EPFL

Thèse n° 10 253

Understanding and Mitigating Latency Variability of Latency-Critical Applications

Présentée le 26 juin 2020

à la Faculté informatique et communications
Laboratoire des Systèmes de Centres de Calcul
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

Mia PRIMORAC

École
polytechnique
fédérale
de Lausanne

2020

Il vaut mieux tard que mal, et cela en tout genre.
— Voltaire

Acknowledgements

This PhD journey has been a great one due to all the support I received from all my mentors, colleagues, friends and family.
First of all, I'd like to thank my advisors, Prof. Edouard Bugnion and Prof. Katerina Argyraki, for guiding me, but also for believing in me and giving me freedom to choose my own research direction. I'm very grateful to have been able to work with them and learn from them; we were a great team.
Ed's practical industry experience and his amazing intuition and technical expertise were extremely helpful and taught me a lot about systems. He always encouraged me to bravely tackle difficult problems and conduct data-driven research. I really appreciate that, despite Ed's multiple roles and functions, whenever I needed help or advice regarding either work or life, he was always very generously offering it.
From Katerina I learned a lot about finding good research problems, shaping ideas and presenting them in an understandable and coherent way. Her enthusiasm for research and love for teaching truly inspired me. I'm very grateful for all the times when, despite all her obligations, she helped with deadlines when it mattered the most, shared her deep understanding of computer networks with me, or offered a career advice.
I'd like to thank Prof. Margo Seltzer, Prof. Anne-Marie Kemarrec, Prof. James Larus, and Dr. Stephen Rumble, for dedicating their time to be in my thesis committee and offering their valuable feedback for this thesis.
It was great to formally be a part of two labs, DCSL and NAL, and I'm lucky to have a large network of amazing people that also spans across VLSC, DSLAB, and beyond. I'd like to thank all of them for sharing this journey with me, especially: Adrien Ghosn, Marios Kogias, Dmitrii Ustiugov, Sam Whitlock, Jonas Fietz, Stanko Novaković, George Prekas, Georgia Fragkouli, Luis Pedrosa, Pavlos Nikolopoulos, Arseniy Zaostrovnykh, Yotam Harchol, Zeinab Shmeis, Stuart Byma, Bogdan Stoica, David Aksun, Sahand Kashani, Mahyar Emami, Endri Bezati, Solal Pirelli, and Rishabh Iyer. Thank you for brainstorming with me, attending my long dry runs, and offering valuable feedback, but also for introducing me to your own exciting areas of research. I'm also thankful for all the fun moments that we shared during our lab events, conferences, and in our free time.
Thank you to Maggy and Céline for ensuring that everything ran smoothly in DCSL and NAL, but also for always being kind and helpful. Special thanks to Maggy for always cheering for each and every one of us students, encouraging us, and making sure that we're doing well.
Before starting my PhD I spent an amazing summer in LABOS at EPFL where I worked with

Prof. Willy Zwaenepoel and Amitabha Roy. This experience was crucial for me embarking on this PhD journey and I am very thankful for having had the opportunity to learn from them.
During my PhD I had two fruitful collaborations for which I thank Maaz Mohiuddin, Eleni Stai, Prof. Jean-Yves Le Boudec, and George Prekas. These projects broadened my horizons and taught me a lot, as well as my internship in Intel Labs. Thank you to all the members of Networking Platforms Lab that I worked with, especially to Ren Wang and Mesut Ergin. This great experience encouraged me to pursue industrial research and inspired a big part of the work in this thesis.
It was a pleasure to be a part of the PolyDoc committee and I'd like to thank Sarah van Rooij, Louis Martin-Monier, Gael Lederrey, Nick Morgan, and others, for all the team work that resulted in many social and educational events. I'd also like to thank Miranda Kreković and Marta Martinez-Camara for giving me the opportunity to mentor young girls in GirlsCoding and introduce them to the art of programming. Also, thank you to my Smoozies and all the organizers of Exposure Film Hackathon for this amazing and unique experience that taught me a lot about science communication.
Many thanks to all my friends in Croatia, Switzerland, and around the world, for all the great moments that we've had together during the past (almost) six years, especially to: Ines Akrap and Tanja Kojić for their long-lasting friendship that started long before I knew anything about computers; Jasmina Malićević for her contagious enthusiasm and for being my best conference buddy; Elvira Teran and Ida Amini for brining fun to my internships and maintaining our long-distance friendship; Hermina Petrić Maretić, Helena Peić Tukuljac, and Miranda Kreković for their friendship and support; Vlatka Pavišić, Olivier Blanvillain, Aggelos Biboudis, Denys Shabalin, Jorge Vicente Cantero, Guillaume Martres, Séb Doeraene, Fredrik Lilkaer, Olaf Geirsson and Ronja Hesthammer, and all the others who helped me fill my life at EPFL with sports, games, travels, and great memories.
This journey would be impossible without the love and support of my family. I can't thank enough to my parents, Vesna and Perica, for unconditionally supporting me my entire life. Due to their encouragement to always pursue my dreams I was determined to start and finish this journey. My siblings, Eva Maria and Juraj, grew up during the years of my absence and I'd like to thank them for not letting the distance stand between us. Thank you to my uncle and the rest of my extended family for cheering for me on this journey. I'd like to give special thanks to all my grandparents for always being there for me and supporting me, particularly to my *dida* Miško from whom I inherited my ambition for research and studying and who has always been an inspiration to me.
Finally, I'd like to thank Georg who wholeheartedly supported me all these years and who filled this journey with joy and happiness.
If you keep reading this thesis you will discover a whole lot about "hiccups". Every system has them, but also every long journey like this one has them. While it's important to acknowledge them, it's more important to move on and fulfill all your goals despite them.

Lausanne, June 7, 2020 M. P.

Abstract

A major theme of IT in the past decade has been the shift from on-premise hardware to cloud computing. Running a service in the public cloud is practical, because a large number of resources can be bought on-demand, but this shift comes with its own set of challenges, *e.g.,* customers having less control over their environment. In scale-out deployments of on-line data-intensive services, each client request typically executes queries on many servers in parallel, and its final response time is often lower-bounded by the slowest query execution. There are numerous hard-to-trace reasons why some queries finish later than others.

Latency variability appears at different timescales. To understand its cause and impact we need to measure it correctly. It is clear that we need hardware support to measure latencies on the nanosecond-scale, and that software-based tools are good-enough for the millisecond-scale, but it is not clear whether software-based tools are also reliable at the µs-scale, which is of growing interest to a large research and industry community.

Measuring is a first step towards understanding the problems that cause latency variability, but what if some of them are extremely complex, or fixing them is out of reach given a service provider's restrictions? Even large companies that own datacenters and build their own software and hardware stack sometimes suffer from hard-to-understand or hard-to-fix performance issues. Medium-sized companies that simply use shared public infrastructure, and do not themselves develop most of the code running on the machines, have limited capabilities and often cannot rule out each and every source of system interference. This reality inspired the line of research on what is known as *hedging* policies. By using redundant requests we can reduce overall latency at the cost of consuming additional resources.

This dissertation characterizes latency variability of interactive services, shows how to measure it and how to mitigate its effect on end-to-end latency without sacrificing system capacity. Concretely, this dissertation makes three contributions: First, it shows how to measure µs-scale latency variability with both low cost and high precision, with the help of kernel bypass and hardware timestamps. Second, it empirically derives a lower bound on the tail latency that any implementable hedging policy might achieve for a given workload. Through evaluating our lower bound on a large parameter space, we determine when hedging is beneficial. Lastly, we describe and evaluate a practical policy, LÆDGE, that approximates our theoretical lower bound and achieves as much as half of its hedging potential. We show the applicability of our solution to real applications deployed in the public cloud where LÆDGE outperforms the state-of-the-art scheduling policies by up to 49%, averaged on low to medium load.

Keywords : cloud computing, latency measurement, RPC scheduling, load balancing, hedging

Résumé

Un thème majeur de l'informatique au cours de la dernière décennie a été la migration vers le cloud. L'hébergement d'un service informatique dans cloud est pratique, car les ressources peuvent être allouées à la demande. Cependant, ce changement s'accompagne de ses propres défis et limitations, notamment la perte de contrôle pour les clients sur leur environnement. Dans les déploiements *scale-out* de services en ligne (des services OLDI), chaque requête de clients est exécutée sur de nombreux serveurs en parallèle, et son temps de réponse final est souvent limité par l'exécution de la requête la plus lente. Il existe de nombreuses raisons, difficiles à localiser, pour lesquelles certaines requêtes se terminent plus tard que d'autres.

La variabilité de la latence apparaît à différentes échelles de temps. Pour comprendre sa cause et son impact, on doit la mesurer correctement. Nous avons clairement besoin d'un support hardware pour mesurer des nanosecondes ; les logiciels, quant à eux, sont assez bons pour des mesures à l'échelle des millisecondes, mais sont-ils assez précis pour mesurer des microsecondes (qui intéressent de plus en plus une grande communauté de chercheurs et d'industriels) ?

La mesure est la première étape vers la compréhension de la variabilité de la latence, mais que doit-on faire si l'origine de ces latences est extrêmement complexe, ou impossible à éliminer? Même les grandes entreprises qui possèdent leurs propres centres de données et créent leur propre logicielle et hardware souffrent parfois de problèmes de performance difficiles à comprendre ou à résoudre. Les entreprises qui utilisent une infrastructure publique partagée sont encore moins à même de résoudre chaque source de variabilité ou de latence. Cette réalité a inspiré la recherche en direction de ce qu'on appelle *"hedging"*. En utilisant des requêtes redondantes, on peut réduire la latence au prix de la consommation de ressources supplémentaires.

Cette thèse caractérise la variabilité de latence des services interactifs, montre comment la mesurer et comment atténuer son effet sur la latence de bout en bout sans sacrifier la capacité du système. Concrètement, cette thèse apporte trois contributions : Tout d'abord, elle montre comment précisément mesurer la variabilité de la latence à l'échelle des microsecondes à faible coût. Deuxièmement, elle dérive empiriquement une borne inférieure de la latence que le *hedging* en général peut atteindre. En évaluant notre borne inférieure sur un large champ de paramètres, nous déterminons quand le *hedging* est bénéfique. Enfin, nous décrivons et évaluons une politique pratique, LÆDGE, qui se rapproche de notre borne inférieure théorique et réalise jusqu'à la moitié de son potentiel. Nous montrons l'applicabilité de notre solution aux applications réelles déployées dans le cloud public. LÆDGE surpasse les politiques de planification de pointe de 49% en moyenne, pour une charge du système faible à moyenne.

Mots-clefs : informatique en cloud, mesure de latence, planification RPC, équilibrage de charge, *hedging*

Zusammenfassung

Die breite Einführung von Cloud Computing stellt einen der bedeutendsten IT Trends des vergangenen Jahrzehnts dar. Unternehmen aller Größen betreiben ihre Softwaredienste zunehmend mit Hilfe von Cloudanbietern, weil diese es erlauben Rechenkapazitäten kurzfristig und nach Bedarf einzukaufen. Die Migration in die Cloud bringt allerdings eine Reihe von Herausforderungen mit sich – allem voran haben Kunden von Cloudanbietern weniger Kontrolle über die Umgebung in der ihre Software ausgeführt wird.

In skalierbaren Installationen von datenintensiven Onlinediensten (OLDI Applikationen) führt jede Userabfrage typischerweise zu weiteren Abfragen die parallel auf vielen Servern ausgeführt werden. Die langsamste dieser untergeordneten Abfragen bestimmt dabei häufig die insgesamte Dauer, d.h., die Gesamtlatenz, der Userabfrage. Warum manche dieser Abfragen mehr Zeit in Anspruch nehmen ist oft schwer nachzuvollziehen und wird von zahllosen Faktoren beeinflusst.

Diese Variabilität in Latenzzeiten tritt in verschiedensten Größenordnungen auf. Um ihre Ursachen und Auswirkungen zu verstehen, braucht es exakte Werkzeuge zur Messung der Latenz. Es ist klar, dass wir dedizierte Hardware benötigen, um Latenzen im Nanosekundenbereich zu messen, und dass software-basierte Tools für den Millisekundenbereich ausreichen. Für Latenzen im Mikrosekundenbereich ist dies eine offene Frage mit der sowohl Forscher als auch Praktiker zunehmend konfrontiert sind.

Die Messung von Latenzvariabilität ist ein wichtiger erster Schritt um Onlinedienste zu optimieren, aber in vielen Situationen nicht ausreichend: Oftmals sind die Applikationen zu komplex, oder die Störfaktoren liegen außerhalb des Einflusses des Dienstbetreibers. Doch selbst große Unternehmen, die ihre eigenen Rechenzentren betreiben und volle Kontrolle über Hardware und Software besitzen, kämpfen damit die Performancecharateristika ihrer Dienste zu verstehen. Mittelständische Unternehmen, die öffentliche Cloudinfrastruktur nutzen, und den größten Teil ihres Codes nicht selbst entwickeln, sind davon umso mehr betroffen. Dieser Umstand bildet den Ausgangspunkt für eine Technik namens *Hedging* in Onlinediensten: Mit Hilfe von mehreren redundanten Abfragen (und auf Kosten eines höheren Ressourcenverbrauchs) kann die Latenz von Userabfragen reduziert werden.

In dieser Dissertation charakterisieren wir die Latenzvariabilität interaktiver Onlinedienste, zeigen wie sie gemessen werden kann, und schließlich wie ihr negativer Einfluss auf Gesamtlatenzen mitigiert werden kann ohne die Kapazität des Dienstes wesentlich zu verringern. Konkret leistet diese Dissertation drei Beiträge: Zunächst zeigen wir, wie sich Latenzvariabilität im Mikrosekundenbereich messen lässt, im Speziellen ohne kostspielige, spezielle Hardware und mit hoher Präzision durch Kernel-Bypass Networking und Hardware-Zeitstempel. Zweitens leiten wir empirisch eine Untergrenze für die Gesamtlatenz ab, die eine praktische Hedging-Technik für eine gegebene Auslastung bestenfalls erreichen kann. Dank dieser Untergrenze können wir für einen großen Raum an Parametern bestimmen wann und zu welchem Grad Hedging vorteilhaft ist. Zuletzt beschreiben und evaluieren wir eine konkrete Hedging-Technik, LÆDGE, die sich unserer theoretischen Untergrenze annähert und deren Potenzial bis zur Hälfte ausschöpft. Wir zeigen die Anwendbarkeit unserer Lösung auf reale Onlinedienste in der öffentlichen Cloud. Dabei übertrifft LÆDGE existierende Techniken bei geringer bis mittlerer Auslastung um bis zu 49%.

Stichwörter: Cloud, Latenzmessung, RPC-Scheduling, Lastverteilung (Load-Balancing), *Hedging*

Contents

List of Figures

List of Tables

1 Introduction

Today's world is hardly imaginable without the Internet and its multitude of services defining modern life. Internet services play a large role in our businesses, education, social life and many other aspects of our lives. Especially since social distancing recently became an unavoidable part of our lives [99], we rely on Internet services more than ever.

On-line payment, web search, image search, social-networking applications, e-mail services, video streaming, advertising and recommendation services are just a few examples of Internet services that have billions of users around the globe. Put together, these applications generate an exponentially-growing amount of data, traversing the global network infrastructure between service providers and the devices of users. Service providers typically store the data in large facilities called datacenters that have vast amounts of storage, compute and networking resources. To turn this "big data" into a valuable business resource, the information needs to be adequately organized and promptly accessible.

Not all Internet services have the same performance requirements. This dissertation focuses on interactive services that need to access large amounts of data and need to do it under tight latency constraints. These bounds primarily arise from the client-oriented nature of such applications, also known as Online Data-Intensive (OLDI) services [89]. OLDI services interactively ("On-Line") involve interaction with a user, but also require processing large amounts of data for each user request ("Data-Intensive"). The user can either be an end-user or a company using the service (a business-user). Some of the most important OLDI services are web-search, image-search, social-networking and recommendation services [12, 122].

OLDI services typically operate under hard-to-meet service-level objectives (SLOs), often expressed in terms of tail latency [11, 12, 28, 30, 60, 65, 68]. Each SLO captures a customer expectation, and failing to meet it has concrete consequences, *e.g.,* a hit to the service provider's reputation and a drop in advertising revenue [16, 25, 115, 126].

To access the vast amount of data needed for an OLDI service under these tight latency constraints, the service needs to keep the data in memory and access it in parallel. OLDI services are therefore scaled-out to many servers—each of which contains only a part, a *shard*, of the whole dataset. As a consequence, the user request needs to query many servers in parallel. The root node (also known as a central scheduler or a dispatcher) multiplies the original user request into as many queries as there are shards and sends them to the corresponding leaf servers. The leaf servers process the incoming queries according to some scheduling discipline. The root node can typically send the final aggregated response for the user only once *all* the leaf servers have replied to their query. In other words, the final response time to a user's request is lower-bounded by the slowest of the leaf servers. In practice, such a communication pattern causes problems for service providers trying to meet the tight latency constraints, because a small fraction of slowed-down queries (on a single server) can affect a large number of user requests. In the literature, this problem is called the tail-at-scale problem, introduced by Dean *et al.* [28]. The severity of the tail-at-scale problem is proportional to the number of servers. When more leaf servers are used in parallel, the same small fraction of slowdowns on a single server affects more end-to-end user requests.

The general causes of latency variability in large-scale services are well understood: One is variable *queuing delay, e.g.,* due to load fluctuations [28, 33, 55, 103]. Another one is variable *service time*, which, in turn, comes from two distinct sources:

1. *Inherent query complexity*: Different queries may take different amounts of time to execute on a given hardware and software stack, because of different complexities [57, 72, 87, 135]. For example, in case of an in-memory key-value database a range query will typically take longer than a simple lookup.

2. *Software and hardware system events*: Different instances of the same query may take significantly different amounts of time to execute on a given system because of system events that are unrelated to the service itself: decisions made by an operating-system scheduler or power-management algorithm, interrupts, garbage collection, virtualization effects, or simply competing for CPU and/or cache with other applications [17, 28, 32, 43, 69–71, 76, 78, 80, 88, 92, 98, 119, 128, 134].

Dean *et al.* focus on large service providers that develop their own infrastructure and entire software stacks. Such companies spend a lot of engineering time and resources trying to eliminate the sources of slowdowns as they arise, but latency variability due to unpredictable software and hardware system events is often non-trivial and hard to trace at scale [17, 28]. Finding and fixing its root causes can take a long time—sometimes even a few months [119].

A wide range of service providers are concerned with latency variability. In this dissertation, we look at this problem not only from the perspective of a large-scale service provider, but primarily from the perspective of a service provider that is also a user of a public cloud. Businesses of all sizes, and from a range of very different domains, are increasingly deploying their IT services in the cloud instead of hosting their own on-premise IT infrastructure. The massive shift of company infrastructures to the cloud has been a real game changer in many different industries over the past decade. The advantages of using the public cloud are numerous: it is easy to elastically scale the service up or down, service providers do not need to buy or maintain the machines, and the availability requirement is taken care of by the public cloud providers. However, using a shared infrastructure gives service providers less control over some of the sources of the slowdowns in query processing, such as obscure operating-system decisions or a co-located noisy tenant.

This shift to the cloud is happening partly due to more and more companies experiencing the "big data" problem. Even numerous startups and medium-size companies offer services that can be considered OLDI services and have similar latency requirements—while running in the shared public cloud.

Latency variability is causing problems not only for OLDI-service providers, but also for network-service providers that are trying to deploy software network functions in production. A network function is a building block of a network implementing a simple function, such as a firewall or a router. Historically, one network function in a network was one hardware appliance. In the scope of the Network Function Virtualization (NFV) paradigm [53], network functions are implemented in software for reasons such as flexibility and ease of innovation. Implementing network functions in software makes them much more sensitive to slowdowns, compared to previously-used hardware appliances. At timescales at which these packet-processing applications operate, every microsecond matters.

In today's computer systems it is more challenging to mitigate latencies at the microsecond-scale than at the millisecond-, or even nanosecond-scale, for two main reasons [13]: First, millisecond-scale latencies can be mitigated with various software techniques, such as context switching, that are too slow for the microsecond-scale. Second, techniques implemented in hardware that mitigate nanosecond-scale latencies, such as out-of-order execution and branch prediction, do not scale well to the microsecond regime.

Measuring microsecond-scale latencies brings up a similar set of problems like mitigating them: First, as latency mitigation using software techniques is much more challenging at the microsecond- than at the millisecond-scale, so is latency measurement using software techniques more challenging at the microsecond than at the millisecond-scale. Second, measurement tools that come in a form of proprietary hardware are designed to measure hardware appliances at the nanosecond-scale. These tools naturally work well at the microsecond-scale, but sometimes using them reduces the flexibility of experiments and they can be unjustifiably expensive when lower precision is sufficient for a certain experiment.

Every microsecond counts and counting them incorrectly can lead to misinterpretation of benchmarking results, wrong conclusions and the loss of engineer's time. Having reliable tools to measure microsecond-scale latency is therefore increasingly important. To make reliable latency measurement tools accessible to a larger community of researchers and engineers, they should also be affordable.

1.1 Problem Definition

1.1.1 Measuring Latency Variability

Correctly measuring latency variability is the key to understanding its source and impact. The timescale at which a service provider cares about latency variability depends on the service itself. Network services and microservices operate at microsecond-timescale, while, for example, a document search operates at millisecond-timescale.

To benchmark low-latency appliances, the industry has traditionally used hardware-based tools, namely Spirent [121] and Ixia [64], which provide nanosecond-scale accuracy, but neither flexibility nor low cost. Researchers, on the other hand, typically use software tools, which provide low cost and flexibility, but their accuracy is unclear, if not downright questionable [19]. Using software tools to measure millisecond-precision latency is sufficient, while using tools relying on hardware is obligatory to measure nanosecond-precision latency. When we want to measure microsecond-precision latency, it is not clear what type of tools one should be using. Kernel bypass, as the means to faster I/O [34, 111, 112], and hardware timestamping, now increasingly available in commodity Network Interface Cards (NICs), are creating new opportunities for building better inexpensive traffic generators and measurement tools that can potentially replace the expensive ones.

1.1.2 Dealing with Latency Variability

Precisely measuring latency variability is a step closer to understanding and eliminating it — but only when eliminating it turns out to be feasible. Eliminating each and every source of latency variability is an expensive, unpredictable and never-ending endeavor because of the extreme complexity of today's software and hardware, and the fast pace at which they evolve: Understanding the interactions within the systems to maintain strict latency constraints is very challenging when, at the same time, new code is being pushed to the systems daily and weekly, operating system kernels are changing on weekly and monthly bases, new hardware is added every few quarters *etc.* When we add to the picture running a service in the public cloud and sharing resources with unknown tenants, guaranteeing latency performance is even harder.

An alternative to trying to eliminate each and every source of latency variability is mitigating (*i.e.,* hiding) latency variability, *e.g.,* via redundancy, which is the main focus of this dissertation. Redundancy can mitigate various sources of latency variability arising from the queuing delays in the network [28, 125, 133], as well as the service time in the processing nodes [28, 68, 125]. Redundancy *per se* cannot reduce latency variability caused by inherent query complexity, but it can be combined with software or hardware techniques that do address that latency component (such as adaptive parallelism [57]).

Techniques for mitigating latency variability through redundancy, also called *hedging* techniques, usually leverage the existence of multiple replicas of the same shard [28, 68, 125], or the existence of multiple equal-length paths between each source-destination pair in datacenter network architectures [1, 50, 125]. Both of these sources of redundancy (in both network and computing resources) normally exist in every datacenter deployment of an OLDI service, typically for load-balancing and fault-tolerance purposes. Hedging techniques redundantly issue the same query to more than one replica of the same shard, or send the query through multiple network paths, and wait for the fastest response to arrive. One server replica might reply to a query faster than the others if, for example, its waiting queue was shorter than the waiting queues of the other replicas, or if the query processing on the other replicas was slowed down by a system event or by another concurrently-running application, to name a few possibilities. The gist of hedging is reducing the probability of a query being slowed down by running it multiple times or sending it through multiple paths.

A problem with hedging-based techniques is the increased load that they generate in the system and in the network by redundantly issuing queries. If the additional load is not appropriately controlled, it can significantly deteriorate the performance of the system. For instance, if every query is issued twice the system capacity is halved. Trivially reissuing every incoming query is optimal only when the system load is close to zero [125]. Existing state-of-the-art hedging techniques are more advanced than the naive technique of simply reissuing every single query, but they all fall into one of the two categories: either they issue too many redundant queries and significantly reduce the maximum sustainable throughput of the system, or they do not issue enough redundant queries and fail to mitigate end-to-end latency. This is mostly caused by them lacking awareness of the current system load and relying on sub-optimal push-based load-balancing techniques.

The goal of load balancing is to reduce queuing delays by evenly distributing the load, and, consequently, improving end-to-end latency. If the service does not exhibit any service-time variability, and the network paths do not get congested, the optimal thing to do is to deploy the best load-balancing technique considering the underlying infrastructure properties, such as network latency. Load-balancing policies come in two main flavors: push- and pull-based policies. Traditionally, pull-based (single-queue) load balancing, hereafter referred to as Per-Shard Queuing, has been avoided primarily due to its sensitivity to network overheads, but also due to limited buffering in the previously deployed load-balancing hardware appliances. When network latency is negligible, we know from queuing theory that a single-queue yields minimal queuing delay [74, 103]. Push-based (multi-queue) load balancing mitigates the network delay, but it can lead to load imbalance in the processing nodes when queries exhibit service-time variability. A push-based load balancer needs to decide where to process queries at the time of each request arrival. In cases when it is feasible (given the communication overhead and other constraints), workstealing can ameliorate the load imbalance caused by a badly made decision of the push-based load balancer [31, 44, 79, 103].

Per-Shard Queuing and other single-queue policies introduced in this dissertation are logically centralized only *within* a shard. The scheduler can therefore seamlessly run either on a single machine or on as many machines as there are shards. The degree of shard replication is often limited due to high costs of DRAM [89], and it typically varies between two and six replicas per single shard [68, 107].

1.2 Thesis Statement

This dissertation characterizes latency variability of interactive services, shows how to measure it, and how service providers can mitigate its effect on end-to-end latency with additional use of idle resources in the cloud, but without sacrificing system capacity. We demonstrate that µs-scale latency variability can be measured in a reliable and affordable manner using commodity NICs and servers, with the help of kernel bypass and NIC timestamps. We measure and define intra-query service time variability as latency variability of the same query executing on the same software and hardware stack on different machines. By combining hedging and load-balancing techniques we mitigate intra-query service time variability and set a lower bound for tail latency that a hedging-based policy can achieve. With the help of our lower bound we evaluate a large parameter space of workloads and determine when hedging is a good idea. We design and deploy a practical hedging policy and show, through simulation and system evaluation, that single-queue load balancing and load-dependent hedging are the key to achieving a significant portion of our lower bound.

1.3 Summary of Thesis Contributions

This dissertation makes three main contributions in the area of network measurement and scheduling of datacenter applications. These contributions are especially of interest to service providers of interactive services, as well as developers and researchers working on software network functions.

The contributions are the following:

1. **How to measure microsecond-scale latency variability at low cost and high precision**

   The first part of this dissertation answers when one can use software-based latency-measurement tools, as opposed to reaching out for tools backed up with commodity or proprietary hardware features. The main focus here is on the microsecond-precision latency measurement— which is increasingly important for understanding and optimizing the *killer microseconds* [13] causing latency variability of latency-critical applications, *e.g.*, network services [53]. In reality, latency distributions of network functions are in the µs-scale and can be relatively complex.

   Concretely:

   - In our setup, we show that we can close the gap between commodity tools and proprietary hardware-based tools with the help of kernel bypass (up to 99^{th} percentile latency) and commodity NICs (up to 99.99th percentile latency).

- We answer which traffic generators correctly capture the latency distribution of a simple network function under subtle operating-system configuration changes that are commonly found in deployments of network functions.

2. **Answering when hedging is a good idea through Idealized Hedge**

   Unlike previous approaches that try to tackle both queuing delays and service-time variability through redundancy, this dissertation leverages Per-Shard Queuing to tackle the queuing delays, and uses redundancy to tackle only the service time variability due to, what we call, intra-query service-time variability or IQ-jitter (Section 2.3). A system experiences IQ-jitter when different instances of the same query executing on the same data take different amounts of time to process. We design a theoretical policy that hedges *only* when the current (fast-changing) load allows it and uses a *perfect predictor* of query completion times to free-up the resources when they are required to process pending requests. In that, our theoretical policy, called **Idealized Hedge**, is a lower bound. Here are some properties of Idealized Hedge:

   - Idealized Hedge can determine the maximum end-to-end latency reduction that *any* implementable hedging policy can achieve in our setup.
   - The gist of Idealized Hedge is combining hedging and centralized per-shard queuing.
   - Idealized Hedge always prioritizes new over reissued work and will only create redundant work when there are idle resources available.

   We experimentally compare Idealized Hedge to Per-Shard Queuing through a discrete-event simulation. Per-Shard Queuing yields optimal queuing delay and does not hedge. This allows us to identify regimes where hedging has the potential to improve tail latency, and to upper-bound the potential improvement. In other words, through a wide range of synthetic OLDI workloads and cluster sizes, and by using Idealized Hedge as the lower bound, this dissertation given an answer to the question *"When is hedging a good idea?"*.

3. **Improving the state-of-the-art of latency mitigation techniques with LÆDGE**

   The second part of this dissertation improves over state-of-the-art scheduling techniques for mitigating latency variability of OLDI services in the presence of IQ-jitter. Idealized Hedge cannot be implemented in practice, because it relies on a perfect predictor of query completion times. We contribute three realistic *approximations* of Idealized Hedge and show on a range of workloads for which OLDI deployments these approximations make sense. Concretely, this dissertation introduces **Load-Aware Hedge (LÆDGE)** (pronounced like *"ledge"*) and explores its benefits with different types of cancellations. A cancellation is a remote-procedure call that cancels a query execution on a processing node. We differentiate two types of cancellations: (1) a cleanup cancellation cancels a copy of an already finished query, and (2) a pre-emptive cancellation speculatively cancels one of concurrently running hedged queries. We explore LÆDGE in the following three variants:

- LÆDGE without cancellations,
- LÆDGE with cleanup cancellations, and
- LÆDGE with cleanup and pre-emptive cancellations.

LÆDGE has the following properties:

- It approximates Idealized Hedge and, like Idealized Hedge, relies on centralized queuing.
- It achieves a larger part of the latency reductions of Idealized Hedge than more complex state-of-the-art policies with learned parameters.
- LÆDGE is simple to use and works in practice: We implemented LÆDGE within Google's OLDI framework [49] and deployed it in the public cloud. Our best system result shows that LÆDGE can reduce tail latency of an enterprise search engine by 49% compared to Per-Shard Queuing, averaged on up to 60% utilization.

1.4 Thesis Roadmap

The rest of the dissertation is organized as follows:

Chapter 2 presents the background on OLDI services and the importance of reducing latency variability in such services to achieve a better user experience. The chapter then categorizes different sources of latency variability and discusses prior work in the area of scheduling OLDI services — with the focus on their ways of handling different sources of latency variability. Finally, the chapter gives an introduction to different software and hardware techniques for measuring latency variability.

Chapter 3 evaluates different software and hardware measurement techniques for measuring latency variability of simple network services (a single network function). Concretely, we measure the discrepancy between expensive proprietary appliances and tools that leverage new opportunities in commodity servers.

Chapter 4 explains the motivation behind hedging-based techniques and presents our theoretical policy, Idealized Hedge. The chapter then uses Idealized Hedge on different modeled interactive services and their different deployments to gain insight into the applicability of hedging, as a general policy, to OLDI services.

Chapter 5 introduces our attempt to approximate Idealized Hedge with a realistic hedging policy called Load-Aware Hedge (LÆDGE) that adapts to system utilization. The chapter compares LÆDGE to Idealized Hedge through the same range of workloads from Chapter 4.

Chapter 6 describes the system implementation of our scheduling techniques in an OLDI framework and its evaluation in the public cloud.

Chapter 7 gives an overview of the related work in the area of latency-measurement and latency-reduction techniques.

Finally, Chapter 8 discusses and concludes the dissertation and provides insights for future research in this area.

1.4.1 Bibliographic notes

Chapter 3 largely consists of material I originally published in ACM SIGCOMM KBNets in 2017 [105] with my advisors, Prof. Edouard Bugnion and Prof. Katerina Argyraki, and for which we received the Best Paper Award. This work was also invited for publication in ACM SIGCOMM Computer Communication Review in October 2017 [106].

The work presented in Chapters 4 and 5 is currently under submission.

This dissertation does not include the joint work on energy proportionality and workload consolidation for latency-critical applications, where I designed the Pareto-optimal frontier for the control plane of IX dataplane operating system. This was a joint project with Dr. George Prekas and our advisor Prof. Edouard Bugnion at EPFL, published in ACM Symposium on Cloud Computing in 2015 [104].

This dissertation also does not include the work I did on Fast and Consistent Controller-Replication (FCR) scheme for distributed SDN controllers. I contributed to the design and and implemented the FCR system. This was a joint project with Dr. Maaz Mohiuddin, Dr. Eleni Stai, and Prof. Jean-Yves Le Boudec at EPFL, published IEEE Access in 2019 [95].

