that "expands" to the right-hand side.

Example 3.4.2. We now write down a formula for the *non-compliance* of the single query Q above, with respect to policy $\mathcal{V} = \{V_1, V_2, V_3\}$ from §3.3, according to Definition 3.3.7.

Let D_1 and D_2 be databases that follow the example schema, and write down FOL encodings, $\mathsf{V}_1^{D_i}, \mathsf{V}_2^{D_i}, \mathsf{V}_3^{D_i}$, and Q^{D_i}, for the views and query on database D_i $(i = 1, 2)$ as in Example 3.4.1. Then, the desired formula F is the conjunction of:

$$\begin{aligned}
&\forall\overline{x}.\mathsf{V}_1^{D_1}(\overline{x}) \leftrightarrow \mathsf{V}_1^{D_2}(\overline{x}), && (V_1(D_1) = V_1(D_2))\\
&\forall\overline{x}.\mathsf{V}_2^{D_1}(\overline{x}) \leftrightarrow \mathsf{V}_2^{D_2}(\overline{x}), && (V_2(D_1) = V_2(D_2))\\
&\forall\overline{x}.\mathsf{V}_3^{D_1}(\overline{x}) \leftrightarrow \mathsf{V}_3^{D_2}(\overline{x}), && (V_3(D_1) = V_3(D_2))\\
&\exists\overline{x}.\neg\left[\mathsf{Q}^{D_1}(\overline{x}) \leftrightarrow \mathsf{Q}^{D_2}(\overline{x})\right], && (Q(D_1) \neq Q(D_2))
\end{aligned}$$

where $\overline{x}$ denotes a distinct sequence of variables. If formula F is *satisfiable*, then we have found a pair of databases D_1 and D_2 that are equal under all views in $\mathcal{V}$ but not under query Q, indicating that Q is *not compliant*. Conversely, if formula F is *unsatisfiable*, then Q is *compliant*.

For ease of exposition, we omitted the encoding of primary key constraints or consistency with prior query–output pairs in these examples, but they can be encoded similarly.

Handling Practical SQL Queries

While the translation from relational algebra to first-order logic is straightforward, it does not trivially extend to real-world SQL due to two major *semantic gaps*:

1. While the translation assumes that relational algebra is evaluated under *set semantics*, in practice databases use a mix of set, bag, and other semantics when evaluating queries. For example, a SQL `SELECT` clause can return duplicate rows, while `UNION`, a set operator, removes duplicates.

2. SQL supports operators beyond basic RA operators including aggregations and sorting. These non-relational operators cannot be trivially translated into first-order logic.

To handle more complex SQL semantics, we first need to assume that the tables in our database contain *no duplicate rows*. It suffices for the schema to contain a primary key in each table, which is the case in many web applications since object-relational mapping (ORM) libraries like Active Record[4] and the Django ORM[5] add primary keys by default.

Given this assumption, our general approach is to rewrite complex SQL queries into *basic queries* (§3.4) that we can handle. We describe rewriting strategies for many classes of queries encountered in applications, and point out SQL features that our rewriting does not currently handle (§3.4).

Basic SQL Queries

Definition 3.4.3. We call a SQL query *basic* if it:

1. Is a `SELECT`–`FROM`–`WHERE` clause on base tables, where the `WHERE` conditions are composed of logical and equality operators;[6] or a `UNION` of such `SELECT` clauses; and,

2. Never returns duplicate rows.

It follows that a basic SQL query, when executed on a database with no duplicate rows, corresponds directly to RA under set semantics, and thus can be translated into FOL as in §3.4. We could have also allowed the `MINUS` operator in basic queries; we omitted it as it is not used in our evaluation.

To determine that a query is basic, we first ensure that it falls into one of the two syntactic categories. And then, for each category, we ensure that the query returns no duplicate rows:

1. If the query consists of a single `SELECT` clause, we check that it satisfies one of these sufficient conditions:

    - It contains the `DISTINCT` keyword.

[4] `https://edgeguides.rubyonrails.org/active_record_basics.html`

[5] `https://docs.djangoproject.com/en/3.2/topics/db/models/`

[6] In fact, we can also support any operator implemented by an SMT solver (e.g., integer comparison).

- It projects primary / unique key column(s) from every table in `FROM`, e.g., `SELECT UId, Name FROM Users`.
- Its returned rows are constrained by primary key / uniqueness in its `WHERE` clause. For example, for the following query to return the value x multiple times:

  ```
  SELECT e.EId FROM Events e, Attendances a
  WHERE e.EId = a.EId AND a.UId = 2
  ```

  there must be multiple occurrences of the row $(2, x, ?)$ in the *Attendances* table, which is impossible.

2. If the query is a `UNION` of `SELECT`'s, it cannot return duplicate rows as the SQL `UNION` operator removes duplicates.

In our experience, policy views can typically be defined directly as basic queries (e.g., V_1—V_3 from §3.3), and so we currently only support policy views defined as basic queries.

Rewriting Into Basic Queries

When an application issues a complex query, the checker *rewrites* it into a basic query. Ideally, we would like the rewritten query to be equivalent to the original. When this is not possible, we approximate a complex query Q with a basic query Q' such that Q' reveals *at least as much information* as Q does.[7] Such approximation preserves soundness but might sacrifice completeness.

We now describe how to rewrite several classes of complex queries that we encounter in our evaluation.

Inner joins. A query of the form:

```
SELECT ... FROM R1 INNER JOIN R2 ON C1 WHERE C2
```

is equivalently rewritten to the basic query:

```
SELECT ... FROM R1, R2 WHERE C1 AND C2
```

Left joins on foreign key. For a query of the form:

```
SELECT ... FROM R1 LEFT JOIN R2 ON R1.A = R2.B ...
```

If `R1.A` is a foreign key into `R2.B`, then the query can be equivalently written as an inner join, which is handled above.

Order-by and limit. We ensure that the `ORDER BY` columns are included in the query output, and can then safely discard the `ORDER BY` clause. We also discard any `LIMIT` clause but, when adding this query to the trace, use a modified condition $O_i \subseteq D(Q_i)$ (instead of "=") to indicate that the checker may have only observed a partial result.

Aggregations. We approximate `SELECT COUNT(*) FROM R` with `SELECT * FROM R`.

[7] We simply guarantee that Q can be computed from the result of Q'.

Left joins. Left joins of the form:

```
SELECT DISTINCT A.* FROM A LEFT JOIN B ON C1 WHERE C2
```

can be equivalently rewritten to the basic query:

```
(SELECT A.* FROM A INNER JOIN B ON C1 WHERE C2)
  UNION
(SELECT * FROM A WHERE C3)
```

where `C3` is obtained by replacing each occurrence of `B.?` with `NULL` in `C2` and simplifying the resulting predicate.[8]

Nulls. SQL `NULL`s exhibit special semantics—most comparisons involving `NULL` return *unknown*, which is propagated through logical connectives according to the three-valued Kleene logic [18]. We currently model `NULL` as just another regular value in the data type. This leads us to *over-approximate* the information fetched by a basic query, which is sound for the query being checked. While not sound for policy views in general, this model does preserve semantics for the class of views we encounter in our evaluation—basic queries whose `WHERE` clauses are a conjunction of equalities with at least one non-`NULL` operand. In future work, we plan to model `NULL`s systematically using a two-valued semantics of SQL [61, § 6].

Feature not supported. We currently do not support SQL features such as `GROUP BY` clauses, the `ANY` and `EXISTS` keywords, etc., although it is possible to similarly approximate these features using basic queries. Prior work has studied other formalisms that model more complex SQL semantics [29, 35, 143, 142, 149, 34, 151], and in the future we plan to leverage them to model SQL queries more precisely.

Finding Small Models for Non-compliance

As we show in §3.7, the SMT formulation from §3.4 allows off-the-shelf SMT solvers to check *compliant* queries within hundreds of milliseconds. On *non-compliant* queries (i.e., satisfiable formulas), however, the solvers time out on moderately sized schemas where tables have up to tens of columns. Empirically we found that finite model finders in CVC4 [121] and Vampire [119] return quickly when tables have few columns, but either time out or run out of memory on larger tables. While it is sound to block a query whose compliance cannot be decided, the developer is left unsure whether compliance is violated or simply cannot be ascertained.

To check non-compliant queries faster, we rely on the observation that non-compliant queries encountered in practice typically enjoy a *small counterexample property*—there exist databases D_1 and D_2, where each table has a small number of rows, that violate the conditions set forth in Definition 3.3.7.

Driven by this observation, we formulate a special *size-bounded* SMT formula where the size of each table in databases D_1 and D_2 is bounded by a constant. For example, a *Users* table with a size bound of 2 can be represented as follows:

[8] As long as `C2` contains no negations, it is safe to treat a `NULL` literal as `FALSE` when propagating through or short-circuiting `AND` and `OR` operators.

UId	Name	IsAdmin	Exists?
$x_{u,1}$	$x_{n,1}$	$x_{a,1}$	b_1
$x_{u,2}$	$x_{n,2}$	$x_{a,2}$	b_2

where the x variables are symbols representing table content, and each b_i is a Boolean encoding whether row i exists.

Under this size-bounded table encoding, we can translate relational algebra into a simpler SMT formula *without using any relation symbols or quantifiers*. For example, the query:

```
SELECT IsAdmin FROM Users WHERE UId = 5
```

can be written as:

$$\mathsf{Q}(\mathsf{x}_a) \coloneqq \bigvee_{i=1}^{2} \left(x_{u,i} = 5 \wedge x_{a,i} = \mathsf{x_a} \wedge b_i\right),$$

while the regular formulation (§3.4) would have used a relation symbol and an existential quantifier. As we show in §3.7, such simple formulas can be discharged by solvers quickly.

The main downside to this formulation is that when two tables are joined, the formula size grows multiplicatively in table sizes. Therefore, this encoding is only feasible with small size bounds. In our evaluation, we set a table's size bound to be two plus the number of rows required to produce the previous query results, although we can increase this bound if needed.

Other Optimizations

We now describe two optimizations for the compliant case.

Column-based filtering. If the policy contains a view of the form `SELECT C1, C2, ..., Ck FROM R`, then any query that references only columns `R.C1`, ..., `R.Ck` must be compliant. Privoxy accepts such queries without SMT solving. As we show in §3.7, this optimization filtered out 7 % to 42 % of queries on our evaluation benchmarks (Table 3.1).

Strong compliance. We tweak Definition 3.3.7 to get the definition of "strong compliance" by replacing query/view equality with containment.

Definition 3.4.4 (Strong compliance)**.** A query Q is *strongly compliant* to policy $\mathcal{V}$ given trace $\{(Q_i, O_i)\}_{i=1}^{n-1}$ if for each pair of databases D_1, D_2 that conform to the schema and satisfy:

$$\begin{aligned} Q_i(D_1) &= O_i, && (\forall 1 \le i \le n-1) \\ Q_i(D_2) &= O_i, && (\forall 1 \le i \le n-1) \\ V(D_1) &\subseteq V(D_2), && (\forall V \in \mathcal{V}) \end{aligned}$$

we have $Q(D_1) \subseteq Q(D_2)$.

It follows that strong compliance implies compliance. We found that the Z3 and Vampire can verify strong compliance faster (when it holds), and so we use it as a fast-path check. We still solve the regular formula with CVC4.

Remark 3.4.5. In the other direction, compliance *does not* imply strong compliance. In fact, when the trace is empty and the views $\mathcal{V}$ and query Q are conjunctive queries (CQs), compliance holds iff $\mathcal{V}$ *determines* Q (§3.3), and strong compliance holds iff Q has a CQ *rewriting* using $\mathcal{V}$ [87, 68]. It has been established that for CQs, determinacy does not coincide with the existence of a rewriting [108, 3]; this implies that compliance and strong compliance are not equivalent.

However, strong compliance coincides with compliance for every query encountered in our evaluation. This observation leads us to hypothesize that the two notions are equivalent for a large class of queries encountered in practice.

3.5 Decision Generalization and Caching

While SMT solvers can verify a wide range of queries, they often take significant time to do so. As we show in §4.7, while a solver typically checks a query within 200 ms, a page load can consist of *tens of queries*, resulting in *seconds* of overhead.

Therefore, we aim to to *avoid calling solvers* whenever possible. To do so, we rely on the observation that, while an application can issue an unbounded set of queries at run time, it only exhibits a finite number of different behaviors. For example, two users viewing different events on a calendar app trigger two distinct sequences of queries. However, the two sequences are generated by the same program logic, and are thus likely to be identical in structure while differing only in parameters (e.g., event ID). If a solver deems the first sequence compliant, we can *generalize* this knowledge to conclude that the second sequence is also compliant without calling any solvers again.

This generalization step is the central challenge we tackle in this section: Given a query that has been deemed compliant with respect to a trace (§3.3), how to abstract this query–trace pair into a *decision template* such that (1) any query–trace pair that matches this template is compliant, and (2) the template is generic enough to match traces produced from similar requests. We do not generalize *non-compliant* decisions, which are less performance-sensitive as they typically indicate a bug in the application, to be fixed by human intervention.

We present a concrete example of such a decision template in §3.5. We then describe how we generalize compliance decisions into generic templates by extracting *unsatisfiability cores* from the solvers (§3.5). Finally, we introduce Privoxy's *decision cache*, which stores these templates and uses them to declare new queries compliant, without calling solvers, whenever a new query–trace pair matches a template (§3.5).

Motivating Example

Let us return to our running example, the calendar application (§3.3). Suppose a user whose $UId = 1$ visits the URL `/events/42` to view Event #42. To serve this request, the application issues a trace of SQL queries, shown in Figure 3.2 (every query except the last is followed by

1. `SELECT * FROM Users WHERE UId = 1`
   - `(UId=1, Name=John Doe, IsAdmin=false)`
2. `SELECT * FROM Attendances WHERE UId = 1 AND EId = 42`
   - `(UId=1, EId=42, ConfirmedAt="05/04 1pm")`
3. `SELECT * FROM Events WHERE EId = 42`

Figure 3.2: An example trace of queries generated by a calendar application when a user (with user ID 1) views Event #42. Every query except the last is followed by its returned rows.

```
SELECT * FROM Attendances
WHERE UId = |?MyUId| AND EId = |?0|
```

- `(UId = ?MyUId, EId = ?0, ConfirmedAt = *)`

```
SELECT * FROM Events WHERE EId = |?0|
```

Figure 3.3: The decision template generated from the trace in Figure 3.2.

its returned rows). The current query is Query #3, which, as explained in Example 3.3.6, is compliant because Query #2 has established that the user is an attendee of Event #42. As a result, the solver checking this query trace returns "unsatisfiable" (§3.4).

Our goal is to generalize this trace into a template that applies to another user viewing a different event. Figure 3.3 shows the decision template in this case. The notation is read as follows: If each (parameterized) query–output pair above the horizontal line has a match in a trace $\mathcal{T}$, then any query of the form below the line is compliant given $\mathcal{T}$. This particular example codifies the knowledge that, after we ensure that a user x attends an event y, user x can view event y for any x and y.

