EPFL

Thèse n° 8017

Hardware and Software Support for RPC-Centric Server Architecture

Présentée le 5 septembre 2022

Faculté informatique et communications
Laboratoire d'architecture de systèmes parallèles
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

Mark Johnathon SUTHERLAND

Acceptée sur proposition du jury

Prof. M. Jaggi, président du jury
Prof. B. Falsafi, Prof. A. Daglis, directeurs de thèse
Prof. M. Silberstein, rapporteur
Dr S. Reinhardt, rapporteur
Prof. E. Bugnion, rapporteur

École
polytechnique
fédérale
de Lausanne

2022

To my darling Laurianne.

Acknowledgements

Completing a PhD has unequivocally been the most challenging thing I have ever done, and moreover the journey was a task for which I was undoubtedly not mentally prepared to undertake. Entering the program, I naïvely expected only academic and intellectual growth. Upon leaving, I now realize that the personal growth required to complete the journey has reached almost every area of my character. It was only possible for me to finish this work with the support of my advisors, and a great many friends, family members, and colleagues, whom all are owed an extreme debt of gratitude and I will thank by name throughout this section. However, I wish to begin this section by mentioning the most important lesson I learned in the past six years, from the most influential person in my life.

If anyone would be first,
let him be last of all,
and servant of all.
— Jesus of Nazareth

It is only now, on the far side of the PhD journey, that I realize the depth and practical implications of Jesus' teaching in that statement. Although the value of this was difficult to see at first, and attempting to embody it was even more difficult, I look forward to attempting to live and work with as much of an "others-focused" mindset as I can possibly muster. To those readers who know me, and have tolerated my many lapses towards a selfish attitude, I want to thank you for your patience and forgiveness! To those who do not, I can only say that I have benefitted from many people who have sought my good over or beside their own, and those experiences are the ones I am most thankful for and wish to create for others.

Acknowledgements

At this point, I wish to thank my two advisors, Babak and Alex, for all of their effort, collaboration, and perseverance in working with me and helping me grow as a professional and researcher. Babak and I share the trait of perfectionism, and his method of doing research results in some of the most consistently high-quality output that I know of. From him, I learned how to communicate ideas much more clearly, focus only on the important questions, and prioritize all the various interests and possibilities bouncing around in my brain. Although these lessons were undoubtedly important for me, I am the most thankful for two specific moments where he encouraged me to keep pushing, despite my lack of belief that I could reach the finish line. I am also grateful for the fact that Babak plays as hard as he works. All of his students get to enjoy arguably the best coffee at EPFL (and maybe even beyond), lab dinners, ski trips, and movie nights, all of which objectively made my PhD experience more enjoyable. He also made sure that we absolutely never needed to worry about funding our research, freeing us up to work in any direction for which we can make a strong case.

Alex is one of the most knowledgeable and well-rounded people I have ever met, and I am very thankful that he joined my PhD journey as an official co-advisor in my fourth year. He was kind enough to invest a major amount of his time into working with me and improving my research, all in addition to his own students at Georgia Tech. Additionally, being my immediate predecessor in the lab working on the systems and networking research angle, I benefitted significantly from his insights in both of those areas. In particular, Alex had a knack for asking questions about my ideas or results that revealed shortcomings in the work, with a remarkable degree of prescience. We also were extremely productive in working together, and found that our writing styles meshed extremely well. Alex is also the closest person I know to attaining five nines of availability – there were many times I wrote to him at five in the morning on a Sunday, not expecting a response for days, and I heard back within hours. I have no idea how he keeps going, but I know that I am thankful for all those responses. His students are very blessed to have an advisor like Alex.

The next two people I wish to thank were my first contacts in the world of computer architecture: Natalie Enright Jerger and Andreas Moshovos, at the University of Toronto. Natalie was kind enough to extend an offer to me to supervise my Master's thesis, and consistently exceeded my expectations in treating me like a colleague and a fellow researcher even though

I was not a PhD student. Natalie's expertise, demeanour, and infectiously positive attitude made my first research experience outstanding. Without the phenomenal experience I had in her group, I might not have continued on in the research world. She also continues to stay in touch even long after I left her group in Toronto. Andreas taught me to love the intimate details of out-of-order processor microarchitecture (and everything related to speculation!), and even gave me the opportunity to help teach it by assisting his class. His recommendation to choose EPFL over other offers helped me become the person I am today.