2 Background

This chapter first provides background on interactive cloud services necessary for Chapter 4 and Chapter 5. In particular, the chapter focuses on On-Line Data-Intensive (OLDI) services, and the importance of latency variability in such services. OLDI services are particularly sensitive to latency variability because of their scale-out architecture and fan-out communication pattern. The chapter gives an overview of typical sources of latency variability and how their effect translates to the end-to-end performance of OLDI services. We will see which scheduling techniques service providers have been using and how these techniques address latency variability.

The chapter then gives an overview of software and hardware techniques that make measuring latency variability more reliable, providing the necessary background for Chapter 3 that evaluates them.

2.1 Online Data-Intensive Services

This section introduces Online Data-Intensive (OLDI) services such as web search (searching through inverted document indices), content-based image similarity search, recommendation services, advertising services, and social applications. Figure 2.1 shows high-level overview of users accessing a service deployed in a datacenter. OLDI services are "online" services because of their client-oriented nature: End-users of such services are typically connected via their devices and impatiently waiting for the results of the service, while their business-users could be using the service as a building block for another higher-level service that has tight latency expectations from its components.

After a user request traverses the Internet and enters a web server inside the datacenter, it is typically dispatched to a cluster of servers where the relevant data are stored. Figure 2.2 shows a simplified architecture of an OLDI service, *i.e.,* the cluster where the actual processing of the application happens. The cluster that serves OLDI applications consists of a tier of root nodes that serves client requests, while a tier of leaf nodes stores a part of application data sets [12].

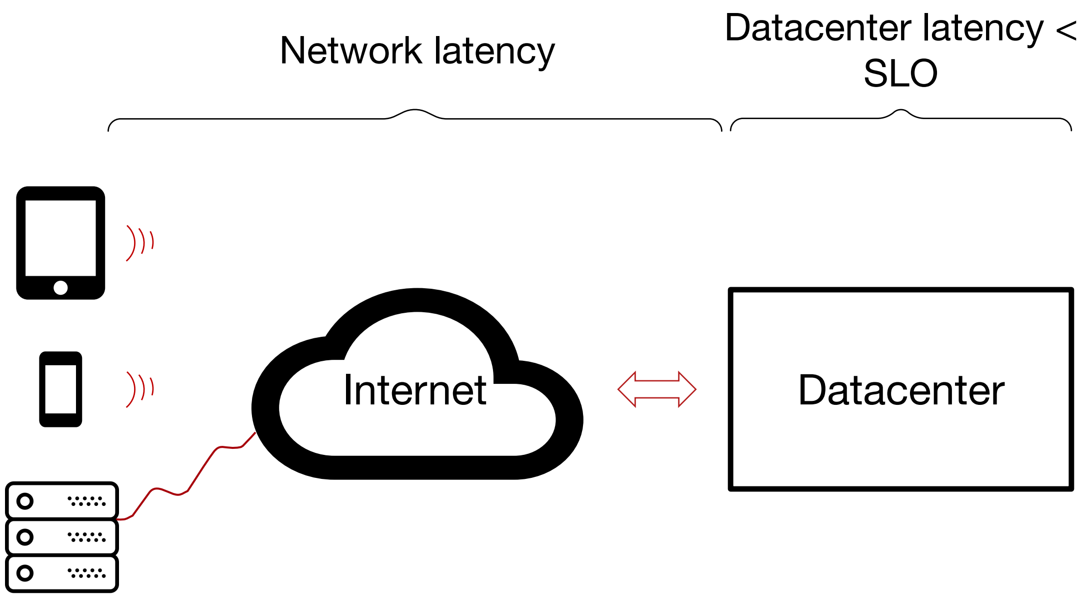


Figure 2.1 – Service providers deploy OLDI services in datacenters. Users of OLDI services have tight latency constraints and a service-level objective (SLO) is a time limit that tries to capture the users' expectations a service provider needs to meet within the datacenter, excluding the network latency required to reach the datacenter.

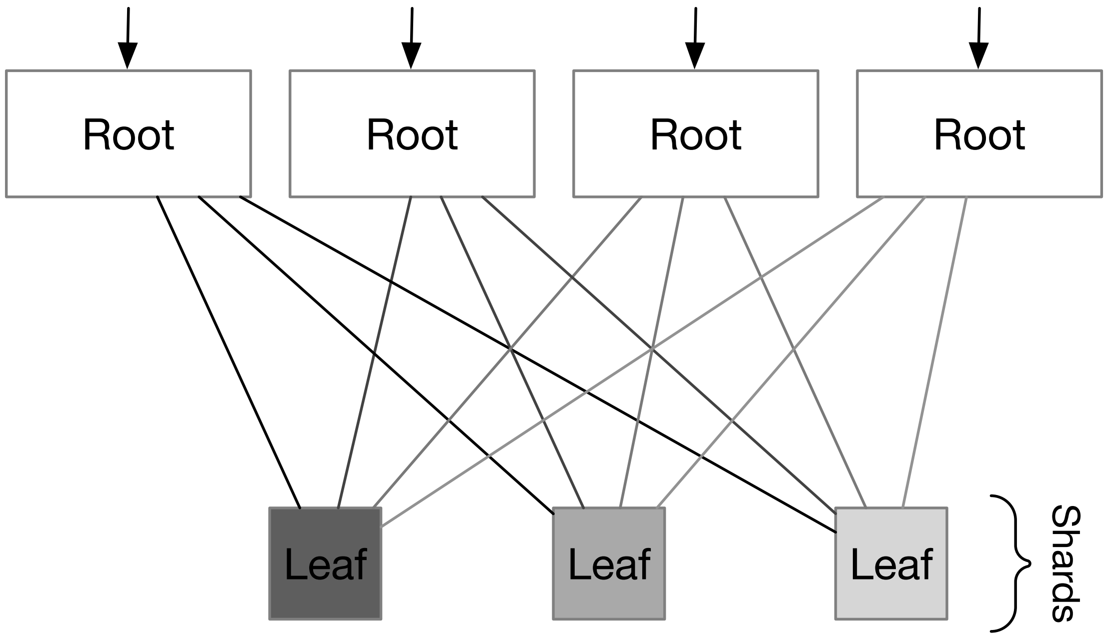


Figure 2.2 – Architecture of an OLDI service without replicated shards. Every root receives user requests and schedules them onto the leaf servers. The data is split into shards and each leaf node holds (at least) one shard of the data.

Such services involve hundreds or thousands of “leaf” nodes, each holding a part (“shard”) of the data needed to answer client requests; a tier of “root” nodes receives client requests, breaks each client request into distinct queries, forwards the queries to different leaves and waits for their replies. This dissertation assumes that for the accuracy of the final results in the client response *at least one reply needs to be received from each distinct shard* (as we will see in Chapter 7, this assumption can sometimes be relaxed). The root therefore waits for the slowest query to finish to aggregate the query responses received from all the shards into the final client response. As a result, distinct queries are often expected to finish within a few milliseconds [28, 56] or even microseconds in case of microservices [122].

2.2 The Effect of Latency on End-Users

End-users expect OLDI services to respond instantaneously. If something goes wrong and the end-to-end response is delayed, users leave the page. The loss of users due to their impatience was directly translated to their reduced engagement (and therefore revenue losses) by Google and Amazon and the results were striking: additional delays as small as 200 ms can already impact user satisfaction and double the time-to-click (which measures user engagement) [115]. This study was among the first ones to highlight the importance of (very) low latency for good quality of service (QoS) experience.

In the context of emerging augmented-reality and virtual-reality applications, the level of responsiveness needs to be even higher to keep the Quality of Experience (QoE) of users high through seamless interactivity [28].

To keep the end-users satisfied, service providers typically quantify the targeted service performance using service-level objectives (SLOs). SLOs need to adequately match the expectations of end-users that lead to high-enough QoS user experience. With business-users, unlike with end-users, service providers establish bounding contracts about the performance of their service, called service-level agreements (SLAs). SLAs are binding, unlike the SLOs, and violating of an SLA leads to financial compensation of the business-user.

SLOs and SLAs are predicates over a set of measurable performance indicators, also known as service-level indicators (SLIs) [94]. If an SLI is the user-facing 99^{th} percentile latency, its SLO could be a threshold that the metric should not exceed (*e.g.,* 300 ms). Figure 2.1 highlights the part of the request path that an SLO is typically referring to. If a service provider violates either an SLA or an SLO, that (explicitly in the former and implicitly in the latter case) translates to losses in revenue [16, 25, 115, 126]. This makes end-to-end latency metrics of today’s services increasingly important. Hereafter we will address the service-provider latency constraints only by the term SLO.

2.3 Sources of Latency Variability

Latency is a time delay between two measurement points. Unless stated differently, latency (in the context of OLDI services) is the delay between the moment when the root node fans-out a user request in the form of queries and the moment when the root node receives at least one query reply from each distinct shard of the data.

The causes of latency variability in large-scale services are well understood: One is *variable queuing delay, e.g.,* due to load fluctuations [28, 33, 55, 103]. Another one is *variable service time,* which, in turn, comes from two distinct sources:

1. Different queries may take different amounts of time to execute on a given hardware and software stack, because of different complexities. An example of this is a range query versus a simple lookup query in an in-memory database. We call this type of latency variability *inter-query service time variability.*
2. Different instances of the same query may take significantly different amounts of time to execute on a given system, because of system events that are unrelated to the service or the data itself: decisions made by an operating system scheduler or power-management algorithm, interrupts, garbage collection, virtualization effects, or simply competition for CPU and/or cache with other applications [17, 28, 32, 69–71, 76, 78, 80, 88, 92, 119, 128, 134]. We call this type of latency variability *intra-query service time variability* (*IQ-jitter* for short).

Let us accordingly define the service time component. Similarly to Mirhosseini *et al.* [92], we express the service time S experienced by a query as the sum of two random variables:

$$S = P + J,$$

where P is the inter-query service time variability determined by query complexity and shard content/size, while J, the intra-query service time variability, is determined by various system events that are independent of the query and the service itself—in general, IQ-jitter depends on the current software and hardware state of the entire leaf node executing the query.

2.4 The Tail-at-Scale Problem

In Section 2.1 we saw that the sheer size of the data that typically needs to be processed by an OLDI service requires a high degree of parallelization across the leaf nodes to meet strict SLOs. With the growing number of leaf servers that need to all work in parallel, the probability of at least one of the leaf servers failing to deliver a query result on-time grows proportionally. The communication pattern of OLDI services in which the root node needs to aggregate the responses from *all* distinct shards amplifies the effect of latency variability within a single node to end-to-end response times.

Consider the architecture of an OLDI service in Figure 2.2. If the number of shards, N, grows and the latency distribution within each leaf server stays the same — the end-to-end latency of the service measured from the root node will increase. For example, if a leaf node replies in time 99.9% of the time and the root node fans-out a query to N such leaves, the aggregate response will finish in time $(0.999^N * 100)\%$ of the time: ~90.48% and ~36.77% of the time for $N = 100$ and $N = 1000$, respectively. At large scale, typically-rare tail-latency scenarios become the common case at scale and exacerbate the end-to-end response time of the end-user request.

This problem is known as the *tail-at-scale problem*, named after the paper by Dean *et al.* in which it was first addressed [28]. The tail-at-scale problem emphasizes the importance of building latency-tolerant services, because in today's complex systems the service providers cannot realistically guarantee that none of the potential sources of latency variability on, for example, the server-level, will not slow down sending the final response to end-users.

2.5 How to Schedule RPCs in OLDI Services?

	Policy	**Queuing model**	**Hedging probability**	**Load balancing**
a)	*Naïve Hedge* [28, 125]	push	1	random
b)	*d-Hedge (with or without cancellations)* [28, 68]	push	$Pr(RTT > d)$	random
c)	*p-Hedge* [68]	push	$q \cdot Pr(RTT > d)$	randomization + JSQ
d)	*Random load balancing*	push	0	none
e)	*Join-shortest-queue (JSQ)* [51]	push	0	JSQ
f)	*Join-bounded-shortest-queue (JBSQ)* [74]	push & pull	0	JBSQ
g)	*Per-Shard Queuing (PSQ)*	pull	0	per-shard centralized queue

Table 2.1 – Existing hedging and load-balancing policies.

2.5.1 Replication in OLDI Services

Let us extend the system architecture in Figure 2.2 by shard replicas. The new OLDI architecture we consider is depicted in Figure 2.3 – the main difference is that multiple leaves can serve the same shard. It is not uncommon that shards are replicated a few times across different leaf

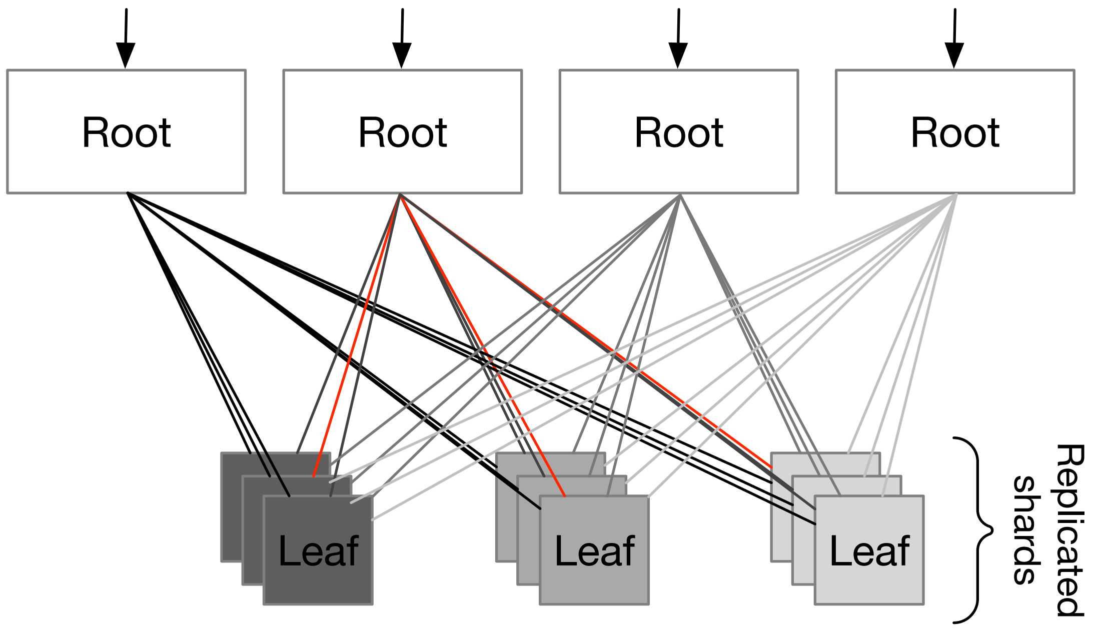


Figure 2.3 – Architecture of an OLDI service with replicated shards. Every root receives user requests and schedules them onto the leaf servers. Each leaf node holds a part of the data. The red lines illustrate the *minimum* amount of communication when a request arrives to a root node. At least one replica of each shard needs to be accessed in order to aggregate a correct final response.

nodes to provide additional throughput capacity and maintain availability in the presence of failures [28, 89]. In reality, the degree of shard replication is often limited due to the high cost of DRAM [89], and it rarely goes beyond six replicas per shard [68, 107].

Introducing replication in a distributed system with read and write operations raises concerns about consistency - if one request writes some data to a shard, and another request subsequently reads from a different replica of the same shard the update will not be visible unless the replicas are in sync. This type of problems are out of the scope of this dissertation. We focus on read-only workloads that include a wide range of OLDI workloads.

Scheduling requests *within* the same replicated shard is an interesting problem that researchers have addressed with two main types of techniques: *hedging* and *load balancing*. Table 2.1 shows an overview of hedging and load-balancing policies from the literature. The rest of this section describes each of them.

2.5.2 Hedging

Since we typically have replicated shards, any query for a distinct shard can be replicated and sent to at least two nodes that serve that shard. This way, the system *hedges its bets*, as it needs to wait only for the fastest response to each query. Hedging was widely adopted in combination with performance monitoring, to improve job completion time of map-reduce-style [29] jobs that may take tens to hundreds of seconds [4–6]. More recently, however, it is being proposed as a means to reduce tail latency in large-scale services, including OLDI services [28, 42, 56, 65, 67, 68, 108, 116, 125, 127, 132].

Hedging can mitigate latency variability in the context of interactive services as long as system noise is independent per leaf, or as long as there is queue imbalance in the leaves. Two or more leaves hosting the same data shard are unlikely to both suffer a query slowdown while serving the same query. Hence, by replicating a query across multiple leaves and collecting the response that arrives first, we reduce the probability of the query being slowed down by a performance hiccup, or by other requests in the queue.

In this thesis, we consider three representative hedging policies (top three rows in Table 2.1) and study how they can help OLDI applications:

- **Naïve Hedge [125]:** The root *always* sends each query to *all* the leaves that host the corresponding shard in a FCFS manner. This is conceptually the simplest hedging policy as it does not require storing any state at the root. It has been applied in many different contexts, from map-reduce jobs [4–6, 23, 136] to DNS queries, database servers, and packet forwarding [125].

- **Delayed Hedge (d-Hedge) [28, 68]:** For each query, the root randomly picks a leaf that hosts the relevant shard and sends the query to it; if the reply does not arrive within a pre-configured, fixed delay d, the root replicates the query to another leaf. The value of d can be tuned by the system operator to control the number of replicated queries in the system. In addition, d-Hedge is typically implemented with cleanup cancellations: whenever a query response arrives, every pending reissued query is cancelled either by the root/scheduler or by the leaf replica from which the response has arrived. This is the policy that was proposed by Dean *et al.* when they introduced the “tail at scale” problem [28] and was used as a baseline by Kaler *et al.* [68].

- **Probabilistic Hedge (p-Hedge) [68]:** This is similar to d-Hedge, but introduces an extra tuning knob: the probability q of replicating each delayed query; both the probability q and the delay d are trained based on the workload. This was proposed by Kaler *et al.* in their recent study of hedging policies for datacenters [68] and is the most sophisticated hedging policy that we found in the literature.

2.5.3 Load Balancing

Besides hedging, the standard approach to managing latency is load balancing (LB). Load balancing aims to evenly distribute network traffic across multiple servers and that can happen

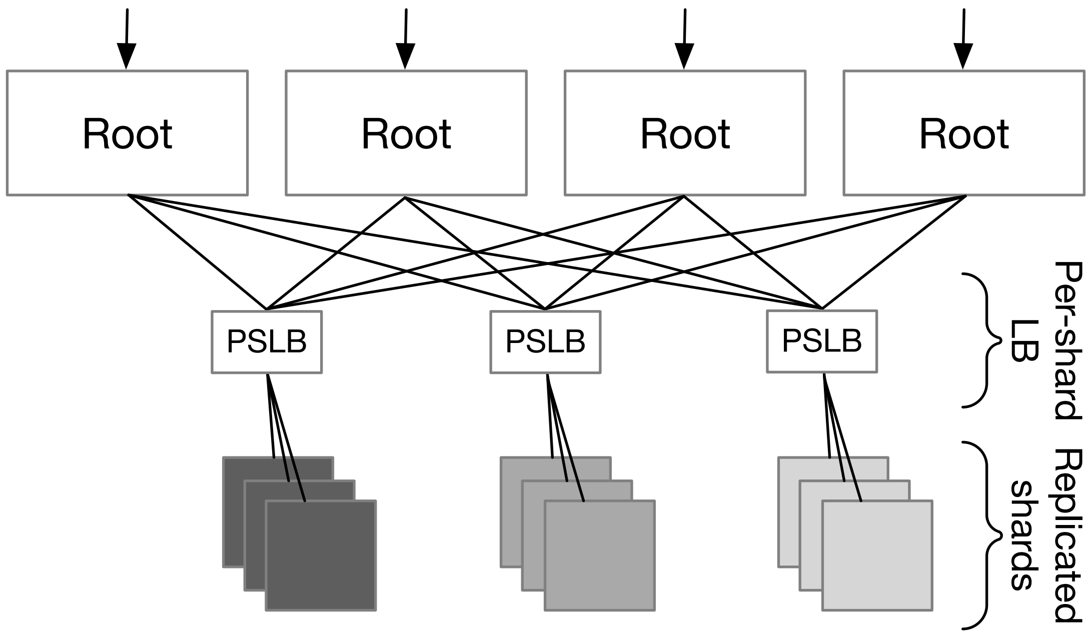


Figure 2.4 – Architecture of an OLDI service with per-shard load balancers. The roots forward user requests to per-shard load balancers that schedule the queries onto the replicated shards.

at various layers of the OSI network stack. The most commonly known are Layer 4 and Layer 7 load balancing. Layer-4 load balancers, also known as network load balancers, balance connections to servers, and they do not take into account the number of requests within each connection. A Layer-7 load balancer balances requests to servers. In this dissertation we consider solely Layer 7 load balancing.

Load balancers have traditionally been implemented in hardware, but due to costly upgrades and limited scalability, a lot of research has been done on reliable and scalable load balancers implemented in software. Today it is a common practice to deploy software load balancers in cloud environments [36, 74, 97].

In the context of OLDI applications, service providers deploy load balancers in two main ways: First, a *pre-root* load balancer can receive the requests before the root nodes and assign them to one particular root node according to some scheduling discipline. Second, in addition to the pre-root load balancer, a service provider can also deploy a *per-shard* load balancer that receives the requests for a particular shard and schedules the requests within the shard onto the leaf nodes. We focus on per-shard load balancing with the focus on its end-to-end tail-latency behavior. Figure 2.4 depicts an architecture of an OLDI service with per-shard load balancers deployed after the root nodes.

Note that different logical per-shard load balancers do not have to be located on distinct physical machines as suggested in Figure 2.4. When the load-balancing decision does not require any additional state (for example, simply picking one random replica from each shard), the load-balancing logic can be implemented within the root nodes themselves. For implementing stateful load-balancing policies (such as Join-shortest-queue [51]) a per-shard load balancer is indeed located between the root nodes and the leaf nodes (if there are multiple root nodes), similarly to the setup in Figure 2.4, but multiple logical load balancers for different shards may be co-located on the same physical machine.

A per-shard load balancer makes an important decision on how to leverage multiple replicas of the same shard. The scheduling policy implemented in the load balancer can largely influence both throughput and latency behavior of the entire service. The load-balancing scheduling policy can be either push-based (*i.e.,* multi queue) or pull-based (*i.e.,* single queue). A push-based load-balancing policy speculatively decides where to schedule a query at the moment of its arrival and sends it to a replica. This means that the queues are not in the load balancer but in the replicas (*i.e.,* in the leaves). Opposite of a push-based is a pull-based load-balancing policy that lazily decides where to schedule a query at the moment when a replica becomes available. The queues in the leaves are in this case very shallow, and requests are buffered in the per-shard load balancer itself.

In this dissertation, we consider four per-shard load-balancing polices (listed in rows (d) to (g) in Table 2.1):

- **Random:** For each query, and for each shard, the per-shard load balancer, randomly picks a leaf that hosts the relevant shard and sends the query to it.

- **Join-shortest-queue (JSQ) [51]:** The root tier sends each query to a per-shard load-balancer, which immediately sends the query to a leaf node; of all the leaves that host the relevant data shard, the load-balancer picks the one with the smallest number of pending queries for that shard. JSQ is known to outperform Random but is far from optimal for FCFS servers with highly-variable job sizes [51, 58, 59]. We picked it because it is simple to implement and popular in the industry, *e.g.,* it is widely deployed in reverse-proxies of server farms [51, 97].

- **Per-Shard Queuing (PSQ):** The root tier sends each query to a per-shard load-balancer, which stores the query until a leaf that can serve it becomes available. In other words, the load balancer dispatches a query to a leaf *only* if the leaf is idle. From a queuing-theory perspective, Per-Shard Queuing corresponds to a single-queue $M/G/k$ model, where k is the number of leaves to choose from. This policy in theory outperforms any LB policy that uses multiple distinct queues (*e.g.,* JSQ) in the presence of non-deterministic service times [118]. The reason why single-queue solutions have not been very popular in practice is that they requires synchronization between the per-shard load balancer and the leaves (so that the load balancer knows when a leaf becomes idle); this exposes the round-trip latency between the load balancer and the leaves, hence may significantly impact throughput. On the other hand, in a modern datacenter running OLDI services, the round-trip time between the root tier (or the per-shard load-balancing tier) and the leaves is typically significantly smaller than the service time of OLDI queries (for example, it takes < 20µs to perform an RPC between two virtual machines in Microsoft Azure [41]), which makes Per-Shard Queuing an excellent candidate for load balancing [92, 100].

- **Join-bounded-shortest-queue (JBSQ) [74]:** This policy combines the advantages of PSQ and JSQ, by splitting the job of queuing pending queries between a centralized scheduler and the leaves. It takes a parameter n, which specifies the number of pending requests that each leaf can hold, *e.g.,* JBSQ(1) is equivalent to PSQ, while JBSQ(∞) is equivalent to JSQ. The value of n can be configured so as to hide the round-trip latency between scheduler and leaves and enable full throughput.

Note that both hedging and load-balancing policies can be implemented in the per-shard load balancer.

In the rest of this dissertation we will come back to the scheduling techniques introduced above, evaluate them, and challenge them in order to better mitigate latency variability and achieve lower end-to-end response time.

Let us now focus on one particular server and the process of measuring latency variability of that server.

2.6 Overhead of Latency Measurements

Measuring latency variability is the first step towards understanding its cause and fixing it or, alternatively, finding the best technique to mitigate its effect. In this section we will give an overview of latency measurement techniques, some of which we will evaluate later in Chapter 3.

Latency can be measured at different levels of the hardware and software stack: on the wire [139], in the Network Interface Controller (NIC) [38, 73], in the kernel driver [139], in the in-kernel socket layer [73], in the application implemented on top of a kernel bypass framework [27], or in the application implemented using a standard POSIX API [96]. Measuring closer to the wire naturally results in more precise measurements.

Figure 2.5a shows the components of a latency measurement experiment and the path that a packet goes through on its way from the sender, through a device under test (DUT), before it reaches the receiver. The sender and receiver can be the same or a different entity. In general, a DUT can be either a hardware appliance, or a software application running on a commodity server. Let us take a closer look at 2.5a and the most important moments (marked with T_1 to T_8) in a lifetime of a single measurement packet sent with a traditional networking stack:

- The packet is formed by the sender's user-space application, which then issues a system call to the kernel after T_1.
- At T_2, the kernel queues the packet in the NIC.
- At T_3, the first bit of the packet is sent over the network to the DUT.
- T_4 and T_5 are the moments of receiving the first incoming bit and sending the first outgoing bit of the packet on the DUT, respectively.
- On the way back the packet enters the NIC at T_6.
- After a hardware interrupt, the packet reaches the kernel space at T_7.
- When the sender application gets scheduled, the packet reaches the user space once again at T_8.

Software-based latency measuring tools can suffer from the same set of problems as the software under test related to operating system overheads between two consecutive measurements, interference with co-located applications, *etc.* Noisy measurements can be misleading and lead to wrong conclusions and the loss of an engineer's time.

The standard tools that engineers use, such as netperf [96], use the traditional networking stack and CPU timestamps in user space, *i.e.,* they measure the packet latencies as $T_8 - T_1$ in 2.5a. This exposes the measurements to many overheads.

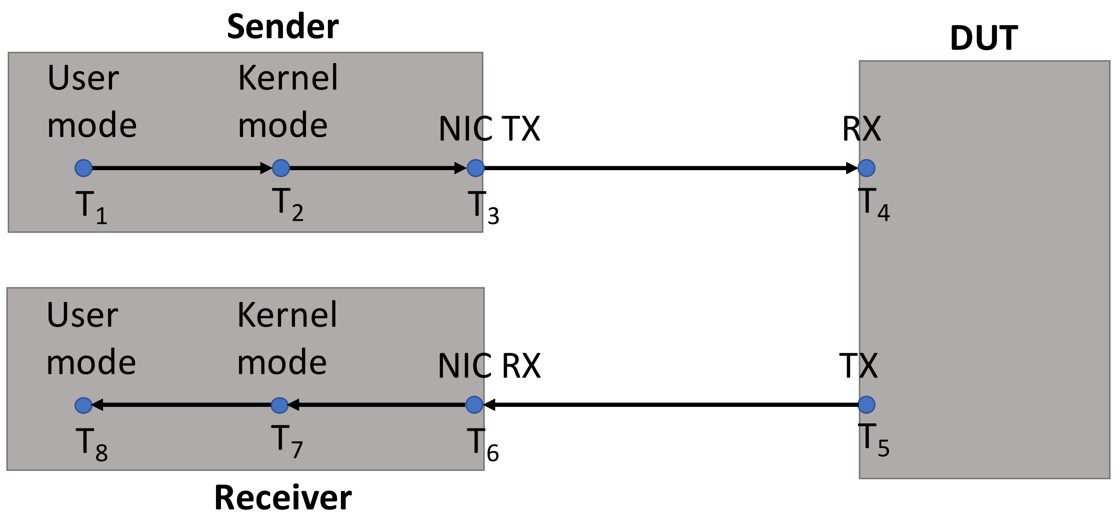
The overheads of network stacks implemented in the kernel are widely known [15, 54, 61, 66, 82, 111]. POSIX socket operations are typically implemented as system calls and involve unnecessary data copies, since the application buffers are not explicitly exchanged with the kernel, the packet processing can experience bad cache locality due to decoupling networking stack from the application, *etc.* These are just a few examples of how a measurement could be distorted due to latency variability of the measurement tool itself.

Kernel bypass, as the means to faster I/O [34, 111], and hardware timestamping, now increasingly available in commodity Network Interface Cards (NICs), create new opportunities for building better traffic generators and latency measurement tools.

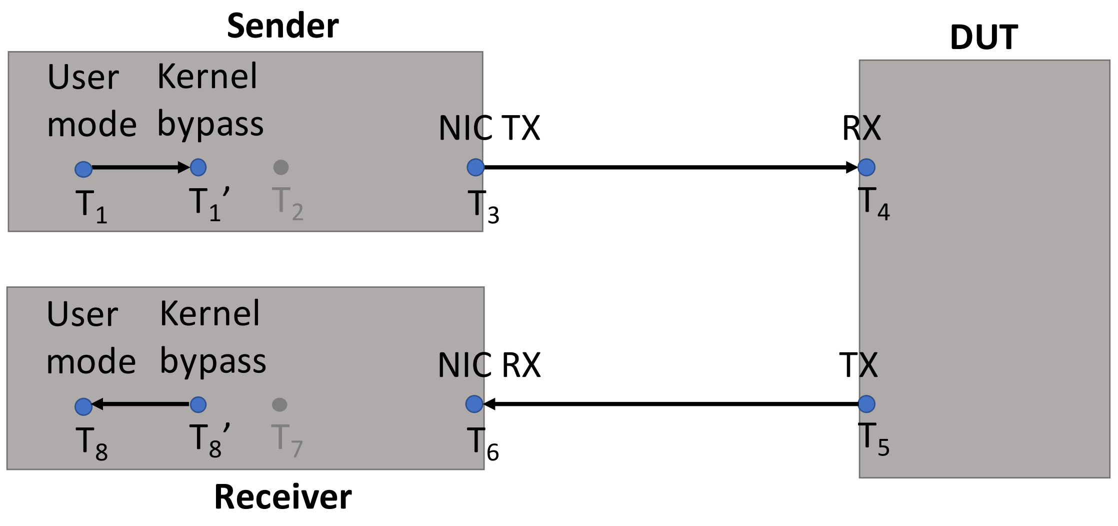
2.6.1 Kernel Bypass I/O

Kernel bypass significantly reduces OS overheads in packet processing by moving protocol processing to user space. Control of the I/O device is also moved to user space, which is why kernel bypass solutions are typically hardware dependent. We consider only Ethernet I/O.

Popular kernel bypass frameworks such as Intel's DPDK (Data Plane Development Kit) [34] and netmap [111] are widely used to develop packet processing applications with high throughput



(a) Traditional networking stack



(b) Kernel-bypass networking

Figure 2.5 – Different points in hardware and software stack where one can measure latency.

and low latency requirements, such as network functions [137] or servers with microsecond-scale requirements [74]. DPDK dedicates the entire NIC to an application, while netmap maps packet queues directly to application address space. Packet processing on top of kernel bypass is not standardized and it is up to the developer to choose an appropriate protocol stack or develop her own.

Traffic generators such as TRex [27] and Pktgen [124] leverage kernel bypass to increase the precision of latency measurements, and achieve higher throughput, but they use only software timestamps (*i.e.,* CPU timestamps).

Figure 2.5b depicts the path of a measurement packet with the sender's measuring application implemented on top of a kernel-bypass framework. The time it takes between the two timestamps in this case is $T_8 - T_1 - (T_3 - T_1') - (T_8' - T_6)$.

2.6.2 Hardware Timestamping

The two variants of hardware timestamping that we consider in this dissertation are (1) commodity NIC timestamps and (2) FPGA-based timestamps built into proprietary hardware applicances. In both cases the time it takes between the two timestamps, according to Figure 2.5, amounts to $T_6 - T_3$.