Two transformations took place as we went from the concrete trace to the template: (1) We *pruned* the first query, which is inconsequential to the compliance decision; and (2) We replaced the concrete values with *parameters* like `?0`. In this template, occurrences of `?0` constrain the event ID checked previously to be the same as the event ID fetched in the current query. The `ConfirmedAt` value should have been given its own parameter `?1`;

1. `SELECT * FROM Users WHERE UId = |?MyUId|`

   - `(UId = ?MyUId, Name = ?0, IsAdmin = ?1)`

2. `SELECT * FROM Attendances`
   `WHERE UId = |?MyUId| AND EId = |?2|`

   - `(UId = ?MyUId, EId = ?2, ConfirmedAt = ?3)`

---

3. `SELECT * FROM Events WHERE EId = |?2|`

$?MyUId = 1$ $?0 =$ `"John Doe"` $?1 =$ `false`
$?2 = 42$ $?3 =$ `"05/04 1pm"`

Figure 3.4: The base template for the trace in Figure 3.2.

but since it appears only once, we write it as "*", which denotes that an arbitrary value would work. If a concrete value is material to the compliance decision (e.g., `IsAdmin=true`), it would be kept as-is in the template.

In the next subsection, we describe how to systematically extract decision templates like Figure 3.3 from concrete traces.

Extract Templates Using Unsat Cores

The key to producing a generic template from a concrete trace is to extract a small set of assumptions in the trace that is sufficient to render the query compliant, and to keep only those assumptions in the template. To this end, we rely on the *unsatisfiability core* [129] (or "unsat core") produced by an SMT solver on an unsatisfiable formula. An unsat core is a subset of clauses in the formula that remains unsatisfiable even with all other clauses removed; the assumptions included in the unsat core are the ones we keep in the template.

To produce a useful unsat core, we must *expose* each individual assumption (e.g., $MyUId = 1$) to the solver as a *separate clause* in the formula, to give the solver the opportunity to exclude it from the unsat core should it be extraneous.

Therefore, we formulate a slightly different *generalization formula* for unsat core extraction. The first step is to produce a *base template* by abstracting the concrete trace into a *parameterized trace*, with equal values being given the same parameter; we reuse any parameters from the request context (`?MyUId`). We still track the concrete value for each parameter in case it is relevant to compliance. Figure 3.4 shows the base template generated from the trace in Figure 3.2.

Given a base template, we can write down our generalization formula for unsat core extraction by asserting non-compliance in the style of §3.4.

Definition 3.5.1 (Generalization formula)**.** Given a base template consisting of a parameterized trace $\{(Q_i, O_i)\}_{i=1}^{n-1}$, a parameterized query Q, and parameter–value pairs $\{(p_i, v_i)\}_{i=1}^{k}$, the *generalization formula* is the conjunction of these clauses:

$$\begin{array}{lcr} [LQ_i] & Q_i(D_1) \supseteq O_i \wedge Q_i(D_2) \supseteq O_i, & (1 \le i \le n-1) \\ [LV_i] & p_i = v_i, & (1 \le i \le k) \\ & V(D_1) = V(D_2), & (\forall V \in \mathcal{V}) \\ & Q(D_1) \neq Q(D_2), & \end{array}$$

where D_1 and D_2 are a pair of databases. The clauses on the first two lines are labeled LQ_i and LV_i for easy reference.

We write $Q_i(D_1) \supseteq O_i$ instead of "=" so that the resulting template matches queries that return more rows than needed.

A main difference between this formula and the regular compliance formula (§3.4) is that the structure of the queries and returned rows (LQ_i) are now *decoupled* from the values in those queries and rows (LV_i). This allows the solver to include individual LQ_i's and LV_i's separately in the unsat core.

Returning to Figure 3.4, a solver invoked on its generalization formula can produce the unsat core $\{LQ_2\}$, indicating that (1) Query #1 is irrelevant, and (2) all concrete values are irrelevant as long as the equality relations between them hold. This unsat core corresponds to the template in Figure 3.3.

Trimming equalities. On some traces, the approach described above can *over-constrain* equalities between values, harming generalization. This happens when the same parameter is assigned to two unrelated values that happen to be equal (e.g., if the user ID and event ID are both 42), causing the template to also unnecessarily require this equality. To mitigate this problem, we adopt two optimizations.

First, we assign a fresh parameter to each returned cell whose column name does not appear in the `WHERE` clause of any policy view. Such columns do not affect compliance so long as they do not appear in the schema constraints either.[9]

Second, in our evaluation we configure the database to start the auto-increment ID for each table at a different offset (e.g., 10^6 apart). This prevents rows from different table from sharing the same primary key (at least initially, during cache population), thus avoiding equalities between these IDs.

To speed up template matching (§3.5), we also omit equalities between parameters whose value appears only in returned rows and not queries. In our evaluation we did not encounter policies that depend on such equalities.

[9] Our prototype does not consider schema constraints in this optimization (although support can easily be added). If this optimization fails, we fall back to the original.

The Decision Cache

After decision templates are generated, they are stored in the *decision cache*. When a new query arrives, the cache attempts to *match* the templates against the current query and trace and, if a match is found, declares the query compliant. In case of a cache miss, the query–trace pair is sent to the solvers and, if compliant, is generalized into a new template to be inserted into the cache. The decision cache use a hash map to index the templates by their (parameterized) query, and performs template matching through recursive backtracking.

In **??** we discuss some additional avenues for improving Privoxy's decision cache.

3.6 Implementation

We implemented Privoxy as a custom Java Database Connectivity (JDBC) driver, which wraps around an underlying database connection and intercepts queries and responses. Our prototype only supports applications that can run on the JVM, and run as a Java library within the web server, although our design does not constrain where the checker resides.

The checker parses and manipulates SQL queries using Apache Calcite [14] and consults the compliance decision cache. In case of a cache miss, it uses Z3's Java binding [104] to generate SMT formulas and serialize them in the SMT-LIB 2 format [12]. As an optimization, we pre-generate and serialize the common formula "preamble" stating that $\forall V \in \mathcal{V}$, $V(D_1) = V(D_2)$ (for compliance, Definition 3.3.7) or $V(D_1) \subseteq V(D_2)$ (for strong compliance, Definition 3.4.4), and then append query-specific assertions for each query. We provide additional information about our formulas in **??**.

The checker then invokes an ensemble of solver binaries in parallel. Our ensemble consists of Z3 [41] (v4.8.10) and CVC4 [13] (v1.8) using default configurations, and Vampire [84] (v4.5.1) using a configuration from its CASC portfolio that proved effective for us.[10] If a query is not compliant, or all solvers time out after 2 s, the checker conservatively rejects the query by raising a Java `SQLException`. For unsat-core generation, we only use CVC4, which we found to produce smaller cores than Z3; Vampire does not produce unsat-cores.

Our prototype does not check that basic queries return no duplicate rows (§3.4); instead, we manually verified that this is the case for the queries encountered in our evaluation.

3.7 Evaluation

We evaluate Privoxy by using it to enforce privacy policies on diaspora* [46], an open-source social network web application in Ruby on Rails with 12.7 k stars on GitHub.[11] We devised a privacy policy for diaspora* and made minor modifications to its source code (§3.7), and measured its performance running under Privoxy, reporting on page load times (§3.7), performance breakdown (§3.7), and decision template generalization (§3.7). Finally, we provide microbenchmarks on checking non-compliant queries (§3.7).

To highlight our evaluation results:

- Porting diaspora* to Privoxy required changing less than 10 lines of code (LoC) (§3.7).

[10] `https://github.com/vprover/vampire/blob/master/CASC/Schedules.cpp\#L281`.

[11] `https://github.com/diaspora/diaspora`.

- Assuming compliance decisions are cached, Privoxy incurs 7 %–11 % overhead to page load times (§3.7).
- Over 90 % of the decision templates produced generalize to similar requests; the rest generalize to restricted cases (§3.7).

Setup

Deployment. We host diaspora* (v0.7.14) on an AWS EC2 c5.4xlarge instance running Ubuntu 18.04. Because our JDBC-based prototype only supports applications running on the JVM (§3.6), we run diaspora* on JRuby [78] (v9.2.6.0), a Ruby interpreter on the JVM (we use OpenJDK 11). The application runs on the Puma web server in production mode over HTTPS behind NGINX (which serves static files directly), and stores data in MySQL running on the same EC2 instance.

Schema and policy. We model 36 out of 52 tables in the diaspora* database schema; tables not modeled include Rails's internal tables and those related to OAuth and cross-pod functionalities. Our schema model also includes 64 primary key and uniqueness constraints, 39 foreign key constraints, and 4 others (e.g., that a reshared post must be public). On top of this schema, we devised a privacy policy consisting of 108 view definitions, many of which can be auto-generated (e.g., those that grant admins full access to each table).

Application modifications. We modified diaspora* to communicate the current user ID to Privoxy at the start of each request (4 LoC) by issuing a special SQL `SET` command, and modified one raw SQL query in the code to quantify field names with table names due to a parsing limitation (1 LoC).

The application sometimes fetches information inaccessible to the user, although it does not display such information (§3.2). We modified the `Profile` model to fetch only the public fields of a person's profile, and to fetch potentially private fields only when needed (2 LoC using the `lazy_column` gem [95]). This change can sometimes incur an additional SQL query to fetch the non-public fields. We disabled background jobs, which are not modeled by our privacy policy as they are not always performed on behalf of a user.

Benchmarks. We use five page loads to exercise various aspects of application behavior:

1. **Simple post**: View a post, shared with the current user, with two likes, two comments, and one tag.
2. **Complex post**: View a public post with a poll where 30 users have voted and commented.
3. **Prohibited post**: Attempt to view a post that the user is not authorized to view.
4. **Conversation**: View a conversation with another user, which consists of five messages.
5. **Other's profile**: View another user's profile, with basic information and two posts from the user's stream.

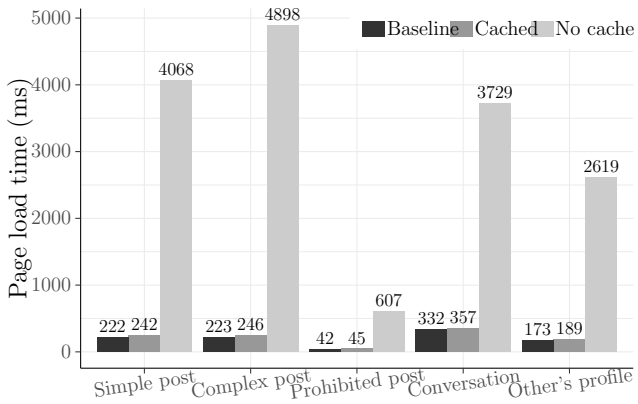


Figure 3.5: Page load times (PLT) for five pages. When compliance decisions are cached, Privoxy incurs 7 %–11 % overhead.

The queries issued by these page loads are all compliant to the privacy policy. We will use these benchmarks to study performance and cache generalization.

Page Load Times

We start by measuring the performance of loading the five benchmark pages. From a client VM (c5.4xlarge) in the same AWS region as the server, we measure the *page load time* (PLT) of each page by using the Selenium [134] Python binding to drive a headless Chrome browser to visit each page. The PLT is reported as the time elapsed between `navigationStart` and `loadEventEnd` as defined in the `PerformanceTiming` interface [152]. Note that these experiments measure a best-case baseline performance, as clients outside the region / cloud provider are likely to experience higher network latency.

For each benchmark page we report the PLT under three settings: "baseline" (without Privoxy), "cached" (with Privoxy enabled, every query hitting the decision cache), and "no cache" (with decision caching disabled). For each experiment, we warm up the OpenJDK HotSpot VM on the server by performing 100 loads first, and then measure the PLT using another 100 loads, reporting the median.

Figure 3.5 shows the PLT for each page under the three settings. When compliance decisions are cached, Privoxy incurs 7 %–11 % overhead resulting from parsing queries and decision cache template matching. When caching is disabled, Privoxy incurs an order of magnitude higher PLT as a consequence of invoking the solvers on every query.

Performance Breakdown and Analysis

To gain a better understanding of the performance, we provide more fine-grained data for each page load.

Fetch latency. Rendering each page potentially requires requesting multiple URLs, some in the background (e.g., loading comments for a post in diaspora*). The page load time, however, does not include the time taken to complete such asynchronous requests. We therefore map out the URLs fetched in each page load and measure the time taken to *fetch*

Table 3.1: Privoxy incurs up to 13.1 % overhead to fetch latency assuming compliance decisions are cached, and the decision templates it produces generalize well. A description of the table can be found in §§ 3.7 to 3.7.

		Fetch Latency (median)			Query Stats (Cold Cache)				Decision Templates	
Page Load	URL	Baseline	Cached	No Cache	Filtered	Miss	Hit	Total	Minimal	Generalizable
Simple post	`/posts/1`	184 ms	191 ms	4.0 s	7	21	1	29	16 / 21	18 / 21
	`/posts/1/comments`	48.8 ms	50.8 ms	814 ms	2	4	1	7	4 / 4	4 / 4
	`/notifications`	56.6 ms	61.1 ms	2.3 s	6	8	0	14	6 / 8	7 / 8
Complex post	`/posts/2`	180 ms	190 ms	4.8 s	4	26	3	33	19 / 26	23 / 26
	`/posts/2/comments`	220 ms	228 ms	27.8 s	3	6	29	38	6 / 6	6 / 6
Prohibited post	`/posts/3`	32.9 ms	33.1 ms	594 ms	1	4	0	5	4 / 4	4 / 4
Conversation	`/conversations?id=1`	285 ms	292 ms	3.6 s	6	18	0	24	7 / 18	18 / 18
Other's profile	`/people/30c2`	129 ms	136 ms	2.5 s	4	14	0	18	11 / 14	14 / 14
	`/people/30c2/stream`	65 ms	73.6 ms	3.5 s	4	11	7	22	10 / 11	9 / 11

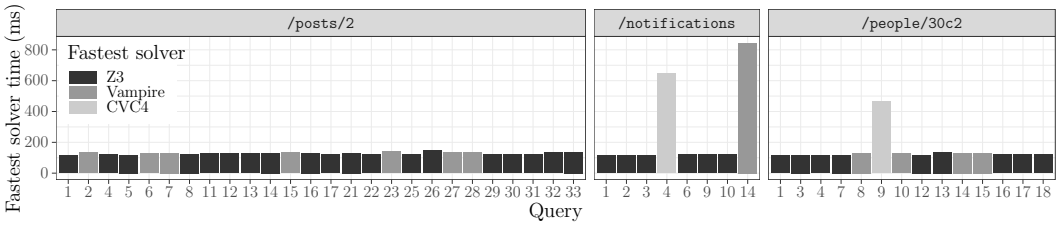


Figure 3.6: Shortest solver time for queries during cache miss, and the solver that achieved it. No solver is fastest on all queries.

each URL alone (i.e., not including the page rendering time in the browser); we call this the "fetch latency", and we measure it using Python's `requests` library.

Table 3.1 shows, for each page load, the fetch latency for each URL fetched.[12] At median, Privoxy incurs up to 13.1 % of overhead assuming compliance decisions are cached. If caching is disabled, the overhead ranges from 10×—125×.

The highest "no cache" overhead is incurred by the URL `/posts/2/comments`, which returns a list of 30 comments on the complex page. To serve this URL, diaspora* issues three queries to fetch all comments on the post as well as the `People` and `Profile` rows of each commenter (each query returns 30 rows). It then issues *30 additional queries* to fetch the "mentions" on each comment; the SMT formulas for each query contains the 90 rows fetched previously, making it costly to generate and serialize ($\sim$ 160 ms versus the usual 10 ms—15 ms),[13] and to solve (taking roughly 500 ms on Vampire). Fortunately, if decision

[12]The `/notifications` URL is fetched by all but the "prohibited post" page; we only show it once in the table.