As one can see, I have been blessed with a great number of outstanding instructors, from Toronto to EPFL. At EPFL, I wish to thank Ed Bugnion, George Candea, and Katerina Argyraki for teaching me about computer systems, and broadening my horizons beyond traditional computer architecture. From the first few days of their course called "Principles of Computer Systems" (POCS), I knew that I wanted to work on systems for the long term. During my PhD, I had the opportunity to be part of POCS three times – once as a student and twice as a teaching assistant – and I learned a huge amount each time. I cannot say enough about how rich my POCS experience was; it was without a doubt the best course I have ever taken.

I would also like to thank the many collaborators I had who reside a bit further abroad than my immediate EPFL orbit. Specifically, I would like to thank Dionisios Pnevmatikatos at NTUA, and Virendra Marathe at Oracle Labs. Both of them graciously gave their time to help me during my first publication, Dionisios from a network perspective and Virendra from a distributed memory one. Additionally, I thank my thesis committee members Ed Bugnion, Mark Silberstein, and Steve Reinhardt for graciously investing their time in reviewing my thesis proposal, giving me feedback, and helping guide my final two years towards a more productive conclusion. Thank you also to Martin Jaggi for agreeing to serve as my thesis jury president on such short notice.

Next, I have to thank the people in two labs at EPFL, beginning with all of my fellow PARSA students. We are all walking the same journey, and although I was slightly less connected to the lab's social life than others were, I greatly appreciated the moments where we were able to share in the positives and negatives of the PhD life together. In particular, I want to thank Arash Pourhabibi and Dmitrii Ustiugov, who were my two most frequent collaborators

other than my advisors. As a completely green first-year student, I will never forget my first experience of trying to do high-impact research work with Dmitrii. Dmitrii is probably the only person with whom I can technically argue at levels of intensity that would usually be considered far too high-volume for an office environment, and still have both of us come out smiling and believing we had a productive meeting. He is also one of the most persistent people I have ever met – he will never, ever give up. I am so proud of his many career successes, and I look forward to continuing to try to convince him of the concept of objective morality in our lunch meetings to come!

Arash and I worked together daily for two of the last three years of my PhD, and shared two publications. He is the exact opposite of Dmitrii in his level of personal intensity, but no less in his level of technical proficiency and persistence. I will always remember how the two of us worked in completely opposite time zones, both in the dark – me before the sunrise, and him after the sunset. During our years of working together, I learned a lot about software development that I would have not encountered otherwise – in particular, I have tried to take onboard his love for testing and good development practices even when it is inconvenient. Finally, Arash knows about good food, where to find it, and how to make legitimately good dishes, which I have been lucky enough to share many times. I am thankful for the time we spent working together and look forward to more in the future.

Next, I want to thank the rest of the students that were co-running threads during my time in PARSA: Javier, Mario, Hussein, Sid, Ognjen, Simla, Nooshin, Ali, Shanqing, Yuanlong, Rafael, Zilu, Dina, Canberk, and Fatih. Javier was the only other student as obsessed with out-of-order microarchitecture as me, until Ali came along... Javier, *put that coffee down!* Mario was our lab's resident programming languages expert and the only other person in the lab who followed the NHL. I still can't understand half the languages he knows, or how he knows more about hockey than Canadians. Sid is the master of getting from A to B, no matter the cost. If something must be done, and he is convinced of the need, he will give you a positive result on time. If I had to choose one person to get something done under a deadline with major consequences for failure, I am picking him with no hesitation.

Hussein and Ognjen were two students I shared many discussions about FPGAs with – Hussein

because of his love for them, and Ognjen because of his frustration. Although Hussein moved on to the enterprise world early, we shared a good year with our desks in the lab lounge – I will never forget being hit on the head many times with a rubber ball in the middle of work, because he decided to throw it at me from across the room without calling out. Ognjen might be the only person I know who worries about everything as much as me, and I am really thankful for the long discussions we had in an attempt to stop overthinking many issues. I really enjoyed all my lunch discussions with Rafael about finance and geopolitics, and arguing about subjects on which we both were objectively uneducated. Although I did not have as many interactions with Nooshin, Simla, Shanqing, Yuanlong, Zilu, Canberk, Dina, and Fatih as I would have liked, I enjoyed the few good times we had, such as at lab dinners.