NIC Timestamps

Timestamping a packet directly in the NIC provides highly precise measurements, but the overhead and the generality of the implementation can vary from hardware to hardware. Many commodity NICs, such as Intel's x710 10GbE NIC [63] or 82599 10GbE, support hardware-based packet timestamping, but their implementation is narrowly tailored to the precise requirements of IEEE 1588 time synchronization, *i.e.,* the implementation relies on matching one outstanding PTP timestamping packet with its reply which restricts the type of packets that can be timestamps and significantly restricts the throughput of timestamped packets. Intel's NIC 82580 [62], provides hardware support to timestamp all incoming packets. Unfortunately, outgoing packets must still be timestamped in software by the application. More recent Mellanox NICs, such as ConnectX4 [90], offer general-purpose hardware timestamping support for all incoming and outgoing packets.

Recent traffic generators such as LANCET [73] and MoonGen [38] leverage NIC timestamps of commodity NICs in order to obtain highly precise latency measurements.

Hardware Appliances

Proprietary hardware-based measuring devices, such as Spirent [121] and IXIA [64], guarantee nanosecond-precision latency measurements. Their primary purpose is testing hardware switches and middlebox appliances, and they have been widely adopted in telecommunication industry. Such devices typically have FPGA devices that can timestamp all outgoing and incoming packets even at high speeds (40 Gbps and beyond). Unfortunately, their programmability is quite limited as they are tailored to standard RFC benchmarks.

3 How to Measure Microsecond-Scale Latency Variability

One accurate measurement is worth a thousand expert opinions.
— Grace Hopper

Many network applications have tight end-to-end latency requirements, including VoIP and interactive video conferencing, high-frequency trading, high-performance computing [75], as well as different network services [53, 117]. In these applications, even microsecond-scale latency variations can be intolerable.

To understand the causes of latency variability in latency-critical applications that have service times in the order of microseconds, such as microservices or network services, one needs to measure it correctly. This chapter focuses on network services and serves as a guideline for network researchers, as well as software developers, that work on reducing latency variability in such applications.

Network researchers need tools to generate traffic and measure latency and throughput. The ideal tool would combine low cost, flexibility, and accuracy: it would be inexpensive to obtain and usable with commodity components; enable the generation of arbitrary traffic patterns and the testing of arbitrary protocols; and provide latency – including tail-latency – measurements at the µs-scale. An eager client for such a tool today would be the community researching network function virtualization (NFV), whose goal is to study the latency and throughput of network functions [39, 53].

The industry has traditionally used hardware-based tools [64, 121], which provide accuracy, but neither flexibility nor low cost: they are excellent for validating Application Specific Integrated Circuits (ASICs) using standardized approaches [21], but they cannot test arbitrary protocols, and they are too expensive for most researchers. For the price of a hardware traffic generator that is able to saturate a link with tens of Gbps, one can buy tens of commodity servers with multiple NICs.

Researchers, on the other hand, typically use software tools, which provide low cost and flexibility, but their accuracy is unclear, if not downright questionable [19]. Many researchers have experienced the frustration of using software traffic generation and measurement – because that is the only option – while worrying about noise and repeatability, especially when the Linux networking stack and socket-based interface are involved [19]. With datacenter and cloud operators chasing the killer microsecond [10], researchers increasingly report results in µs-scale tail latencies [80, 114, 138]; but such results can be trusted only if they are obtained with a tool that provides accuracy at the same scale.

Kernel bypass, as the means to faster I/O [34, 111], and hardware timestamping, now increasingly available in commodity Network Interface Cards (NICs), are creating new opportunities for building better traffic generators and measurement tools. For instance, MoonGen – a scriptable, high-speed packet generator built on top of Intel DPDK (Data Plane Development Kit) can provide precise latency measurements while executing user-provided Lua scripts per packet [38]. It relies on many modern NICs having hardware-based packet timestamping, some of which are tailored to the precise requirements of IEEE 1588 time synchronization. Recent work shows that precise RTT measurements with hardware timestamps can be highly beneficial even for datacenter congestion control [22, 77, 93]. However, the precision of the NIC hardware timestamps has its limits [38, 77] and, to the best of our knowledge, we were the first to evaluate it against a commercial hardware appliance [106].

We ask the following two questions:

1. *How close do state-of-the-art commodity solutions get to bridging the gap between hardware and software and providing accurate* µ*s-scale tail latency measurements?*
2. *Are the measurements sufficiently accurate to study the latency distribution of software network functions?*

We answer the first question based on a simple observation: the latency of a constant-rate flow going through an ASIC-based switch is expected to be constant. We first use a proprietary hardware-based measuring device to confirm that it indeed hardly varies. We then use this measured latency as the ground truth and determine up to which percentile, different software tools measure it correctly.

We answer the second question by sending constant-rate flows through a software network function that simply forward packets. We use different hardware and software tools to observe the impact of operating system (OS) configurations on the network function's tail latency up to the 99.9999^{th} percentile. We quantify the mismatch between the hardware ground truth and other tools.

We also contribute the following results:

- The use of NIC-based hardware timestamps on commodity NICs, evaluated on Intel's X710 10GbE, provides accurate readings up to the 99.99^{th} percentile, but not beyond.

Tool	Characteristics	Latency Measurements	Measured at Granularity
Spirent [121]	Commercial hardware appliance, commonly used for standardized RFC2544 performance tests	FPGA-based or proprietary	$10ns$, 1µs, 5µs
MoonGen [38]	Dataplane using DPDK and Lua	On many NICs determined by the NIC leveraging IEEE 1588 support (if at all available)	$10ns$ (hardware), $100ns$ (software)
TRex [27]	DPDK dataplane	Determined by the CPU	$100ns$
netperf [96]	Socket-based interface	Determined by the CPU	$100ns$

Table 3.1 – Overview of evaluated traffic generators.

- A tuned DPDK solution such as TRex introduces 5µs to 10µs overhead in readings, yet does allow study of the impact of operating-system configuration changes in network forwarding devices.
- POSIX-based solutions that rely on blocking I/O introduce almost 20µs overhead at 50^{th} percentile and have a 50µs long tail, hence should be avoided when measuring µs-scale latencies.
- Our study suggests that bidirectional hardware support is highly beneficial to accurately measure µs-scale latencies.

3.1 Overview of Traffic Generators

Table 3.1 lists the traffic generation and measurement tools that we consider in this thesis. Our goal is not a comparison of all the available tools – we consider only a subset that we deemed sufficient for understanding where kernel bypass lands between traditional hardware and software tools when it comes to latency measurements.

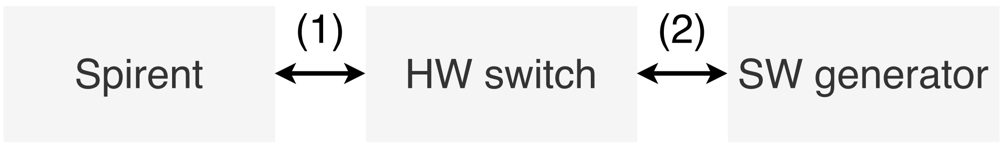
Spirent [121] represents state-of-the-art hardware-based tools. It was designed to accurately measure ns-scale latency, but it is customized for a fixed set of predefined, standardized tests such as the ones specified in RFC 2544 [21]. It is possible to configure traffic generation to some extent, through GUI or scripts written in high-level languages, some of which require an extra license that bears a substantial cost.

MoonGen [38] represents state-of-the-art software tools that leverage kernel bypass and hardware timestamping at the NIC. It is built on top of DPDK and LuaJIT, and it is fully scriptable. Its best reported performance result is 178.5 Mpps with 64-byte packets running on twelve CPU cores at 2 GHz and twelve 10Gbps links while executing user-provided Lua scripts per packet.

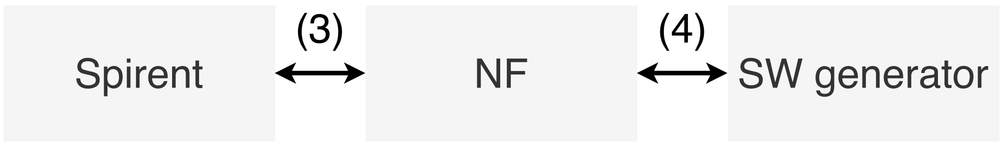
TRex [27] is a software tool that leverages kernel bypass as well, but it relies on software timestamps. We use the stateless version, whose best reported performance result is that it can generate 10-20 Mpps with 64-byte packets while running on one core.

Finally, netperf [96] represents traditional software tools that use blocking POSIX API and conventional network drivers. Even though it was designed to measure performance and not as a full-fledged traffic generator, netperf can generate constant-rate UDP traffic of configurable message size, burst size, and inter-message time. This flexibility is good enough for assessing the benefit of kernel bypass over traditional I/O for latency measurements.

3.2 Experimental setup



(a) Measuring a hardware switch.



(b) Measuring a network function.

Figure 3.1 – Experimental setups.

We now describe our experimental setup, including the configuration of any hardware and software tools.

3.2.1 Hardware setup

We use four devices: a Cisco SG500X-48 switch [26] ("HW switch"), an FPGA-based Spirent SPT-3U chassis [121] ("Spirent"), and two x86 machines, one acting as a software traffic generator and measurement node ("SW generator"), the other as a network function ("NF"). The nodes are connected with two meter long 10 GbE cables. The x86 machines are dual socket Intel Xeon CPU E5-2699 v4 @ 2.20GHz with hyperthreading disabled, each with two Intel x710 10 GbE NICs [63].

We experiment with the four configurations depicted in Figures 3.1a and 3.1b and enumerated accordingly:

1. Spirent + HW switch: to measure the true latency of the Cisco switch (Figure 3.1a).
2. SW generator + HW switch: to measure the accuracy of latency measurements achievable with software tools (Figure 3.1a).
3. Spirent + NF: to measure the true latency of our network function (Figure 3.1b).
4. SW generator + NF: to determine whether software tools can accurately measure the latency of our network function (Figure 3.1b).

In all experiments, we use two distinct physical ports from each device, and each port is both sending and receiving traffic (so, we have two independent end-to-end flows). In the case of the x86 machines, the two ports are located in different NICs, but attached to the same CPU socket's PCIe root complex.

3.2.2 Software setup

The "NF" machine implements port forwarding. The OS is Fedora Linux 23 with kernel version 4.4.9. The forwarding software uses DPDK version 17.02 and consists of two forwarding streams in opposite directions, running on two cores that each have a dedicated RX and TX queue. Unless otherwise stated, we configure the machine to minimize jitter: we disable all the power-saving options such C-states and P-states, NUMA balancing, transparent huge pages, kernel audit, and interrupt moderation, and we run all the cores at the nominal frequency (but not TurboBoost).

The "SW generator" machine has similar configuration, but runs one of the following programs:

- **MoonGen with hardware timestamping**:
  We use the version from GitHub referenced in [37]. Hardware timestamping on Intel x710 10 GbE was designed for IEEE 1588 [35] time synchronization, and it can only timestamp one outstanding single packet at a time due to resynchronization requirements, therefore can do sampling only of the latency distribution. It uses a separate hardware queue for the non-timestamped traffic.
- **MoonGen with software timestamping**:
  MoonGen also works with software timestamps. In this case, there is no sampling limitation – the latency of all packets can be captured. We keep a $100ns$-granularity histogram.
- **Netperf**:
  We use the standard netperf tool, but replaced its histogram implementation with our own, more fine-grained one ($100ns$). We use a UDP request-response benchmark with histograms and inter packet time control enabled.

- **TRex**:
  We use the stateless version of TRex [27]. We created a control plane experiment to fit our benchmark requirements. As with netperf, we replaced its original coarse-grain histogram implementation (10µs granularity) with our own (100ns granularity).

We further isolate the CPUs on which we run the traffic generators, and pin the forwarding tasks to these CPUs. We also make sure the cores, ports, and allocated memory are on the same socket.

In all experiments, the (hardware or software) traffic generator produces two independent UDP flows of 64-byte packets, each one at a rate 1Gbps. We calibrate all the tools to the same line rate using Spirent as a sink. We report the data from 5 independent runs of each experiment. Each run executes the benchmark for 120 seconds after a warm-up of 30 seconds.

Measurement granularity depends on the tool. The Spirent chassis has 16 adjustable-size histogram buckets, which we set after calibration to 10ns in Section 3.3 and between 1µs and 5µs in Section 3.4. Hardware timestamps in MoonGen have the precision of 10ns. The software solutions (MoonGen-SW, TRex, netperf) keep a 100ns-granularity histogram of latencies as measured using the processor's cycle counter.

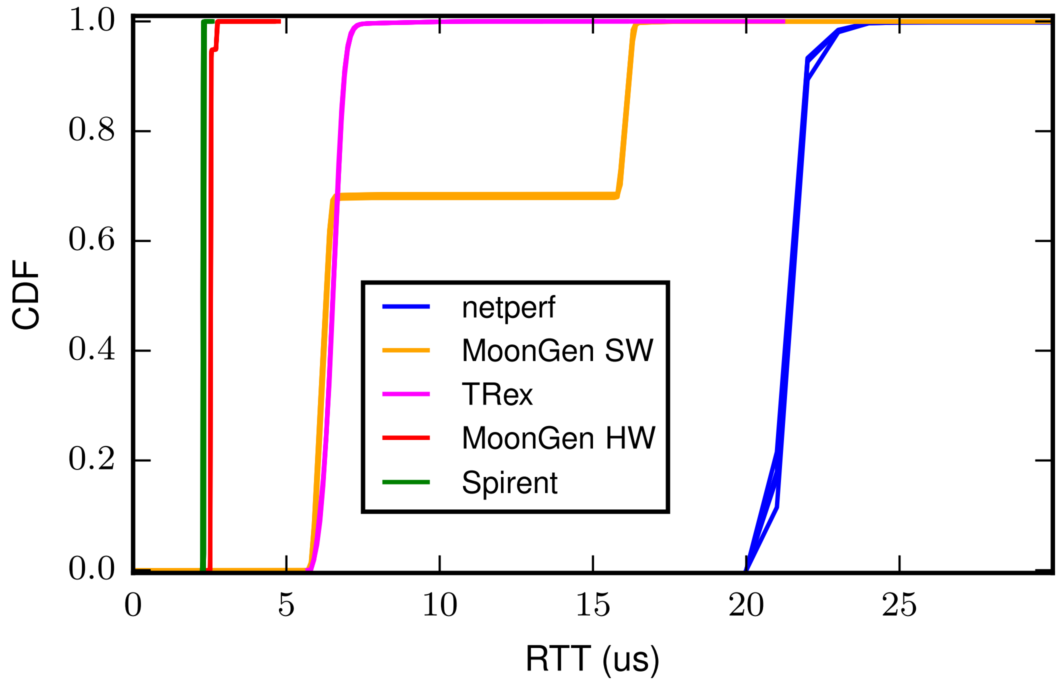
3.3 Closing the HW/SW gap

We now answer the following basic question: when measuring µs-scale latency, how far are state-of-the-art software tools from traditional hardware-based tools?

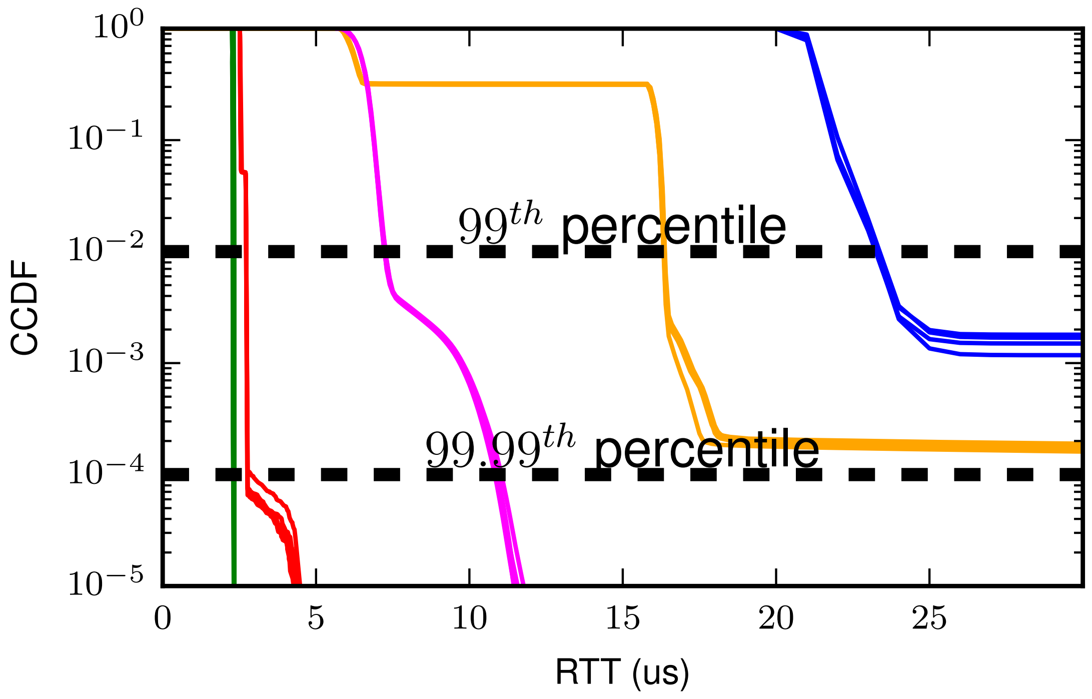
To answer this first question, we use the testbed configurations in Figure 3.1a to measure the latency of the HW switch. The idea is that any modern ASIC-based switch is expected to offer per-packet latency of a couple µs with insignificant jitter; we use testbed configuration (1) to confirm this, and testbed configuration (2) to test whether the software tools can measure µs-scale latencies.

Figure 3.2 shows the switch's latency distribution as reported by the different tools. The same data is presented in two different ways, akin to [80]: Figure 3.2(a) shows the cumulative distribution function (CDF) of the latency, while Figure 3.2(b) shows the complementary cumulative distribution function (CCDF), which provides a more explicit view of tail latency (shows which fraction of measurements exceed a given latency value). The CCDF is presented on a log scale to highlight the effects of tail latency. In case the figure is viewed in black and white, the labels are ordered by ascending accuracy, i.e., the top-most label (netperf) corresponds to the right-most (least accurate) latency distribution. We report data up to the 99.9999th percentile ($1-10^{-6}$).

First, we confirm that the HW switch provides stable, if not exceptionally low, latency, as expected from an ASIC-only datapath: Spirent reports minimum, mean, and maximum latency



(a) CDF of the latency



(b) Complementary CDF

Figure 3.2 – Latency measurements of a hardware switch using different tools.

of 2.24µs, 2.26µs, and 2.52µs, respectively. The spread across more than 400 million measurements is, therefore, less than 300ns. The reported CCDF (left-most one in Figure 3.2(b)) is near vertical up to the 99.9999th percentile.

Second, we see that the combination of kernel bypass and hardware timestmaps comes very close to the ground truth provided by Spirent: MoonGen-HW reports minimum, median, and 99.99th percentile latency of 2.466 µs, 2.524 µs, and 2.723 µs, respectively. The reported CCDF (second from the left in Figure 3.2(b)) is less than a µs away from the ground truth up to the 99.99th percentile. Beyond that, however, the error increases. For instance, the 99.999th percentile ($1-10^{-5}$) latency is 4.379 µs, a noticeable increase, most likely due to the imperfect synchronization between the two different NICs of the SW generator (the one where each measured packet departs and the one where it arrives) [38].

Third, we see that kernel-bypass alone is not enough, hardware timestamps are necessary to get this close to the ground truth: MoonGen-SW reports latency between 5.225 µs and 7.1 µs for 60% of the measurements, but significantly higher for the rest. TRex does better, overlapping with MoonGen-SW for 60% of the measurements, including the median of 6.5 µs, but reporting stable latency up to the 99th percentile of 7.3 µs, only a 28% increase from the minimum value. Unless hardware timestamps are available, implementing your software generator in a managed language comes at a high price.

Finally, we see the limits of using the standard POSIX API and conventional network drivers for latency measurements: Netperf (the right-most curve in both graphs) is at least 17µs off the ground truth. We attribute the gap to highly variable latency of interrupt dispatching and thread wakeups on multicore machines. Our conclusions are reproducible even with NIC ports directly connected. For more details please refer to our repository [86].

3.4 Measuring Subtle Differences in OS Configurations of Network Functions

We now answer the second question: are software tools accurate enough for measuring the latency of software network functions?

To answer the second question, we build on the insights of Section 3.3 to measure the latency distribution of our network function (DPDK port forwarding). For these experiments we use the testbed configurations in Figure 3.1b.

We want to experiment with scenarios that introduce non-trivial latency and jitter, but are also realistic and interesting to the networking community. So, instead of introducing artificial latency and jitter ourselves, we consider four OS-level configurations that have latency implications:

1. **local**: a baseline "out-of-the-box" OS configuration, where the network function (CPU

and memory) runs on the same NUMA socket that has the PCIe root complex of the NIC. The NIC/memory interactions are therefore all local to the same socket.

2. **remote**: also a baseline OS configuration, but the network function runs on the remote NUMA node relative to the PCIe root complex of the NIC. All NIC/memory interactions must therefore go through the QPI interface between sockets.

3. **local+isolset**: we augment "local" to further use the `isolcpu` and `taskset` features of the Linux scheduler to explicitly isolate the network function and ensure that no other application is ever scheduled on the same core.

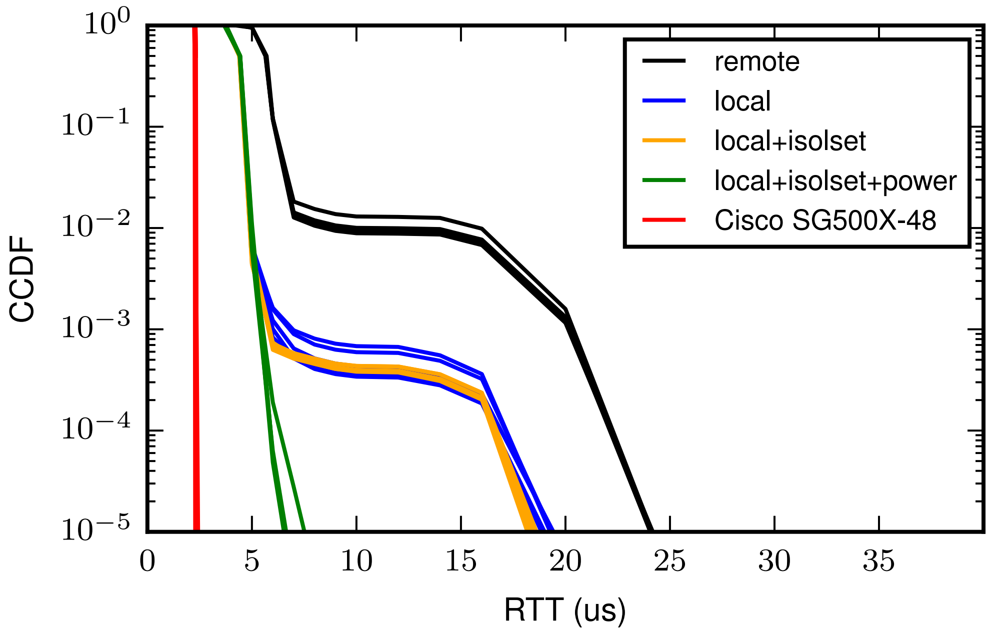
4. **local+isolset+power**: we further disable power-saving options including P-states (and TurboBoost), C-states and PCIe Active State Power Management.

Figure 3.3 shows the latency distribution of the network function, for each of the four OS configurations, as reported by different tools. Each subfigure shows the latency CCDF of the network function for the four OS configurations, as well as the latency CCDF of the HW switch, which is used as a reference. Each subfigure reports data captured by a different tool; we omit netperf from this evaluation due to its limitations shown in Figure 3.2. The labels are ordered by ascending accuracy, i.e., the top-most label (remote) corresponds to the right-most (least accurate) distribution.

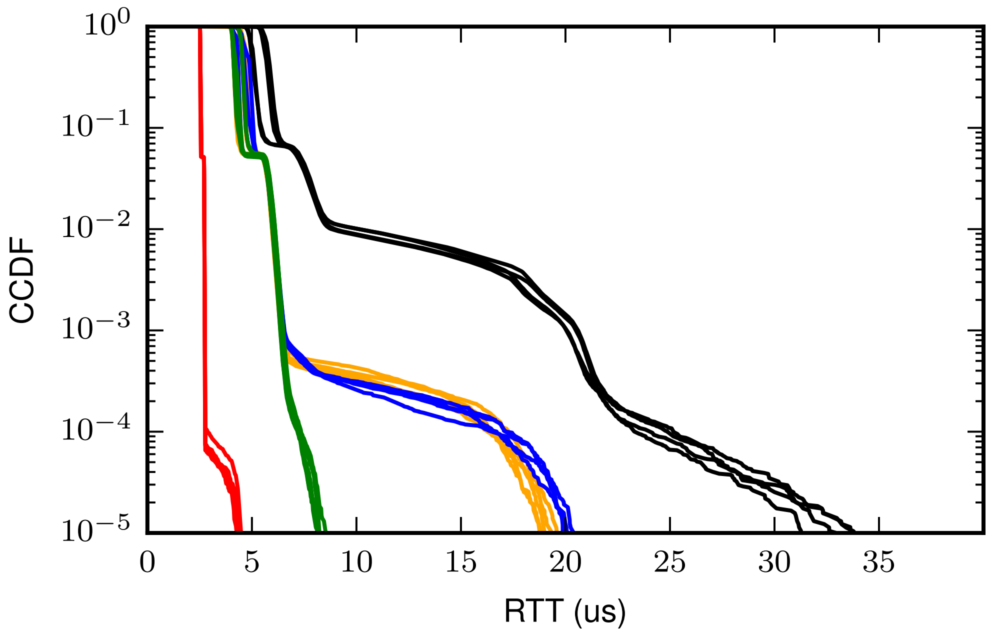
First, we establish the ground truth: Figure 3.3a shows the NF's latency CCDF as reported by Spirent (testbed configuration (3) in Figure 3.1b), which is the most precise of the considered tools (Figure 3.2). We see that, while the NF is clearly slower than the HW switch, they can both deliver relatively low jitter. We also clearly see the impact of OS configuration on latency: when considering minimum, or even median latency, it is necessary and sufficient to ensure that the local socket is consistently used; when considering tail latency, however, it is essential to further control power settings. For instance, at the 99.99^{th} percentile, the appropriate power settings reduce tail latency by a factor of 2.6, which is consistent with prior observations [80]. The "local+isolset+power" configuration has the lowest latency and jitter, with a minimum latency of 3.73 µs and a maximum latency of 10.72 µs, and a smooth CCDF near-vertical line between the two.

Next, we assess how well the software tools can measure the same NF latency: Figures 3.3b-3.3c-3.3d (testbed configuration (4) in Figure 3.1b) show how MoonGen-HW, TRex, and MoonGen-SW, respectively, report gradually noisier latency distributions. Still, both MoonGen-HW and TRex are accurate enough to capture the impact of OS configuration on NF tail latency; TRex may be off by several µs in absolute terms, but it does capture correctly the relative overhead introduced by each OS configuration. MoonGen-SW (as well as netperf, not shown), on the other hand, does not.

Measuring is a first step towards understanding the problems that cause latency variability, but when they are extremely complex or fixing them is out of reach given a service provider's restrictions, latency-mitigating techniques are the next logical step. The next chapter describes how can we know whether latency mitigation can pay off and up to how much—before we deploy our service and implement latency-mitigation techniques.

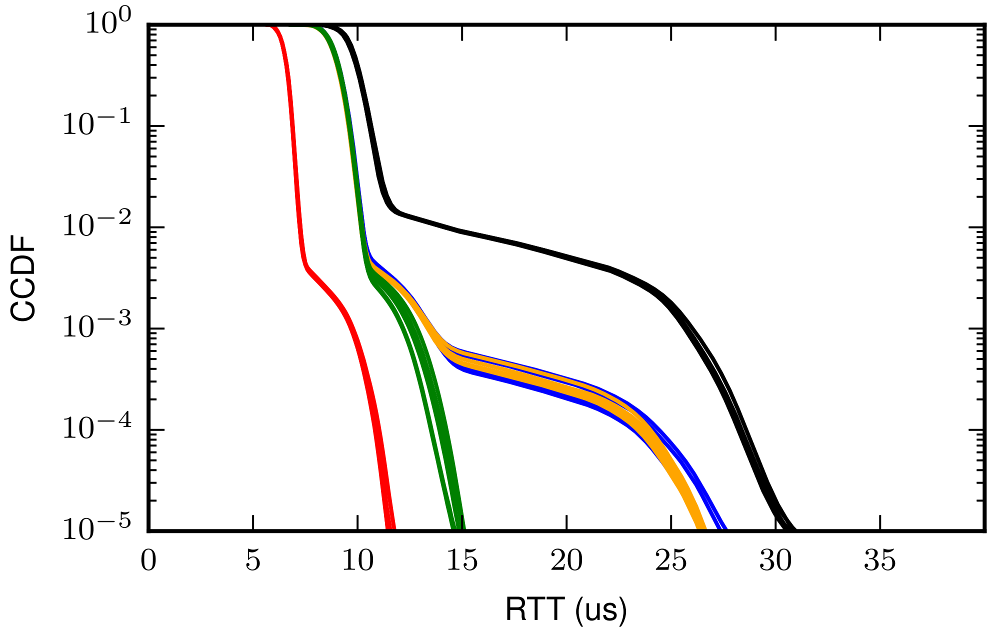


(a) client: Spirent

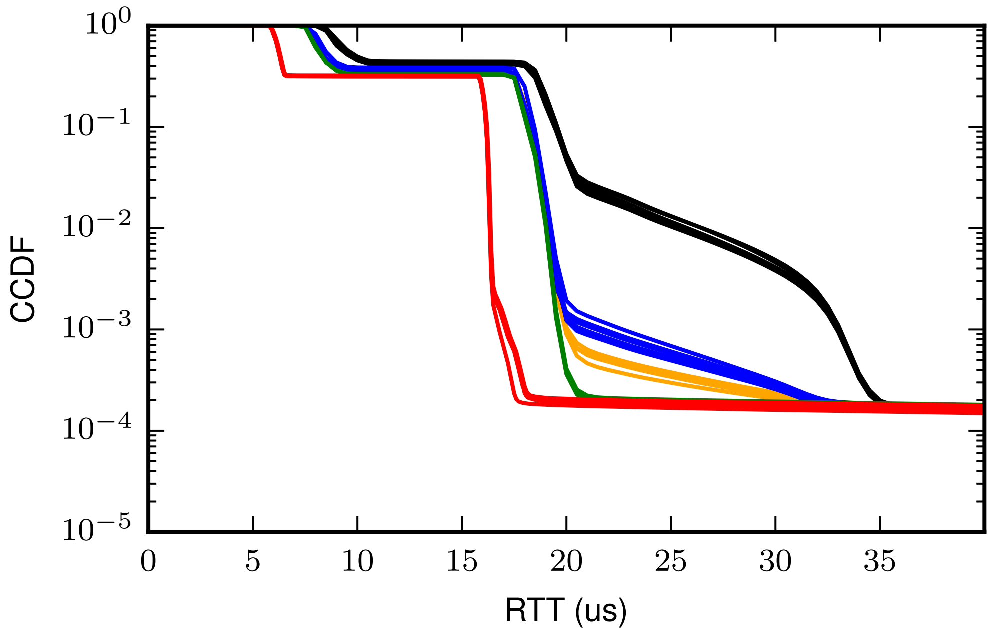


(b) client: MoonGen-HW

Figure 3.3 – Effect of different OS and application configurations of the software port-forwarding application, as measured by different client tools. The hardware switch is added for comparison.



(c) client: TRex



(d) client: MoonGen-SW

Figure 3.3 – (*cont.*) Effect of different OS and application configurations of the software port-forwarding application, as measured by different client tools. The hardware switch is added for comparison.

4 When to Hedge in Interactive Services?

In large-scale deployments of on-line data-intensive (OLDI) services, each client request typically executes on many servers in parallel. As a result, system slowdowns ("hiccups"), although rare within a single server, can interfere with a large number of client requests and cause revenue losses. Service providers have long been fighting this "tail at scale" problem by hedging, i.e., issuing redundant requests to mitigate the service-time overheads of hiccups. But what if the queuing overheads introduced by hedging are actually hurting tail latency more than the rare "hiccups"?

This chapter challenges the state-of-the-art in analyzing and developing hedging and load-balancing techniques. We contribute a new metric, intra-query jitter, in the absence of which hedging never makes sense. We suggest integrating hedging into the best load-balancing policy—pull-based load balancing with a single queue per shard—whereas existing hedging techniques are push-based and use multiple processing queues. Based on this approach, we design an ideal hedging policy and use it to gain insight into the applicability of hedging, as a general technique for OLDI services.

OLDI services, as Section 2.1 describes, involve hundreds or thousands of "leaf" nodes, each holding a part ("shard") of the data needed to answer client requests; a tier of "root" nodes receives client requests, breaks each client request into distinct queries, forwards the queries to different leaves, and waits for the slowest query to finish in order to create the final client response. This results in the tail-at-scale problem—latency variability within a single server gets amplified at scale.