[13]One way to speed up formula serialization is to aggressively reuse parts of a formula when generating the next one. We currently only reuse the "preamble" (§3.6), which includes the views but not the trace.

caching is enabled, much of this overhead disappears *even on a cold cache*—as the last 30 queries are identically structured, all but the first will hit the cache.

Query statistics. The "Query Stats" part of Table 3.1 shows query statistics for each URL when starting from a cold cache. The URL with the fewest queries (5) is for the "prohibited post" page, which performs permission checks on the requested post and immediately returns a 404. The page with the most queries (38) is `/posts/2/comments`, which is explained above. The "Filtered" column shows the number of queries handled by column-based filtering (§3.4), ranging from 7 % to 42 % of total queries. The "Hit" column shows the number of cache hits. Since these experiments start from a cold cache, the only possible cache hits are those on previous queries in the same request. We previously explained the case of `/posts/2/comments` (29/38 hits). As another example, `/people/30c2/stream` issues two structurally identical sequences of queries to fetch two posts in the stream; the second sequence of queries all experience cache hits (7/22 hits).

Solver running times. During a cache miss (or if caching is disabled), Privoxy invokes an ensemble of solvers to verify query compliance. Figure 3.6 shows, for three URLs, the running time of the fastest solver for each query. While most queries can be handled within 200 ms, some take up to over 800 ms. We also observe that no one solver is fastest on all queries; this is why Privoxy invokes three solvers in parallel and waits for the fastest to finish.

Template Generalization

With caching enabled, every cache miss places a new decision template into the cache (§3.5). We manually inspected every template generated by the URLs and counted (1) how many are derived from a minimal unsat-core, and (2) how many generalize to a similar URL (e.g., another user viewing a similar post); these numbers are reported under "Decision Templates" in Table 3.1. For five out of the nine URLs, 100 % of the templates generalize; the rest achieve a generalization rate of at least 80 %. Taken together, over 90 % of the templates generalize fully.

Even templates that do not generalize fully do apply in restricted scenarios (exemplified below); no template is specific to a user ID, post ID, etc.

Solver Performance: Non-compliant Queries

Recall that Privoxy uses size-bounded formulas to find small models for non-compliant queries (§3.4). We evaluate such formulas' performance on Z3 on three hypothetical non-compliant queries in the setting of diaspora*:

1. Viewing post: Return a special error if a post doesn't exist (versus exists but unauthorized to view).

2. Viewing profile: Always return extended profile regardless of the person's privacy setting.

3. Viewing profile: Show the number of posts by the person. (This query can leak information on private posts not shared with the viewer.)

For all three cases, Z3 returns "satisfiable" on the size-bounded formula within 600 ms, while the regular formulas time out after one minute on all solvers.

3.8 Related Work

View-based access control. View-based access control has been studied extensively in the database community [**rosenthal01:perm**, 16, 15, 103, 123, 124]. Privoxy differs from these works in two main ways:

- While some existing works focus on *single-query* checking [16, 15], we check a *trace* of queries (§3.3), which is crucial for supporting web applications.
- While some mechanisms (e.g., `GRANT SELECT ON` views in SQL) require queries to explicitly use view names (e.g., V_1, V_2), we support queries issued directly against the "base tables" (e.g., *Users*, *Events*). This enables us to support existing applications, which have no knowledge of the policy views.

Information flow control (IFC). The property that the application leaks no secret information under policy $\mathcal{V}$ coincides with the *noninterference* property in IFC [38, 55] if we define the low equivalence relation as database equivalence under $\mathcal{V}$. It can be shown that if an application request handler performing database reads only issues compliant queries (Definition 3.3.7), then it satisfies noninterference.

Query rewriting. Query rewriting [**stonebraker74:acl**] is a dynamic disclosure control mechanism: If a query returns secret values, a run-time enforcer replaces them with placeholders (or drops the rows altogether). Such mechanism is implemented in commercial databases [21, 99], as well as in academic works such as Jacqueline [154] and Qapla [**mehta17:qapla**]. While query rewriting allows the programmer to issue queries freely without regard to any privacy policy, it can alter query semantics, which may give rise to unexpected behaviors at run time [150, 123].

Static enforcement. Alternatively to dynamic checking, several systems have been proposed to *statically* verify policy compliance, e.g., SeLINQ [127], SIF [32], Swift [33], UrFlow [30], Hails [51, 52], LWeb [112], and Lifty [114]. These static systems incur no run-time overhead and can be more precise than Privoxy as they analyze the source code; some can even extend to client code. However, they typically require using a specialized language or framework like Jif [105] or Ur/Web [31], making them less compatible with existing applications.

Alternative compliance formulation. Another query compliance formulation is *instance-based determinacy* [123, 62, 83, 157], according to which a query is compliant if the information it reveals is contained in the views *given all accessible information in the database*. (In contrast, our formulation assumes only information retrieved by previous queries in the request.) Although instance-based determinacy would allow us to safely accept more queries (e.g., Q_1' from Example 3.3.5 if the current user is an attendee), checking instance-based determinacy requires information outside what is being queried by the application and can be costly (e.g., a *data complexity* of co-NP complete for conjunctive queries [83]).

3.9 Conclusion and Future Directions

Privoxy is a privacy enforcement system that supports legacy web applications and incurs low overhead. While we have demonstrated the feasibility of our approach, here we outline two future directions.

Specialized decision procedures. While SMT solvers are flexible and versatile, they may not be the most performant tool for verifying query compliance. One alternative is the chase procedure [5, 94], which has been used to study query determinacy [54, 106] and can be naturally extended to compliance. We hypothesize that for the class of views and queries encountered in the privacy use case, compliance holds exactly when the chase terminates in a small number of rounds. We defer refining and proving this hypothesis, as well as experimenting with various chase implementations [17], to future work.

Checking updates. We plan to extend Privoxy to verify privacy compliance for database updates. We envision this extension taking two steps. First, we check updates against separate "update" policies, which can differ from the "read" policies as as some information should be accessible but not modifiable. We expect the update policies to be much easier to specify and check, since SQL `INSERT INTO`, `UPDATE`, and `DELETE` statements are simpler than `SELECT`. Second, we ensure that the updates maintain schema constraints, which are assumed to hold by Privoxy's checking of `SELECT`. Some constraints (like primary keys) are maintained automatically by the database, while others we have to check in Privoxy.

Chapter 4

How to Train your DNN: The Network Operator Edition

4.1 Introduction

Deep Neural Networks have gained significant traction in both academia and industry. One type of deep neural network – feedforward convolutional neural networks (CNNs) – have been the driving thrust behind critical applications such as image recognition [133, 70], drug discovery [148], and medical diagnosis [148, 139, 90, 80]. The accuracy of CNNs often requires frequent retraining, so it is important to reduce CNN training time. Efforts to do so have targeted nearly all layers of the software and hardware stack, and increasingly involves distributing training across machines in a cluster. Early work on distributed CNN training adopted the *parameter server model* [88] where computation is performed by several worker nodes and one or more *parameter servers* are used to aggregate and distribute results from individual workers. Recent work has also looked at a variety of topics, such as improving the performance of distributed CNN training through the use of better scheduling [69], and improving network transfers [147]. In this paper we focus on optimizations that involve the cluster network. A variety of network oriented solutions have been proposed, but with little resulting clarity about which proposal (or combination of proposals) achieves the best end-to-end performance. This paper seeks to answer this question.

We first observe that one can divide network optimizations into two broad categories: those that change the network fabric and those that only change software at end hosts. This is a useful distinction because changing the network fabric is typically more difficult (involving router software and perhaps hardware), while host software changes are significantly easier to implement. The first fabric-based change we consider is using IP multicast (a feature that is supported by most routers but often not enabled). IP multicast has been used previously to accelerate HPC workloads using MPI [26, 155, 73], which in turn has been used as a communication primitive by a number of distributed CNN frameworks [147, 9]. The other class of fabric-based optimization is in-network aggregation, as can be implemented using programmable switches. The use of in-network aggregation for CNNs has already been proposed in Daiet [125] and Luo [92], and here our goal is to understand its performance impact relative to other network optimizations.

The host-based techniques we consider move away from the parameter server model. These include ring-reduce [10] and all-reduce (*e.g.,* Rabenseifner [116] or butterfly mixing [25]) which avoid the use of a centralized server for aggregation and have been successfully used to speed up HPC jobs in the past. We provide a more detailed list of such approaches in §4.2.

We analyze these various approaches – in isolation and in combination where possible – to answer four questions. First, how do these various optimizations rank in terms of effectiveness? Second, given this ranking, is it necessary for us to resort to fabric-based mechanisms, or do host-based mechanisms suffice? Third, how robust are these results to possible future changes in CNNs (*e.g.,* more layers)? Fourth, how robust are these results to possible future changes in host behavior (*e.g.,* changes in TensorFlow)?

We rely on trace driven simulation to address these questions. The use of simulations allowed us great latitude in testing a variety of proposals including ones which require changes to the network hardware and were hence infeasible for us to test in practice. The use of a simulation also allowed us to avoid certain approximations and non-determinism that would have been difficult to consider in a tractable analytical model. Thus, simulation provided us with a good balance between the accuracy of our results and the ability to try out a wide range of optimizations. To further ensure realism for our results we seed our simulations with traces generated from training CNNs on real hardware using distributed TensorFlow. We describe our techniques for generating traces and the actual design of our simulator in greater detail in §4.4.

We ran our simulator on four image recognition models (described in greater detail in §4.6), and we present evaluation results from these runs later in the paper. At a high level we found that in the typical case using fabric-based mechanisms to speed up training in the parameter server model has lower benefits than using host-based mechanisms that abandon the parameter server model in favor of other reduce strategies. We found that this held even when we combined both fabric-based mechanisms. Our basic conclusion is that optimizing communication for CNN training does not necessitate changes to the network fabric.

4.2 Background

In this section, we begin by discussing the computational model for CNN training. Following this, we provide an overview of communication paradigms for distributed CNN training. Overall, we provide overviews of the following mechanisms that we explore in this paper: in-network aggregation, IP multicast, ring-reduce, ring-reduce with multicast, and butterfly mixing. Next, we discuss how changes to the network fabric could be used in conjunction with the aforementioned communication paradigms – and thus further accelerate training.

Distributed Training Steps

We now consider the computational process: how the model computation (forward pass and back propagation) interleave with the communication. We consider data parallel approaches to distributed learning, where each worker operates on the entire model but uses different training data. Data parallel learning is the most prevalent approach today.

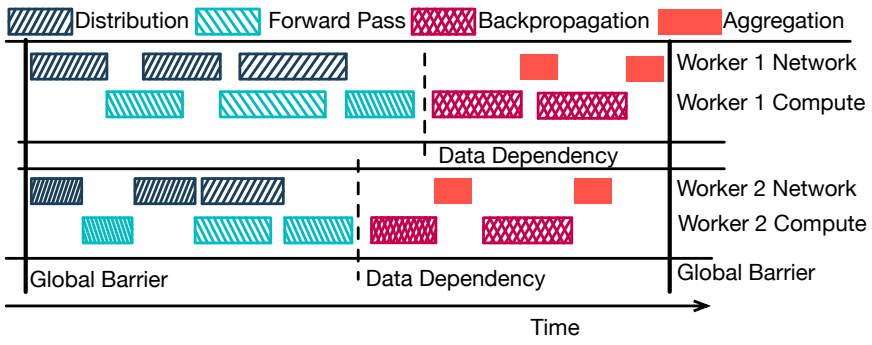


Figure 4.1: Steps in a distributed training job with param. server

Steps

Distributed CNN training proceeds in four steps:

Distribution: When using parameter servers, the parameter server updates to the worker constitutes the distribution phase. For butterfly mixing or ring-reduce, this would be the last communication phase, right before all workers receive the same updated model.

Forward Pass: Each worker selects a sample of the training data and uses the parameters to compute labels for this training sample. This computation is commonly referred to as the *forward pass* since each layer operates on the training image in order. The training data used in this step is loaded concurrently during the distribution step and is not on the critical path. As a result, our analysis does not consider time taken loading training data.

Backpropagation: Next, each worker uses the supplied labels (from the training set) and computed labels to determine each layer's contribution to training error, and then computes an appropriate change to the layer. This computation is commonly referred to as *backpropagation* and proceeds from the last layer in the neural net to the first one. Backpropagation utilizes results computed during the forward pass and, as a result of this data dependency, it cannot progress until the worker has finished the forward pass.

Aggregation: As the backpropagation progresses, the worker will send updates to a reducer. In the parameter server model, parameter servers are responsible for both aggregating and applying updates. An iteration is considered to have completed only when the parameter servers have received updates from *all* workers, thus enforcing a global barrier (across all workers) between the distribution and aggregation step.

Within the parameter server model, each iteration of the algorithm consists of the four steps (shown in Figure 4.1). These four steps can be partially pipelined, in the following two ways. First, the forward pass proceeds layer-by-layer, and as a result a worker can begin the forward pass step as soon as it has received the parameters for the first layer of the CNN. Second, backpropagation also proceeds layer-by-layer, and workers can begin sending updates as soon as they have computed updates for a layer. To the best of our knowledge, all commonly used CNN frameworks employ pipelining to improve training performance.

For end-host mechanisms (i.e., butterfly mixing and ring-reduce), the forward pass is not

pipelined with the distribution phase. The back-propagation pipelining operates the same way.

Algorithms for Efficient Reduction

Training on Parameter Servers

One way to facilitate distributed CNN training makes use of one or more centralized parameter servers [88]. These algorithms are implemented as a part of several CNN frameworks including TensorFlow [47], Caffe2 [58], and MXNet [28].

For parameter server based training, there can be two distinct, communication phases during each training iteration: (i) a distribution phase where CNN parameters are distributed to workers, which then execute a local training algorithm using these parameters, and (ii) an aggregation phase where each worker sends the local training algorithm's updates to one or more parameter servers.

The distributed CNN training algorithms we consider are iterative. Distributed training algorithms can be further classified into synchronous and asynchronous algorithms. *Synchronous* training algorithms require that all workers agree on the model at the beginning of a training iteration; this is implemented by having the parameter server impose a barrier across workers. *Asynchronous* training algorithms do not impose consistency requirements across workers. While asynchronous training algorithms decrease the time taken by each iteration, they increase the total number of iterations required and can thus slow down overall training time [27]. Some companies like Google tend to favor synchronous training algorithms [2]. In this paper we focus on synchronous training algorithms because the presence of synchronization barriers allows us to more easily reason about iteration time.

Training without Parameter Servers

Parameter servers present several issues to practitioners, which begin with selecting the correct ratio between the number of parameter servers and workers. Rather than aggregating parameter updates in centralized nodes, others have advocated efficient all-reduce algorithms that exchange parameters merely between worker nodes. In the context of training CNNs, two algorithms are most commonly discussed: ring-reduce [74] and butterfly mixing [25].