I also thank the administrative and technical staff that keep PARSA running on a day-to-day basis. In particular, I want to acknowledge Stéphanie and Valérie for handling scheduling, travel, administration, and even sometimes translation. Both of them are not only helpful, but they do their jobs with a smile, which was more appreciated than they perhaps know. For the short time we were in PARSA together, I saw the wizardry of Rodolphe as a sysadmin. Then, when he moved on to new opportunities and left me the task of maintaining the infrastructure, I saw his wizardry even more clearly...

I want to also thank the students in the other labs that I was close to: Marios, Adrien, Rishabh, and Georg. Marios was sometimes like a third advisor to me, who was interested in incredibly similar problems and technical ideas. Like clockwork, I would think of a system connection to one of my ideas, and he had either considered the exact same and rejected it for solid reasons, or could give me helpful advice on the spot. Thank you Marios for all the time you spent answering my questions and early stage ideas. Adrien was one of the most open people I have ever met, almost brutally honest about himself and others. I think he is the only other EPFL student I know who is as emotional as me, something that I greatly appreciated. Rishabh has always impressed me with his ability to dive into a completely different research field than he originally intended to study, and still publish top-tier papers. Although I didn't get the chance to work with either of them much, I am looking forward to staying in touch in the future, and I am extremely grateful for the time they spent with me when I was completing the last few chapters of this thesis. Their feedback made the work objectively better. Georg, I appreciated

our connection in my first year very much, particularly your spontaneous invitations – this made me feel very welcome as a North American new to Europe.

I am also incredibly indebted to the wonderful people I met through Westlake Church Lausanne. When I first arrived in Lausanne on a Saturday, I searched on Google for "english speaking church Lausanne", showed up the next day, and have hardly missed a Sunday morning since then. It is not an exaggeration to say that without the many connections I made through Westlake, I would not have reached the end of my PhD, or have the life I have today. If I tried to list every person who encouraged me, or every conversation I enjoyed, I would go on for twenty pages. Let me just say that I love all of you, our church community, and hope that I can continue to be an encouragement to all of you for the long run. Among the many people I have come to know at Westlake, I have to specifically thank my closest friends for standing with me through every circumstance, friends that are so close I think of them like family: Ruan, Iddo, Anne-Lize, Clara, Noel, and the Rizzos.

During the past six years, Ruan and Iddo have been like the brothers I never had in my family, with whom I could share my competitive nature and intensity, but also my soft side. I have always been able to count on Ruan to be a calming influence, for relentless support, for our shared obsession with cycling and riding up mountains, and our many times of prayer at all hours of the day. I am deeply honoured that he stood with me as the best man at my wedding. Iddo's thoughtfulness and skill in active listening has always impressed me, in addition to his honesty in sharing the lessons which he learned from his father with all of us. Also, I could always count on Iddo to appreciate a nice bottle of red wine together. I will always treasure our conversations during the multiple times you came to visit, as well as on our shared holidays. Thank you both for all your support, and I very much look forward to the future of our friendship.

Anne-Lize, Noel, and Clara have become big and little sisters to me, who were there for me during many occasions. All three of them have always been a major encouragement to me in their attitude, faith, and kindness. In particular, I want to acknowledge Anne-Lize and her family, who probably prayed more for me than I did myself. Also, the shared obsession Noel and I have for Bible Project podcasts and videos led to many diversions from this thesis work

that refreshed my concentration and motivation. I have the privilege of being one of Tom and Karen Rizzo's many pseudo-adopted children, and I want to thank them as well as their daughter Fiona, and son-in-law Adrian for being there for me when recovering from a sudden hospitalization (twice). I will never forget the support and care I felt from all of them.

Next, I want to thank my parents, Gord and Kathy, as well as my sister Sam, for all of their support and patience while I was on my PhD journey, especially because I moved so far away from small-town Alberta to do it. My parents taught me almost everything foundational that helped me grow into the person that I am. Additionally, I am fortunate enough to not just have them as parents, but also now more so as friends. My dad was one of the earliest sources of my involvement with computers, and probably he did not ever imagine that I would one day be receiving a PhD in computer science because of our home PC on which I played Minesweeper, Rogue, and Clusterball. He taught me the importance and dignity of hard work, the value of showing up and being there for others, and was there countless times as support when I moved away from home to study, and then even further away to do a PhD. I imagine my mom did not much appreciate my spending time inside staring at a screen, but her patience with that part of my young life definitely was a foundation in me choosing computers as a career. Also, I am grateful that my mom and I share an interest in exercise physiology and sports science, which we often have long text conversations about. Sam has been a constant source of humility to me, and although that may seem to be a cliché, I actually deeply appreciate it. Without her, I might have still believed that I could be an athlete. Despite carrying some disadvantages from our family's unfortunate lack of height, her work ethic almost got her to the Canadian national women's hockey team, and I am so proud of her.