Even though a lot can and has been done to reduce latency variability, completely eliminating its causes has proved elusive. In a modern cloud environment where different services unavoidably share resources, there always exist some unexpected performance "hiccups". Debugging these hiccups is notoriously difficult. For instance, there is the case of an application suffering random $12ms$ scheduling delays, because a kernel feature caused the jitter of interrupt requests to be significantly higher than the timer interval [17]; or the case of non-work-conserving scheduling, in which the kernel was throttling programs that exceeded a misconfigured purchase quota [119]. But even when performance hiccups are easy to debug, fixing them is often beyond the control of the interested parties: most service providers do not own a datacenter and do not develop their own operating systems and entire software stacks that they can easily control—yet they still offer interactive services that need to meet strict tail-latency SLOs.

Hedging masks service-time variability at the cost of extra system load (caused by the replicated queries), hence an extra queuing delay. So, if we take any standard hedging policy and any standard load-balancing policy (that tries to minimize latency without hedging), and we measure latency as a function of system load, we expect to see a tipping point: at first, hedging will outperform plain load balancing; however, for some offered system load, the cost of replicating queries will start to outweigh the benefit and the situation will be reversed. The challenge, then, is knowing *when* to hedge so as to operate before the tipping point. When a service involves coarse-grained jobs that may take tens to hundreds of seconds, it can make informed decisions about when to hedge by monitoring the progress of job execution [4–6]. These techniques, however, would not make sense in OLDI services with (sub)millisecond-timescales.

The rest of this Chapter studies *when* hedging makes sense: when does it have the potential to improve tail latency relative to plain load balancing and by how much? To answer this question, we define Idealized Hedge, an idealized hedging policy that outperforms, by construction, any implementable hedging policy. We experimentally compare Idealized Hedge to Per-Shard Queuing, which is the load balancing policy that, given our setup, yields optimal queuing time and does not hedge. This allows us to identify regimes where hedging has the potential to improve tail latency and to bound the potential improvement.

4.1 OLDI Setup

The OLDI architecture we consider is depicted in Figure 2.4. We assume that all scheduling disciplines that we explore are implemented within per-shard load balancers.

A client request typically requires accessing multiple data shards. When a root node receives a client request, it creates at least one query per distinct shard that needs to be accessed to answer the request and sends the query to a leaf node that holds that shard. Each leaf is equipped with a queue and processes queries first-come, first-served (FCFS), which is optimal for light-tailed service-time distributions [130] and used in many of today's low-latency frameworks [15, 34, 68]. Prior work shows that FCFS is the best non-preemptive scheduling policy when tail latency is the most important metric [15, 68, 80, 92, 130]. To compute the final client response, the root needs one response per distinct shard.

The *end-to-end latency* of a client request is the amount of time that elapses between the moment a root node fans-out the original request into multiple queries and the moment when the root has received the first response from each distinct shard. It is equal to the service time plus queuing delay experienced by the slowest query mandatory for aggregating the final response (the last of the *first* queries that finish on a yet-unprocessed shard).

4.2 Hedging and Load Balancing: State of the Art

In this dissertation, we consider three representative hedging policies, previously introduced in Section 2.5.2 and listed in the top three rows of Table 2.1, namely:

- Naïve Hedge [125],
- Delayed Hedge with cleanups (d-Hedge w/ CC) [28, 68], and
- Probabilistic Hedge (p-Hedge) [68].

Ideally, a hedging policy must walk the fine line between (a) reissuing too many queries and adding too much system load (hence queuing delay), and (b) not reissuing enough queries and failing to mitigate the effect of IQ-jitter on tail latency. Naïve Hedge errs toward the former (it always reissues as much as possible); Vulimiri *et al.* [125] have showed that this policy helps reduce tail latency, but only when system load is below 30% [125]. d-Hedge and p-Hedge provide knobs for controlling the added load; however, as we will see, they still do not enable a proper balance between (a) and (b), *i.e.,* they can still unnecessarily overload the system or fail to mitigate the effect of IQ-jitter, even if their knobs are properly tuned (Section 4.6).

Outside hedging, the standard approach to managing latency is load balancing (LB); hence, it makes sense to compare hedging policies against standard load-balancing policies. Here we consider four such policies, previously described in Section 2.5.3 and listed in the bottom four rows of Table 2.1, namely:

- Random,
- Join-shortest-queue (JSQ) [51],
- Per-Shard Queuing (PSQ), and
- Join-bounded-shortest-queue (JBSQ) [74].

4.3 Simulation Setup

In the results in the rest of this section we rely on discrete event simulation to compare hedging against the best of standard load-balancing policies across a large parameter space. The goal is not to evaluate precisely how these policies perform in real systems (we use different experiments for that in Chapter 5), but to understand some of their fundamental properties, *e.g.,* how does hedging compare to Per-Shard Queuing in an idealized setting (where PSQ is the optimal load-balancing policy)?

Our simulation setup mimics a simple OLDI application deployed in a medium-sized cluster:

(a) There are 100 leaves processing queries, all with the same hardware and software configuration. There are 5–500 distinct data shards, each replicated in 2, 3 or 6 leaves (depending on the experiment). Similarly to Kaler *et al.* [68], each leaf processes queries FCFS. Leaves never drop queries (in policies where the leaves store pending queries, *e.g.,* JSQ, the queue is always large enough).

(b) The root tier sends queries to each per-shard load balancer, which fans out queries to leaves according to the simulated (hedging or load-balancing) policy. Network latency (from the root to each per-shard load balancer, and vice versa) is zero.

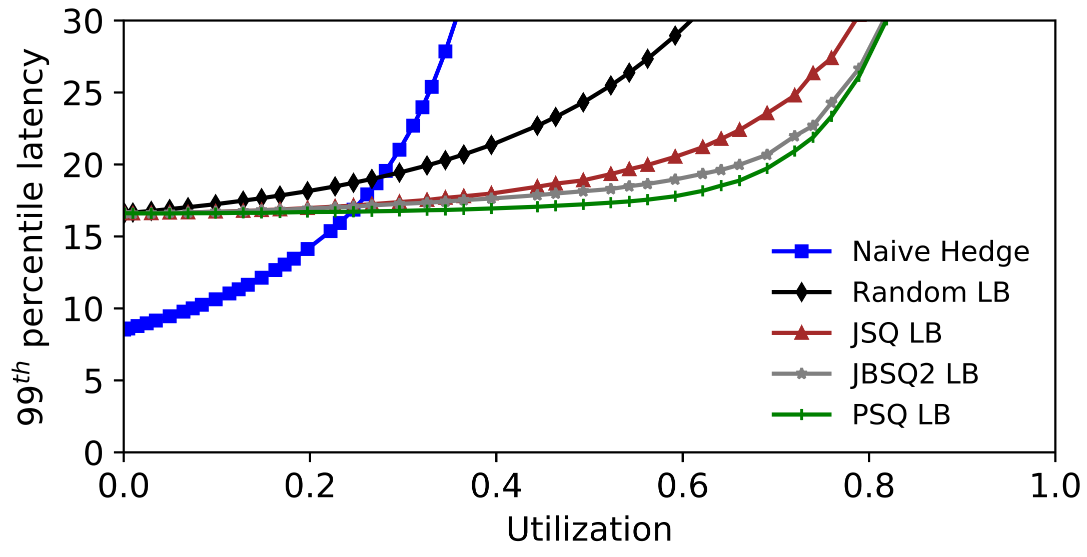
(c) Client requests arrive at the root tier following an open-loop Poisson arrival process.

(d) We model query service time S experienced by a query as the sum of two components, P and J, as per Section 2.3. Unless otherwise noted, P follows an exponential distribution (motivated by Mirhosseini *et al.* [92] as well as our system results in Chapter 5), while J (IQ-jitter) follows a bimodal distribution. This means that, in any single experiment, an instance of a query executing at a leaf node experiences a performance hiccup with some probability (*e.g.,* 10^{-3} in our running example in the following Section 4.4), while performance hiccups have the same duration (*e.g.,* $15 \times \bar{P}$ in our running example).
Across experiments, we vary hiccup probability and duration to approximate a range of real problematic system events reported in the literature: $15 \times \bar{P}$ has occurred as the result of badly configured timer intervals [17], $30 \times \bar{P}$ as a result of non-conserving job-to-core allocation [119], and $100 \times \bar{P}$ due to the operating system throttling a user-space application after it exceeded its CPU quota [119]. This particular distribution and the choice of parameters also reflects our observations from our system implementation described in Chapter 5.

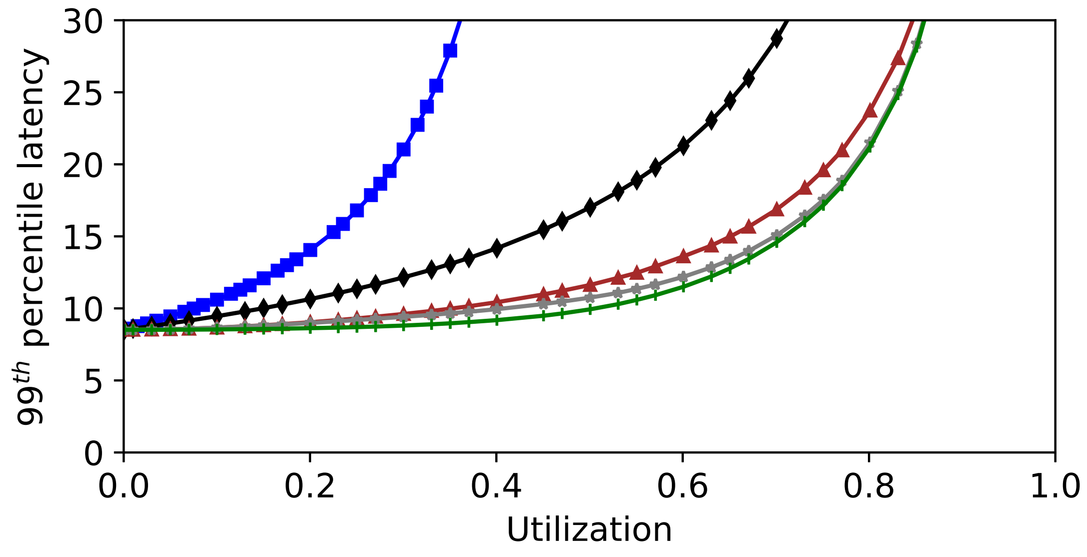
We report tail latency as a multiple of $\bar{P}$ (the average service time without IQ-jitter introduced in Section 2.3). For example, when 99^{th} percentile latency totals 20, 1% of client requests experience end-to-end latency higher than $20 \times \bar{P}$. This provides more insight than an absolute number, especially since (in these particular experiments) we are measuring a simulated, idealized setup. In reality, $\bar{P}$ is in the order of milliseconds for OLDI applications and in the order of microseconds for microservices [122].

4.4 Two Simple Observations

We start by comparing all the load-balancing policies and Naïve Hedge on our running example: Figure 4.1 shows 99^{th} percentile latency as a function of system utilization in a cluster

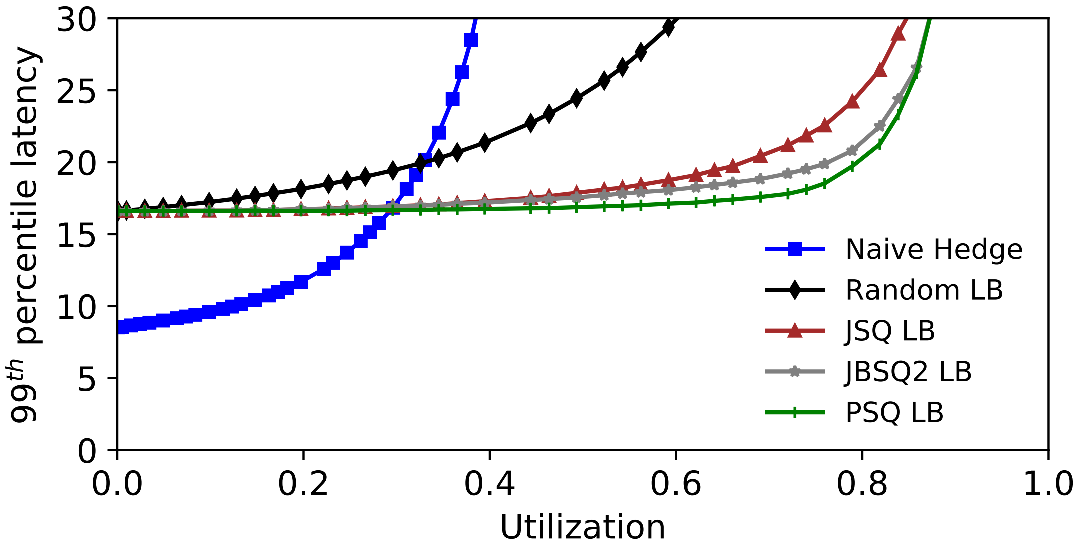


(a) IQ-jitter with hiccup probability $\frac{1}{1000}$, hiccup duration $15 \times \bar{P}$.

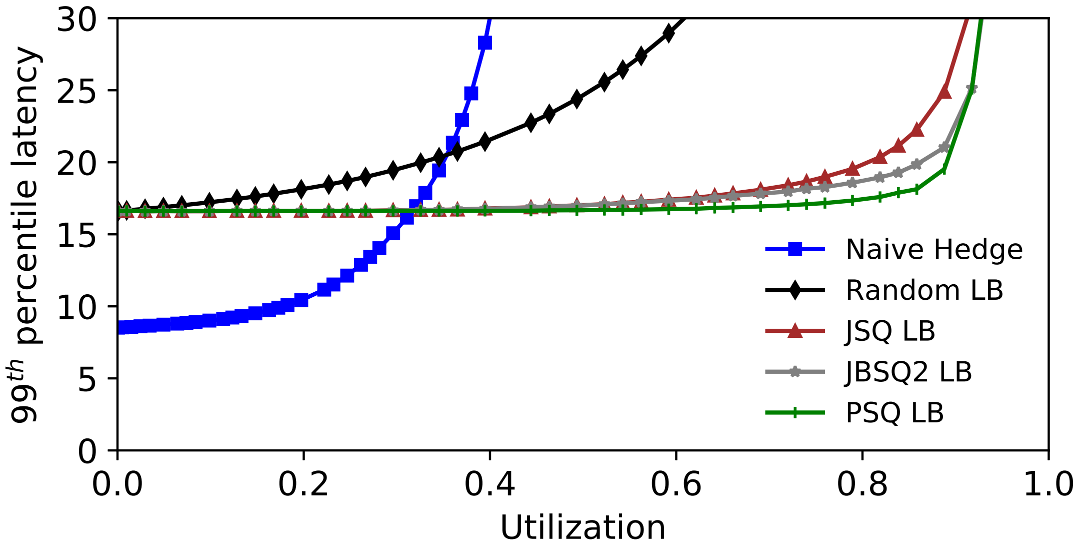


(b) No IQ-jitter.

Figure 4.1 – 99^{th} percentile latency as a function of utilization, in a cluster with 50×2 leaves, with and without IQ-jitter.



(a) 50 × 3 leaves (3 replicas per shard)



(b) 50 × 6 leaves (6 replicas per shard)

Figure 4.2 – 99th percentile latency as a function of utilization, with hiccup probability $\frac{1}{1000}$, and hiccup duration $15 \times \bar{P}$. The trend in behavior of hedging and load-balancing policies from Figure 4.1a persists even with larger number of replicas.

with 50 shards, each replicated in 2 leaves, first with IQ-jitter of hiccup probability 10^{-3} and hiccup duration $15 \times \bar{P}$ (Figure 4.1a), then without IQ-jitter (Figure 4.1b). The utilization of 0.0 naturally corresponds to zero queries per time unit, while the utilization 1.0 (or 100%) corresponds to $\frac{1}{\bar{P}}$ queries per time unit, for each replica. For example, if the mean service time without IQ-jitter equals to $\bar{P} = 1\ ms$, 1000 queries and 2000 queries per second correspond to 100% utilization with one and two replicas per shard, respectively. Note that the maximum utilization does not depend on the number of shards, because all shards are queried in parallel, but solely on the number of replicas per shard and $\bar{P}$.

We derive two simple, yet important observations:

First, Per-Shard Queuing outperforms all the other load-balancing policies in all situations. This is expected from queuing theory (given our idealized simulation setup). The performance difference is greater in the presence of IQ-jitter (Figure 4.1a), which makes sense given that system noise increases the dispersion of the service-time distribution—and more service-time dispersion creates a bigger challenge for non-centralized load-balancing policies.

Second, in the presence of IQ-jitter (Figure 4.1a), there is a clear “turning point”: At low utilization, Naïve Hedge (despite its naïveté) delivers significantly lower tail latency than any load-balancing policy. For instance, in an unloaded system, tail latency is $8 \times \bar{P}$ with Naïve Hedge and $17 \times \bar{P}$ with PSQ—close to a $2\times$ improvement. However, when utilization exceeds 25%, the load-balancing policies deliver lower tail latency (as well as higher throughput).

As a side note, in the absence of IQ-jitter (Figure 4.1b), hedging does not improve tail latency relative to load balancing, for any system utilization. This makes sense given that, in the absence of IQ-jitter, the same query always takes the same amount of time to execute on two identical leaves.

These observations persist with the larger number of replicas, as Figure 4.2 demonstrates: The main property of the graphs that changes with the growing number of replicas is the location of the “knee” of the latency curve, especially for more advanced load-balancing policies, *i.e.,* the saturation point of the system before tail latency spikes is shifted to the right in case of JSQ, JBSQ and PSQ policies. It is not surprising that load-balancing policies can do better load-balancing with the growing number of replicas, however their behavior at low loads did not drastically change comparing to the basic two-replica case, and, when compared to hedging, hedging still offers significant latency improvement in the presence of IQ-jitter. Note that we keep the number of reissued requests of Naïve Hedge to two, regardless of the total number of replicas, which still “only” halves the system capacity.

Our observations suggest that a policy that combines hedging and Per-Shard Queuing and adapts the fraction of hedged queries to system utilization, might achieve lower tail latency than either hedging or load balancing alone.

4.5 The Design of Idealized Hedge

Idealized Hedge is an idealized policy that, by construction, achieves lower latency than any implementable hedging or load-balancing policy. It has the following properties:

1. A leaf is never idle when it can serve any pending query, whether that query is currently being served by another leaf or not.
2. A leaf serves a hedged (replicated) query only when it cannot serve a non-hedged (yet-unserved) one.

The second property ensures that hedged queries never increase the queuing delay experienced by non-hedged queries. The two properties together ensure what we might call *work conservation in the presence of hedging*: no resources are idle when they could be doing useful work, and no resources are dedicated to hedging when they could be used for other work.

Idealized Hedge is not implementable because it requires perfect prediction of the completion times of all the currently executing queries: To ensure that the two properties stated above always hold, the system may need to cancel one copy of a hedged query currently executing on a leaf (so that the leaf can serve a new, non-hedged query that just arrived). To ensure that no actual policy could outperform Idealized Hedge, the system must always cancel the copy that will take longer to complete, hence the need for perfect completion-time prediction.

We simulated Idealized Hedge as follows: A per-shard load balancer maintains a queue with all the pending queries in order of arrival, and it knows the status of each leaf and which query it is processing (if busy). Moreover, if two copies of a hedged query are executing on different leaves and a new query arrives (that can be served by the same leaves), the scheduler (within the load balancer) perfectly predicts which copy will finish executing first. With this knowledge, the per-shard load balancer performs the following operations:

- Dispatches queries to the leaves in an FCFS manner using a centralized queuing discipline *within* each shard.
- Hedges a query as soon as a leaf that can serve it becomes idle.
- Cancels any copy of a hedged query if another copy finishes first (we call this a *cleanup cancellation* or CC for brevity).
- Cancels one copy of a hedged query upon arrival of a new query that can be served by the same leaf (we call this a *pre-emptive cancellation* or PC).

4.5.1 Idealized Hedge for Two Replicas

Figure 4.3 shows the finite-state machine of Idealized Hedge for a given shard that is replicated in two leaves:

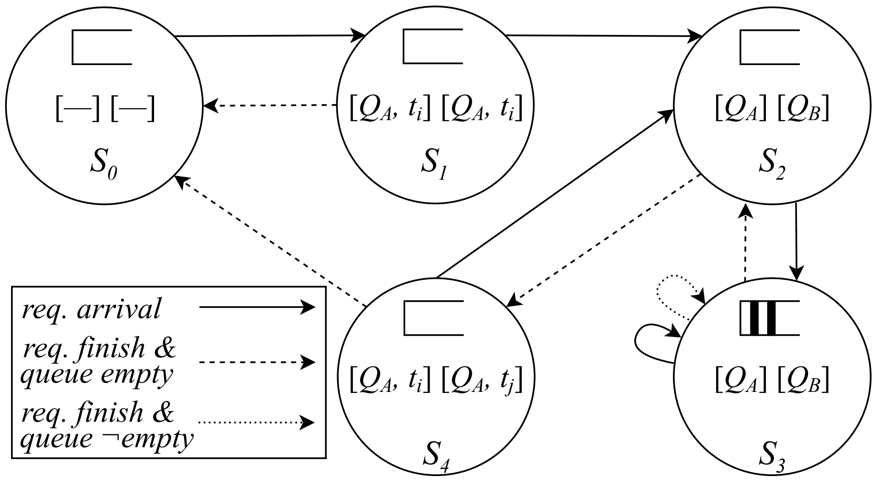

[—] [—]
S_0
$[Q_A, t_i]$ $[Q_A, t_i]$
S_1
$[Q_A]$ $[Q_B]$
S_2
req. arrival
req. finish & queue empty
req. finish & queue ¬empty
$[Q_A, t_i]$ $[Q_A, t_j]$
S_4
$[Q_A]$ $[Q_B]$
S_3

Figure 4.3 – The finite-state machine of the Idealized Hedge policy on a single shard with two replicas. The states show: whether the centralized queue is empty, whether replicas are running the same or different queries (Q), and whether the execution on the replicas started at the same time (t). This state machine also corresponds to the LÆDGE policy with both CC and PC.

S_0 Both leaves are idle. There are no pending queries.

S_1 (Initial hedging) Both leaves are serving the same query Q_A, which they started to serve simultaneously; there are no other pending queries. This state occurs on a transition from S_0, following the arrival of query Q_A.

S_2 (No hedging) The two leaves are serving different queries, Q_A and Q_B, and there are no additional pending queries. This state occurs on a transition from S_1, following the arrival of query Q_B, at which point one copy of Q_A (the one that would have finished later) was pre-emptively cancelled.

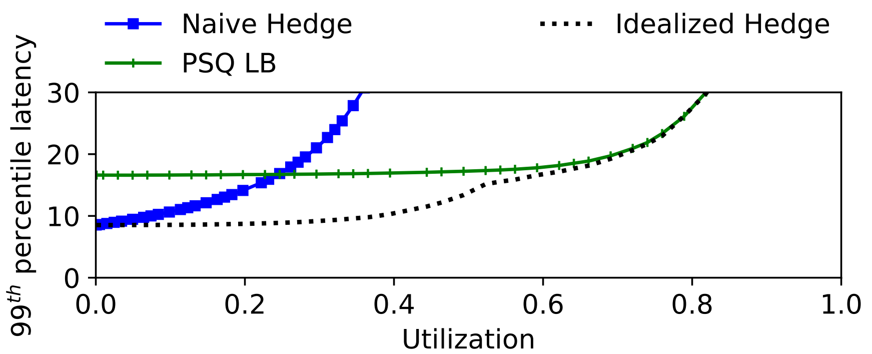
S_3 (Load balancing with a single queue) The two leaves are serving different queries, Q_A and Q_B, and there are additional pending queries. In this state, when a leaf finishes serving its query, it pulls the next one from the head of the queue (without hedging), exactly as in the PSQ policy.

S_4 (Delayed hedging) Both leaves are serving the same query Q_A, which they started to serve at different times; there are no other pending queries. This state occurs on a transition from S_2, following the completion of one query, at which point the other query (Q_A) is reissued to the otherwise idle leaf.

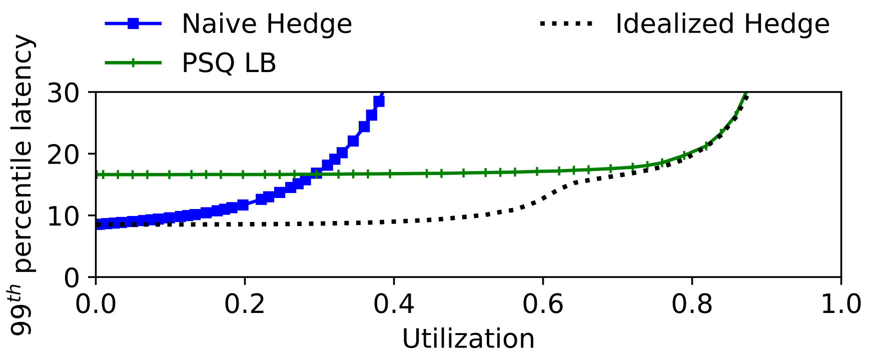
4.5.2 Generalized Idealized Hedge

Two design choices are required to extend the design of Idealized Hedge from two to any number of replicas:

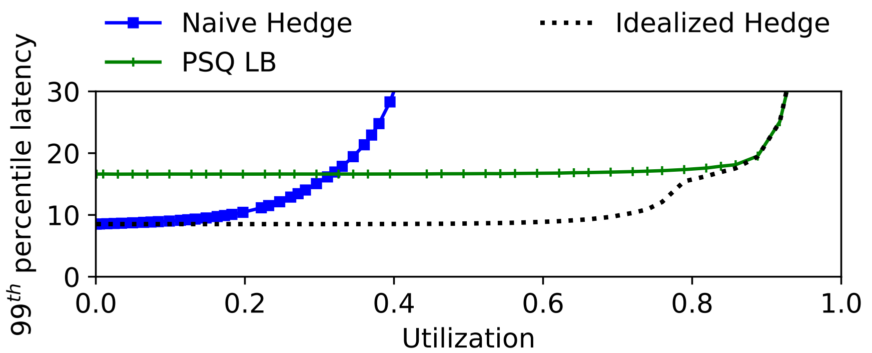
First, we need to decide on how many replicas can a query be running concurrently. We



(a) 50 × 2 leaves (2 replicas per shard)



(b) 50 × 3 leaves (3 replicas per shard)



(c) 50 × 6 leaves (6 replicas per shard)

Figure 4.4 – Demonstration of the behavior of Idealized Hedge for different number of replicas: 99^{th} percentile latency as a function of utilization, with hiccup probability $\frac{1}{1000}$, and hiccup duration $15 \times \bar{P}$. Idealized Hedge continues to achieve the best of hedging and load balancing even for larger number of replicas.

Algorithm 1: Generalized Idealized Hedge Algorithm: Request Arrival

```
$SQ[i] \leftarrow [\,],\ i \in [1,\ldots,n_{shards}]$ ;                 // Initialize a shard queue (SQ) per shard
on new request $r$ arrival
    for each shard $s$ do
        if available replicas of shard $s \geq 2$ then
            send $r$ and $r'$ to 2 random replicas of shard $s$ ;    // Replication on arrival
        else if available replicas of shard $s == 1$ then
            send $r$ to the only available replica of $s$ ;          // No replication
        else
            if $\exists$ (concurrently running identical requests on $s$) then
                $request\_to\_cancel \leftarrow -1$ ;                // Req. that would save most time
                $max\_time\_saved \leftarrow -1$ ;                   // Time saved by $request\_to\_cancel$
                // $end(r)$ is a function that perfectly predicts query completion times
                for each hedged request-pair $r_1$ and $r_2$ on $s$ s.t. $end(r_2) \geq end(r_1)$ do
                    $time\_saved \leftarrow end(r_2) - end(r_1)$ ;   // $end(r)$ is the finishing time of $r$
                    if $time\_saved > max\_time\_saved$ then
                        $max\_time\_saved \leftarrow time\_saved$ ;
                        $request\_to\_cancel \leftarrow r_2$ ;
                end
                replace $request\_to\_cancel$ with $r$ ;            // Prioritization of new requests
            else
                enqueue $r$ to $SQ[s]$ ;                             // Enqueue to the shard queue
            end
        end
    end
end;
```

decided to limit the number of concurrently run replicas to two replicas, similarly to Kaler *et al.* [68]. Since the probability of a system slowdown is low, running on more than two replicas concurrently rarely yields better results than simply limiting the policy to use just two replicas.

Second, if in total we have more than two replicas per shard, we need to decide which request to reissue with delay. This corresponds to the state transition $S_2 \rightarrow S_4$ in the finite state machine in Figure 4.3. Among all the single requests, we choose one that started the longest ago since it most-probably got delayed.

Algorithm 1 and Algorithm 2 give the details of Idealized Hedge in case of a request and response arrival, respectively. Prioritization of new requests as well as work conservation with redundancy is visible in Algorithm 1. Note that our first multi-replica assumption of keeping the number of hedged requests to two, is not fully compatible with work conservation in the presence of hedging, *i.e.,* we can have idle resources in Idealized Hedge when in total we have more than two shard replicas. For example, when a request arrives to a fully idle system, it will occupy at most two replicas, while the others will remain idle. Therefore, when we have more than two replicas, to maintain the property that *no resources are idle when they could be doing useful work*, we define simultaneously serving more than two copies of the same query not to be useful.

Algorithm 2: Generalized Idealized Hedge Algorithm: Response Arrival

```
on response $q$ arrival after request $r$ finished on node $n$ and shard $s$
    if ∃ (replica of $r$, $r'$, still running on node $n'$ and shard $s$) then
        // 2 slots on shard $s$ available, both nodes $n$ and $n'$
        if size($SQ[s]$) == 0 then
            // Delayed replication
            if number of running non-replicated requests on replicas of shard $s \geq 2$ then
                replicate the 2 oldest non-replicated requests on shard $s$ to $n$ and $n'$, respectively ;
            else if number of running non-replicated requests on replicas of shard $s == 1$ then
                replicate the only non-replicated request on shard $s$ to $n$ ;
                free-up shard $s$ on $n'$ ;
            else
                free-up shard $s$ on $n$ and $n'$ ;
            end
        else if size($SQ[s]$) == 1 then
            // Delayed replication and queuing mode
            pop a pending request $p$ from $SQ[s]$ ;
            if number of running non-replicated requests on replicas of shard $s \geq 1$ then
                replicate the the oldest non-replicated request on shard $s$ to $n$ ;
                replace $r'$ with $p$ on shard $s$, node $n'$ ;
            else
                replicate $p$ to both $n$ and $n'$ (cancel $r'$ on $n'$) ;
            end
        else
            // Queuing mode
            pop 2 pending requests, $p_1$ and $p_2$ from $SQ[s]$ ;
            send $p_1$ and $p_2$ to shard $s$ on node $n$ and $n'$, respectively (cancel $r'$ on $n'$) ;
        end
    else
        // 1 slot on shard $s$ available, only node $n$
        if size($SQ[s]$) > 0 then
            // Queuing mode
            pop a pending request $p$ from $SQ[s]$ and send it to shard $s$ on node $n$ ;
        else
            // Delayed replication
            replicate the the oldest non-replicated request on shard $s$ to $n$ ;
        end
    end
end;
```

4.6 Idealized Hedge versus State of the Art

Figure 4.5 compares Idealized Hedge against the three hedging policies (Naïve Hedge, d-Hedge, p-Hedge), as well as Per-Shard Queuing (the best load-balancing policy). The experimental setup matches that of Figure 4.1a (50 × 2 leaves and IQ-jitter with hiccup probability of 10^{-3} and hiccup duration $15 \times \bar{P}$).

As expected, Idealized Hedge outperforms the real policies, and exhibits the following behavior: at low utilization (until around 5%), when the leaves are mostly in states S_0 and S_1, it behaves like Naïve Hedge; at high utilization (from around 60%), when the leaves are mostly in state S_3, it converges to PSQ; in between, it clearly outperforms (by up to $8 \times \bar{P}$) all the real policies.

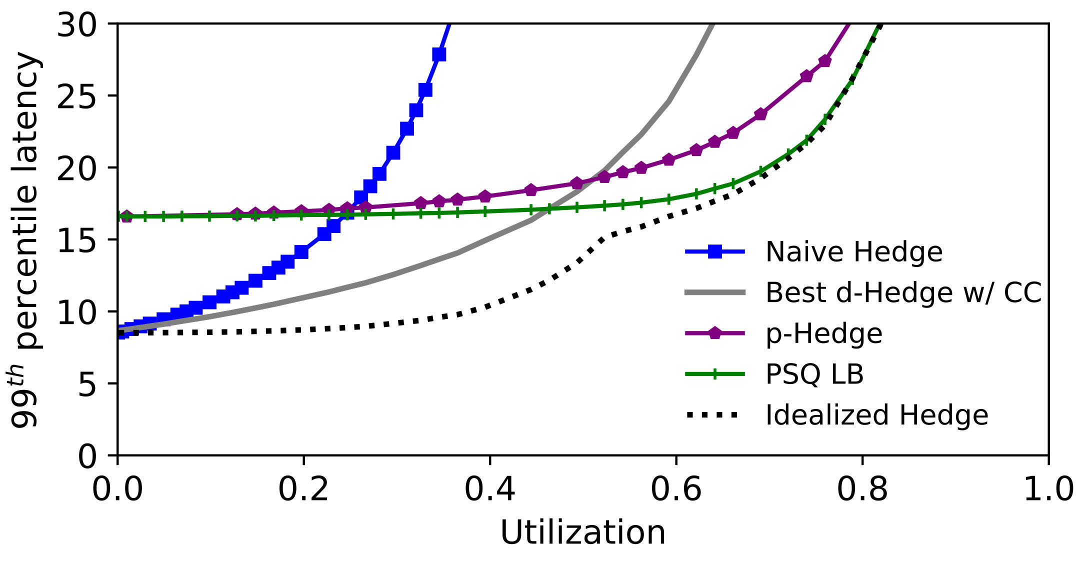


Figure 4.5 – 99th percentile latency as a function of utilization. Idealized Hedge vs. existing hedging policies and Per-Shard Queuing. Same setup as in Figure 4.1a.