Ring-reduce, popularized by Uber's implementation (Horovod [74]), requires that the workers connect in a ring. There are two communication phases. First, parameters in the model are assigned to each worker in a round robin fashion. In the first phase (analogous the aggregation phase), each worker begins computing gradient updates; when they complete the computation for the parameter assigned to it, it immediately sends the parameter to the next node in the ring. Upon receiving the update, each worker incrementally averages the update with it's locally calculated gradient. Once the last worker in the ring has received the parameter, it has the complete averaged model parameter from the entire cluster. In the second phase, this exact updated model is passed around the ring a second time such that all workers now possess this updated model.

On the other hand, butterfly mixing performance scales logarithmically with the number of workers. At each phase within butterfly mixing, each worker simultaneously sends the

entirety of its model to one other worker. After receiving an update, the worker averages its local model with the received model. Suppose there are four workers: A, B, C, D. In the first phase of communication, A and B simultaneously exchange their entire model, while C and D follow suit. At this point, A and B contain the same averaged model, as does C and D. In the second phase of communication, A sends its averaged model to C (and vice versa), while B sends its averaged model to D. This concludes the all-reduce. Butterfly mixing reduces the number of communication phases required during updates at the cost of sending a larger amount of data over the network fabric.

Mechanisms to Accelerate Training

The network fabric is capable of providing support for distributed training at multiple levels. Here, we discuss two proposals and how they can used in conjunction with the all reduce algorithms presented earlier.

IP Multicast

Multicast [43] implementations are designed to be bandwidth efficient. For CNN training, multicast ensures that the amount of data traversing any network link does not scale with worker count. In practice, deploying multicast requires addressing a litany of considerations (e.g., membership, discovery, reliability). However, we do not address them in this paper, noting only that CNN training tasks involve bulk transfers (making reliability easier to solve since one has time to identify and recover from errors, and there are various reliable multicast implementations available) and tend to run over several hours with a constant set of workers, simplifying the management problems.

When using a parameter server, IP multicast assists solely with the distribution phase. During the distribution phase each parameter server sends each worker all of its parameters – w copies of the same data when deployed in a cluster with w workers. Enabling IP multicast would allow each parameter server to send a single copy of its parameters. It is also possible to use IP multicast with ring-reduce during the second ring during it's model distribution.

In-network Aggregation

In-network aggregations improve training time when using parameter servers. The data sent during the aggregation phase varies by worker. To reduce traffic in this phase, the network needs to implement the model's aggregation semantics. This is possible either through the use of software switches operating on an overlay network [93] or through the use of programmable switch ASICs [125] such as Barefoot Tofino [11]. In this approach the network buffers worker updates and aggregates them before sending them to parameter servers. The benefits of this approach mirror those achieved through multicast; this mechanism also presents several deployment challenges including requiring the use of new, specialized switching hardware, allocation of compute and memory resources on switches, and mechanisms for isolating CNN training traffic. We also do not address these issues in the paper, and refer the interested reader to recent discussions on this topic [125].

4.3 Analytical Models Insufficient

Our initial attempts to derive an analytical model for training performance provided *intuition* about mechanism scaling, but failed to identify the reasons about the relative performance of the various mechanisms. In particular, it fails to capture two specific components of CNN training that significantly affect performance.

First, the backpropagation phase consists of fine-grained, causal, interleavings between compute and network. This set of interleavings is highly *model specific*. Each model has *hundreds* of composite operations that are interspersed over highly uneven units of time. Moreover, radical new CNN proposals appear frequently; thus developing a novel analytical model for each new CNN design is highly burdensome. Rather than attempt to model this complex interaction, our simulator uses a comprehensive set of empirical traces that are easy to collect.

Secondly, different phases of computation (e.g., distribution and aggregation) overlap non-deterministically. Recall that while there is a *global barrier* at the parameter server, each worker unit has a *local barrier* before it initiates backpropagation. If we assume 1.) every local barrier is reached simultaneously and 2.) there is no variance in the compute part of backpropagation, this would – within the constraints of our computational model – maximize the amount of incast on the parameter server. On the other hand, any delta between workers hitting the local barrier would reduce each worker's overlap between the backpropagation phases; this delay reduces incast. We refer to this delta as **backpropagation staggering**. Backpropagation staggering is influenced by two factors that are extremely difficult to model analytically. First, it requires reasoning across both the overlapping distribution and aggregation phases. The mechanism used to distribute parameters, as well as the qualities of the individual parameters themselves (i.e., how long it takes to compute and send over the network) affect the amount of backpropagation staggering. Second, there is natural variation in worker processing time that further influences the staggering. This is not limited to the parameter server framework; in the case of end-host mechanisms like butterfly mixing, pipelined parameter mixing between workers can interfere with each other. In Section 4.7, we will show in much greater detail how the amount of backpropagation staggering influences how the mechanism and model performs. Thus, an analytic model which fails to capture this phenomenon will not be sufficient to answer the questions we posed.

4.4 Trace-driven Simulator Design

We seek to develop a simulator that captures the performance nuances described in Section 4.3, while simultaneously being sufficiently flexible to accommodate a limited scope of potential changes in feedforward CNN models and Tensorflow. Thus, we moved towards developing a trace driven simulator which we describe in greater detail in this section.

To ensure that the trace collection is simple and effective, a trace must have two attributes. First, it must be network agnostic, so the same trace can be used for different network settings. Second, it must accurately model pipelining. The traces were generated

by adding minor instrumentation to TensorFlow 1.4. Computation in TensorFlow is represented as a dataflow graph, where individual *operations* are represented as nodes on the graph, while parameters (i.e., tensor) are transferred along the edges of the graph. Our instrumentation records *send* operations along the dataflow graphs. The trace is very simple. For each operation, it shows five attributes. 1.) event time 2.) parameter name – identified by an edge name (e.g., *conv1/weights/read* tells us that the first convolution operation is being read by the worker) 3.) size of the parameter ready to be queued on to the network. 4.) source device (e.g., worker0) 5.) destination device (e.g., parameter server). This trace is simple to generate, parse, and possibly even modify if the operator wants to run simulations on synthetic models.

For each neural net model, we collected a different representative trace. On each iteration, we partitioned the events into two traces: an aggregation trace and a distribution trace.

For the aggregation trace, we identified specific send operations that are *gradient* operations triggered automatically by TensorFlow's SyncReplicaOptimizer. The aggregation trace allows us to accurately model how the parameters are sent over the network. Recall from the computational model that due to extensive pipelining, the calculated gradients are delivered to the parameter server as soon as the parameter gradient is calculated. In the gathering of this data, note that not every operation from the worker to the parameter server falls on the critical path. The beginning of the aggregation is marked by a *dependency operation* that varies from model to model. We only consider the send operations that occur following the dependency operation, since all other send messages prior to that dependency operation are pipelined with the worker's forward pass; thus, they do not fall on the critical path. For example, it is common for CNN training to use *batch normalization*, which normalizes the inputs to each layer. The worker will send the result of its batch normalization computation to the parameter server so the PS can compute moving averages over multiple iterations. Such operations must be removed from the aggregation trace. There are other operations that must be filtered out that arise occasionally on a per-model basis (such as the use of *auxiliary logits*).

For the distribution trace, we used the TensorFlow timeline tool to identify the particular send operations originating from the parameter server that triggered forward pass operations on the worker. The purpose of the distribution trace is to obtain the *order* in which parameters are queued on the network. We demonstrate in the evaluation (§4.7) how the ordering of parameters affects the end-to-end performance of distributed training. Additionally, we profiled the forward pass time on a single GPU. We later use this for emulating the pipeline effects in the forward pass, which we find plays an insignificant role in typical use cases; this is due to the one-to-many communication overhead arising from the parameter server.

To simulate normal performance computational performance variance across GPUs, we recorded traces for each worker in clusters of difference sizes and across several distinct clusters. To ensure that the trace is agnostic to the network and size of the cluster, the recorded time of a trace event is relative to the first event in that trace. Different workers in a cluster finish receiving the model at different times due to network topology and parameter ordering. The absolute times in the aggregation trace are affected by the network conditions,

CNN Name	1 PS	2 PS	4 PS	8 PS
VGG-16 Sim	21.0	22.5	19.3	18.2
VGG-16 Real	22.5	22.8	20.8	19.3
Inception-v3 Sim	2.29	2.29	1.37	0.852
Inception-v3 Real	2.16	2.16	1.49	1.3
Resnet-200 Sim	7.15	3.34	2.3	2.29
Resnet-200 Real	5.89	2.3	1.71	1.71
Resnet-101 Sim	4.57	2.37	1.52	1.5
Resnet-101 Real	3.7	1.58	0.855	0.9

Table 4.1: Comparison of measured (Real) iteration time compared to simulation prediction times (sim) on a cluster of 8 workers

so aggregation times are recorded relative to the first aggregation event. Thus, we are able to simulate the network effects across a wide variety of cluster sizes.

4.5 Simulator Validation

We validated our simulator by comparing the results of the simulator against actual runs. In a cluster of 8 workers, our results are shown in Table 4.1. For the most part, our simulation results accurately predict the performance trend with more parameter servers. In many of these cases that use multiple parameter servers, the CNN weights are not evenly distributed among the parameter servers, causing the performance improvements to plateau. Our simulation effectively reflects this behavior. There are two notable points at the far ends of the spectrum – Inception-v3 with 8 PSs, and Resnet-200 with 1 PS – where our simulation fails to match our empirical measurement, although both still capture the general scaling trend. There are several possible explanations for this. First, our simulation makes an assumption that parameters in the distribution phase occurs in a round-robin fashion over the workers. While this assumption is sufficient for most of the settings that we looked at, our observations of the actual distribution send-traces reveal that there are some minor overlaps in the way that worker parameters are being sent. This suggests that the round robin assumption is slightly stronger than reality – especially in the case of 1 PS where the amount of backpropagation staggering is most pronounced.

4.6 Model Characterization

Which *CNN model characteristics* influence a mechanism's performance benefit? We've identified four such model attributes that impact the extent to which acceleration mechanisms will benefit performance and show where our CNN models fall on those dimensions. Other, non-deterministic factors not related to the CNN characteristics may influence training performances, but are discussed earlier in Section 4.3. With the exception of the forward pass, these attributes are all extracted directly from the trace. We discuss this in the context of the four distinct, commonly deployed image classification models we deployed and analyzed:

CNN Name	# Layers	# Weights	Model Size (Gb)	# of FLOPs
Inception-v3	21	2.5×10^6	0.715	1.1×10^{10}
VGG-16	22	1.9×10^8	6.58	5.4×10^{10}
Resnet-101	103	2.2×10^6	1.42	1.3×10^{10}
Resnet-200	202	2.2×10^6	2.06	2.8×10^{10}

Table 4.2: Complexity of the CNN models we considered, including both weight and pooling layers in our layer count.

CNN	Fwd Pass Comp	Bkprop Comp	Bkprop Net, 25 Gbps	Comp:Net Ratio
Inception-v3	0.176 sec	0.296 sec	0.028 sec	10.6
VGG-16	0.169 sec	0.024 sec	0.263 sec	0.09
Resnet-101	0.176 sec	0.180 sec	0.052 sec	3.46
Resnet-200	0.357 sec	0.34 sec	0.082 sec	4.14

Table 4.3: Compute and network times during the backpropagation of the model. Note that the backprop compute time does *not* include the time to calculate the first layer of backpropagation.

Inception-v3, Resnet-200, Resnet-101, and VGG16. They were trained using ImageNet data.
Distribution of parameter sizes over model, in particular the size of the last parameter. Many models exhibit a very parameter heavy fully connected last layer, which represents a significant fraction of the model size. Inception-v3 and VGG16 both possess very memory expensive fully connected layers, while Resnet-200 and Resnet-101 are relatively even throughout.
Computation/network bottleneck *after* the first layer of back-propagation. For all future references to backpropagation compute/network ratio, the first layer of backpropagation compute is not included. Once the first layer of backpropagation has been calculated, how interspersed are the computational and network elements of a CNN model? Table 4.3 shows the amount of time spent in communication and computation, and a *compute:net* ratio. VGG16 spends nearly all it's computational time calculating the first back propagation parameter, thus exhibiting the smallest compute:network ratio. On the other hand, Inception-v3, a model which is similarly skewed, is compute intensive even after the first layer of back propagation is computed.
Forward Pass Time: See Table 4.3
Raw size of the model. We show model sizes in Table 4.2. They range from very large (6.58Gb) to small (0.7Gb)
As we will show in Section 4.7, the first two characteristics listed above heavily influence the amount of backpropagation staggering, as they both affect the overlapping distribution and

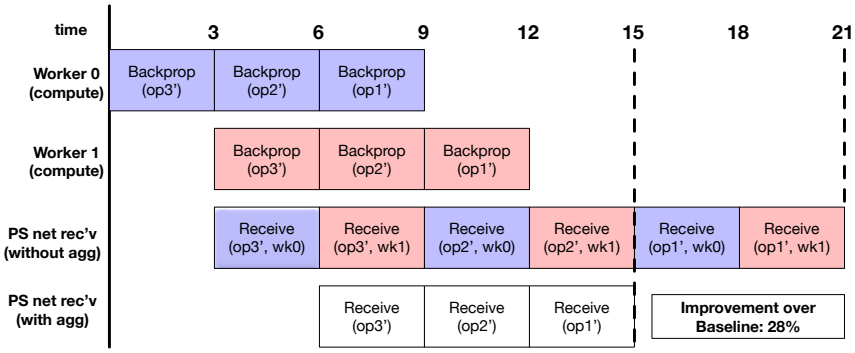


Figure 4.2: Aggregation phase of **computationally/communication even**, 3 layer (op) toy model, but with staggered backpropagation start times. PS = parameter server

Model Name	Aggregation Only	Multicast Only	Multicast + Aggregation
Inception-v3	1.34x	1.69x	3.28x
VGG-16	1.89x	1.94x	22.0x
Resnet-101	1.65x	1.79x	6.07x
Resnet-200	1.65x	1.85x	6.7x

Table 4.4: Factor speedup of network support models relative to a baseline model with no network support. 32 workers, 25 Gbps

aggregation phases.

4.7 Evaluation

In this section, we evaluate the efficacy of the mechanisms described in previous sections, and explore how the CNN model characteristics interact with the mechanism. Finally, we explain how they jointly impact performance. We also present head to head comparisons of competing mechanisms. Finally, we show that our rankings and intuitions generalize to two types of future training conditions: 1. larger models and 2. faster processors.

The traces used in the simulations were derived from clusters in AWS EC2 running Tensorflow 1.4, with a fixed batch size of 32 training instances per worker unit.

Our simulations show that in all cases, an end-host mechanism (ring-reduce) offers performance improvements greater than or equal to any in-network mechanism. Thus, rather than having to make a trade-off between performance and infrastructure cost, operators can instead deploy software mechanisms running on the end-host and expect to get equal or better performance.

In-Network Optimization

We look at network fabric optimizations – in-network aggregation and multicast – which primarily are used to accelerate training *when using a parameter server.*.

In-network Aggregation

Factor 1: Large Last layer of CNN reduces impact: In section 4.3, we defined backpropagation staggering. Recall that increased backpropagation staggering reduces incast, which consequently reduces the impact of in-network aggregation. The amount of backpropagation staggering is positively correlated with the time taken to execute the the penultimate layer(s) of the CNN. During the distribution phase, parameters are distributed between workers in a round robin manner. Thus, the forward pass on each worker is blocked until that single parameter (which can be over 5 Gb in the case of VGG16) reaches the worker in its entirety. Larger final parameter(s) causes each worker node to begin it's individual backpropagation at increasingly staggered times. To illustrate this, we consider a simple example with a 3 operation CNN model. Each operation takes three seconds to compute and three seconds to send over the network. In the case where all the workers start backpropagation simultaneously (not shown), aggregating the new model takes 21 seconds. When using in-network aggregation, aggregation takes 12 seconds (a 43% improvement). In contrast, Figure 4.2 shows aggregation in the same setup when backpropagation start-time is staggered between workers. While performance *without* in-network aggregation stays constant (21 seconds), in-network aggregation only improves performance by 28%. Why? In-network agg reduces parameter server network bottleneck by aggregating parameter updates from *all workers*. This aggregation can't proceed until the *last worker* has communicated that parameter update.

Both VGG16 and Inception-v3 have large last layers that will further stagger the time at which each worker begins backpropagation, making them particularly susceptible to this effect.

Factor 2: Network dominated backpropagation time ***increases*** *impact*: Recall that during the back-propagation process, parameters updates are sent as soon as they are calculated. If the calculation time between parameter updates is relatively short, there will be fewer parameter updates queued inside the network to be sent to the parameter server. Assessing the degree to which the backpropagation is network-bound cannot be evaluated solely by the overall time of backpropagation. VGG16 has the longest *overall* backpropagation time, yet enjoys the largest percentage improvement in performance from in-network aggregation. For VGG16, the majority of the backpropagation computation is spent on computing the *first backpropagation layer*, an expensive fully connected layer. Once this first computational step completes, the remaining model parameters are quickly calculated and queued in the network. Conversely, Inception-v3 spends a significant amount of time doing backpropagation computation even after the first backprop layer is computed.

Results: For 32 workers and 25 Gbps network, Table 4.4 demonstrates the performance improvements derived from using just in-network aggregation to accelerate distributed training. Inception-v3 (which has a compute-intensive backpropagation) experiences the least performance gain from in-network aggregation, while VGG16 (which has a network-intensive backpropagation) experiences the most.

Multicast

Factor 1 – Model Size increases impact Unlike in-network aggregation, the distribution phase is initiated with a global barrier. All the parameters are ready simultaneously, so there is no pipelining on the send-side as in the case of in-network aggregation. Thus performance gains from multicast are more directly a function of model size.
Factor 2 – Forward Pass Pipelining decreases impact (slightly) As the worker receives model parameters, it partially executes the forward pass. However, the forward pass is unlikely to be the bottleneck in the distribution phase, especially as the number of workers grows. Recall that the simulated parameter server distributes parameters to the workers in a round robin fashion. More workers will allow for more computation time between parameter distributions. Consequently, forward pass pipelining has a very slight (if any) impact on multicast performance.
Non-Factor – decreased backpropagation staggering: When using multicast, workers receivers parameters at roughly the same time (within minimal link latency). Thus backpropagation staggering is decreased, as we expect workers to initiate backpropagation simultaneously (approximately). Does the resultant increased incast hurt iteration performance? We find this not to be the case unless the following condition is met. Let D be the delay between worker backpropagation start times, B be the *full backpropagation time* (i.e., including both the computation and network transfer time), and C be the compute time for the backpropagation time of the first layer. In order for the decreased backpropagation staggering to give back performance, the following must hold: $D > B - C$. This is highly unlikely in the multicast case; multicast should beget a very minimal D.

Results: Again, we refer to Table 4.4. Multicast impact is more directly proportional to the size of the model. Resnet-101 (1.78x) and Resnet-200 (1.85x) are both larger than Inception-v3 and smaller than VGG16. Their performance gains likewise fall between the performance gains of those models.

Head to Head: In-network Aggregation vs. Multicast

As shown in Table 4.4, multicast outperforms or approximates the performance gains from using in-network aggregation in all cases. Later, we will show that both approaches are weaker than end-host based acceleration mechanisms, but we briefly provide intuitions for why multicast is more effective than in-network aggregation. Fundamentally, in-network aggregation is tied to backpropagation and multicast is tied to the forward pass. Generally speaking, the forward pass is significantly faster than the backpropagation; refer to Table 4.5. Moreover, the extent to which the forward pass is bottlenecked on compute decreases with more workers. On the other hand, the compute:network ratio of backpropagation remains constant with the amount of workers. In fact, increasing workers only leads to staggered backpropagation start times, which reduces the benefits of in-network aggregation. **Multicast individually is more impactful than in-network aggregation alone.** Table 4.4 indicates that multicast provides larger performance gains than in-network aggregation

CNN Name	GPU Model	Forward Pass	Backprop
Resnet-200	Maxwell Titan X	170 ms	384 ms
Resnet-200	Pascal Titan X	315 ms	520 ms
VGG-16	Maxwell Titan X	173 ms	416 ms
VGG-16	Pascal Titan X	98.2 ms	260 ms
Resnet-101	Maxwell Titan X	109 ms	190 ms
Resnet-101	Pascal Titan X	162 ms	258 ms
Inception-v1	Maxwell Titan X	91.3 ms	141 ms
Inception-v1	Pascal Titan X	57.5 ms	85.9 ms

Table 4.5: Forward Pass vs. Backpropagation Benchmarks [36]

across the board.

Multicast *plus* in-network aggregation

Finally, what happens when multicast is *combined* with in-network aggregation? Because multicast supports the distribution phase and in-network aggregation supports the aggregation phase, both approaches can be simultaneously used to improve performance. In fact, multicast *puts in-network aggregation in the best position to succeed* because it strongly decreases backpropagation staggering. Table 4.4 shows clearly that using multicast with in-network aggregation yields substantially better performance gains than using either just multicast or just in-network aggregation.

Results: Multicast plus in-network aggregation benefits VGG16 the most, as the performance gains increase from 1.9x to 21.2x. For all models, using both multicast with in-network aggregation results in *more than additive* performance gains from the individual mechanisms, for reasons described in the previous paragraph.

Summary of In-network Changes

Based on just looking at in-network mechanisms, we find that optimizations can be ranked as: multicast + aggregation, multicast, aggregation. This leads us to conclude that if one is required to use the parameter server model, then using multicast jointly with in-network aggregation yields the largest performance improvements. While both multicast and in-network aggregation offer their own set of deployment challenges, multicast outperforms in-network aggregation in the CNNs we tested. As evidenced by the CNN characteristics that factored into each mechanism's impact, the interaction between the distribution phase and aggregation phase *across all workers* is key. While the aggregation phase holds more complexity in terms of network/compute interleavings, it is actually the distribution phase optimizations which dictate how much the aggregation phase sits on the critical path. Future work should not solely evaluate the efficacy of accelerating either the distribution phase or aggregation phase in isolation.

Model Name	Ring-Reduce	Ring-Reduce + Multicast	Butterfly Mixing
VGG-16	24.6x	24.6x	11.3x
Resnet-200	6.75x	6.76x	6.79x
Resnet-101	6.55x	6.71x	6.46x
Inception-v3	3.35x	3.41x	3.41x

Table 4.6: Ring-reduce is most effective when parameters are evenly distributed, while butterfly mixing performs well at 25 Gbps.

End-Host Mechanisms

We analyze two all-reduce algorithms: Horovod ring-reduce and butterfly mixing. Table 6 shows speedup over baseline for both algorithms when run with 32 workers at 25 Gbps links. The table also shows improvements when ring-reduce is combined with multicast.

Ring-reduce

Recall that in ring-reduce, the parameters are assigned to workers round robin. One critical issue that must be addressed when doing this is that often a significant fraction of a model's raw size comes from a single parameter (e.g., VGG16, Inception-v3). Analytically, the communication overhead of ring-reduce is $2(W-1) * (max\ parameter)$ where W is the worker count. This overhead can be prohibitively large when a model has a single huge parameter, e.g. VGG16's 5.4 Gb fully-connected layer. This is consistent with our simulation with 32 workers at 10 Gbps where ring-reduce iteration time was 34.0 seconds. CNN models (especially those ending in a fully connected layer) tend to have a few layers that take up a significant percentage of the overall model size; thus even an optimal assignment of model parameters would still result in parameter size imbalances for a worker on a ring.

To address this, we modified our simulator to use parameter messaging, where parameters are partitioned evenly between workers (discussed more in §4.8). In the rest of this section, all ring-reduce results make use of this messaging mechanism. What model characteristics positively influence ring-reduce performance relative to baseline?

Factor: Network-dominated backpropagation increases impact: For ring-reduce, each worker begins backpropagation at the same time. This is imposed by a global barrier. In the optimal case for ring-reduce, each worker sends it's assigned model parameter updates at the same time. Then, every link in the ring would be nearly identically utilized and network contention would be minimized. However, each worker can only send its model parameter update when that gradient has been calculated. Longer time for backpropagation computation on each parameter results in deviation from the optimal case. Table 4.6 shows the model with the most compute-bound back-propagation, Inception-v3, has the lowest performance improvement from ring-reduce (3.3x). In contrast, VGG16, the model with the largest performance improvement (24.6x), has the most network-bound back-propagation process.

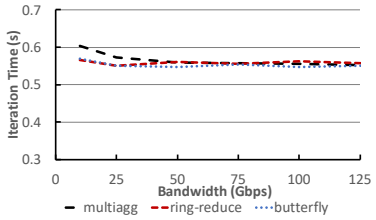


Figure 4.3: **Varying Bandwidth**: Mechanism Rankings for Inception-v3 training time on 32 workers

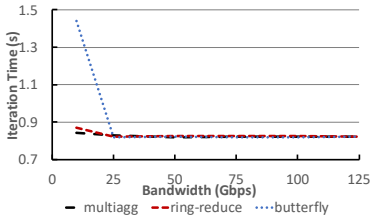


Figure 4.4: **Varying Bandwidth**: Mechanism Rankings for Resnet-200 training time on 32 workers

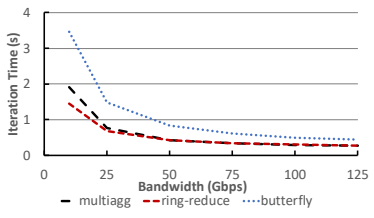


Figure 4.5: **Varying Bandwidth**: Mechanism Rankings for VGG16 training time on 32 workers

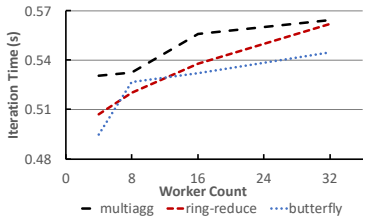


Figure 4.6: Varying Worker Count: Inception-v3 performance at 25 Gbps

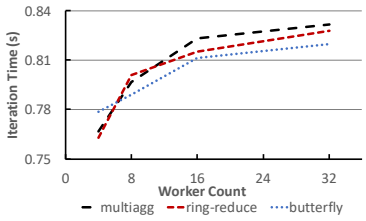


Figure 4.7: Varying Worker Count: Resnet-200 performance at 25 Gbps

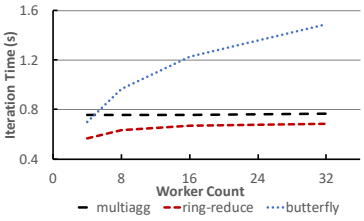


Figure 4.8: Varying Worker Count: VGG16 performance at 25 Gbps

Butterfly Mixing

Recall that the first phase of butterfly mixing merely involves a forward pass. At the point the forward pass begins, each worker already received the complete set of model parameters. Once each parameter is calculated, it will be sent $log(W)$ times, where W is the number of workers. While this communication is sequential, this process can be pipelined between workers. What factors impact performance for butterfly mixing?

Factor: Compute dominated backpropagation increases impact: During backpropagation, longer gradient computations between parameters gives workers a chance to pipeline communications among the $log(W)$ steps. However, models with network dominated backpropagation will *still experience a substantial boost with butterfly mixing*; for example, Table 4.6 indicates that VGG16 still gets a 11.3x speedup.

Butterfly Mixing vs. Ring-Reduce with messaging

Butterfly mixing and ring-reduce are both impacted by an CNN model's backpropagation compute/network interactions, but in opposite ways. Butterfly mixing helps more for compute bound backpropagations, and ring-reduce helps more for more network bound back-

propagation. In Table 4.6, ring-reduce and butterfly mixing perform comparably for all models except VGG16, which has the most network bound backpropagation. This leads to stronger improvement from ring-reduce. Compare this to a lower bandwidth, 10 Gbps. Ring-reduce and butterfly mixing only perform comparably for Inception-v3, which has the most compute bound backpropagation. Conversely, ring-reduce outperforms butterfly mixing for Resnet-200 (Figure 4.4), and VGG16 (Figure 4.5).

Ring-reduce offers superior or equal performance impact as butterfly mixing.

Comparing End-host and In-Network Strategies

In this section, we present overall simulations results for the top performing mechanisms seen so far: ring-reduce with messaging and multicast/in-network aggregation. We include results for butterfly mixing as another competitive reference point, but the focus of this section is on ring-reduce vs. multicast/in-network aggregation.

First, we fix the number of workers to 32 and vary bandwidth. The results for Inception-v3, Resnet-200, and VGG16 are shown in Figure 4.3, Figure 4.4, and Figure 4.5, respectively. Next, for the same acceleration mechanisms, we show how performance scales with number of workers, while keeping bandwidth fixed at 25 Gbps. The results for Inception-v3, Resnet-200, and VGG16 are shown in Figure 4.6, Figure 4.7, and Figure 4.8, respectively. Resnet-101 is left off this panel of graphs but shows consistent trends with Resnet-200.

These results indicate that with 32 workers – across all bandwidths – ring-reduce outperforms the combination of multicast and in-network aggregation. For Resnet-200, Resnet-101 and Inception-v3, ring-reduce and multicast+in-network aggregation perform very similarly. However, ring-reduce holds a key advantage in the VGG16 model. As shown in Figure 4.8 ring-reduce has nearly a 2x performance advantage with 4 workers, and a 1.3x performance advantage with 32 workers. While the gap closes with more workers, we do not observe a case with VGG16 where multicast+in-network aggregation *outperforms* ring-reduce. The gap between the two mechanisms is also more pronounced at low bandwidths, as seen in Figure 4.5.

Ring-reduce has one key advantage over multicast + in-network aggregation. The first and second ring of ring-reduce are equivalent to the parameter server's aggregation and distribution phase, respectively. While there is a per-worker local barrier between the aggregation and distribution phase, no such barrier exists for ring-reduce. The second *distribution* ring of ring-reduce can proceed even while other parameters are still circulating their first ring. Thus the two phases can be pipelined, which enhances performance impact. The difference is particularly stark for models with disproportionately large penultimate layers, such as VGG16 in Figure 4.8.

Ring-reduce offers superior or equal performance impact as multicast with in-network aggregation.

All-reduce with network support

In this subsection, we explore the possibility of using in-network support to accelerate all-reduce algorithms. The primary combination we discuss in this section is ring-reduce with multicast. Note that multicast does not offer any performance benefits to butterfly mixing.