The most interesting comparison is between Idealized Hedge (black dotted line) and Per-Shard Queuing (green solid line with vertical lines), because it provides an upper bound on the tail-latency reduction that can be expected from *any* form of hedging. This comparison indicates two points:

- There exists a significant utilization range (from ~60% and up, in our setup) where no real hedging policy may bring any significant benefit relative to Per-Shard Queuing.
- Outside this range, hedging *may* bring significant benefit, but the two state-of-the-art hedging policies cannot fulfill this potential. Only Naïve Hedge (blue solid line with squares) achieves all the benefit that hedging could achieve, but only at low utilization (until around 5%, in our setup). d-Hedge (gray solid line) slightly outperforms the other real policies for a short utilization range (20-30%), but it remains far from Idealized Hedge. p-Hedge (purple solid line with pentagons) is outperformed by Per-Shard Queuing in all situations.

Of course, the behavior of d-Hedge and p-Hedge depends dramatically on how their configuration parameters are tuned; we followed all the instructions in the relevant literature, and we did our best to maximize their performance.

In p-Hedge, we trained the parameters d and q, using the most successful of the methods explored in [68]: for each level of system utilization, we computed a “reissue budget” (the percentage of hedged queries in the system) using their iterative algorithm; then, for each level of system utilization, we trained d and q on a sample of latency measurements using

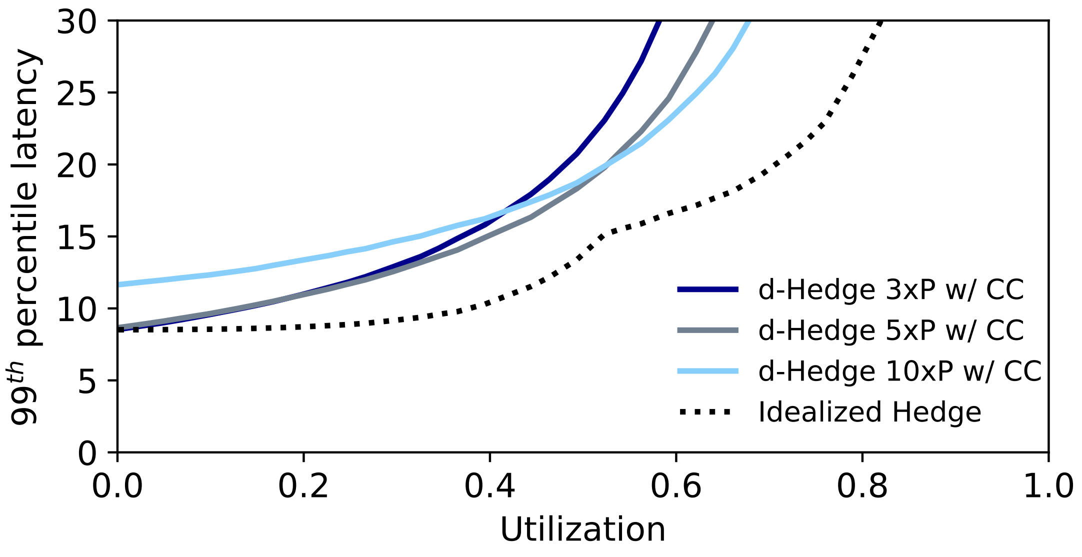


Figure 4.6 – 99^{th} percentile latency as a function of utilization. Comparison of different d-Hedge configurations with cleanup cancellations. Same setup as in Figure 4.1a.

their proposed training algorithm that accounts for queuing delays for a fixed reissue budget. We tried sampling rates up to 80%; the results we show are for a sampling rate of 60%, because increasing the sampling rate further did not significantly change the results.

In d-Hedge we implemented cleanup-cancellations, following the advice of Dean *et al.* [28]: Whenever a query response arrives, and its reissued query is still pending, the reissued query is cancelled. We experimented with a range of parameters for the delay parameter d, and the best trade-off between achieving low latency and achieving the maximal throughput is achieved for $d = 5 \times \bar{P}$, and is shown as best d-Hedge in Figure 4.5. We experimentally found that further increasing the delay d does not fully mitigate latency at lower loads, while further reducing the delay d reduces the system capacity even more. To illustrate that, Figure 4.6 demonstrates what happens when the delay is increased to $d = 10 \times \bar{P}$ or decreased to $d = 3 \times \bar{P}$.

The light blue in Figure 4.6 line corresponds to the former case ($d = 10 \times \bar{P}$) and it illustrates that increasing the delay can achieve higher system utilization before the latency spikes, but at the cost of higher latency at low loads. Further increasing the delay results in latency mitigation insignificant to the tail latency.

The dark blue line in Figure 4.6 shows what happens in the latter case, when the delay d is set to $3 \times \bar{P}$. We see that the more aggressive hedging does not achieve lower latency than the best d-Hedge, but it pays the price in lower system utilization before the latency spikes.

d-Hedge with clean-up cancellations is still far from Per-Shard Queuing and Idealized Hedge in terms of throughput, and it is still far from Idealized Hedge in terms of tail latency. Removing the clean-up cancellations further reduces the throughput of d-Hedge by roughly 40%.

All this may not prove that d-Hedge and p-Hedge could not perform any better, but it illustrates the difficulty of tuning them so as to achieve a desired balance between too little and too much hedging.

4.6.1 Beyond One Example

We now extend our observations beyond the specific setup of Figure 4.5: how much potential does hedging have to improve tail latency as the cluster size and nature of IQ-jitter vary?

We consider the following scenarios: clusters of 5×2, 50×2 and 500×2 leaves, *i.e.,* small, medium, and large; hiccup probability ranging from 10^{-1} to 10^{-5}; and hiccup duration $15 \times \bar{P}$, $30 \times \bar{P}$, and $100 \times \bar{P}$. Regarding the latter, our choice of parameters is motivated by the literature, as described in Section 4.3. We maintain the same reporting methodology which focuses on the 99^{th} percentile tail latency as a function of the utilization on the leaf servers, for an open-loop Poisson arrival process.

We summarize our results in Figure 4.7 in three sets of heat maps: (1) for hiccup duration $15 \times \bar{P}$ (Figure 4.7a), (2) for $30 \times \bar{P}$ (Figure 4.7b), and (3) for $100 \times \bar{P}$ (Figure 4.7c). Each heat map illustrates the relative improvement in 99^{th} percentile latency that Idealized Hedge brings relative to Per-Shard Queuing: the x-axis is system utilization, the y-axis is hiccup probability (on a logarithmic scale), and the intensity of each data point is the relative improvement in the 99^{th} percentile latency (so, a darker data point indicates higher potential for hedging to improve tail latency). We show improvement only if greater than 20% to focus on scenarios with significant improvement potential. Each column corresponds to a different cluster size: 5×2, 50×2 and 500×2 leaves, from left to right. The dashed horizontal line in Figure 4.7a, middle heat map (so, 50×2 leaves) corresponds to the setup of Figure 4.1a and Figure 4.5.

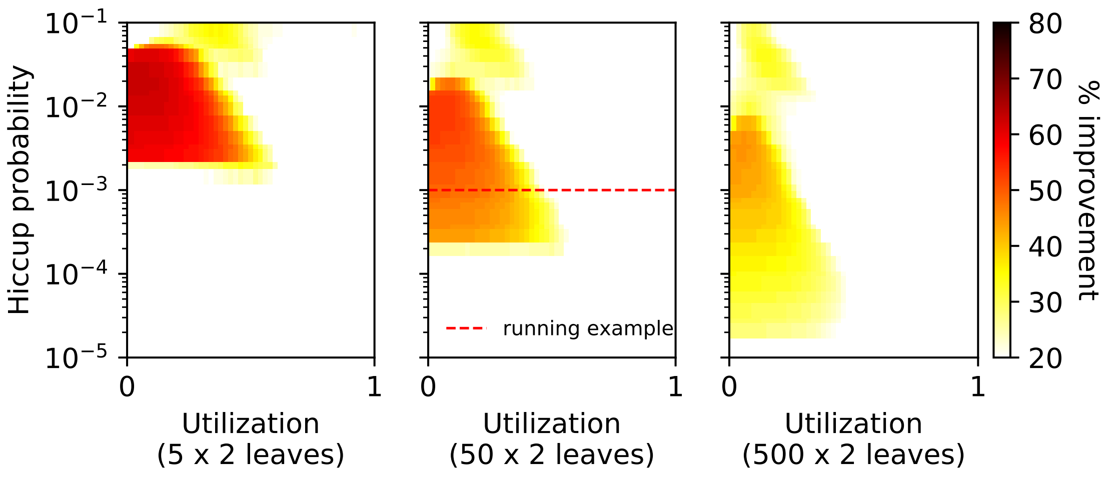
First, we observe that **hedging cannot significantly improve tail latency when system utilization exceeds** ~60**%** (the heat maps are mostly empty beyond ~60% utilization). Beyond this point, hedging improves latency by at most 20%, especially in Figures 4.7a and 4.7b, independently from cluster size and hiccup probability or duration (in Figure 4.7c the improvement above 60% utilization can occasionally reach ~40%).

The intuition is simple: as system utilization increases and leaves become busier, opportunities for hedging disappear; as a result, Idealized Hedge eventually converges to Per-Shard Queuing. The point of diminishing returns corresponds to medium-heavy utilization, where queues are starting to form, and below the point where the well-known heavy-traffic approximation determines behavior irrespective of service-time distribution [52]. In the former two scenarios (Figures 4.7a and 4.7b), between ~60% and ~80% utilization hedging provides at most 10% improvement (not visible in the heat maps), and beyond ~80% utilization, it provides no improvement at all. In the third scenario (Figure 4.7c), when the hiccup duration is by two orders of magnitude longer than $\bar{P}$, the improvement is a bit more significant, but only for very rare and very frequent hiccups.

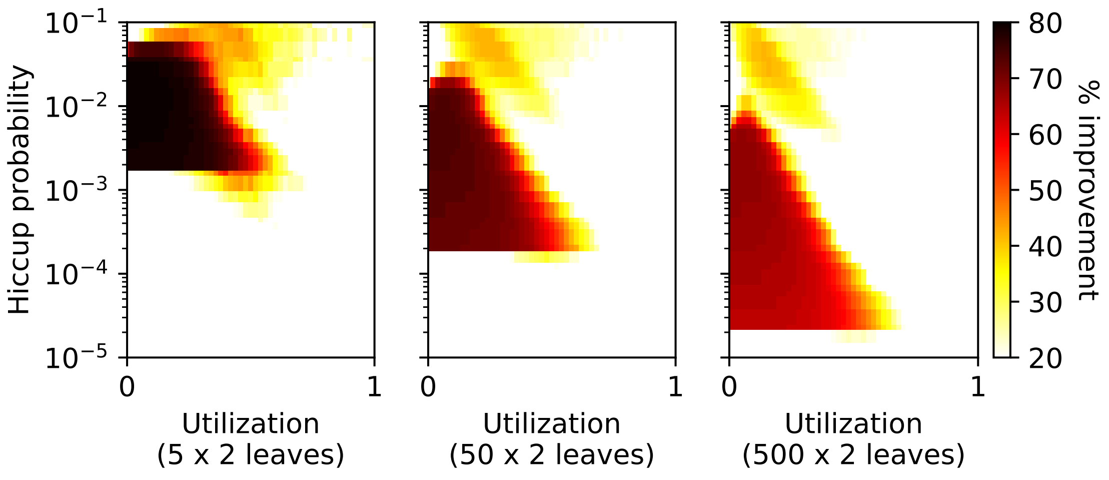
Second, **hiccup duration mostly affects the amount of potential improvement (the heat-map intensity), rather than the existence of potential improvement (the heat-map shape)**. Compare any two heat maps for the same cluster size in Figures 4.7a and 4.7b: they shade mostly the same (x, y) surface, but the heat map on the right (longer hiccup duration) is darker than the one on the left. The intuition is that, for any given cluster size, hedging can be useful only within a given hiccup probability range; outside this range, performance hiccups are either too rare or too frequent for hedging to make any difference, and this is independent of hiccup duration. The third heat map looks slightly different due to the extreme difference between $\bar{P}$ and the hiccup duration ($100 \times \bar{P}$), compared to the first two scenarios. We can see that for longer hiccup durations hedging can bring much more improvement, regardless of the cluster size, but still within the limits.

Third, **the larger the cluster size, the smaller the hiccup probability for which hedging may be useful**. For example, when the hiccup probability is between 10^{-4} and 10^{-5}, hedging may be useful only in the 1000-leaf cluster (and would be useful in larger clusters as well); in the two smaller clusters, each request needs access to fewer distinct shards, and the probability of IQ-jitter slowing down the serving of one or more of these shards becomes insignificant. Conversely, the higher the hiccup probability, the smaller the cluster size for which hedging may be useful. For example, when hiccup probability exceeds 10^{-2}, hedging may be helpful in all three clusters (but less so in the 1000-leaf cluster, where it may improve tail latency by at most 35%).

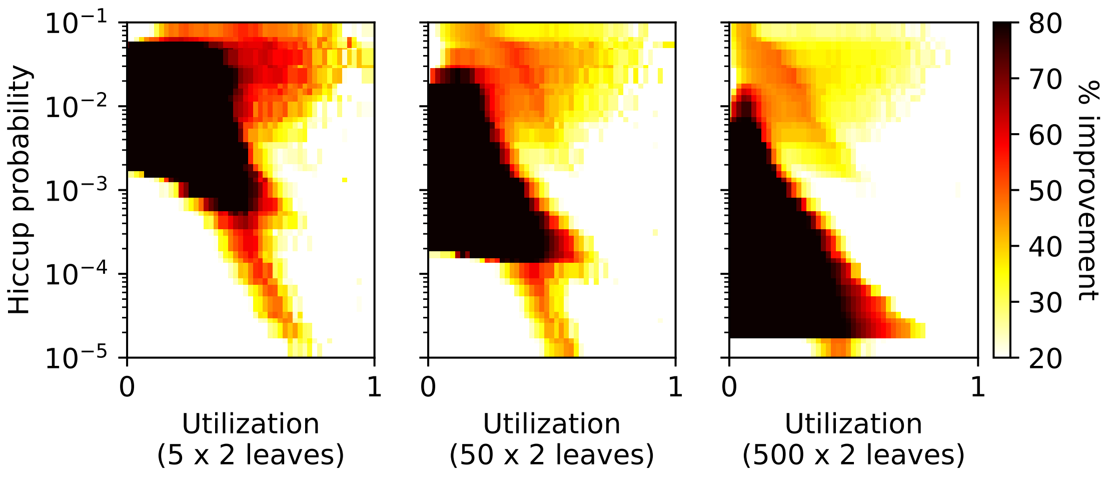
In order to apply this methodology beyond simulations we need to create a policy that can be implementable in practice. The next chapter describes how we approximate Idealized Hedge with a realistic hedging policy that can achieve a significant part of the improvements promised by the Idealized Hedge theoretical policy.



(a) Idealized Hedge, hiccup duration $15 \times \bar{P}$



(b) Idealized Hedge, hiccup duration $30 \times \bar{P}$



(c) Idealized Hedge, hiccup duration $100 \times \bar{P}$

Figure 4.7 – Heat maps showing how much Idealized Hedge improves the 99^{th} percentile latency relative to Per-Shard Queuing. x-axis is system utilization; y-axis is hiccup probability.

5 Walking the LÆDGE Between Hedging and Load Balancing

So far we have motivated the need for our lower bound, Idealized Hedge, using which we can determine whether it pays off to implement hedging-based scheduling for a particular workload through simulation and workload modeling. As we saw in Chapter 4, our Idealized Hedge relies on a perfect predictor of query completion times in order to pre-empt duplicated requests; replacing such a predictor with a good-enough heuristic, or altogether removing the need for it, brings us to a solution that can be implemented in practice.

This chapter goes beyond estimating the theoretical benefits of hedging-based policies and introduces an implementable policy, LÆDGE, that can reach as much as half of the hedging potential that we quantified in the previous chapter. In this chapter we evaluate LÆDGE in our discrete-event simulator using the same range of workloads from Chapter 4 (later in Chapter 6 we show that our approach also works in a real system).

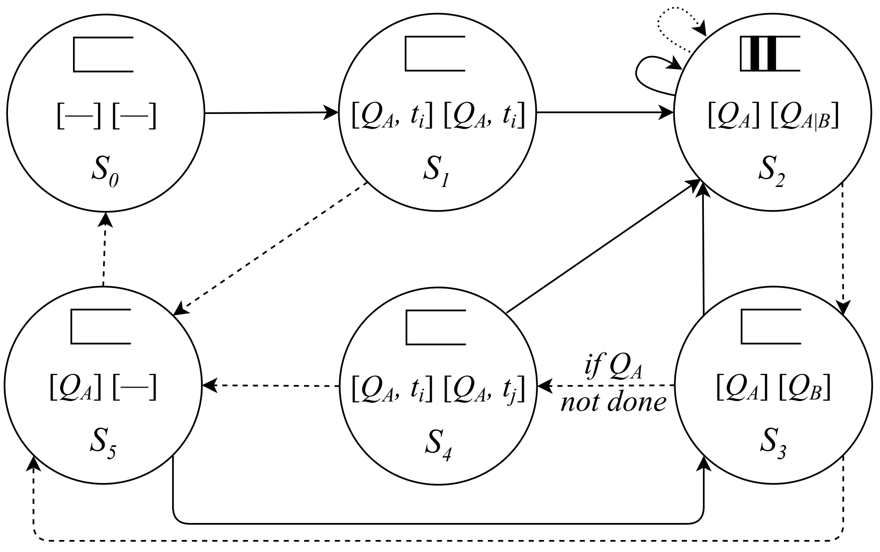
5.1 Design of Load-Aware Hedge

We propose **Load-Aware Hedge (LÆDGE)**, which has the following properties:

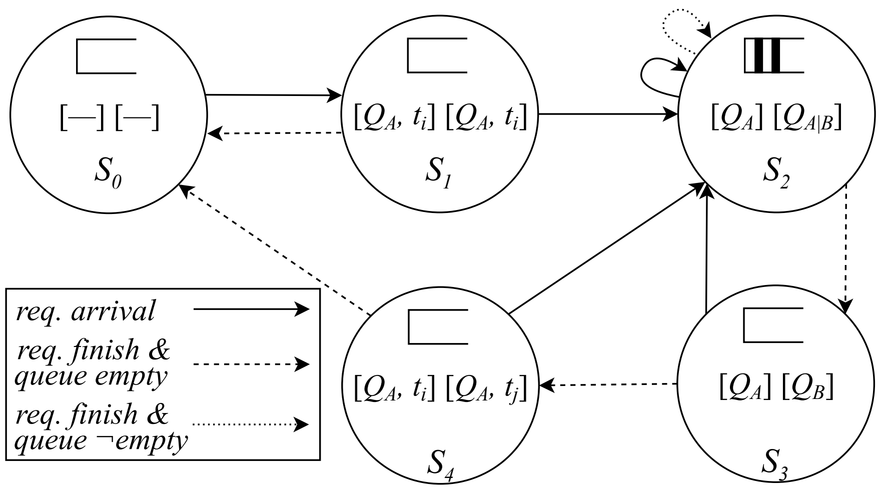
- A leaf is never idle when it can serve a pending query currently being served by at most one other leaf.
- A leaf *starts* to serve a hedged query when it cannot serve a non-hedged (yet-unserved) one.

Unlike Idealized Hedge, the plain variant of LÆDGE does not guarantee that a leaf serving a hedged query could not be serving a non-hedged one at all times. This would require both cleanup cancellation (CC), which corresponds to cancelling the replicated query after the original one finishes, and pre-emptive cancellation (PC), which corresponds to speculatively cancelling one of the hedged queries while both of them are running in parallel. LÆDGE only guarantees that when a leaf *starts* to serve a hedged query it could not be serving a non-hedged one – this only requires knowledge of all pending queries in the system.

We implemented three variants of LÆDGE, which differ mostly in the type of query cancellation that they require:



(a) Plain LÆDGE



[—] [—]
S_0
[Q_A, t_i] [Q_A, t_i]
S_1
[Q_A] [$Q_{A|B}$]
S_2
req. arrival
req. finish & queue empty
req. finish & queue ¬empty
[Q_A, t_i] [Q_A, t_j]
S_4
[Q_A] [Q_B]
S_3

(b) LÆDGE with cleanup cancellation

Figure 5.1 – The finite-state machines of the LÆDGE and LÆDGE-with-CC policies on a single shard with two replicas.

1. **Plain LÆDGE (no cancellations):**
   Similar to Per-Shard Queuing, a per-shard load balancers dispatches queries to leaves from a per-shard queue; similar to Idealized Hedge, a leaf serves both non-hedged and hedged queries, prioritizing the former. There are no query cancellations. An efficient implementation of this policy (as well as other LÆDGE policies and Per-Shard Queuing) requires a single queue per shard with µs-scale round-trip latency to the

leaves, which is commonly found in datacenter environments [41,74]. Figure 5.1a shows the corresponding state machine.

2. **LÆDGE with CC :**
   This policy augments plain LÆDGE with cleanup cancellations (CCs), *i.e.,* all copies of a hedged query are cancelled when another copy finishes executing first. An efficient implementation of this policy requires a low-overhead mechanism for interrupting a query and cleaning up its side effects. Whether such a mechanism exists or not depends on the application itself (for instance, Boucher *et al.* [20] enabled efficient µs-scale cancellations for microservices [2,47,91] written in Rust). Figure 5.1b shows the state machine of LÆDGE with CC.

3. **LÆDGE with CC+PC :**
   This policy adds pre-emptive cancellations (PCs), *i.e.,* one copy of a hedged query is cancelled when a new query arrives that can be served by the same leaves. The state machine is similar to that of Idealized Hedge, shown in Figure 4.3. The only difference is the lack of perfect completion-time prediction; instead, this policy cancels the copy that started executing most recently. One can imagine a more sophisticated heuristic that leverages prior knowledge of the service-time distribution, but we do not explore this.

Like in Idealized Hedge, all cancellations in our discrete-event simulator are zero cost, in the sense that they introduce no extra processing delay and no extra communication between the centralized scheduler and the leaves. While this is not a realistic assumption, it allows us to analyze the upper bound on the potential gains of cancellations. In other words, if zero-cost cancellations do not yield notable benefits, neither will the realistic ones.

5.2 LÆDGE versus Idealized Hedge

Let us compare LÆDGE with the best performing policies that we have analyzed so far. Figure 5.2 shows utilization on the x-axis, and 99th percentile end-to-end latency on the y-axis, normalized by mean service time $\bar{P}$. The experimental setup matches that of Figures 4.1a and 4.5. The new lines in Figure 5.2 are the pink starred line, that represents plain LÆDGE, and the purple line with crosses, that represents LÆDGE with cleanup cancellations. Figure 5.2 compares the two LÆDGE variants against Idealized Hedge, as well as the best existing real policies (among the simulated ones): d-Hedge was the best so far up to ~40% utilization, and Per-Shard Queuing beyond that.

First, we observe that plain LÆDGE reduces the gap to Idealized Hedge from at most $7.5 \times \bar{P}$, to at most $3.8 \times \bar{P}$ (while hiccup duration is $15 \times \bar{P}$, in this setup). At low utilization (up to 15%), it behaves like the best existing real alternative (which happens to be d-Hedge). From some point on (~50%), it converges to Per-Shard Queuing. In between 5% and 50% utilization, it outperforms the best of the existing policies, closing the average gap to Idealized Hedge from $4.94 \times \bar{P}$ to only $2.16 \times \bar{P}$.

Second, we observe that adding cleanup cancellations to LÆDGE improves tail latency only marginally (given our assumption of zero-cost cancellations, it could not increase it). Note that the best hedging policy, d-Hedge with cleanup cancellations, performs significantly worse without cleanup cancellations. Since d-Hedge is a push-based policy, it needs to decide where to run the original query at the time of its arrival. Also, it needs to decide where to run the replicated query after the delay *d* has expired. In both cases, queries need to first queue in the leaf nodes for an unknown amount of time. Queuing delays of d-Hedge are therefore higher than that of pull-based policies. The intuition with cancellations is that cleanup cancellations mitigate this by reducing the processing time added with hedging and they shorten the queues in the leaves. LÆDGE is a pull-based policy designed to hedge queries only when the current load allows it, thus the lack of cancellations does not affect it as much. At lower utilizations, CCs are not crucial for LÆDGE because there is always sufficient capacity in the system, while at higher utilizations, CCs are less important because there are fewer hedged queries executing in the system, since LÆDGE adapts to the system load.

Interestingly, adding pre-emptive cancellations to LÆDGE-with-CC *increases* tail latency at some utilization levels. The next section dives in deeper into this problem.

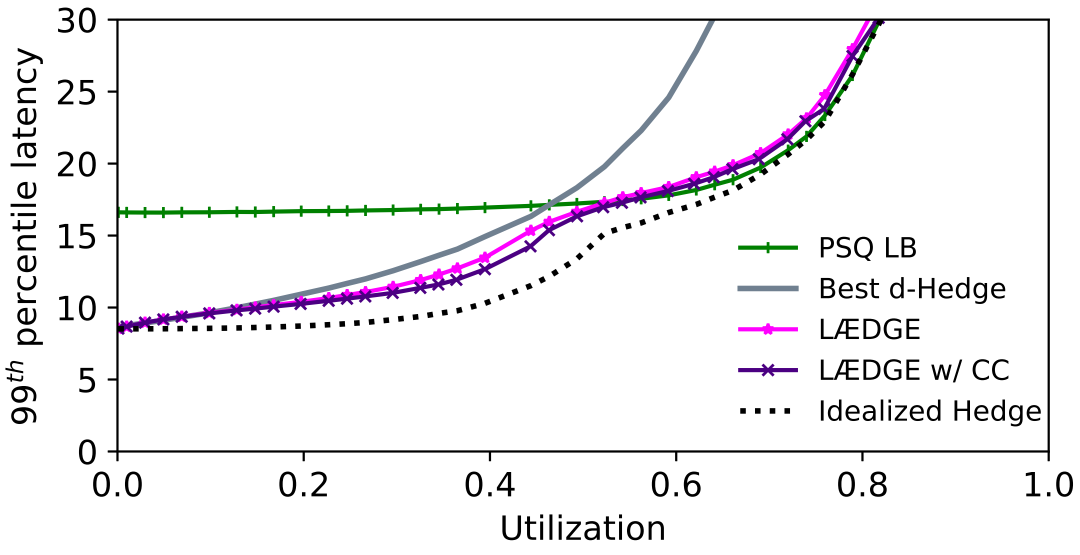


Figure 5.2 – 99th percentile latency as a function of utilization. LÆDGE flavors vs. Idealized Hedge and the best existing policies. Same setup as in Figure 4.1a.

5.2.1 To Cancel or Not to Cancel

Surprisingly, we observe that cancellations do not significantly help LÆDGE, and may actually hinder it.

We demonstrate that in Figure 5.3, which uses the same experimental setup as Figure 5.2 and shows all three variants of LÆDGE and Idealized Hedge. The golden line with hexagons shows the result for LÆDGE with both cleanup and pre-emptive cancellations. We have already seen that cleanup cancellations improve the tail latency of LÆDGE only marginally but, interestingly, Figure 5.3 shows that adding PCs to LÆDGE-with-CC *increases* tail latency at some utilization levels, starting from ~30%. It turns out that cancelling the wrong copy of a hedged query (the one that would have finished first) is an expensive mistake; without any sophisticated completion-time predictors one is better off not cancelling at all.

To better understand these results, we completed a careful analysis of PCs and their effect on tail latency. We define the "accuracy" of our LÆDGE policy with both CC and PC as the proportion of PCs that correctly cancel the copy that would finish later. Overall, our policy achieves > 99% accuracy—this is unsurprising given the rarity of IQ-jitter events. However, once we consider only the PCs where at least one copy of the cancelled query experiences IQ-jitter, our policy performs notably worse. For instance, at 40% system utilization, when transitioning from S_1 to S_2 (*i.e.,* from the state with hedged queries that started at the same time to the state with no hedging) in the presence of IQ-jitter, our policy achieves ~50% accuracy. In this case the queries started at the same time and we have no extra knowledge about them—so we randomly cancel one of them with 50% probability of it being the right one (50% out of the total number of times when IQ-jitter stalled one of the queries while in state S_1). When transitioning from S_4 to S_2 (*i.e.,* from the state with hedged queries that started at different times to the state with no hedging), also at 40% system utilization, accuracy drops to ~19% (out of the total number of times when IQ-jitter occurred in S_4). Our policy does not attempt to predict whether an IQ-jitter event is likely to have occurred and simply picks the most-recently started copy. In the common case this is the right thing to do because the query that started later would likely finish later (since the same queries running on replicas of the same shard *typically* finish after a similar amount of time). In rare cases when a query that stated first experiences IQ-jitter, our policy makes the wrong decision. We tried a number of different heuristics including reverting the decision and cancelling the older query when transitioning from S_4 to S_2: the heuristic that we chose yields better results than the other ones we tried since it makes the right decision in the common case without IQ-jitter.

The bottom line is that cancellations pay off only if we can assume a good predictor of performance hiccups due to system events—while this may sometimes be possible according to Hao *et al.* [56], LÆDGE was designed in the absence of such assumptions.

5.3 Generalized LÆDGE Design

Extending the design of LÆDGE to an arbitrary number of replicas (*i.e.,* more than two replicas) is relatively straightforward: We follow the design principles from Idealized Hedge in Subsection 4.5.2 and (a) limit the number of simultaneously running hedged requests to two, and (b) prioritize replication of the requests that have been running for a longer time, when there are multiple choices available.

Algorithm 3 presents the design of the plain LÆDGE policy, *i.e.,* LÆDGE without cancellations,

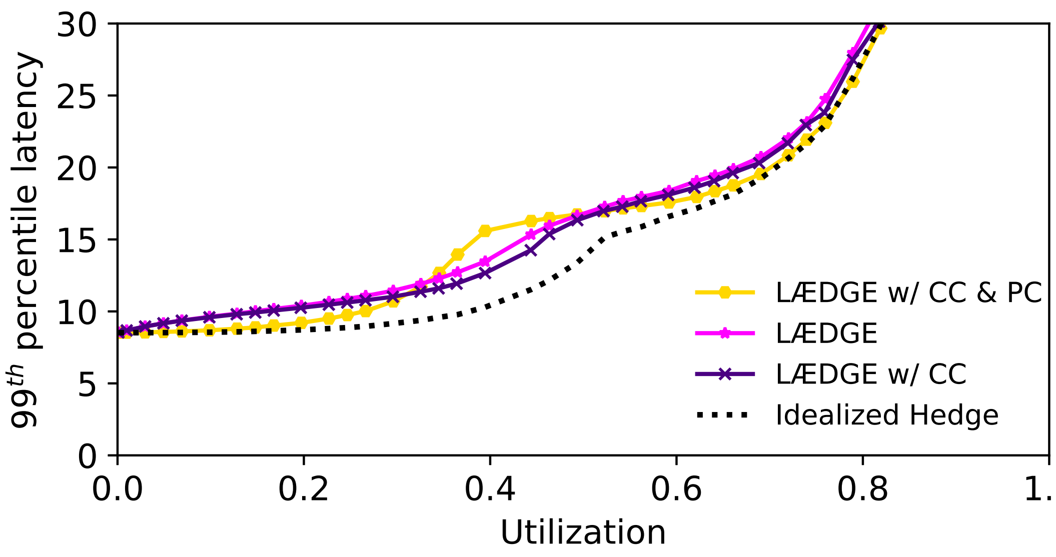


Figure 5.3 – 99^{th} percentile latency as a function of utilization. Comparing LÆDGE with CC+PC vs. other LÆDGE flavors and Idealized Hedge. Same setup as in Figure 4.1a.

in the event of both request and response arrivals. Algorithm 3 is much simpler than the corresponding Algorithm 1 and Algorithm 2, because it does not include preemption of already running requests. The other elements of Idealized Hedge design are present in LÆDGE as well: replication on arrival only when resources are available, queuing-up requests per shard, prioritizing pending enqueued requests over the replicated ones (to the extent possible without implementing the cancellations), and, finally, delayed replication of already running requests once the resources become available.

Generalized LÆDGE with cleanup cancellations behaves like LÆDGE in Algorithm 3 on request arrival, and like Idealized Hedge in Algorithm 2 on response arrival. Furthermore, the generalized algorithm of LÆDGE with cleanup and pre-emptive cancellations is identical to that of Idealized Hedge, with the main difference being the implementation of the function $end(request)$; in case of Idealized Hedge this function is the implementation of the non-existent perfect predictor of query completion times, while in case of LÆDGE with CC+PC this is the implementation of our simple and rather accurate heuristic.

Figure 5.4 shows that LÆDGE continues to achieve a significant part of the tail latency reduction of Idealized Hedge, even for the larger number of replicas. The simulation behind the Figure 5.4 is set up the same way as that in Figure 4.4 in the previous chapter. As we increase the number of replicas from 2 to 6 we notice the following:

First, adding cleanup cancellations brings around 3× more tail latency reduction with 6 compared to 2 replicas, but only for a very narrow range of utilization.

Second, as in Idealized Hedge, the tail latency curves of all the LÆDGE policies are shifted

Algorithm 3: Generalized LÆDGE

```
SQ[i] ← [ ], i ∈ [1,...,n_shards] ;                    // Initialize a shard queue (SQ) per shard
on request r arrival
    for each shard s do
        if available replicas of shard s ≥ 2 then
            send r and r' to 2 random replicas of shard s ;   // Replication on arrival
        else if available replicas of shard s == 1 then
            send r to the available replica ;                  // No replication
        else
            enqueue r to SQ[s] ;                               // Enqueue to the shard queue
        end
    end
end;
on response q arrival from node n serving shard s
    if size(SQ[s]) > 0 then
        pop a pending request p from SQ[s] ;
        send p to n ;                                          // No replication
    else
        if ∃ a non-replicated unfinished request r_i on shard s then
            replicate the oldest r_i to node n ;               // Delayed replication
        end
    end
end;
```

to the right; that is, in the case of a larger number of replicas, LÆDGE also achieves higher utilization before the latency spikes. For example, in Figure 5.4c the tail latency stays the same as the minimal latency of Naïve Hedge and the latency of Idealized Hedge, all the way up to 60% system utilization.

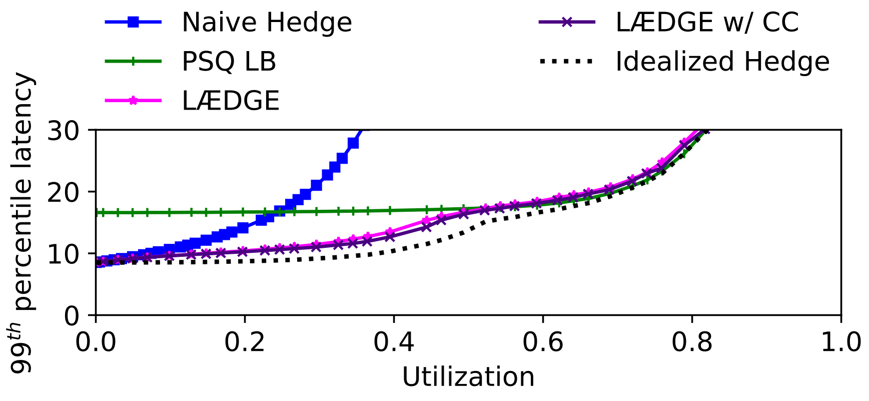
Third, with the larger number of replicas we can see that the gap between Idealized Hedge and the LÆDGE policies gets a bit wider, before LÆDGE quickly converges back to Idealized Hedge and Per-Shard Queuing. This is due to the lack of pre-emptive cancellations in the shown LÆDGE policies (we omit LÆDGE with CC+PC due to the negative result analyzed in Section 5.2.1).

5.4 Beyond One Example

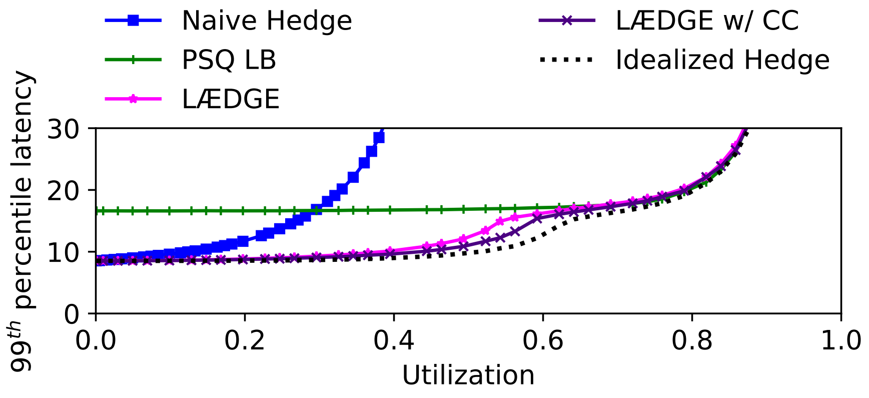
We now extend our observations beyond the specific setup of Figures 5.2 and 5.3: how well does LÆDGE fulfill the hedging potential as the cluster size and nature of IQ-jitter vary?

We consider the same scenarios as in Section 4.6.1 and summarize our results in a similar set of heat maps (Figure 5.5). To assess how well LÆDGE approximates Idealized Hedge, we have to compare the heat maps. To simplify the comparison, we introduce Table 5.1, which summarizes the comparison of figures where hiccup duration is $15 \times \bar{P}$ (Figures 4.7a, 5.5a, 5.5c, and 5.5e), and $30 \times \bar{P}$ (Figures 4.7b, 5.5b, 5.5d, and 5.5f).

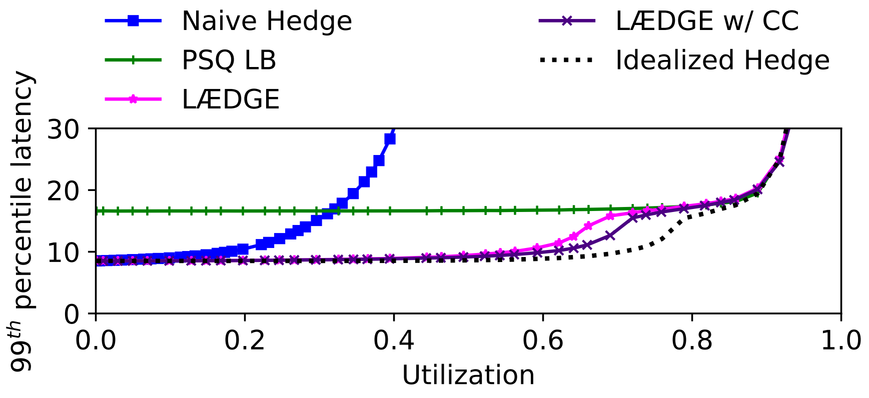
Table 5.1 shows the percentage of the "surface" of each heat map that indicates improvement



(a) 50 × 2 leaves (2 replicas per shard)



(b) 50 × 3 leaves (3 replicas per shard)



(c) 50 × 6 leaves (6 replicas per shard)

Figure 5.4 – Demonstration of the behavior of LÆDGE and LÆDGE with CC for different number of replicas: 99^{th} percentile latency as a function of utilization, with hiccup probability $\frac{1}{1000}$, and hiccup duration $15 \times \bar{P}$.

		Idealized Hedge			LÆDGE			LÆDGE with CC			LÆDGE with PC		
	% reduction w.r.t. PSQ→	> 20	> 30	> 40	> 20	> 30	> 40	> 20	> 30	> 40	> 20	> 30	> 40
$15 \times \bar{P}$	5 × 2 leaves	26.3	**19.6**	15.6	9.8	**8.3**	6.4	13.6	11.8	9.6	14.4	11.6	10.1
	50 × 2 leaves	29.9	**21.5**	15.4	13.4	**9.8**	3.8	16.8	12.2	5.2	15.9	12.3	9.2
	500 × 2 leaves	31.9	**17.9**	4.4	12.6	**4.2**	0.2	14.8	5.2	0.2	15.5	8.0	0.9
$30 \times \bar{P}$	5 × 2 leaves	36.9	**28.1**	22.3	9.6	**7.6**	6.2	18.9	13.4	11.8	21.8	15.6	12.6
	50 × 2 leaves	40.5	**29.7**	22.5	15.2	**12.6**	11.1	22.6	17.4	15.5	23.3	18.2	14.5
	500 × 2 leaves	43.1	**33.4**	26.7	20.9	**19.0**	16.7	27.2	22.2	20.5	25.1	21.2	17.9

Table 5.1 – Percentage of the total surface (in Figure 5.5 and Figure 4.7) that has more than 20%, 30% and 40% reduction in 99^{th} percentile tail latency over Per-Shard Queuing for the workloads with the hiccup duration of $15 \times \bar{P}$ and $30 \times \bar{P}$.

above a certain threshold (20%, 30% and 40%). For instance, consider the row that corresponds to hiccup duration $30 \times \bar{P}$ and cluster size 50×2 leaves, and the two columns that correspond to Idealized Hedge and LÆDGE, with > 30% latency improvement; the two cells where this row and columns intersect indicate that Idealized Hedge achieves such improvement for 29.7 % of the data points, while LÆDGE does for 12.6 % of data points.

To summarize, LÆDGE fulfills as much as half of the hedging potential, depending on the setup. Consider again the columns that correspond to Idealized Hedge and LÆDGE with > 30% latency improvement, and compare the values of these two columns that are in the same row; LÆDGE improves from 23% to 56% of the data points that are improved by Idealized Hedge (*i.e.,* that could possibly be improved through hedging). In general, LÆDGE is closer to Idealized Hedge for the medium and large clusters and the longer hiccup duration.

On a side note, LÆDGE does not deteriorate lower latency percentiles compared to Per-Shard Queuing (including the median), but, as we saw, it improves the tail. The rare hiccups that we analyzed, however, only influence the tail — not, for example, the 50^{th} percentile latency.

Finally, we should note that, out of curiosity, we experimented with two more types of application-independent noise (other than bimodal): exponential and bimodal+exponential. For the former, not even Idealized Hedge can improve tail latency, Per-Shard Queuing is the best policy, and LÆDGE performs almost the same as Per-Shard Queuing; this is not surprising, given that hedging was invented to deal with noise due to unpredictable system events, which is better modeled with a bimodal distribution. For the latter, the results were almost identical to the ones we got for bimodal noise.

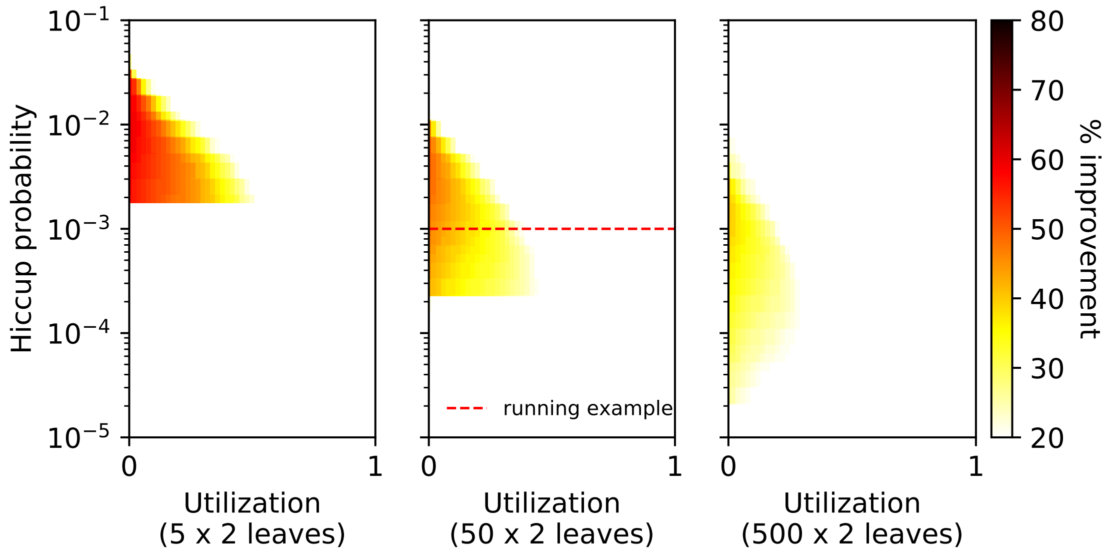
Next, we examine the impact of cancellations—do they, in general, improve tail latency enough to be worth the effort?

5.4.1 Are Cancellations Worth the Effort?

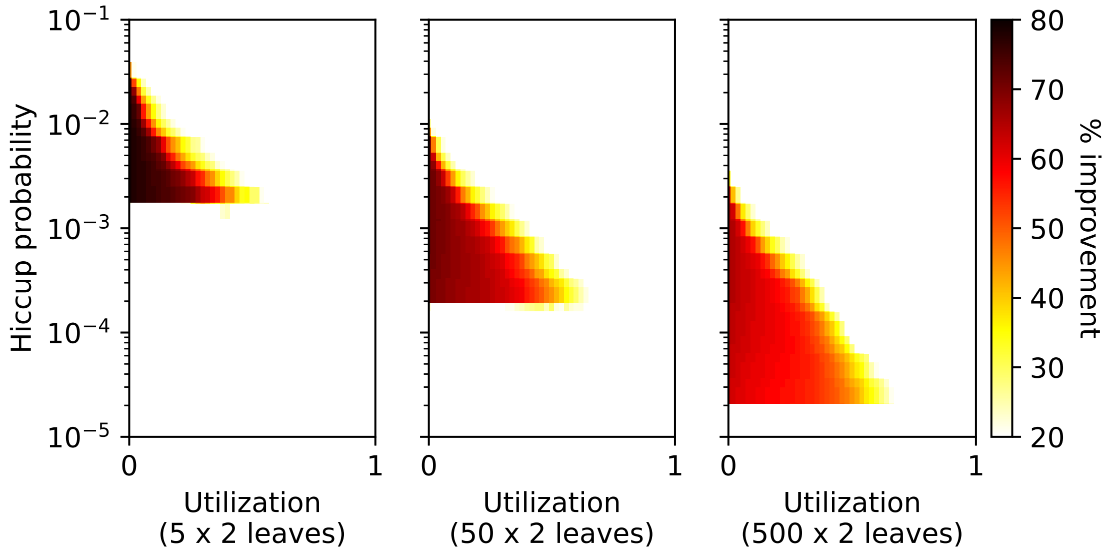
First, our sensitivity analysis confirms that cleanup cancellations (CCs) improve LÆDGE only marginally. At lower utilizations, CCs are not needed because there is always sufficient capacity in the system, while at higher utilizations CCs are less important because there are fewer hedged queries executing anyway due to the load-awareness of LÆDGE. While Figure 5.2 made this clear for one specific setup, Figure 5.6 demonstrates it for the full parameter space. The intensity of each data point in Figure 5.6 shows the improvement in 99^{th} percentile latency that LÆDGE with CC brings relative to LÆDGE without any cancellations; the darker the area the bigger the improvement of the tail latency. To quantify this improvement consider the columns of Table 5.1 that correspond to LÆDGE and LÆDGE with CC; on average, LÆDGE with CC offers the same improvement to 2.3% more data points than LÆDGE when hiccup duration is $15 \times \bar{P}$, and to 5.6% more data points than LÆDGE when hiccup duration is $30 \times \bar{P}$.

Second, our sensitivity analysis confirms that pre-emptive cancellations (PCs) have a marginal impact on LÆDGE, which is sometimes positive and sometimes negative. Similarly to Figure 5.6 that evaluates the cleanup cancellations. Figure 5.7 shows the impact of *pre-emptive* cancellations on the full parameter space (beyond our running example from Figure 5.2). The intensity of each data point in Figure 5.7 shows the improvement in 99^{th} percentile latency that LÆDGE with CC+PC brings relative to LÆDGE with CC; areas where the improvement is positive are grey, while areas where it is negative (*i.e.,* PCs increase tail latency) are red. While the impact is mostly marginally positive, there exist a few areas of highly negative impact. We have already discussed the intuition in Section 5.2; this graph indicates that the effect persists across the parameter space.

In a way, the non-effectiveness of cancellations is good news for application developers: Cancellations can be complicated and expensive to implement, requiring additional interaction between the centralized scheduler and the leaves. The fact that our simulated zero-cost cancellations bring no significant improvement to tail latency over plain LÆDGE without cancellations, suggests that real-life cancellations are not worth the effort in our context.

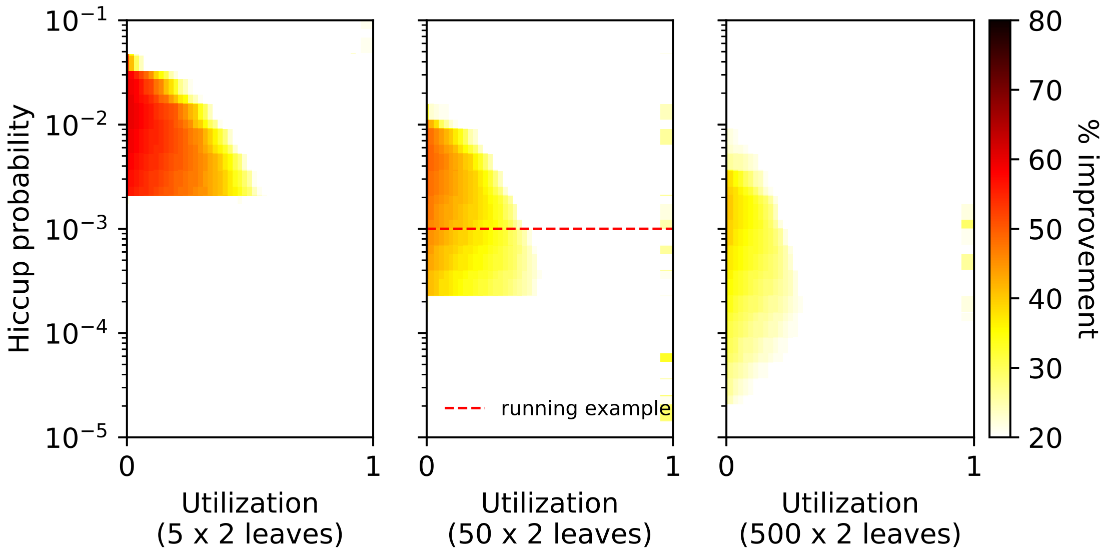


(a) LÆDGE, hiccup duration $15 \times \bar{P}$

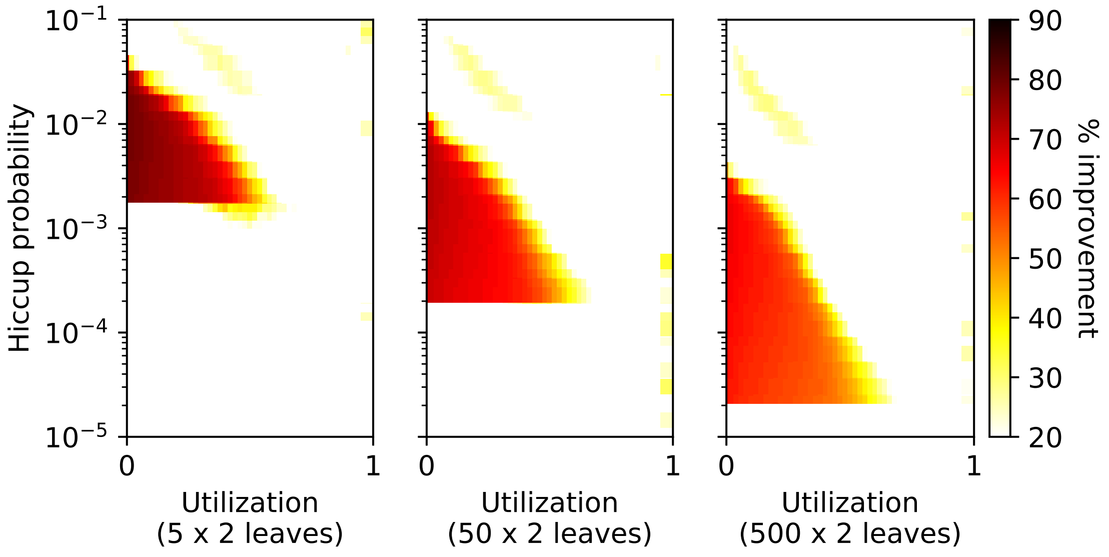


(b) LÆDGE, hiccup duration $30 \times \bar{P}$

Figure 5.5 – Heat maps showing how much various LÆDGE policies improve the 99^{th} percentile latency relative to Per-Shard Queuing. x-axis is system utilization; y-axis is hiccup probability.

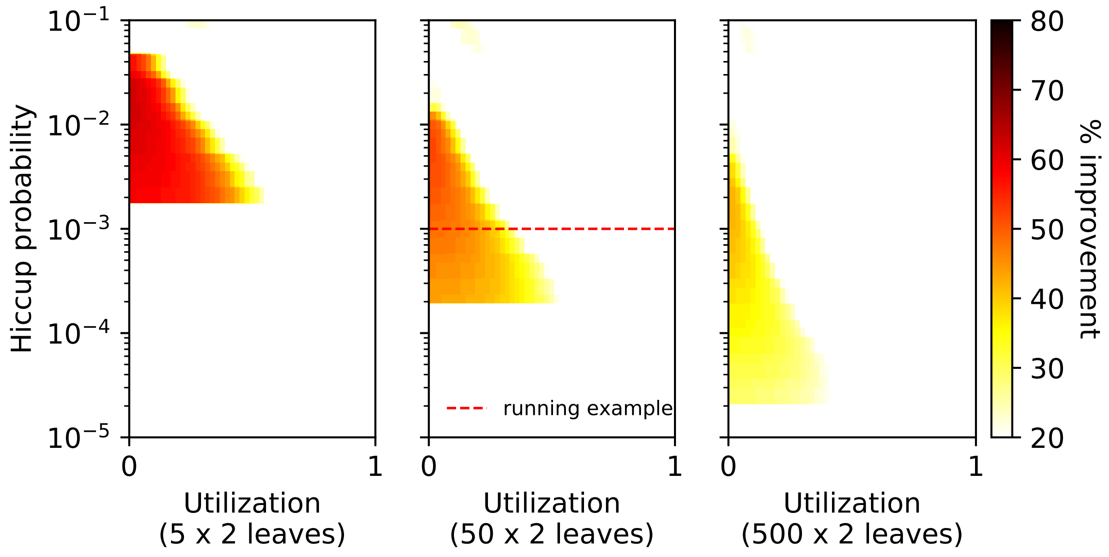


(c) LÆDGE with CC, hiccup duration $15 \times \bar{P}$

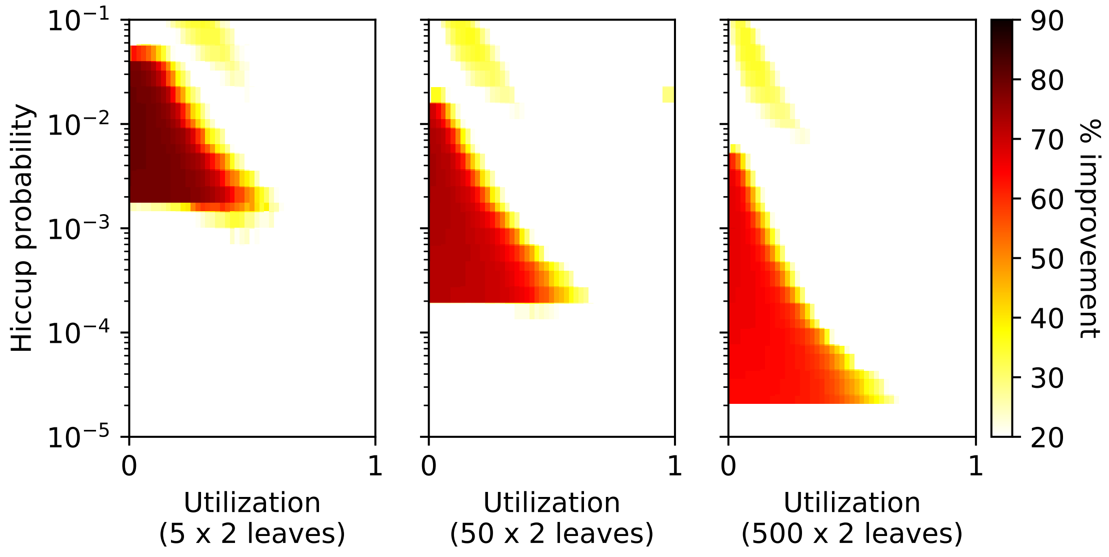


(d) LÆDGE with CC, hiccup duration $30 \times \bar{P}$

Figure 5.5 – (*cont.*) Heat maps showing how much various LÆDGE policies improve the 99^{th} percentile latency relative to Per-Shard Queuing. x-axis is system utilization; y-axis is hiccup probability.

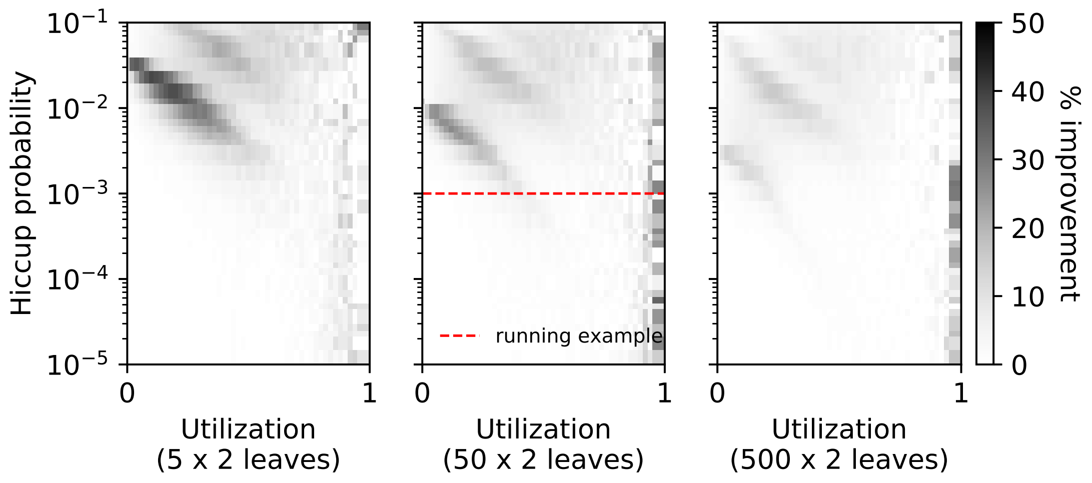


(e) LÆDGE with CC+PC, hiccup duration $15 \times \bar{P}$

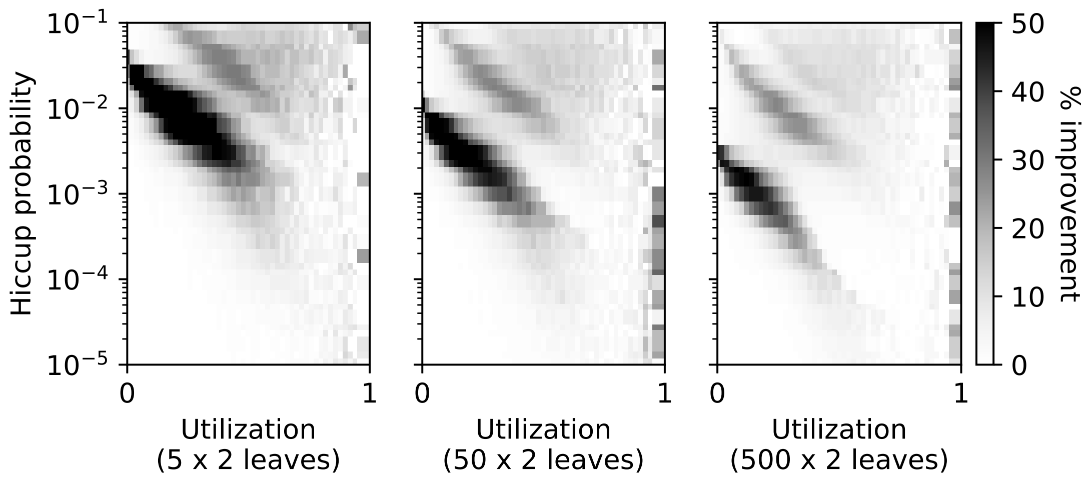


(f) LÆDGE with CC+PC, hiccup duration $30 \times \bar{P}$

Figure 5.5 – (*cont.*) Heat maps showing how much various LÆDGE policies improve the 99th percentile latency relative to Per-Shard Queuing. x-axis is system utilization; y-axis is hiccup probability.

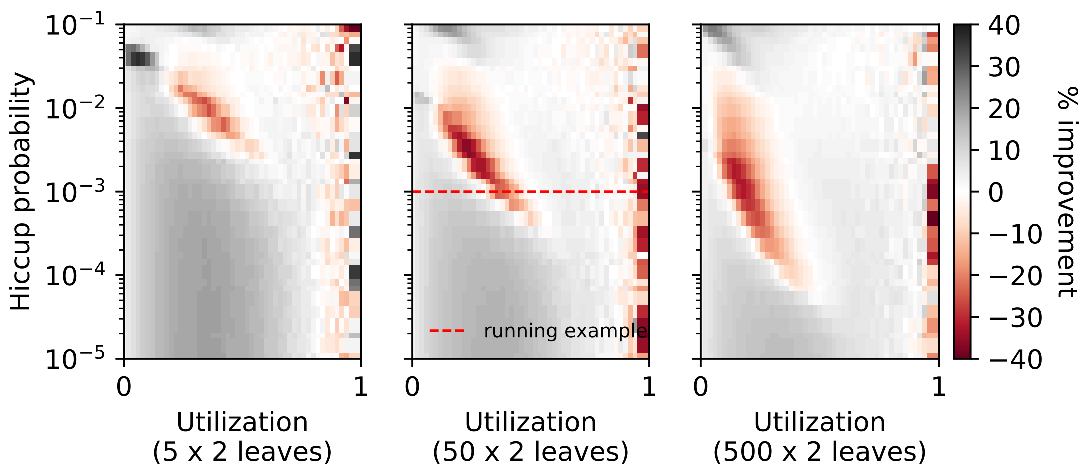


(a) Latency reductions through Cleanup Cancellations; hiccup duration: $15 \times \bar{P}$

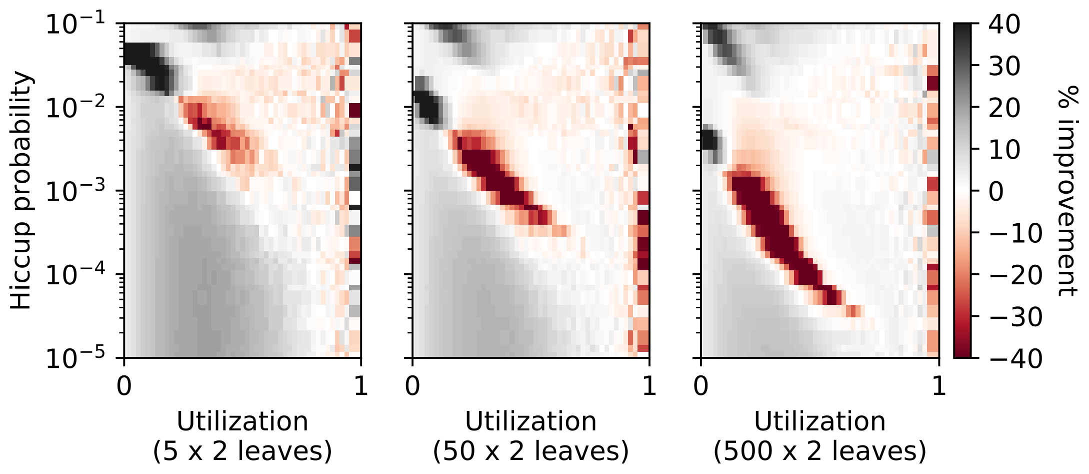


(b) Latency reductions through Cleanup Cancellations; hiccup duration: $30 \times \bar{P}$

Figure 5.6 – Heat maps showing how much LÆDGE with CC improves the 99th percentile latency relative to the plain LÆDGE policy. x-axis is system utilization; y-axis is hiccup probability.



(a) Latency reductions through Pre-emptive Cancellations; hiccup duration: $15 \times \bar{P}$



(b) Latency reductions through Pre-emptive Cancellations; hiccup duration: $30 \times \bar{P}$

Figure 5.7 – Heat maps showing how much LÆDGE with CC+PC improves the 99^{th} percentile latency relative to LÆDGE with CC. x-axis is system utilization; y-axis is hiccup probability.