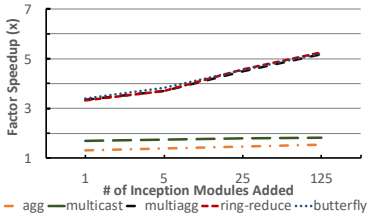


Figure 4.9: Synthetic Model (Network-heavy): Speedups relative to baseline increase as more network heavy layers are added

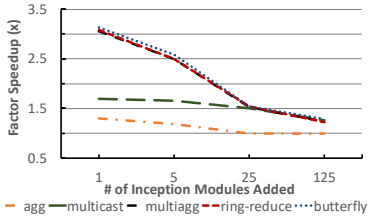


Figure 4.10: Synthetic Model (Compute-heavy): Speedups relative to baseline decrease as more compute heavy layers are added

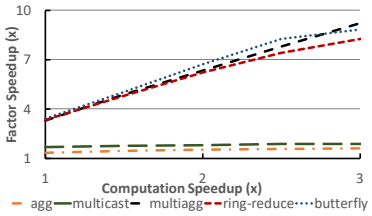


Figure 4.11: Faster GPU: Inception-v3 performance gains as compute speeds increase

The same applies for in-network aggregation for both ring-reduce and butterfly mixing. When using ring-reduce, multicast could be used to improve the performance time of the *second ring*. When parameters have been distributed between workers equally, we find in our simulations that multicast in the second phase of ring-reduce has very limited impact on performance. This can be reasoned about analytically. The communication overhead in the second loop is: $\frac{Model\ size \times (Workers-1)}{Workers \times Bandwidth}$. When using multicast, the communication overhead is: $\frac{Model\ size}{Bandwidth}$ Note the bottleneck is on the receiving worker side. When the number of workers is large, these result in very similar performance. Our simulation results (Figure 4.6) confirm that across the board, ring-reduce with multicast performs equivalently with just ring-reduce.

Future (i.e., Synthetic) CNN Models

To this point in the paper, we have demonstrated that ring-reduce outperforms other proposals. Next we ask, how could these models change over time and would our results still hold? Recent model proposals have simply modified existing CNN models through the addition of convolutional layers [70, 71, 49]. To test this, we add between 1 and 125 modules to the Inception-v3 model. We capture the full breadth of possible layers by adding one of two types of modules: Compute Intensive (35x35x288 module) and Network intensive (17x17x768 module). Again, we run our simulations with 32 workers with 25bps. First, we observe in Figure 4.10, 4.9 that our mechanism ranking are preserved over both strains of synthetic models. However, the relative impact of the speedup mechanisms change. With the compute intensive synthetic model, the performance impact of in-network aggregation quickly drops to to zero, since pipelining in backpropagation provides ample time for parameters to be sent over the network between computations. In contrast, multicast provides the same amount of performance impact between layers, and ultimately equals the performance of the other mechanisms with only 25 layers added. If models trend towards becoming computationally expensive, **operators dedicated to using parameter servers could turn**

away from in-network aggregation for good, as multicast alone equals the impact of using both multicast and in-network aggregation. For the network intensive synthetic model, multicast with in-network aggregation, ring-reduce, and butterfly mixing grow linearly with the number of layers. Even in the extreme case with 125 additional layers, neither in-network aggregation or multicast alone reaches it's maximal performance gain of 2x. While this is hardly an exhaustive list potential model changes, creating synthetic models from existing model traces is extremely simple. Operators and developers can easily modify existing traces to *project* necessary changes to the network fabric or end-host design.

Faster Computational Capabilities

Next, we examine the case of potential enhancements in compute capabilities (e.g., faster hardware accelerators). In particular, do faster computations change the relative effectiveness of the acceleration mechanisms? In our examples, we increased the speed of convolutions by various factors.

Inception-v3 is shown in Figure 4.11 and Resnet-200 is shown in Figure 4.12. First, we observe that there is a point where pipelined phases of the parameter server model become *so network bound* that the parameter server paradigm (i.e., multicast with in-network aggregation) wins out. For most models, this point occurs at around 2.5x speedup, where multicast/in-network aggregation surpasses the performance of both ring-reduce and butterfly mixing. One standout trend from Figure 4.12 is in Resnet-200 at 3x computation speedup, where butterfly mixing results in only a 10x speedup while multicast with in-network aggregation leads to a 19.5x speedup. In fact, for all models except Inception-v3, butterfly mixing gains flags with faster computation. This is consistent with our observation that butterfly mixing impact decreases as backpropagation becomes more network bound. Inception-v3 happens to be so compute-bound that even at 3x compute speedup, butterfly mixing keeps pace with ring-reduce and multicast/in-network aggregation.

Our results indicate that faster compute capabilities could lead to better performance impact when jointly using multicast and in-network aggregation. We hasten to mention that performance is a function of many factors (e.g., bandwidth, memory, etc.) that grow at their own pace. The complexity of these assessment reinforces the need for a simple simulator that can be used to assess the current state of models and hardware.

Evaluation Summary

From most to least performance impactful, our mechanism rankings are as follows: 1.) ring-reduce, 2.) multicast with in-network aggregation, 3.) butterfly mixing, 4.) multicast, 5.) in-network aggregation. Not only have we demonstrated that an **individual end-host mechanism outperforms joint usage of network support**, these performance rankings should continue to hold as models evolve and computation becomes increasingly accelerated. Ultimately, we hypothesize that the reason for this ranking is that current implementations of end-host mechanisms more effectively overlap distinct pipelined phases, thus exploiting available bandwidth more effectively over the entire iteration. This difference in pipelining efficiency grows more pronounced when the penultimate layers of the model

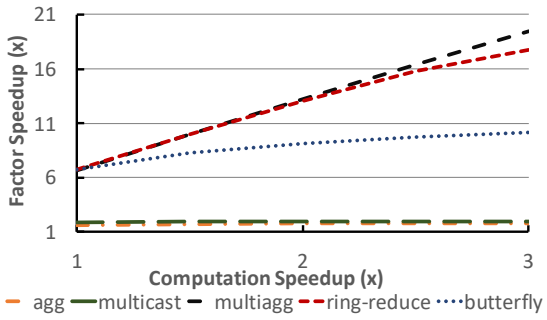


Figure 4.12: Faster GPU: Resnet-200 performance gains as compute speeds increase

Num PS	CNN Model	Min %	Max %	Ideal %
4	VGG-16	$4 \cdot 10^{-5}$	0.918	0.25
8	VGG-16	$1 \cdot 10^{-5}$	0.859	0.125
4	Inception-v3	0.0007	0.451	0.25
8	Inception-v3	0.0003	0.292	0.125
4	Resnet-200	0.165	0.33	0.25
8	Resnet-200	0.002	0.22	0.125

Table 4.7: Empirical Measurements for how parameter assignment to parameter servers. The columns show the percentage (in terms of bytes) of weights that are placed on the most and least occupied parameter server. Resnet-101 is similar to Resnet-200

are disporportionately large. As models inevitably change, operators must carefully examine the qualities of the final CNN layers when considering how to accelerate training.

4.8 Robustness of Evaluation

Much like how we expect the CNN models to change, training software running on the end host will also change. During the simulator development, we identified several *end host* design decisions that influence performance outcomes. Ultimately, we find that our findings are (mostly) robust to these end-host configurations. First, we look at *equal* assignment of parameters to parameter servers. Next, we look at three design configurations that are not currently integrated into Tensorflow: Parameter Distribution, Message Pipelining, and removing parameter server side global barrier. In this section, we rely on simulations so to gain intuition about host level changes from a communication overhead point of view. Our simulations do not include system-level overheads for these approaches, and we leave evaluating this to future work.

Parameter Assignment to Parameter Servers

By default, TensorFlow iterates through the parameters in the model and assigns those parameters to the PS in a round robin fashion. While this effectively balances the *number*

CNN Model	Multiagg (s)	8 PS Multiagg (s)	Ring-Reduce (s)
VGG-16	0.765	0.539	0.683
Resnet-200	0.830	0.820	0.824
Resnet-101	0.598	0.551	0.556
Inception-v3	0.569	0.549	0.562

Table 4.8: Using 8 parameter servers with a theoretically optimal distribution for multicast + aggregation does not provide substantive gains over ring-reduce. 32 workers, 25 Gbps

of parameters per PS, the weights on each of them can be vastly different. The uneven distributions across several models are shown in Table 4.7. For example, in VGG-16, the fully connected layer alone consists of 5.44 Gb (out of 6.58 Gb total over the entire model). Through actual executions of TensorFlow, the simulations we have explored to this point accurately assigns parameters to parameter servers based on TensorFlow's default heuristic. Here, we explore this possibility of dividing parameters evenly.

To simulate this fairly, we aggressively split each parameter between 8 parameter servers and 32 workers. Our results are shown in Table 4.8. With the exception of VGG16, ring-reduce continues to equal the combined efforts of multicast and in-network aggregation. If this end-host design change comes to fruition *and* models trend towards VGG16 characteristics (i.e., short backpropagation and an exceptionally large last layer), operators should act accordingly to consider hardware network support. With the exception of VGG16, parameter assignments does not change the fact that end-host acceleration mechanisms outperform in-network acceleration.

Message Pipelining

In both the distribution and aggregation phases of distributed training, the node typically waits for the entirety of a parameter to arrive before sending it forward. This can be inefficient when individual parameters are large; for example the largest parameter in VGG16 model is in excess of 5 Gb. Instead, these parameters can be split up into smaller messages and forwarded when ready.

We incorporated message pipelining into our application and found, surprisingly, that for **all models we explored, communication within the parameter server model do not benefit whatsoever from message pipelining**. The improvements that result from message pipelining are swallowed by compute in the backpropagation. Only ring-reduce benefits significantly from messaging, thus our prior ring-reduce evaluatino in section 4.7 keep equipped with messaging. Overall, the conclusions we presented in the evaluation are robust to message pipelining.

Presence of a Global Barrier

Up to this point in the paper, we have used a global barrier in the parameter server. While this creates greater flexibility in operations that can be conducted over the entire model, this

CNN Model	Multiagg (s)	Ring-reduce (s)	Multiagg no barrier (s)
VGG-16	1.53	1.37	1.76
Resnet-200	1.65	1.65	1.65
Resnet-101	1.17	1.13	1.08
Inception-v3	1.14	1.13	0.988

Table 4.9: Removing the global barrier improves multicast + aggregation iteration time, but does not cause a decisive lead over ring-reduce. 32 workers, 25 Gbps

inhibits pipelining between iterations. Alternatively, this global barrier could be removed: when the parameter server receives all updates (from workers) to a parameter, it can immediately forward this update [156]. To fairly capture the performance change of removing the global barrier, we run three iterations and measure the time between a.) when the parameter server receives all updates from the first parameter during the latter part of the first iteration and b.) when that first parameter is received from all workers during the latter part of the third iteration.

Table 4.9 shows that the removal of the global barrier increases the impact of multicast plus in-network aggregation to the point that it becomes roughly equal to the impact of ring-reduce. The global barrier improves training time because the aggregation phase can be pipelined with the distribution phase. However, note that some of that improvement is returned because the worker cannot initiate forward pass until the first model layer arrives. That first layer is the final parameter computed in the backpropagation. Taking out the global barrier evens out the impact of ring-reduce and multicast/in-network aggregation, but our initial claim continues to hold.

Parameter Distribution Order

In the parameter server model, there are two ways of distributing parameters: **round-robin distribution** (one model parameter at a time), and **block distribution** (send all model parameters in entirety to each worker, one at a time). While parameter distribution has been observed to be random [69], simulator validation has shown that round-robin distribution order closely mimics actual empirical experiments. Surprisingly, in our experiments, we found that block distribution outperforms round-robin distribution.

When using round robin distribution, workers progress at roughly the same pace. Thus, until one of the workers begins the backpropagation process, the parameter server ingress bandwidth is un-utilized. When using block distribution, the parameter server bandwidth is utilized as soon as the first worker completes its forward pass. Moreover, only a single worker is likely to be doing backpropagation at a time, thus reducing incast.

How does block distribution compare to in-network aggregation? Recall that in-network aggregation benefits performance most when backpropagation staggering is minimal. When using block parameter distribution, under what analytical conditions would we *similar performance impact between in-network aggregation and block distribution*? Let B_1 be the com-

2*CNN Name	2*Bandwidth (Gbps)	2*Agg (s)	2*Block Distr. (s)
Inception-v3	10	2.99	3.1
VGG16	10	22.3	21.7
Resnet-101	10	4.9	4.94
Resnet-200	10	7.77	7.79
Inception-v3	100	0.71	0.77
VGG16	100	2.23	2.27
Resnet-101	100	0.89	0.94
Resnet-200	100	1.19	1.45

Table 4.10: With 32 workers, the training time of using in-network aggregation and block (i.e., not round robin) parameter distributions is roughly the same

putation time of just the first layer of backpropagation, B_N be the compute time of the entire back propagation, N be the time to communicate the entire model, Rem_{FP} be the remaining forward pass computation after the worker has received the entire model update from the parameter server. $B_1 + N + Rem_{FP} > Rem_{FP} + B_N$, which simplifies to $B_1 + N > B_N$. Thus, block distribution approximates the performance gains of in-network aggregation for models that have unusually large last layers and network transfer times (depends on model size and available network bandwidth).

Table 4.10 illustrates simulation results which show that block distribution performs similarly, or better, than in-network aggregation at vastly differently bandwidths. Block raises additional questions about the efficiency of using in-network aggregation for distributed CNN training.

4.9 Discussion

Next we briefly discuss the impact of other optimization strategies and systems considerations.

Gradient Compression: Other work [89] has also looked at using gradient compression to reduce the amount of aggregation traffic sent during CNN training. Gradient compression and other compression techniques reduce model size but do not affect the number of network transfers. As a result, applying these methods is analogous to using a smaller CNN and is covered by our analysis.

Asynchronous training: Our analysis thus far has assumed the use of synchronous training algorithms (§4.2). We focused on these algorithms for ease of analysis and exposition, and our simulator and analysis techniques can be applied to asynchronous training algorithms. However, neither of the in-network mechanisms can be used for asynchronous training due to a lack of barriers between iterations.

4.10 Conclusion

We began this work wanting to develop network optimizations that improve CNN training performance. We found that despite a great deal of excitement about this area, little was understood about what types of optimizations were promising, or even how current optimizations impacted end-to-end CNN training performance. Thus we sought to address this question, and in doing so found that end host based solutions, which are arguably easier to deploy, generally provide better improvements than in-network solutions. We developed a trace-driven simulator, that simplifies the analysis of how network changes impact CNN performance. We hope that this simulator will provide a foundation to enable the community to develop and evaluate optimizations for improving CNN performance. We plan to open source the simulator and our data so as to allow the community to leverage and extend our findings.

Chapter 5

Concluding Thoughts and Future Work

In summary, we show several ways in which the datacenter infrastructure could support operators in critical configuration tasks. We first presented AutoTune, a tool that leverages existing orchestrator APIs to tune resource allocations and placements for heterogeneous microservice applications. Next, we discussed Privoxy, a system that dynamically checks that SQL queries are compliant with a operator-defined privacy policies. Finally, we demonstrated that architectural changes to the networking fabric are unnecessary for the purposes of improving end-to-end performance of a class of convolutional neural networks for image recognition.

My primary goal in orienting this research was not to develop the most sophisticated research solutions, but to solve real problems in ways are accessible to organizations with relatively little access to resources or expertise. Thus, we aimed for our solutions that require minimal (if any) changes to the existing code, and to make as little assumptions as possible about the underlying infrastructure. Despite this, our research does nudge organizations in the direction of rethinking how application developer and operators should think about configuring applications. In my opinion, this represents the most significant challenge in the adoption of new technology.