6 Measuring and Mitigating IQ-jitter in the Cloud

This chapter goes beyond the synthetic workloads and simulations from the previous two chapters:

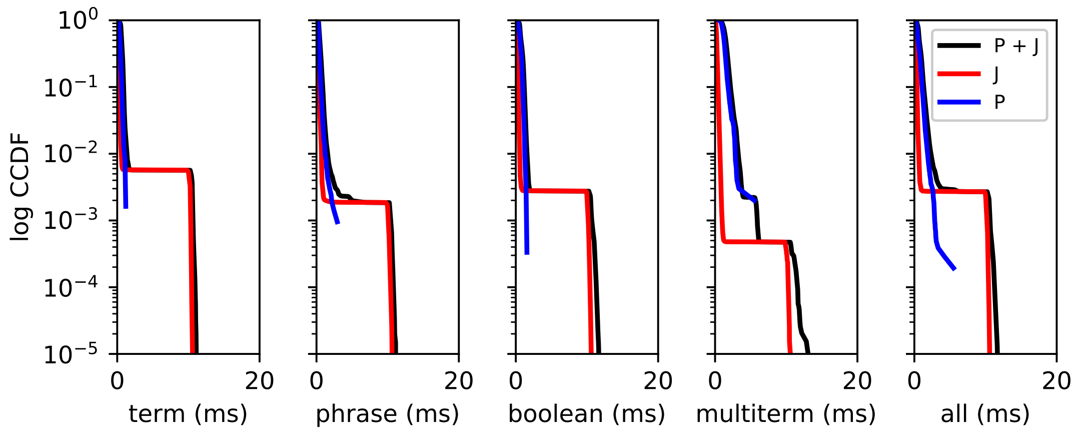
We first show how to measure the distribution of IQ-jitter of a real-world application, which can then be used to estimate the potential benefits of hedging for different deployments like we have seen in the previous two chapters.

A real system includes different overheads that we did not account for in our simulations, for example the networking delay. Could that perhaps change our conclusions and render pull-based policies useless? We then show, in Section 6.2, that this is not the case by implementing the main push and pull-based policies within an existing OLDI framework [48, 49] and by using it to schedule a real-world application in a public cloud.

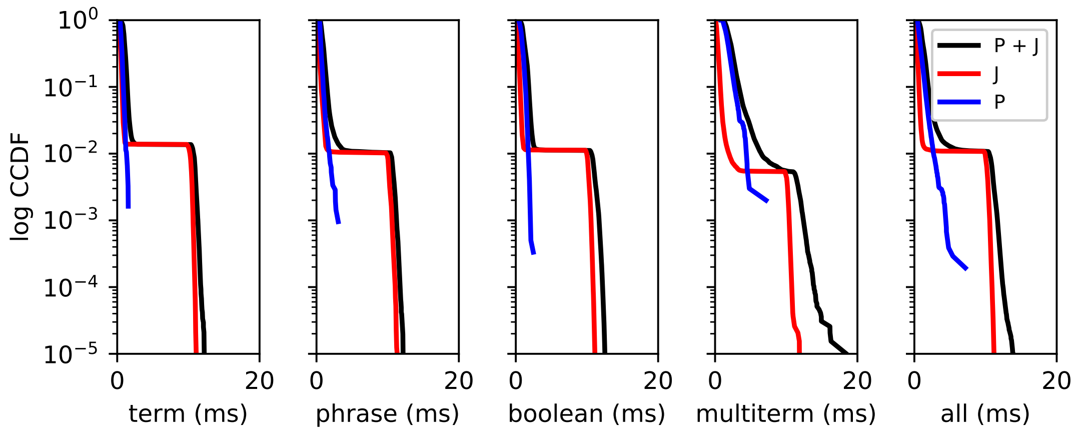
6.1 Empirical IQ-jitter Measurement

Config.	**AVG(P + J)** (ms)	**AVG(P)** (ms)	**AVG(J)** (ms)	**Hiccup prob.**	**Hiccup duration** (ms)
Compute-optimized VM	0.834	0.637	0.198	0.0027	10.162
General-purpose VM	1.367	0.926	0.444	0.0109	10.249

Table 6.1 – Comparison of the service time components of different configurations of Lucene. J is the IQ-jitter of the Lucene workload, while P is its service time without the IQ-jitter. P is measured by executing each query many times and retrieving the minimum execution time for each query, while J is measured as the difference between each execution time and the corresponding P. Hiccup probability and hiccup duration describe J.



(a) Compute-optimized VM



(b) General-purpose VM

Figure 6.1 – Distributions of service-time components measured on two different VM types. The four query types are shown separately from the aggregate distribution ("*all*").

The distribution of IQ-jitter varies based on the workload, the underlying hardware, co-located applications, the initial state of the system, to name a few reasons. While we cannot know the precise distribution of IQ-jitter in the deployment cluster in the public cloud before we run our workload at full scale, we can estimate it by running our application on a single machine in the public cloud using a sample workload.

As an example of a real-world application we use Lucene, which is a popular open-source, enterprise search engine. It is representative of OLDI services because (1) it involves sharding, and (2) client requests are interactive, with expected latency in the millisecond-scale.

Our workload is Lucene's standard nightly regression test, which consists of ~10,000 search queries belonging to four different types: phrase, term, multiterm and boolean [84]. The data consists of an inverted index of 18 million Wikipedia English web pages [131] split into 16 sub-indices (for parallel execution in 16 threads). Lucene comes in C++ and Java implementations; we used the former [85].

So, what is the IQ-jitter experienced by our Lucene workload? To answer this question, we executed the $10,000$ queries of our workload 1000 times each, in a random order, always on the same server type. Consider a specific query Q. For each execution of this query, $Q_i, i = 1...1000$, we measured the service time S_i (which does not include any queuing or network delay). We approximated the application-dependent component of the service time experienced by Q as the minimum service time across executions: $P(Q) = \min_{i=1...1000} S_i$. Then we approximated the IQ-jitter experienced by each query execution Q_i as $J_i = S_i - P(Q)$. By putting together all the IQ-jitter values for a given query type, we obtained the IQ-jitter distribution for this query type.

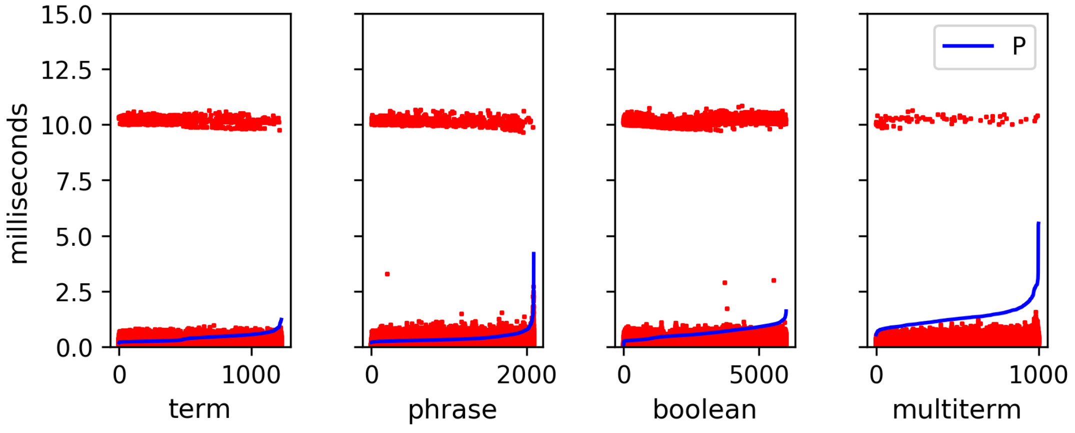
Figure 6.1 shows (in the form of CCDFs) the empirical distributions of P, J, and $P + J$, for the four query types of our workload and for the two different server types. The curves differ in length as P's distribution size depends on the number of queries ($\sim 10,000$), whereas the two other distribution sizes depend on the duration of the measurement experiment. Table 6.1 states the mean values of service time and its P and J components, as well as J's hiccup probability and duration, for each server type.

We observe that IQ-jitter is substantial in both server types and that the empirical distributions are consistent with our simulation setup: the application-dependent component (P) can be well approximated with an exponential distribution (a straight line on a log-based CCDF), while the IQ-jitter component (J) has a clear bimodal nature. This holds across the four query types that vary significantly in complexity (with "multiterm" queries being the most complex ones).

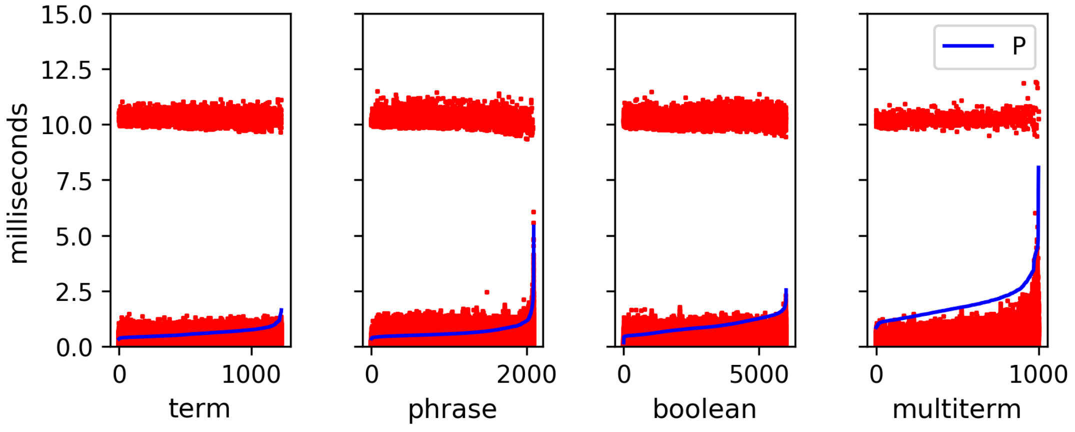
We also observe that the compute-optimized VMs experience IQ-jitter of lower hiccup probability (but similar hiccup duration) than the general-purpose VMs (compare the right-most CCDF in each of Figures 6.1a and 6.1b). We investigated the reason and found that a significant part of IQ-jitter is due to involuntary rescheduling of Lucene threads, which occurs less frequently in the compute-optimized VMs.

Figure 6.2 shows the same data as that in Figure 6.1, but from a different perspective. Once again we show the data for the two VM types in separate figures, and in each figure we show the individual measurements of J for each of the four query types. On the x-axis we show the order number of the query. We can see that different query types are differently represented in the nightly benchmark. One red dot in each of the four subplots in Figures 6.2a and 6.2b is a result of a *single latency measurement* for that query, minus the smallest-ever measurement obtained for that query (which is the value of the blue line, P, for that query). Note that the IQ-jitter samples are ordered by their corresponding values of P in the ascending order.

Of course, different Lucene deployments may experience less IQ-jitter: If the same workload runs on fully-controlled physical machines, *e.g.,* in a dedicated datacenter, IQ-jitter can be reduced by tweaking the OS or the application itself to mitigate the impact of involuntary thread rescheduling on tail latency. We consider our experimental setup as a representative example of a noisy realistic environment for running interactive services in the public cloud.



(a) Compute-optimized VM



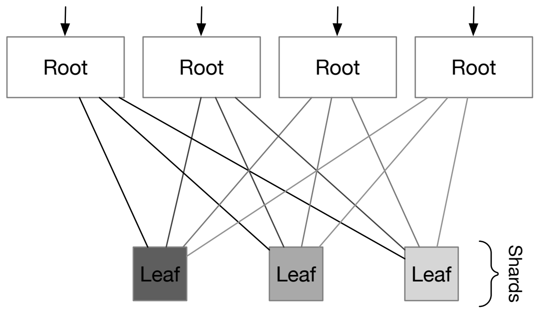
(b) General-purpose VM

Figure 6.2 – IQ-jitter measurements (the red datapoints) on two different VM types sorted by the increasing value of P (the blue line) within each query type. The x-axis shows the order number of the query (not all query types are equally represented), while the y-axis shows the latency in milliseconds.

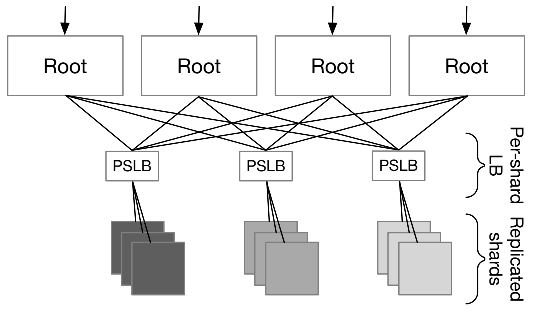
6.2 Mitigating Tail Latency in the Public Cloud

We implemented two load-balancing policies (Random and Per-Shard Queuing) and two hedging policies (Naïve Hedge and LÆDGE) in OLDIsim, Google's open-source OLDI cloud benchmarking framework [49].

The I/O part of the framework stayed unchanged: it uses the event-based `libevent` API [81] on top of vanilla Linux and TCP. We extended the framework with request generation following a Poisson inter-arrival distribution. We changed its architecture to include the notion of shard replication and a per-shard load balancer, *i.e.,* from its original architecture in Figure 6.3a, we changed it so that it fits that in Figure 6.3b. Finally, we implemented our load-balancing and hedging policies.



(a) Original OLDIsim architecture.



(b) OLDI architecure with per-shard load balancers (PSLBs).

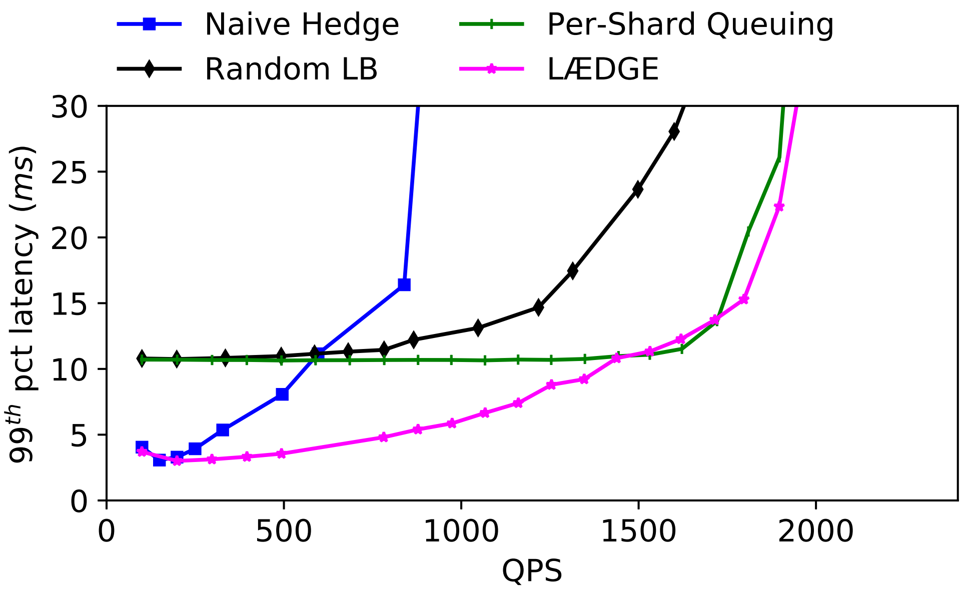
Figure 6.3 – Original OLDIsim architecture and the architecture with PSLBs.

We ran the Lucene workload on AWS EC2 virtual machines (VMs), organized in a “cluster” placement group [3]. We used two VM types: (1) compute-optimized instances with 16 vCPUs @3.0 GHz and 32 GB of memory ($c5.4xlarge$), and (2) general-purpose instances with 16 vCPUs @2.2 GHz and 64 GB of memory ($m5a.4xlarge$). All VMs were running the default Ubuntu 16.04.6 image, kernel version 4.4.0-1092-aws. The round-trip time between any two VMs in our setup amounts to 90 µs, on average. This corresponds to 10.8% of the mean service time of the compute-optimized workload (which is the one with shorter service time).

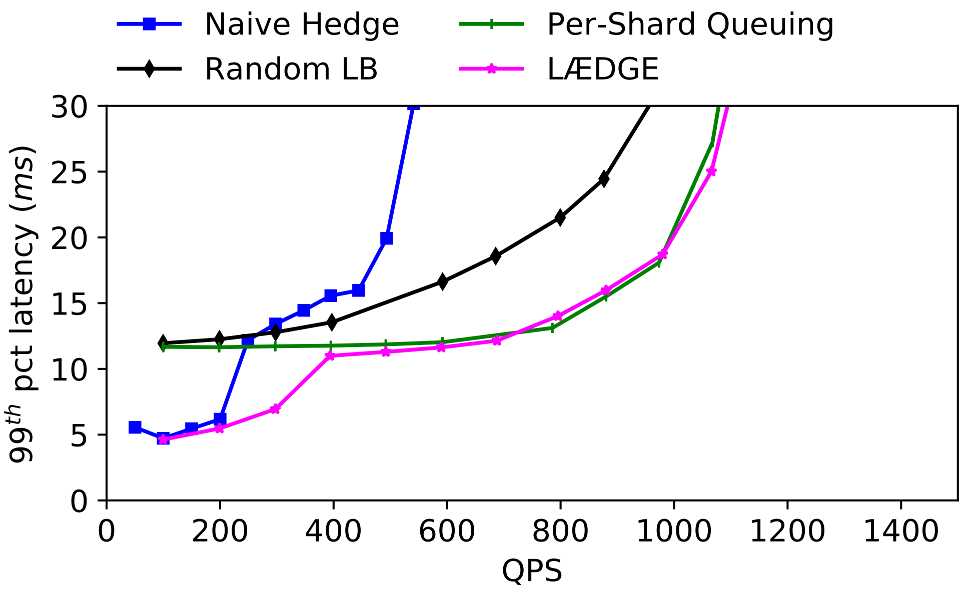
We deployed 5 × 2 leaf servers, *i.e.,* 5 distinct shards, each replicated in 2 leaves. When a leaf executes a query, it uses 16 parallel threads (one per vCPU). To avoid the introduction of data-driven bias in our results, we replicated the same reversed index on all 5 shards (though, from the point of view of the application, they are still distinct shards served by different leaves). This decision simplifies the comparison with the simulation results in Chapter 5, without fundamentally changing the conclusions.

We deployed a single root node and ran 5 distinct per-shard load balancers. We chose to co-locate the per-shard load balancers with the root node. Note that deploying more root nodes is possible when a single one cannot keep-up with the load. In that case one needs to also deploy a pre-root load balancer “before” the root nodes that will assign each request to one of them (see Section 2.5.3). Such a load-balancer was already available in the OLDI framework that we extended, and we did not change its behavior. Note that in our case, additional root nodes were not needed due to the relatively high service time of our chosen application.

We measured the end-to-end latency experienced by our Lucene workload under varying system load. Figures 6.4a and 6.4b show the 99^{th} percentile latency as a function of system utilization, for the two server types, respectively. The x-axis is capped at 100% utilization which is computed from the average service time without the IQ-jitter ($\bar{P}$), as in our discrete-event simulations in the previous chapters (see Section 4.4).



(a) With compute-optimized VMs



(b) With general-purpose VMs

Figure 6.4 – Mitigating the Lucene hiccups in a system implementation deployed on 2×5 leaves in EC2 VMs.

LÆDGE behaves as expected: it matches Naïve Hedge at low utilization, converges to Per-Shard Queuing at some point, and outperforms the best alternative in between. The exact behavior depends on server type: On the compute-optimized VMs, LÆDGE converges to Per-Shard Queuing at about 60% utilization; before that, it improves tail latency by 49%, or 5.3 *ms*, on average, relative to Per-Shard Queuing. On the general-purpose VMs, convergence to Per-Shard Queuing happens quite earlier—at about 27% utilization—and the improvement of tail latency before that point is somewhat smaller (40%, or 4.7 *ms*, on average). This is due to our workload experiencing more frequent hiccups on the general-purpose VMs. General-purpose VMs are recommended for application development and testing, unlike the compute-optimized ones that are recommended for high-performance web and gaming servers [9], which is much closer to our OLDI scenario.

Our LÆDGE implementation (deployed in a public cloud with real system noise and non-zero network latencies) behaved as our simulation predicted: When we use compute-optimized VMs, our experimental setup consists of 5×2 leaves with IQ-jitter of hiccup probability 0.0027 and hiccup duration $15.95 \times \bar{P}$ (Table 6.1). The closest simulated setup is a cluster of the same size with IQ-jitter of the same hiccup probability and hiccup duration $15 \times \bar{P}$. This corresponds to the leftmost heat map in Figure 5.5a, y-axis value 0.0027 (which is close to the base of the triangular shape of the heat map). If we observe this heat map at the given y-axis value, we can see that our simulated LÆDGE policy significantly outperforms PSQ until utilization ~40% and then converges to PSQ at utilization ~60%—exactly the behavior of our LÆDGE implementation.

7 Related Work

7.1 Hardware-Based Techniques for Measuring Latency Variability

In Chapter 3, we study the latency measurement features of one of the most expensive pieces of hardware that engineers use to measure latency, namely a proprietary load generator from Spirent [121]. We also study a commodity NIC that has only the NIC timestamps tailored to the precise requirements of IEEE 1588 time synchronization—which is still the most commonly found type of timestamps in the NICs available today. There is a large spectrum of devices available in between these two extremes that are studied in the literature.

Weber *et al.* [129] re-purposed a programmable P4 switch [8] and implemented a hardware-based traffic generator with similar timestamping and throughput capabilities as the proprietary hardware devices, but with much more programmability regarding *e.g.,* packet inter-arrival times. The price of a P4 switch is significantly lower than that of a high-end specialized hardware appliance with advanced features such as the Poisson inter-arrival times, but still higher than the price of a commodity server. In our analysis in Chapter 3, we expect this tool to perform as well as Spirent.

Caliper [45] and OFLOPS [113] are FPGA-based tools built on top of the NetFPGA platform [46]; we used Spirent instead.

Zilberman et al. [139] focused on dissecting one-way latency from the wire to the application up to the ns-scale latency. Their methodology relies on Data Aggregation and Generation (DAG) cards that have precision comparable to Spirent's. They used a latency-optimized NIC that relies on a proprietary driver and a proprietary kernel-bypass framework [40] and a Solarflare NIC [120]. While they focused on the best possible case in terms of hardware and software setup on the measured side, in Chapter 3 of this dissertation, we evaluated commonly-used techniques in academia and industry on the measuring side. Both sets of results show that kernel-bypass improves tail latency by $20-40\mu$s.

Kogias *et al.* [73] leverage NIC-based hardware timestamping to measure RPC end-to-end latency. They use a new Mellanox NIC (ConnectX-4 [90] or newer) that costs more than the NIC we used in our study in Chapter 3, but that in exchange offers general-purpose timestamping for *all* incoming and outgoing packets. Therefore, unlike the NIC that we used, ConnectX-4 can timestamp even the TCP packets at line rate. This type of hardware feature enables them to measure RPC latency. This is not possible using the NIC timestamps tailored to the precise requirements of IEEE 1588 time synchronization—which are still the most commonly found timestamps in today's commodity NICs.

7.2 Taming Map-Reduce Latency

Early attempts to reduce tail latency studied long-running map-reduce jobs with execution times measured in seconds or minutes [29]. This timescale allows for "observe-then-predict" type of algorithms, with sophisticated execution profiling based on which a decision can be made about when it pays off to hedge [5, 6, 101, 136]. Map-reduce systems such as LATE [136], Mantri [6] and Dolly [5] leveraged the idea of hedging to mitigate the stragglers that would delay the entire phase. Our work focuses on interactive services which lack some pre-requisite metrics (*e.g.,* the number of bytes left to read from the input) and do not lend themselves to heavy-weight profiling.

7.3 Reducing Latency in OLDI Services

The timescales at which OLDI services operate are much smaller than the timescales of map-reduce type of applications. Therefore, many of the techniques discussed in Section 7.2 are not applicable to OLDI services, or applying them is extremely challenging. In this section we give an overview of studies that are also concerned with optimizing latency in OLDI services.

7.3.1 Optimizing Inter-Query Service Time Variability Through Adaptive Parallelism

Many researchers have tried to predict the service time of interactive services and accordingly adjust the level of parallelism in the processing nodes or prioritize short-running queries over long-running ones [57, 72, 87, 122].

Haque *et al.* [57] noticed that blindly parallelizing all queries quickly saturates hardware resources without yielding significant benefit for tail latency of short requests. Through exhaustive offline profiling phase, they extract the policy that adds parallelism based on dynamic system load and execution time progress. The intuition is that parallelism should be added only for long running requests and when the load allows it.

Kim *et al.* [72] do not just follow a fixed policy to incrementally add parallelism, but they run a prediction algorithm on a number of static and dynamic features retrieved from a query and its execution to estimate its duration. The key insight to increase the accuracy of the predictions is to wait a certain amount of time D before running the predictor because short queries are likely to simply finish during that time.

Sriraman *et al.* [122] developed an interesting load adaptation system that curtails microservice tail latency by exploiting inherent latency trade-offs revealed in their taxonomy and use it to transition among threading models.

These type of techniques actually reduces only the predictable component of the service time, P, and therefore reduces inter-query service time variability. This is orthogonal to the topic of this dissertation that focuses on the unpredictable *intra*-query service time variability, in the context of OLDI services.

7.3.2 Early Stop

In some cases it is possible to reply to a user's request before all the query replies have arrived. In the literature, there are two main scenarios in which this is acceptable:

First, in some OLDI setups it is acceptable that the accuracy of OLDI results is sacrificed to achieve lower latency. He *et al.* [60] introduced a scheduling model for interactive services where lower result quality can be traded for shorter execution time. Ravindranath *et al.* [110] implemented a deadline-aware system that enables a server to adapt its processing time to control the end-to-end delay for the request.

Second, in systems with scheduling techniques based on erasure coding the final result can be decoded when *a part* of queries finish. Rashmi *et al.* [109] divide each object into k splits and store them in a $(k+r)$ erasure-coded form. Their encoding is such that any k of the $(k+r)$ splits are sufficient to read an object. Kosaian *et al.* [76] apply a similar technique with the help of machine learning and target it for machine learning workloads to reconstruct failed or missing predictions.

Such approaches are orthogonal to scheduling techniques proposed in this dissertation, and they can be integrated in our LÆDGE policy to further reduce the response times.

7.3.3 Hedging at Low Latencies

Dean *et al.* [28] were the first to demonstrate that policies such as Naïve Hedge and d-Hedge can be applied to mitigate latency variability of interactive services and their Bigtable workload [24]. State-of-the-art reissue policies such as p-Hedge [68] address the throughput limitations of Naïve Hedge and d-Hedge. Mirhosseini *et al.* [92], advocate single-queue solutions (equivalent to Per-Shard Queuing in our work) as a plausible means to reduce tail latency in the presence of jitter. We show in Section 4.6 that approaches based on Per-Shard Queuing outperform carefully-designed hedging approaches that do not use a single queue per shard.

7.3.4 Advanced Load Balancing

Lu *et al.* [83] decouple discovery of lightly-loaded servers from job assignment that works with multiple root nodes. Since it offers no redundancy, this technique is prone to increased tail latency in the presence of IQ-jitter. "Snitching" is another interesting LB technique in which the application (*e.g.,* document aggregator in the root node) monitors request latency and picks the fastest replica [7, 123]. This technique also offers no redundancy and is ineffective in case of bursty noise [56].

7.3.5 Cancellations

Ananthanarayanan *et al.* [5] previously observed that cancellations do not improve tail latency of map-reduce workloads. While recent advances have shown that the future of canceling microsecond-scale RPCs is promising [20], cancellation mechanisms often require non-trivial application changes and language-specific mechanisms to avoid memory leaks and inconsistent application state (*e.g.,* even in [20] memory leaks remain a problem). Bashir *et al.* [14] heavily rely on the use of cancellation and suggest their implementation at multiple points in the software stack. Bashir *et al.* also rely on prioritization of new over duplicated requests, similarly to our policies. Dean *et al.* [28] suggest "tied requests", in the context of cluster-level distributed file system [28], which correspond to Naïve Hedge with CCs. Some state-of-the-art hedging policies rely on application-specific or operating-system-level instrumentation to increase the accuracy of scheduling and cancellation decisions [56]. Our own LÆDGE with PC currently uses a trivial policy which can be deployed everywhere, including in the cloud. We leave the study of the combination with profiling for future work.

7.4 Infrastructure Jitter

Section 2.3 includes numerous examples of system events that cause jitter. Hao *et al.* [56] provide operating system support to cut millisecond-level tail latencies by adding the SLO information to the operating system and up front rejecting the requests that cannot meet the deadline, relying on the scheduler to send the request elsewhere. They observe and quantify noise in EC2 with a focus on disk read and write jitter. Our work focuses on in-memory, CPU-bound applications, which also observe a varying amount of jitter determined in part by the underlying cloud VM.

8 Conclusion

This dissertation advanced the area of latency measurements and latency mitigation of Internet services. We showed how to measure microsecond-scale latency variability both cheaply and reliably. Latency-variability is a indeed a real problem for service providers that try to operate under tight service-level objectives. Even for those running Internet services in the public cloud, there is a large potential in mitigating their latency variability by combining pull-based load balancing with load-aware request-reissuing techniques.

First, in Chapter 3 we addressed the problem of unreliable latency measurements of commodity measurement tools that many researchers and engineers use on a daily basis. We focused on µs- instead of ns-scale measurements for comparing software- and hardware-based tools, because the former is a "gray area" where optimized software techniques might still stand a chance. We compared some commonly used tools and techniques to a high-end hardware appliance to determine how (un)reliable they are. During the "calibration" phase we used a hardware switch that has predictable and constant service times at microsecond granularity under our traffic pattern. This gave us the baseline of the overheads coming from the tools themselves. We then benchmarked subtle differences in the latency distribution of a software function with different operating-system configurations. A good tool should be able to measure and tell apart those configurations that often give a headache to engineers (ideally the measurements should be offset solely by a constant overhead that we retrieved from the calibration phase).

Our results clearly show the benefit of measuring latency of packet requests and responses, and more generally, of remote procedure calls within the NIC as opposed to CPU. We found that kernel bypass and NIC timestamps can be used to measure microsecond-scale latency up to 99^{th} and 99.99^{th} percentile, respectively.

Having reliable tools and knowing their overheads is important to measure latency variability. Latency variability comes from many sources and having the right tools helps guide engineers in the right direction. However, some sources of latency variability are both hard to predict and impossible to eliminate in practice. In this case, one needs to resort to a higher-level technique such as request reissuing. Chapters 4 to 6 address the challenge of mitigating latency variability through redundantly-issued requests. In this well-studied area we noticed that existing techniques reissue either too many or too few requests, mostly because they reissue in a way that does not adapt to the current system load. Our take on hedging therefore includes load awareness and load balancing, *i.e.,* our insight is that one should reissue a request only if there are enough resources available.

In Chapter 4 we motivated the need for combining load balancing and hedging into a load-aware policy. For that purpose we defined Idealized Hedge, our lower bound that perfectly optimizes queuing delays due to Per-Shard Queuing, and hedges as much as possible without delaying the new incoming requests. A part of the latency reductions of Idealized Hedge is due to a perfect predictor of query-completion times that the policy uses to pre-empt the hedged request that would have finished later in the future. In a realistic setting, such a policy is not possible to implement. We use Idealized Hedge to define an upper bound on the gains that a realistic hedging policy could implement.

Having an upper bound on the gains of hedging can help us understand what types of work-loads benefit from this general technique, but in order to achieve these benefits we need a policy that can be implemented in a real system. Our attempt to approach Idealized Hedge is called Load-Aware Hedge (LÆDGE), presented in Chapter 5. LÆDGE comes in three different flavors: without cancellation, with cleanup cancellations, and with both cleanup and pre-emptive cancellations. We showed that combining single-queue load balancing with hedging results in a policy that can outperform the state of the art and closely approach our lower bound even without cancellations.

One of our observations from Chapter 4 is that hedging only makes sense in the presence of *intra-query jitter* (IQ-jitter), which is our metric that formalizes system hiccups. In Chapter 6 we showed how to extract IQ-jitter of a real application. We then validated the benefits of LÆDGE experimentally with a cloud-based deployment of an interactive web search application. We achieved significant reductions in tail latency in a real system, and confirmed that our system results closely match what our simulation predicted.

8.1 Future Work

Some remarks from our work on hedging in Chapter 4, Chapter 5, and Chapter 6 are (i) hedging yields latency reductions only when latency variability is significant, (ii) pull-based scheduling policies are less effective as networking delay increases, (iii) cancellations help significantly only for a particular range of utilization and some hiccup distributions. When an interactive service is deployed in the Cloud, its properties, such as service time variability or network delay between the nodes, might change dynamically. This means that in practice there is no single optimal scheduling policy across the entire execution. Implementing an RPC framework that would *dynamically* decide on the optimal scheduling policy (push or pull based, with or without hedging and cancellations *etc.*) could be a beneficial future research direction for all kinds of distributed applications, and especially for interactive services.

In this dissertation, we focused on read-only workloads. Removing the read-only constraint for the type of workloads to which our work can be applied is also an interesting direction for future research.

8.2 Discussion

8.2.1 NIC Timestamps for Latency Measurements

The results we presented in Chapter 3 are specific to the type of hardware we had at our disposal at the time this study was conducted. One of our main takeaways were that NIC timestamps are highly beneficial, and enabling them at high throughput, and for arbitrary protocols, is a worthwhile goal for future NICs. Fortunately, this hardware feature is already becoming more widely available, for instance in the a new series of Mellaox NICs [90]. Meanwhile, kernel bypass networking has become more widely used and remains the best software-only solution for a latency-measurement tool that does not require changing the kernel.

8.2.2 Scheduling Discipline in the Processing Nodes

In Chapters 4, 5, and 6, we considered only a particular scheduling discipline in the processing nodes, namely FCFS (see Section 4.3). Prior work shows that FCFS is the best non-preemptive scheduling policy when optimizing for tail latency [80, 92, 130]. Note that regardless of the scheduling discipline in the processing nodes, our insights are still applicable (after applying the corresponding changes due to the change in the setup from Section 4.3). The sources of IQ-jitter do not include the inherent complexity of a query; instead, the hiccups from the system and the environment itself is what forms the distribution of IQ-jitter. Therefore, even in the case of, for example, processor-sharing implemented in the leaf nodes, IQ-jitter still causes increases in tail latency. Thus, the need for both hedging and load balancing remains.

8.2.3 Scalability of Per-Shard Queuing

Scheduling techniques based on Per-Shard Queuing naturally scale with more leaf processing nodes as the queues are centralized *only within* a shard, and the degree of shard replication is often small due to high costs of DRAM [89]. Each (logical) per-shard load balancer is in charge of scheduling the requests for its shard. As the dataset grows and more shards are needed, this design scales horizontally by simply adding more machines for per-shard load balancers (see Figure 2.4).

8.2.4 Fault-Tolerance of Per-Shard Queuing

Building fault-tolerant Internet services is a well-researched area for which we claim no novelty. Our system design in Figure 2.4 is resilient to the failure of all of its components. The state in per-shard load balancers is soft and can easily be reconstructed by querying the leaf nodes, but simply starting from a clean slate is not a problem. The worst thing that can happen if a part of the state of a per-shard load balancer gets lost is that short queues can build up in the leaf nodes. The shards hosted by the leaf nodes are replicated across different leaf nodes. Finally, since we also have multiple root nodes, if a root node fails another root node can take over. For example, leaf nodes can have a list of backup root nodes for each particular failed one. In most cases, a request can be fully reconstructed if there is at least one leaf node still processing it. However, for most low-latency services the typical solution is to contain the failure and return an appropriate error message, rather than trying to fully recover user requests.

8.3 Conclusion

In this dissertation we emphasized the importance of latency variability for various applications, from network services to OLDI applications. Understanding the precision of the tools we use to measure them helps us identify the underlying problems. For microsecond-scale latency variability, one should use either hardware-based solutions, or software-based ones, enhanced with kernel-bypass networking.

It is time to integrate pull-based solutions and centralized queuing within OLDI frameworks. We claim that such solutions are now feasible using modern data-center technologies such as kernel-bypass, data-plane operating systems, programmable switches, and RPC-aware protocols [18, 74, 102–104]. While pulling may be inappropriate in the context of µs-scale service times (*e.g.,* in key-value stores), its overhead becomes negligible when the service time is in the 100µs to 100ms range, as is the case with most OLDI applications (*e.g.,* web search [57]). Even when using a commodity vanilla Linux networking stack, we showed in Chapter 6 that latency can still be mitigated.

One can quantify when redundancy (*i.e.,* hedging), as a general technique to reduce latency, can bring latency reductions in a certain service deployment, before the deployment takes place. The key to achieving this is modeling the workload and designing a lower bound that hedges as much as possible and perfectly undoes its wrong decisions with the help of perfect predictor of query completion times.

When hedging makes sense, the decision to hedge must depend on the state of the queues (as in our LÆDGE policy) and be tuned for the amount of resources available. Hedging should not be tied to a (randomized) delay metric (as in d-Hedge or p-Hedge).

Hedging does not solve the general tail-at-scale problem: it can help only when there exists

IQ-jitter, and—our results suggest—only under specific operating conditions, determined by different characteristics of the workload and deployment including system utilization, IQ-jitter distribution, and fan-out. Even though our community has long studied service-time variability on an end-to-end basis, we now need to study and understand IQ-jitter separately from overall service-time variability, as otherwise we cannot design proper hedging policies and understand their benefit.

Bibliography

[1] M. Alizadeh, A. G. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference*, pages 63–74, 2010.

[2] Amazon. AWS Lambda. https://aws.amazon.com/lambda/.

[3] Amazon. Placement groups. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/placement-groups.html.

[4] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Why Let Resources Idle? Aggressive Cloning of Jobs with Dolly. In *Proceedings of the 4th workshop on Hot topics in Cloud Computing (HotCloud)*, 2012.

[5] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective Straggler Mitigation: Attack of the Clones. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 185–198, 2013.

[6] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *Proceedings of the 9th Symposium on Operating System Design and Implementation (OSDI)*, pages 265–278, 2010.

[7] Apache. Cassandra snitches. https://docs.datastax.com/en/archived/cassandra/3.0/cassandra/architecture/archSnitchesAbout.html.

[8] Barefoot Networks. Barefoot Tofino. https://barefootnetworks.com/products/brief-tofino/.

[9] J. Barr. Choosing the right EC2 instance type for your application. https://aws.amazon.com/blogs/aws/choosing-the-right-ec2-instance-type-for-your-application/.

[10] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan. Attack of the killer microseconds. *Commun. ACM*, 60(4):48–54, Mar. 2017.

[11] L. A. Barroso, J. Clidaras, and U. Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2013.

[12] L. A. Barroso, J. Dean, and U. Hölzle. Web Search for a Planet: The Google Cluster Architecture. *IEEE Micro*, 23(2):22–28, 2003.

[13] L. A. Barroso, M. Marty, D. A. Patterson, and P. Ranganathan. Attack of the killer microseconds. *Commun. ACM*, 60(4):48–54, 2017.

[14] H. M. Bashir, A. B. Faisal, M. A. Jamshed, P. Vondras, A. M. Iftikhar, I. A. Qazi, and F. R. Dogar. Reducing tail latency using duplication: a multi-layered approach. In *Proceedings of the 2019 ACM Conference on Emerging Networking Experiments and Technology (CoNEXT)*, pages 246–259, 2019.

[15] A. Belay, G. Prekas, M. Primorac, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. The IX Operating System: Combining Low Latency, High Throughput, and Efficiency in a Protected Dataplane. *ACM Trans. Comput. Syst.*, 34(4):11:1–11:39, 2017.

[16] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy. *Site Reliability Engineering: How Google Runs Production Systems.* " O'Reilly Media, Inc.", 2016.

[17] M. Bligh, M. Desnoyers, and R. Schultz. Linux kernel debugging on google-sized clusters. In *Proceedings of the Linux Symposium*, pages 29–40, 2007.

[18] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: programming protocol-independent packet processors. *Computer Communication Review*, 44(3):87–95, 2014.

[19] A. Botta, A. Dainotti, and A. Pescapè. Do you trust your software-based traffic generator? *IEEE Communications Magazine*, 48(9):158–165, 2010.

[20] S. Boucher, A. Kalia, D. G. Andersen, and M. Kaminsky. Putting the "Micro" Back in Microservice. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, pages 645–650, 2018.

[21] S. Bradner and J. McQuaid. Benchmarking Methodology for Network Interconnect Devices. IETF RFC 2544, Mar. 1999.

[22] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson. BBR: congestion-based congestion control. *Commun. ACM*, 60(2):58–66, 2017.

[23] H. Casanova. Benefits and Drawbacks of Redundant Batch Requests. *J. Grid Comput.*, 5(2):235–250, 2007.

[24] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, 2008.

[25] Chris Jones and John Wilkes and Niall Murphy and Cody Smith. Service Level Objectives. https://landing.google.com/sre/sre-book/chapters/service-level-objectives/.

[26] Cisco Systems. Cisco SG500X-48 48-Port GB with 4-Port 10-GB Stackable Managed Switch. https://www.cisco.com/c/en/us/products/collateral/switches/small-business-500-series-stackable-managed-switches/c78-695646_data_sheet.html.

[27] Cisco Systems. TRex: Cisco's realistic traffic generator. https://trex-tgn.cisco.com.

[28] J. Dean and L. A. Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, 2013.

[29] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[30] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, pages 205–220, 2007.

[31] P. Delgado, F. Dinu, A.-M. Kermarrec, and W. Zwaenepoel. Hawk: Hybrid Datacenter Scheduling. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC)*, pages 499–510, 2015.

[32] C. Delimitrou and C. Kozyrakis. iBench: Quantifying interference for datacenter applications. In *Proceedings of the 2013 IEEE International Symposium on Workload Characterization (IISWC)*, pages 23–33, 2013.

[33] C. Delimitrou and C. Kozyrakis. Amdahl's law for tail latency. *Commun. ACM*, 61(8):65–72, 2018.

[34] Data Plane Development Kit. http://dpdk.org/.

[35] J. Eidson and K. Lee. IEEE 1588 standard for a precision clock synchronization protocol for networked measurement and control systems. In *Sensors for Industry Conference, 2nd ISA/IEEE*, 2002.

[36] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein. Maglev: A Fast and Reliable Software Network Load Balancer. In *Proceedings of the 13th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 523–535, 2016.

[37] P. Emmerich. Moongen's GitHub repository (commit ef3aa3f). https://github.com/emmericp/MoonGen.

[38] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle. MoonGen: A Scriptable High-Speed Packet Generator. In *Proceedings of the 15th ACM SIGCOMM Workshop on Internet Measurement (IMC)*, pages 275–287, 2015.

[39] European Telecommunications Standards Institute. Network Functions Virtualisation – Introductory White Paper. http://portal.etsi.org/NFV/NFV_White_Paper.pdf, 2012.

[40] Exablaze. ExaNIC X10. https://exablaze.com/exanic-x10.

[41] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. M. Caulfield, E. S. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. G. Greenberg. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *Proceedings of the 15th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 51–66, 2018.

[42] K. Gardner, S. Zbarsky, S. Doroudi, M. Harchol-Balter, and E. Hyytiä. Reducing Latency via Redundant Requests: Exact Analysis. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 347–360, 2015.

[43] P. Garraghan, X. Ouyang, R. Yang, D. McKee, and J. Xu. Straggler Root-Cause and Impact Analysis for Massive-scale Virtualized Cloud Datacenters. *IEEE Trans. Serv. Comput.*, 12(1):91–104, 2019.

[44] F. Gaud, S. Geneves, R. Lachaize, B. Lepers, F. Mottet, G. Muller, and V. Quéma. Efficient Workstealing for Multicore Event-Driven Systems. In *Proceedings of the 30th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 516–525, 2010.

[45] M. Ghobadi, M. Labrecque, G. Salmon, K. Aasaraai, S. H. Yeganeh, Y. Ganjali, and J. G. Steffan. Caliper: a tool to generate precise and closed-loop traffic. In *Proceedings of the ACM SIGCOMM 2010 Conference*, pages 445–446, 2010.

[46] G. Gibb, J. W. Lockwood, J. Naous, P. Hartke, and N. McKeown. NetFPGA - An Open Platform for Teaching How to Build Gigabit-Rate Network Switches and Routers. *IEEE Trans. Education*, 51(3):364–369, 2008.

[47] Google. Cloud functions. https://cloud.google.com/functions/.

[48] Google Cloud Platform. OLDIsim repository. https://github.com/GoogleCloudPlatform/oldisim.

[49] Google Cloud Platform Blog. Benchmarking web search latencies. http://cloudplatform.googleblog.com/2015/03/benchmarking-web-search-latencies.html.

[50] A. G. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: a scalable and flexible data center network. *Commun. ACM*, 54(3):95–104, 2011.

[51] V. Gupta, M. Harchol-Balter, K. Sigman, and W. Whitt. Analysis of join-the-shortest-queue routing for web server farms. *Perform. Eval.*, 64(9-12):1062–1081, 2007.

[52] S. Halfin and W. Whitt. Heavy-Traffic Limits for Queues with Many Exponential Servers. *Operations Research*, 29(3):567–588, 1981.

[53] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee. Network function virtualization: Challenges and opportunities for innovations. *IEEE Communications Magazine*, 53(2):90–97, 2015.

[54] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy. MegaPipe: A New Programming Interface for Scalable Network I/O. In *Proceedings of the 10th Symposium on Operating System Design and Implementation (OSDI)*, pages 135–148, 2012.

[55] M. Handley, C. Raiciu, A. Agache, A. Voinescu, A. W. Moore, G. Antichi, and M. Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the ACM SIGCOMM 2017 Conference*, pages 29–42, 2017.

[56] M. Hao, H. Li, M. H. Tong, C. Pakha, R. O. Suminto, C. A. Stuardo, A. A. Chien, and H. S. Gunawi. MittOS: Supporting Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 168–183, 2017.

[57] M. E. Haque, Y. H. Eom, Y. He, S. Elnikety, R. Bianchini, and K. S. McKinley. Few-to-Many: Incremental Parallelism for Reducing Tail Latency in Interactive Services. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XX)*, pages 161–175, 2015.

[58] M. Harchol-Balter. Task assignment with unknown duration. *J. ACM*, 49(2):260–288, 2002.

[59] M. Harchol-Balter, M. Crovella, and C. D. Murta. On Choosing a Task Assignment Policy for a Distributed Server System. *J. Parallel Distrib. Comput.*, 59(2):204–228, 1999.

[60] Y. He, S. Elnikety, J. R. Larus, and C. Yan. Zeta: scheduling interactive services with partial execution. In *Proceedings of the 2012 ACM Symposium on Cloud Computing (SOCC)*, page 12, 2012.

[61] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller. The eXpress data path: fast programmable packet processing in the operating system kernel. In *Proceedings of the 2018 ACM Conference on Emerging Networking Experiments and Technology (CoNEXT)*, pages 54–66, 2018.

[62] Intel Corporation. Intel 82580EB/82580DB Gigabit Ethernet Controller Datasheet. http://www.intel.com/content/www/us/en/embedded/products/networking/82580-eb-db-gbe-controller-datasheet.html. Revision: 2.7, September 2015.

[63] Intel Corporation. Intel Ethernet Controller 710 Series Datasheet. http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xl710-10-40-controller-datasheet.pdf. Revision: 2.9, April 2017.

[64] Ixia. Ixia traffic generator. https://www.ixiacom.com.

[65] V. Jalaparti, P. Bodík, S. Kandula, I. Menache, M. Rybalkin, and C. Yan. Speeding up distributed request-response workflows. In *Proceedings of the ACM SIGCOMM 2013 Conference*, pages 219–230, 2013.

[66] E. Jeong, S. Woo, M. A. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 489–502, 2014.

[67] G. Joshi, E. Soljanin, and G. W. Wornell. Efficient Redundancy Techniques for Latency Reduction in Cloud Systems. *TOMPECS*, 2(2):12:1–12:30, 2017.

[68] T. Kaler, Y. He, and S. Elnikety. Optimal Reissue Policies for Reducing Tail Latency. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 195–206, 2017.

[69] M. Kambadur, T. Moseley, R. Hank, and M. A. Kim. Measuring interference between live datacenter applications. In *Proceedings of the 2012 ACM/IEEE Conference on Supercomputing (SC)*, page 51, 2012.

[70] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Chronos: predictable low latency for data center applications. In *Proceedings of the 2012 ACM Symposium on Cloud Computing (SOCC)*, page 9, 2012.

[71] H. Kasture and D. Sánchez. Ubik: efficient cache sharing with strict qos for latency-critical workloads. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIX)*, pages 729–742, 2014.

[72] S. Kim, Y. He, S. won Hwang, S. Elnikety, and S. Choi. Delayed-Dynamic-Selective (DDS) Prediction for Reducing Extreme Tail Latency in Web Search. In *Proceedings of the 8th International Conference on Web Search and Web Data Mining (WSDM)*, pages 7–16, 2015.

[73] M. Kogias, S. Mallon, and E. Bugnion. Lancet: A self-correcting Latency Measuring Tool. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, pages 881–896, 2019.

[74] M. Kogias, G. Prekas, A. Ghosn, J. Fietz, and E. Bugnion. R2P2: Making RPCs first-class datacenter citizens. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, 2019.

[75] R. R. Kompella, K. Levchenko, A. C. Snoeren, and G. Varghese. Every microsecond counts: tracking fine-grain latencies with a lossy difference aggregator. In *Proceedings of the ACM SIGCOMM 2009 Conference*, pages 255–266, 2009.

[76] J. Kosaian, K. V. Rashmi, and S. Venkataraman. Parity models: erasure-coded resilience for prediction serving systems. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 30–46, 2019.

[77] C. Lee, C. Park, K. Jang, S. B. Moon, and D. Han. Accurate Latency-based Congestion Feedback for Datacenters. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC)*, pages 403–415, 2015.

[78] J. Leverich and C. Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the 2014 EuroSys Conference*, pages 4:1–4:14, 2014.

[79] J. Li, K. Agrawal, S. Elnikety, Y. He, I.-T. A. Lee, C. Lu, and K. S. McKinley. Work stealing for interactive services to meet target latency. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 14:1–14:13, 2016.

[80] J. Li, N. K. Sharma, D. R. K. Ports, and S. D. Gribble. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *Proceedings of the 2014 ACM Symposium on Cloud Computing (SOCC)*, pages 9:1–9:14, 2014.

[81] libevent – an event notification library. https://libevent.org/.

[82] X. Lin, Y. Chen, X. Li, J. Mao, J. He, W. Xu, and Y. Shi. Scalable Kernel TCP Design and Implementation for Short-Lived Connections. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXI)*, pages 339–352, 2016.

[83] Y. Lu, Q. Xie, G. Kliot, A. Geller, J. R. Larus, and A. G. Greenberg. Join-Idle-Queue: A novel load balancing algorithm for dynamically scalable web services. *Perform. Eval.*, 68(11):1056–1071, 2011.

[84] Lucene nightly benchmarks. https://home.apache.org/~mikemccand/lucenebench/.

[85] Lucene++. https://github.com/luceneplusplus/LucenePlusPlus.

[86] M. Primorac. Repository. https://github.com/MihaelaMia/measure-killer-us, 2017.

[87] C. Macdonald, N. Tonellotto, and I. Ounis. Learning to predict response times for online query scheduling. In *Proceedings of the 35th International ACM SIGIR conference on Research and Development in Information Retrieval (SIGIR)*, pages 621–630, 2012.

[88] A. Maricq, D. Duplyakin, I. Jimenez, C. Maltzahn, R. Stutsman, R. Ricci, and A. Klimovic. Taming Performance Variability. In *Proceedings of the 13th Symposium on Operating System Design and Implementation (OSDI)*, pages 409–425, 2018.

[89] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch. Power management of online data-intensive services. In *Proceedings of the 38th International Symposium on Computer Architecture (ISCA)*, pages 319–330, 2011.

[90] Mellanox Technologies. Mellanox ConnectX-4. https://www.mellanox.com/products/infiniband-adapters/connectx-4.

[91] Microsoft. Azure functions. https://azure.microsoft.com/services/functions/.

[92] A. Mirhosseini and T. F. Wenisch. The Queuing-First Approach for Tail Management of Interactive Services. *IEEE Micro*, 39(4):55–64, 2019.

[93] R. Mittal, V. T. Lam, N. Dukkipati, E. R. Blem, H. M. G. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats. TIMELY: RTT-based Congestion Control for the Datacenter. In *Proceedings of the ACM SIGCOMM 2015 Conference*, pages 537–550, 2015.

[94] J. C. Mogul and J. Wilkes. Nines are Not Enough: Meaningful Metrics for Clouds. In *Proceedings of The 17th Workshop on Hot Topics in Operating Systems (HotOS-XVII)*, pages 136–141, 2019.

[95] M. Mohiuddin, M. Primorac, E. Stai, and J.-Y. L. Boudec. FCR: Fast and Consistent Controller-Replication in Software Defined Networking. *IEEE Access*, 7:170589–170603, 2019.

[96] Netperf. http://www.netperf.org/netperf/.

[97] NGINX. https://www.nginx.com/.

[98] D. M. Novakovic, N. Vasic, S. Novakovic, D. Kostic, and R. Bianchini. DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)*, pages 219–230, 2013.

[99] W. H. Organization et al. Coronavirus disease 2019 (covid-19): situation report, 72. 2020.

[100] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: distributed, low latency scheduling. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 69–84, 2013.

[101] X. Ouyang, P. Garraghan, B. Primas, D. McKee, P. Townend, and J. Xu. Adaptive Speculation for Efficient Internetware Application Execution in Clouds. *ACM Trans. Internet Techn.*, 18(2):15:1–15:22, 2018.

[102] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. E. Anderson, and T. Roscoe. Arrakis: The Operating System Is the Control Plane. *ACM Trans. Comput. Syst.*, 33(4):11:1–11:30, 2016.

[103] G. Prekas, M. Kogias, and E. Bugnion. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 325–341, 2017.

[104] G. Prekas, M. Primorac, A. Belay, C. Kozyrakis, and E. Bugnion. Energy proportionality and workload consolidation for latency-critical applications. In *Proceedings of the 2015 ACM Symposium on Cloud Computing (SOCC)*, pages 342–355, 2015.

[105] M. Primorac, E. Bugnion, and K. J. Argyraki. How to Measure the Killer Microsecond. In *Proceedings of the 2017 Workshop on Kernel-Bypass Networks (KBNETS@SIGCOMM)*, pages 37–42, 2017.

[106] M. Primorac, E. Bugnion, and K. J. Argyraki. How to Measure the Killer Microsecond. *Computer Communication Review*, 47(5):61–66, 2017.

[107] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. R. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceedings of the 41st International Symposium on Computer Architecture (ISCA)*, pages 13–24, 2014.

[108] Z. Qiu and J. F. Pérez. Evaluating the Effectiveness of Replication for Tail-Tolerance. In *Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 443–452, 2015.

[109] K. V. Rashmi, M. Chowdhury, J. Kosaian, I. Stoica, and K. Ramchandran. EC-Cache: Load-Balanced, Low-Latency Cluster Caching with Online Erasure Coding. In *Proceedings of the 12th Symposium on Operating System Design and Implementation (OSDI)*, pages 401–417, 2016.

[110] L. Ravindranath, J. Padhye, R. Mahajan, and H. Balakrishnan. Timecard: controlling user-perceived delays in server-based mobile applications. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 85–100, 2013.

[111] L. Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, pages 101–112, 2012.

[112] L. Rizzo. Revisiting Network I/O APIs: The netmap Framework. *ACM Queue*, 10(1):30, 2012.

[113] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore. OFLOPS: An Open Framework for OpenFlow Switch Evaluation. In *Proceedings of the 13th International Conference on Passive and Active Measurement (PAM)*, pages 85–95, 2012.

[114] S. M. Rumble, D. Ongaro, R. Stutsman, M. Rosenblum, and J. K. Ousterhout. It's Time for Low Latency. In *Proceedings of The 13th Workshop on Hot Topics in Operating Systems (HotOS-XIII)*, 2011.

[115] E. Schurman and J. Brutlag. The User and Business Impact of Server Delays, Additional Bytes, and HTTP Chunking in Web Search. https://conferences.oreilly.com/velocity/velocity2009/public/schedule/detail/8523.

[116] N. B. Shah, K. Lee, and K. Ramchandran. When Do Redundant Requests Reduce Latency? *IEEE Trans. Communications*, 64(2):715–722, 2016.

[117] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making middleboxes someone else's problem: network processing as a cloud service. In *Proceedings of the ACM SIGCOMM 2012 Conference*, pages 13–24, 2012.

[118] J. F. Shortle, J. M. Thompson, D. Gross, and C. M. Harris. *Fundamentals of queueing theory*, volume 399. John Wiley & Sons, 2018.

[119] R. Sites. Data Center Computers: Modern Challenges in CPU Design. https://youtu.be/QBu2Ae8-8LM?t=1205.

[120] Solarflare Flareon Ultra SFN8522-PLUS. https://www.solarflare.com/Media/Default/PDFs/SF-116323-CD-LATEST_Solarflare_SFN8522-PLUS_Product_Brief.pdf.

[121] Spirent Communications. Spirent Test Modules and Chassis. https://www.spirent.com/Products/TestCenter/Platforms/Modules.

[122] A. Sriraman and T. F. Wenisch. µTune: Auto-Tuned Threading for OLDI Microservices. In *Proceedings of the 13th Symposium on Operating System Design and Implementation (OSDI)*, pages 177–194, 2018.

[123] P. L. Suresh, M. Canini, S. Schmid, and A. Feldmann. C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection. In *Proceedings of the 12th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 513–527, 2015.

[124] D. Turull, P. Sjödin, and R. Olsson. Pktgen: Measuring performance on high speed networks. *Computer Communications*, 82:39–48, 2016.

[125] A. Vulimiri, P. B. Godfrey, R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker. Low latency via redundancy. In *Proceedings of the 2013 ACM Conference on Emerging Networking Experiments and Technology (CoNEXT)*, pages 283–294, 2013.

[126] J. Wagner. Why Performance Matters. https://developers.google.com/web/fundamentals/performance/why-performance-matters/.

[127] D. Wang, G. Joshi, and G. W. Wornell. Efficient task replication for fast response times in parallel computation. In *Proceedings of the 2014 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 599–600, 2014.

[128] Q. Wang, Y. Kanemasa, J. Li, D. Jayasinghe, T. Shimizu, M. Matsubara, M. Kawaba, and C. Pu. Detecting Transient Bottlenecks in n-Tier Applications through Fine-Grained Analysis. In *Proceedings of the 33rd IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 31–40, 2013.

[129] M. T. Weber. Measuring network service latencies with a programmable dataplane. Technical report, EPFL, 2019.

[130] A. Wierman and B. Zwart. Is Tail-Optimal Scheduling Possible? *Operations Research*, 60(5):1249–1257, 2012.

[131] Wikipedia: Database download. https://en.wikipedia.org/wiki/Wikipedia:Database_download#english.

[132] Z. Wu, C. Yu, and H. V. Madhyastha. CosTLO: Cost-Effective Redundancy for Lower Latency Variance on Cloud Storage Services. In *Proceedings of the 12th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 543–557, 2015.

[133] H. Xu and B. Li. RepFlow: Minimizing flow completion times with replicated flows in data centers. In *Proceedings of the 2014 IEEE Conference on Computer Communications (INFOCOM)*, pages 1581–1589, 2014.

[134] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey. Bobtail: Avoiding Long Tails in the Cloud. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 329–341, 2013.

[135] X. Yang, S. M. Blackburn, and K. S. McKinley. Elfen Scheduling: Fine-Grain Principled Borrowing from Latency-Critical Workloads Using Simultaneous Multithreading. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*, pages 309–322, 2016.

[136] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *Proceedings of the 8th Symposium on Operating System Design and Implementation (OSDI)*, pages 29–42, 2008.

[137] A. Zaostrovnykh, S. Pirelli, L. Pedrosa, K. J. Argyraki, and G. Candea. A Formally Verified NAT. In *Proceedings of the ACM SIGCOMM 2017 Conference*, pages 141–154, 2017.

[138] Y. Zhang, D. Meisner, J. Mars, and L. Tang. Treadmill: Attributing the Source of Tail Latency through Precise Load Testing and Statistical Inference. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*, pages 456–468, 2016.

[139] N. Zilberman, M. P. Grosvenor, D. A. Popescu, N. M. Bojan, G. Antichi, M. Wójcik, and A. W. Moore. Where Has My Time Gone? In *Proceedings of the 18th International Conference on Passive and Active Measurement (PAM)*, pages 201–214, 2017.

Mia Primorac

Address:	Chemin de Renens 52B
	CH-1004 Lausanne, Switzerland
Phone:	+41 78 976 85 97
Date of Birth:	19th January 1991
E-mail:	mia.primorac@gmail.com
LinkedIn:	www.linkedin.com/in/miaprimorac

Research Interests

My main interests are in the area of computer networks, distributed systems and operating systems, including latency optimizations in datacenters and Software Defined Networks (SDNs), as well as network measurements.

Education

Sep 2014 – Jun 2020	PhD student in Computer Science EPFL, Lausanne, Switzerland Thesis topic: Understanding and Mitigating Latency Variability of Latency-Critical Applications Advisors: Prof. Edouard Bugnion & Prof. Katerina Argyraki
Sep 2012 – Jul 2014	Master's degree in Computing *with honors* *Major: Computer Science* University of Zagreb, Zagreb, Croatia Thesis topic: Inferring Presence Status on Mobile Devices Advisor: Prof. Domagoj Jakobović
Sep 2009 – Jul 2012	Bachelor's degree in Computing *Major: Telecommunications and Informatics* University of Zagreb, Zagreb, Croatia

Work Experience

Sep 2014 – Jun 2020	Doctoral Assistant, EPFL, Lausanne, Switzerland *Data Center Systems Lab and Network Architecture Lab*
Jul 2016 – Nov 2016	Graduate Research Intern, Intel Labs, Hillsboro, OR, USA *Networking Platforms Lab* Supervisor: Dr. Charlie Tai; mentor: Dr. Ren Wang Topic: Latency optimization of pipelined network functions
Jul 2013 – Sep 2013	Summer Intern, EPFL, Lausanne, Switzerland *Operating Systems Laboratory* Supervisor: Prof. Willy Zwaenepoel; mentor: Dr. Amitabha Roy Topic: Efficient Python bindings for a graph processing system *labos.epfl.ch/x-stream*

Teaching Experience

2019	*C++ Programming* at EPFL (with Dr. J.-C. Chappelier)
2017 – 2019	*Computer Networks* at EPFL (with Prof. K. Argyraki)
2015 – 2018	*Systems Oriented Programming* at EPFL (with Dr. J.-C. Chappelier)
2013 – 2014	*Introduction to Java Programming Language* at University of Zagreb

Publications and Technical Reports

- M. Mohiuddin, M. Primorac, E. Stai, J.-Y. Le Boudec; IEEE Access Volume 7; 2019:
  FCR: Fast and Consistent Controller-Replication in SDN
- M. Primorac, E. Bugnion, K. Argyraki; SIGCOMM CCR October 2017:
  How to Measure the Killer Microsecond (*invited publication*)
- M. Primorac, E. Bugnion, K. Argyraki; SIGCOMM KBNets '17:
  How to Measure the Killer Microsecond (*Best Paper Award*)
- A. Belay, G. Prekas, M. Primorac, A. Klimovic, S. Grossman, C. Kozyrakis, E. Bugnion;
  TOCS-2016-0065:
  The IX Operating System: Combining Low Latency, High Throughput and Efficiency in a Protected Dataplane
- G. Prekas, A. Belay, M. Primorac, A. Klimovic, S. Grossman, M. Kogias, B. Gütermann, C. Kozyrakis, E. Bugnion;
  EPFL Technical Report 218568, May 2016:
  IX Open-source v1.0 – Deployment and Evaluation Guide
- G. Prekas, M. Primorac, A. Belay, C. Kozyrakis, E. Bugnion; SoCC '15:
  Energy Proportionality and Workload Consolidation for Latency-critical Applications

Patents

R. Wang, M. Primorac, T.-Y. C. Tai, S. Edupuganti, J. J. Browne. 2017. Technologies for Dynamic Batch Size Management. US 2019 / 0007349 A1, filed Jun. 30, 2017. Patent Pending.

Scholarships and Awards

2017	SIGCOMM KBNets '17 **Best Paper Award**
2014 – 2015	EPFL PhD Fellowship
2014	Merit-based scholarship from University of Zagreb
2012 – 2013	National Croatian Excellence Scholarship

Skills and Languages

PROGRAMMING LANGUAGES:
C (proficient), Python (proficient), Java (Oracle Certified Professional), C++ (advanced), SQL (advanced), C# (advanced), PL/SQL (basic), MATLAB and Octave (basic)

OTHER SKILLS:
networking (including SDN, NFV, kernel bypass (Intel DPDK) and TCP/IP), **operating**

systems (Linux), **cloud computing** (AWS, Docker), **version control** (Git), **database** and **data warehouse** design, **data analysis and machine learning**, **NoSQL and Big Data**

LANGUAGES:
Croatian (native), English (C2), French (B2), German (B1). Reference:
en.wikipedia.org/wiki/Common_European_Framework_of_Reference_for_Languages

Invited Talks

02.03.2020	*What to do with idle resources in the Cloud* Microsoft Research, Cambridge, UK
07.02.2020	*When to hedge in interactive services and how to find energy-proportional resource-allocation policies* Oracle Labs, Zurich, Switzerland
21.08.2017	*How to Measure the Killer Microsecond* SIGCOMM KBNets'17, Los Angeles, USA
24.03.2016	*Experience from doctoral study at EPFL for students at Faculty of Electrical Engineering and Computing* IEEE Croatia Education Section (E25), Zagreb, Croatia

Supervised Students

Konstantinos Prasopoulos, semester project, EPFL, 2018/2019: *Implementation and End-to-End Evaluation of the HULL Architecture*

Extra-Curricular

2017 – 2019	*External Relations Manager* in EPFL PhD association *PolyDoc.* PolyDoc website: www.epfl.ch/campus/associations/list/polydoc/ Coordinating with other associations, representing and gathering PhD students from all the faculties.
2018	*Mentor* in *GirlsCoding.org.* Mentoring coding workshops for 6 to 16 years old girls.
2018 – 2019	Finals of a science-communication movie competition *Exposure Science Film Hackathon* - "*Searching for Love*" Screenings: 1st round – Dec. 5th 2018 (Cinema Cityclub Pully); Finals – April 13th 2019 (Auditorium Arditi, Geneva).

References

Thesis director: Prof. Edouard Bugnion, EPFL, edouard.bugnion@epfl.ch
Thesis co-director: Prof. Katerina Argyraki, EPFL, katerina.argyraki@epfl.ch
Collaborator: Prof. Jean-Yves Le Boudec, EPFL, jean-yves.leboudec@epfl.ch
Internship mentor: Dr. Ren Wang, Intel Labs, ren.wang@intel.com