Privoxy represents a fundamental paradigm shift in how developers should think about privacy when writing code. Rather than relying on a system that rewrites queries into compliant code behind the scenes, developers must be intimately familiar with all privacy policies that govern an application's data access – lest their code cannot go into production reliably. Similarly, AutoTune challenges the notion that operators with limited understanding of application behavior can do no more than overprovision and pray. Instead, our research suggests that the literature 1.) understates the robustness of applications to changing workload and 2.) overstates how dynamic the workloads themselves are.

With this in mind, we envision the following future directions for our work.

- Developer Feedback about Privacy Violations: At this time, Privoxy is only able to inform the developer about which query violates a policy. In organizations that may have hundreds of view-based access policies, fixing the query to be compliant will likely be a challenging problem. In the ideal case, we envision Privoxy being used to inform the developer exactly which privacy policies are being violated by a query.

- Using Policy Checking for Attribute Alignment: One key issue of modern data storage setups is that data is stored in a disorganized fashion, with duplicates stored across different databases with different attribute names. This has significant privacy implications as operators may not be aware of how all sensitive private user information is named. How can Privoxy be used to identify such discrepancies in data naming?

- User specified policies: Organizations commonly collect private information about their clients to inform their business activities (e.g., recommender systems). How can Privoxy be extended to support such a heterogeneous set of user-defined policies? Are authentication views the appropriate abstraction for non-technical customers to explicitly and individually specify how an organization utilizes their data?

- AutoTune limitations: We have provided some intuition using the Datadog trace that AutoTune is feasible to deploy even for dynamic workloads. Could AutoTune truly be a generally viable replacement for autoscaling in highly dynamic workloads? Truly, this will be impossible to answer without actual deployment experience or more sophisticated applications and traces to test AutoTune on.

This dissertation seeks to move towards a world where anyone – not just large corporations – can effectively deploy well-running, sparkling web applications. Historically, technology has typically served to widen inequity in society, in spite of it's sensationalist claims to the contrary. Despite the Internet's democratic potential, one of it's primary limitations is the high barrier to entry for the construction of viral, exciting applications. Building on top of the groundbreaking work to automate deployment of applications, I hope that this dissertation work moves the needle on allowing *anyone* to run perspective-altering applications at scale.

Bibliography

[1] *ab - Apache HTTP server benchmarking tool*. https://httpd.apache.org/docs/2.4/programs/ab.html, retrieved 10/27/2017.

[2] Martín Abadi et al. "TensorFlow: A system for large-scale machine learning". In: *OSDI*. 2016.

[3] Foto N. Afrati. "Determinacy and query rewriting for conjunctive queries and views". In: *Theoretical Computer Science* 412.11 (2011).

[4] Rakesh Agrawal et al. "Extending Relational Database Systems to Automatically Enforce Privacy Policies". In: *the 21st International Conference on Data Engineering (ICDE)*. 2005.

[5] Alfred V. Aho, Catriel Beeri, and Jeffrey D. Ullman. "The Theory of Joins in Relational Databases". In: *ACM Trans. Database Syst.* 4.3 (1979), pp. 297–314.

[6] Omid Alipourfard et al. "CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics". In: *NSDI*. 2017.

[7] Amazon Web Services. *Elastic Load Balancing and Amazon EC2 Auto Scaling*. https://docs.aws.amazon.com/autoscaling/ec2/userguide/autoscaling-load-balancer.html.

[8] Warwick Ashford. *Facebook photo leak flaw raises security concerns*. Mar. 2015. URL: https://www.computerweekly.com/news/2240242708/Facebook-photo-leak-flaw-raises-security-concerns.

[9] Ammar Ahmad Awan et al. "Scalable Distributed DNN Training using TensorFlow and CUDA-Aware MPI: Characterization, Designs, and Performance Evaluation". In: *CoRR* abs/1810.11112 (2018).

[10] Baidu Slicon Valley Lab. *Bringing HPC Techniques to Deep Learning*. http://research.baidu.com/bringing-hpc-techniques-deep-learning/.

[11] *Barefoot Technology*. https://barefootnetworks.com/technology/.

[12] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The SMT-LIB Standard: Version 2.6*. Tech. rep. Department of Computer Science, The University of Iowa, 2017. URL: http://www.SMT-LIB.org.

[13] Clark W. Barrett et al. "CVC4". In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 171–177.

[14] Edmon Begoli et al. "Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources". In: *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. Ed. by Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein. ACM, 2018, pp. 221–230.

[15] Gabriel Bender, Lucja Kot, and Johannes Gehrke. "Explainable Security for Relational Databases". In: *Proc. of SIGMOD*. 2014.

[16] Gabriel M. Bender et al. "Fine-Grained Disclosure Control for App Ecosystems". In: *Proc. of SIGMOD*. 2013.

[17] Michael Benedikt et al. "Benchmarking the Chase". In: *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*. Ed. by Emanuel Sallinger, Jan Van den Bussche, and Floris Geerts. ACM, 2017, pp. 37–52.

[18] Leonard Bolc and Piotr Borowik. *Many-Valued Logics 1 – Theoretical Foundations*. Springer, 1992.

[19] Pat Bosshart et al. "P4: Programming Protocol-Independent Packet Processors". In: *SIGCOMM Comput. Commun. Rev.* 44.3 (July 2014), pp. 87–95. ISSN: 0146-4833. DOI: 10.1145/2656877.2656890. URL: https://doi.org/10.1145/2656877.2656890.

[20] A. Brodsky, C. Farkas, and S. Jajodia. "Secure databases: constraints, inference channels, and monitoring disclosures". In: *IEEE Transactions on Knowledge and Data Engineering* 12.6 (2000), pp. 900–919.

[21] Kristy Browder and Mary Ann Davidson. "The Virtual Private Database in Oracle9iR2". In: *Oracle Technical White Paper* (2002).

[22] John S. Bucy et al. *The DiskSim Simulation Environment Version 4.0 Reference Manual*. CMU PDL, 2008.

[23] Brendan Burns et al. "Borg, Omega, and Kubernetes". In: *Commun. ACM* 59 (2016), p. 5.

[24] Martin Burtscher et al. "PerfExpert: An Easy-to-Use Performance Diagnosis Tool for HPC Applications". In: *SC*. 2010.

[25] John F. Canny and Huasha Zhao. "Butterfly Mixing: Accelerating Incremental-Update Algorithms on Clusters". In: *Proceedings of the 13th SIAM International Conference on Data Mining, May 2-4, 2013. Austin, Texas, USA*. 2013, pp. 785–793. DOI: 10.1137/1.9781611972832.87. URL: https://doi.org/10.1137/1.9781611972832.87.

[26] Hsiang Ann Chen, Yvette O. Carrasco, and Amy W. Apon. "MPI Collective Operations over IP Multicast". In: *IPDPS Workshops*. 2000.

[27] Jianmin Chen et al. "Revisiting Distributed Synchronous SGD". In: *CoRR* abs/1604.00981 (2016).

[28] Tianqi Chen et al. "MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems". In: *CoRR* abs/1512.01274 (2015).

[29] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. "Optimizing Database-Backed Applications with Query Synthesis". In: *the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2013.

[30] Adam Chlipala. "Static Checking of Dynamically-Varying Security Policies in Database-Backed Applications". In: *the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 2010.

[31] Adam Chlipala. "Ur: statically-typed metaprogramming with type-level record computation". In: *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*. Ed. by Benjamin G. Zorn and Alexander Aiken. ACM, 2010, pp. 122–133.

[32] Stephen Chong, K. Vikram, and Andrew C. Myers. "SIF: Enforcing Confidentiality and Integrity in Web Applications". In: *the 16th USENIX Security Symposium*. 2007.

[33] Stephen Chong et al. "Secure Web Applications via Automatic Partitioning". In: *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*. SOSP '07. Stevenson, Washington, USA: Association for Computing Machinery, 2007, pp. 31–44. ISBN: 9781595935915.

[34] Shumo Chu et al. "Axiomatic Foundations and Algorithms for Deciding Semantic Equivalences of SQL Queries". In: *Proc. VLDB Endow.* 11.11 (July 2018), pp. 1482–1495. ISSN: 2150-8097.

[35] Shumo Chu et al. "HoTTSQL: Proving Query Rewrites with Univalent SQL Semantics". In: *the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2017.

[36] *CNN Benchmarks*. https://github.com/jcjohnson/cnn-benchmarks#inception-v1.

[37] E. F. Codd. "Relational completeness of data base sublanguages". In: *Database Systems*. Prentice-Hall, 1972.

[38] Ellis Cohen. "Information Transmission in Computational Systems". In: *Proc. of SOSP*. 1977.

[39] Charlie Curtsinger and Emery D. Berger. "Coz: Finding Code That Counts with Causal Profiling". In: *SOSP*. 2015.

[40] *Datadog: Cloud Monitoring as a Service*. `https://www.datadoghq.com/`.

[41] Leonardo De Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 2008.

[42] Jeffrey Dean and Luiz André Barroso. "The tail at scale". In: *Commun. ACM* 56 (2013), pp. 74–80.

[43] Steve Deering. *Host extensions for IP multicasting*. STD 5. `http://www.rfc-editor.org/rfc/rfc1112.txt`. RFC Editor, Aug. 1989. URL: `http://www.rfc-editor.org/rfc/rfc1112.txt`.

[44] Christina Delimitrou and Christoforos E. Kozyrakis. "Paragon: QoS-aware scheduling for heterogeneous datacenters". In: *ASPLOS*. 2013.

[45] Christina Delimitrou and Christos Kozyrakis. "Quasar: Resource-efficient and QoS-aware Cluster Management". In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '14. Salt Lake City, Utah, USA: ACM, 2014, pp. 127–144. ISBN: 978-1-4503-2305-5. DOI: `10.1145/2541940.2541941`. URL: `http://doi.acm.org/10.1145/2541940.2541941`.

[46] Diaspora Foundation. *The diaspora* Project*. URL: `https://diasporafoundation.org/`.

[47] Joshua V. Dillon et al. "TensorFlow Distributions". In: *CoRR* abs/1711.10604 (2017).

[48] Brendan Gregg. *dtrace pid provider overhead*. `http://dtrace.org/blogs/brendan/2011/02/18/dtrace-pid-provider-overhead/` retrieved 05/01/2018. 2011.

[49] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. "Neural Architecture Search: A Survey". In: *arXiv e-prints*, arXiv:1808.05377 (Aug. 2018), arXiv:1808.05377. arXiv: `1808.05377 [stat.ML]`.

[50] Ali Ghodsi et al. "Dominant Resource Fairness: Fair Allocation of Multiple Resource Types". In: *NSDIR*. 2011.

[51] Daniel B. Giffin et al. "Hails: Protecting Data Privacy in Untrusted Web Applications". In: *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*. Ed. by Chandu Thekkath and Amin Vahdat. USENIX Association, 2012, pp. 47–60.

[52] Daniel B. Giffin et al. "Hails: Protecting data privacy in untrusted web applications". In: *J. Comput. Secur.* 25.4-5 (2017), pp. 427–461.

[53] Tomasz Gogacz and Jerzy Marcinkowski. "Red Spider Meets a Rainworm: Conjunctive Query Finite Determinacy Is Undecidable". In: *Proc. of PODS*. 2016.

[54] Tomasz Gogacz and Jerzy Marcinkowski. "The Hunt for a Red Spider: Conjunctive Query Determinacy Is Undecidable". In: *Proc. of LICS*. 2015.

[55] J. A. Goguen and Jose Meseguer. "SECURITY POLICIES AND SECURITY MODELS." In: *Proc. of IEEE S&P* (1982).

[56] Google Cloud. *Google Cloud: Autoscaling groups of instances*. `https://cloud.google.com/compute/docs/autoscaler`.

[57] *GoReplay*. `https://goreplay.org/`, retrieved 1/13/2020.

[58] Priya Goyal et al. "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour". In: *CoRR* abs/1706.02677 (2017).

[59] Robert Grandl et al. "Multi-resource packing for cluster schedulers". In: *SIGCOMM*. 2014.

[60] Matthew Green. *Twitter post: Piazza offers anonymous posting, but does not hide each user's total number of posts*. Oct. 2017. URL: `https://twitter.com/matthew_d_green/status/925053953330634753`.

[61] Paolo Guagliardo and Leonid Libkin. "A Formal Semantics of SQL Queries, Its Validation, and Applications". In: *Proc. VLDB Endow.* 11.1 (2017), pp. 27–39.

[62] Marco Guarnieri and David Basin. "Optimal Security-Aware Query Processing". In: *Proc. VLDB Endow.* (2014).

[63] Arun Gupta. *Microservice Design Patterns*. `http://blog.arungupta.me/microservice-design-patterns/`, retrieved 9/17/2018.

[64] Diwaker Gupta et al. "DieCast: Testing Distributed Systems with an Accurate Scale Model". In: *ACM Transactions on Computer Systems* 29.2 (May 2011), 4:1–4:48. ISSN: 0734-2071. DOI: `10.1145/1963559.1963560`. URL: `http://doi.acm.org/10.1145/1963559.1963560`.

[65] Diwaker Gupta et al. "To Infinity and Beyond: Time Warped Network Emulation". In: *SOSP*. 2005.

[66] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. "Exploring Network Structure , Dynamics , and Function using NetworkX". In: *SciPy*. 2008.

[67] Raju Halder and Agostino Cortesi. "Fine Grained Access Control for Relational Databases by Abstract Interpretation". In: *Software and Data Technologies*. Ed. by José Cordeiro, Maria Virvou, and Boris Shishkov. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 235–249. ISBN: 978-3-642-29578-2.

[68] Alon Y. Halevy. "Answering Queries Using Views: A Survey". In: *The VLDB Journal* 10.4 (2001).

[69] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy H. Campbell. "Communication Scheduling as a First-Class Citizen in Distributed Machine Learning Systems". In: *CoRR* abs/1803.03288 (2018). arXiv: 1803.03288. URL: http://arxiv.org/abs/1803.03288.

[70] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

[71] Kaiming He et al. "Identity Mappings in Deep Residual Networks". In: *CoRR* abs/1603.05027 (2016). arXiv: 1603.05027. URL: http://arxiv.org/abs/1603.05027.

[72] *Helm – the Kubernetes Package manager*. https://helm.sh/, retrieved 9/17/2018.

[73] Torsten Hoefler, Christian Siebert, and Wolfgang Rehm. "A practically constant-time MPI Broadcast Algorithm for large-scale InfiniBand Clusters with Multicast". In: *2007 IEEE International Parallel and Distributed Processing Symposium* (2007), pp. 1–8.

[74] Uber. *Meet Horovod: Uber's Open Source Distributed Deep Learning Framework for TensorFlow*. https://eng.uber.com/horovod/.

[75] Ethan Jackson. *Kelda: An approachable way to deploy the cloud*. https://github.com/kelda/kelda, retrieved 9/17/2018.

[76] Virajith Jalaparti et al. "Speeding up distributed request-response workflows". In: *SIGCOMM* (2013).

[77] Yang Ji et al. "RAIN: Refinable Attack Investigation with On-demand Inter-Process Information Flow Tracking". In: *CCS*. 2017.

[78] jruby.org. *JRuby – The Ruby Programming Language on the JVM*. URL: https://www.jruby.org.

[79] Sangeetha Abdu Jyothi et al. "Morpheus: Towards Automated SLOs for Enterprise Clusters". In: *OSDI*. 2016.

[80] J. Ker et al. "Deep Learning Applications in Medical Image Analysis". In: *IEEE Access* 6 (2018), pp. 9375–9389. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2017.2788044.

[81] Eddie Kohler. *Download PC review assignments obeys paper administrators • kohler/hotcrp@80ff966* Mar. 2015. URL: https://github.com/kohler/hotcrp/commit/80ff96.

[82] Eddie Kohler. *Hide review rounds from paper authors • kohler/hotcrp@5d53abc*. Mar. 2013. URL: https://github.com/kohler/hotcrp/commit/5d53ab.

[83] Paraschos Koutris et al. "Query-Based Data Pricing". In: *J. ACM* 62.5 (Nov. 2015).

[84] Laura Kovács and Andrei Voronkov. "First-Order Theorem Proving and Vampire". In: *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*. Ed. by Natasha Sharygina and Helmut Veith. Vol. 8044. Lecture Notes in Computer Science. Springer, 2013, pp. 1–35.

[85] Gautam Kumar et al. "Hold 'em or Fold 'Em? Aggregation Queries under Performance Variations". In: *EuroSys*. 2016.

[86] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. "High Accuracy Attack Provenance via Binary-based Execution Partition". In: *NDSS*. 2013.

[87] Alon Y. Levy, Alberto O. Mendelzon, and Yehoshua Sagiv. "Answering Queries Using Views (Extended Abstract)". In: *Proc. of PODS*. 1995.

[88] Mu Li et al. "Scaling Distributed Machine Learning with the Parameter Server". In: *OSDI*. 2014.

[89] Yujun Lin et al. "Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training". In: *CoRR* abs/1712.01887 (2017). arXiv: 1712.01887. URL: http://arxiv.org/abs/1712.01887.

[90] Geert Litjens et al. "A survey on deep learning in medical image analysis". In: *Medical Image Analysis* 42 (2017), pp. 60–88. ISSN: 1361-8415. DOI: https://doi.org/10.1016/j.media.2017.07.005. URL: http://www.sciencedirect.com/science/article/pii/S1361841517301135.

[91] David Lo et al. "Heracles: Improving Resource Efficiency at Scale". In: *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*. ISCA '15. Portland, Oregon: ACM, 2015, pp. 450–462. ISBN: 978-1-4503-3402-0. DOI: 10.1145/2749469.2749475. URL: http://doi.acm.org/10.1145/2749469.2749475.

[92] Liang Luo et al. "Motivating In-network Aggregation for Distributed Deep Neural Network Training". In: *Workshop on Approximate Computing Across the Stack*. Xi'an, China: ACM, 2017. ISBN: 978-1-931971-16-4. URL: http://approximate.computer/wax2017/papers/luo.pdf.

[93] Luo Mai, Chuntao Hong, and Paolo Costa. "Optimizing Network Performance in Distributed Machine Learning". In: *HotCloud*. 2015.

[94] David Maier, Alberto O. Mendelzon, and Yehoshua Sagiv. "Testing Implications of Data Dependencies". In: *ACM Trans. Database Syst.* 4.4 (1979), pp. 455–469.

[95] Jorge Manrubia. *jorgemanrubia/lazy_columns: Rails plugin that adds support for lazy-loading columns in Active Record models*. 2015. URL: https://github.com/jorgemanrubia/lazy_columns.

[96] Hongzi Mao et al. "Resource Management with Deep Reinforcement Learning". In: *HotNets*. 2016.

[97] Aniruddha Marathe et al. "Performance Modeling under Resource Constraints Using Deep Transfer Learning". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '17. Denver, Colorado: Association for Computing Machinery, 2017. ISBN: 9781450351140. DOI: 10.1145/3126908.3126969. URL: https://doi.org/10.1145/3126908.3126969.

[98] Mark Maunder. *Vulnerability in WordPress Core: Bypass any password protected post. CVSS Score: 7.5 (High)*. June 2016. URL: https://www.wordfence.com/blog/2016/06/wordpress-core-vulnerability-bypass-password-protected-posts/.

[99] Microsoft. *Row-Level Security - SQL Server*. 2020. URL: https://docs.microsoft.com/en-us/sql/relational-databases/security/row-level-security.

[100] Microsoft Azure. *Azure Autoscale*. https://azure.microsoft.com/en-us/features/autoscale/.

[101] Barton P. Miller et al. "The Paradyn Parallel Performance Measurement Tool". In: *IEEE Computer* 28 (1995), pp. 37–46.

[102] Jeffrey C. Mogul and John Wilkes. "Nines are Not Enough: Meaningful Metrics for Clouds". In: *Proc. 17th Workshop on Hot Topics in Operating Systems (HoTOS)*. 2019.

[103] Amihai Motro. "An Access Authorization Model for Relational Databases Based on Algebraic Manipulation of View Definitions". In: *Proc. of ICDE*. 1989.

[104] Leonardo de Moura. *Z3 for Java*. URL: https://leodemoura.github.io/blog/2012/12/10/z3-for-java.html.

[105] Andrew C. Myers. "JFlow: Practical Mostly-Static Information Flow Control". In: *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*. Ed. by Andrew W. Appel and Alex Aiken. ACM, 1999, pp. 228–241.

[106] Alan Nash, Luc Segoufin, and Victor Vianu. "Determinacy and Rewriting of Conjunctive Queries Using Views: A Progress Report". In: *Database Theory - ICDT 2007, 11th International Conference, Barcelona, Spain, January 10-12, 2007, Proceedings*. Ed. by Thomas Schwentick and Dan Suciu. Vol. 4353. Lecture Notes in Computer Science. Springer, 2007, pp. 59–73.

[107] Alan Nash, Luc Segoufin, and Victor Vianu. "Views and Queries: Determinacy and Rewriting". In: *ACM Trans. Database Syst.* (July 2010).

[108] Alan Nash, Luc Segoufin, and Victor Vianu. "Views and Queries: Determinacy and Rewriting". In: *ACM Trans. Database Syst.* 35.3 (July 2010). ISSN: 0362-5915.

[109] Tony Mauro. *Adopting Microservices at Netflix: Lessons for Architectural Design*. https://goo.gl/DyrtvI, retrieved 01/21/2017.

[110] *Node Todo-App*. https://github.com/kelda/node-todo, retrieved 1/15/2020.

[111] Rong Pan et al. "SHRiNK: a method for enabling scaleable performance prediction and efficient network simulation". In: *IEEE/ACM Transactions on Networking* 13 (2005), pp. 975–988.

[112] James Parker, Niki Vazou, and Michael Hicks. "LWeb: information flow security for multi-tier web applications". In: *Proc. ACM Program. Lang.* 3.POPL (2019), 75:1–75:30.

[113] Daniel Pasailă. "Conjunctive Queries Determinacy and Rewriting". In: *Proc. of ICDT*. 2011.

[114] Nadia Polikarpova et al. "Liquid information flow control". In: *Proc. ACM Program. Lang.* 4.ICFP (2020), 105:1–105:30.

[115] Dan R. K. Ports and Jacob Nelson. "When Should The Network Be The Computer?" In: *Proceedings of the 17th Workshop on Hot Topics in Operating Systems (HotOS '19)*. ACM. Bertinoro, Italy, May 2019.

[116] Rolf Rabenseifner. "Optimization of Collective Reduction Operations". In: *Computational Science - ICCS 2004*. Ed. by Marian Bubak et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 1–9. ISBN: 978-3-540-24685-5.

[117] *Placing pods relative to other pods using affinity and anti-affinity rules*. `https://docs.openshift.com/container-platform/4.1/nodes/scheduling/nodes-scheduler-pod-affinity.html`, retrieved 9/16/2019.

[118] *Redhat Resource Management Guide*. https://goo.gl/hiFDD7.

[119] Giles Reger, Martin Suda, and Andrei Voronkov. "Finding Finite Models in Multi-sorted First-Order Logic". In: *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*. Ed. by Nadia Creignou and Daniel Le Berre. Vol. 9710. Lecture Notes in Computer Science. Springer, 2016, pp. 323–341.

[120] David K. Rensin. *Kubernetes - Scheduling the Future at Cloud Scale*. O'Reilly Media, 2015, All. URL: `http://www.oreilly.com/webops-perf/free/kubernetes.csp`.

[121] Andrew Reynolds et al. "Quantifier Instantiation Techniques for Finite Model Finding in SMT". In: *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*. Ed. by Maria Paola Bonacina. Vol. 7898. Lecture Notes in Computer Science. Springer, 2013, pp. 377–391.

[122] Chris Richardson. *Pattern: Microservice Architecture*. `https://microservices.io/patterns/microservices.html`, retrieved 9/17/2018.

[123] Shariq Rizvi et al. "Extending Query Rewriting Techniques for Fine-Grained Access Control". In: *Proc. of SIGMOD*. 2004.

[124] Arnon Rosenthal, Edward Sciore, and Vinti Doshi. "Security Administration for Federations, Warehouses, and Other Derived Data". In: *Research Advances in Database and Information Systems Security: IFIP TC11 WG11.3 Thirteenth Working Conference on Database Security July 25–28, 1999, Seattle, Washington, USA*. Boston, MA: Springer US, 2000, pp. 209–223. ISBN: 978-0-387-35508-5.

[125] Amedeo Sapio et al. "In-Network Computation is a Dumb Idea Whose Time Has Come". In: *HotNets*. 2017.

[126] Amedeo Sapio et al. "Scaling Distributed Machine Learning with In-Network Aggregation". In: *NSDI*. 2021.

[127] Daniel Schoepe, Daniel Hedin, and Andrei Sabelfeld. "SeLINQ: Tracking Information across Application-Database Boundaries". In: *the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. 2014.

[128] Luc Segoufin and Victor Vianu. "Views and Queries: Determinacy and Rewriting". In: *Proc. of PODS*. 2005.

[129] Daniel Selsam and Nikolaj Bjørner. "Guiding high-performance SAT solvers with unsat-core predictions". In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2019, pp. 336–353.

[130] Supreeth Shastri, Melissa Wasserman, and Vijay Chidambaram. "How Design, Architecture, and Operation of Modern Systems Conflict with GDPR". In: Mar. 2019.

[131] J. Shi et al. "On the Soundness Property for SQL Queries of Fine-grained Access Control in DBMSs". In: *2009 Eighth IEEE/ACIS International Conference on Computer and Information Science*. 2009, pp. 469–474.

[132] Yuri Shkuro. *Take OpenTracing for a HotROD ride*. `https://medium.com/opentracing/take-opentracing-for-a-hotrod-ride-f6e3141f7941`, retrieved 9/17/2018.

[133] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *CoRR* abs/1409.1556 (2014). arXiv: `1409.1556`. URL: `http://arxiv.org/abs/1409.1556`.

[134] Software Freedom Conservancy. *SeleniumHQ: Browser Automation*. 2021. URL: `https://www.selenium.dev/`.

[135] Ben Stock. *Search leaks hidden tags • Issue #135 • kohler/hotcrp*. June 2018. URL: `https://github.com/kohler/hotcrp/issues/135`.

[136] Lalith Suresh et al. "Automating Cluster Management with Weave". In: *arXiv e-prints*, arXiv:1909.03130 (Sept. 2019), arXiv:1909.03130. arXiv: `1909.03130 [cs.DC]`.

[137] Bert Hubert. *tc(8)*. Linux man page – iproute2. 2001.

[138] Jörg Thalheim et al. "Sieve: Actionable Insights from Monitored Metrics in Microservices". In: *CoRR* abs/1709.06686 (2017). arXiv: `1709.06686`. URL: `http://arxiv.org/abs/1709.06686`.

[139] Nathalie-Sofia Tomov and Stanimire Tomov. "On Deep Neural Networks for Detecting Heart Disease". In: *CoRR* abs/1808.07168 (2018).

[140] Amin Tootoonchian et al. "ResQ: Enabling SLOs in Network Function Virtualization". In: *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*. NSDI'18. Renton, WA, USA: USENIX Association, 2018, pp. 283–297. ISBN: 978-1-931971-43-0. URL: `http://dl.acm.org/citation.cfm?id=3307441.3307466`.

[141] Todd Hoff. *Lessons Learned From Scaling Uber To 2000 Engineers, 1000 Services, And 8000 Git Repositories*. `https://goo.gl/1MRvoT`, retrieved 01/21/2017.

[142] Margus Veanes, Nikolai Tillmann, and Jonathan de Halleux. "Qex: Symbolic SQL Query Explorer". In: *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*. 2010.

[143] Margus Veanes et al. "Symbolic Query Exploration". In: *the 11th International Conference on Formal Engineering Methods (ICFEM): Formal Methods and Software Engineering*. 2009.

[144] Kaushik Veeraraghavan et al. "Kraken: Leveraging Live Traffic Tests to Identify and Resolve Resource Utilization Bottlenecks in Large Scale Web Services". In: *OSDI*. 2016.

[145] Shivaram Venkataraman et al. "Ernest: Efficient Performance Prediction for Large-scale Advanced Analytics". In: *NSDI*. 2016.

[146] Mario Villamizar et al. "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud". In: *2015 10th Computing Colombian Conference (10CCC)*. 2015, pp. 583–590. DOI: `10.1109/ColumbianCC.2015.7333476`.

[147] Abhinav Vishnu, Charles Siegel, and Jeff Daily. "Distributed TensorFlow with MPI". In: *CoRR* abs/1603.02339 (2016).

[148] Izhar Wallach, Michael Dzamba, and Abraham Heifets. "AtomNet: A Deep Convolutional Neural Network for Bioactivity Prediction in Structure-based Drug Discovery". In: *CoRR* abs/1510.02855 (2015). arXiv: `1510.02855`. URL: `http://arxiv.org/abs/1510.02855`.

[149] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. "Synthesizing Highly Expressive SQL Queries from Input-Output Examples". In: *the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2017.

[150] Qihua Wang et al. "On the Correctness Criteria of Fine-Grained Access Control in Relational Databases". In: *Proc. of VLDB*. 2007.

[151] Yuepeng Wang et al. "Verifying Equivalence of Database-Driven Applications". In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017).

[152] Zhiheng Wang. *Navigation Timing*. W3C Recommendation. https://www.w3.org/TR/2012/REC-navigation-timing-20121217/. W3C, Dec. 2012.

[153] Neeraja J. Yadwadkar et al. "Selecting the Best VM across Multiple Public Clouds: A Data-Driven Performance Modeling Approach". In: *SOCC*. 2016.

[154] Jean Yang et al. "Precise, Dynamic Information Flow for Database-backed Applications". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2016. New York, NY, USA: ACM, 2016, pp. 631–647.

[155] Xin Yuan et al. "Group Management Schemes for Implementing MPI collective Communication over IP-Multicast". In: *JCIS*. 2002.

[156] Hao Zhang et al. "Poseidon: An Efficient Communication Architecture for Distributed Deep Learning on GPU Clusters". In: *CoRR* abs/1706.03292 (2017). arXiv: 1706.03292. URL: http://arxiv.org/abs/1706.03292.

[157] Zheng Zhang and Alberto O. Mendelzon. "Authorization Views and Conditional Query Containment". In: *Proc. of ICDT*. 2005.