Last, and certainly not least, I want to say thank you to my wife Laurianne and everyone in the Imhof, Eberli, and Geiser families. I was probably not the one they were expecting for Laurianne, but they welcomed me with an extreme amount of kindness and patience, particularly in view of all the struggles I have with expressing myself in French. However, they made me feel welcome, accepted, and like a member of their family even before Laurianne and I made the membership official. Laurianne joined my PhD journey in my third year, and stuck with me through all my self-doubt and difficulty, even though it was initially incredibly confusing for her to understand the many demands of completing a PhD. I could not have

imagined having a more caring girlfriend, fiancé, and now wife, and for that I am eternally grateful. Her consistent support through 5:30AM phone calls for prayer, and her presence with me through many late nights helped me more than she will ever comprehend. Nothing can make me more joyful than thinking about spending the rest of our lives together, and about how I can make her the happiest person in the world.

Finally, I would also like to thank the many individuals and organizations that financially supported my PhD journey through funding arrangements. I would like to acknowledge the EPFL, the Swiss National Science Foundation (SNSF), and the Swiss citizens in general whose taxes supported my work. This thesis was funded by the following grants, projects, and awards: EPFL graduate student funding from the Computer Science department, the SNSF's "Memory-Centric Server Architecture for Datacenters" and "Hardware/Software Co-Design for In-Memory Services" projects, Facebook's "Full-System Accelerated & Secure ML Collaborative Research" program, a Google Faculty Research Award, and Oracle Labs' Accelerating Distributed Systems grant.

Lausanne, August 17, 2022 – MJS.

Abstract

Online services have become ubiquitous in technological society, the global demand for which has driven enterprises to construct gigantic datacenters that run their software. Such facilities have also recently become a substrate for third-party organizations due to the advantages of moving infrastructure to the cloud. The task of developing, releasing, and maintaining software at datacenter scale has given rise to a software architecture employing many independent *microservices*, each accomplishing a single role and communicating using an enforced API, the most common of which is Remote Procedure Calls (RPCs). As microservices have become standard practice for datacenter-scale software, the datacenter's underlying components must support them efficiently.

The increasing adoption of microservice architectures implies a drastic growth in network communication, because each microservice receives and creates many RPCs that often execute for only a few microseconds (µs). Therefore, delivering users an interactive, low latency service becomes more challenging, because each request involves more interactions with the components implementing the communication stack. It is particularly difficult to ensure the latency of the slowest responses, called the "tail latency", is acceptable to the service's users. Datacenter system design is therefore undergoing a rapid shift to enable programmers to reap the benefits of microservices without their performance quandaries.

Handling RPCs from µs-scale software at the line rates of today's NICs – delivering up to 400Gbps – is an open challenge, which will require designing all layers of the communication stack to natively offer support for RPC semantics. Although the performance of the network and protocol layers has drastically improved by prioritizing RPCs as a primary design objective, server hardware has not yet done so. Therefore, we posit that now is the time for an RPC-centric server architecture to emerge to allow server endpoints to match the performance of their surrounding system components.

Abstract

To that end, this thesis introduces hardware and software support for RPC-centric server architecture. We first make the case that today's hardware-terminated network transport protocols grossly over-provision buffering because they are agnostic to the latency constraints inherent in each RPC – simply exposing such RPC-level information to hardware allows 1.25 – 2.2× better performance. Motivated by prior work demonstrating the RPC stack's burdensome cost, we then show how a previously proposed RPC stack accelerator can be integrated with the implementation of our aforementioned NIC protocol. Finally, we propose new NIC-driven load balancing policies that boost microservice throughput via improved locality, while simultaneously maintaining tail latency guarantees. Our proposals improve 99th% tail latency in data stores by $2-5.5\times$, and reduce instruction cache misses in stateless microservices by $1.1-1.8\times$. In summary, we present evidence that designing and implementing a server's NIC hardware to natively support RPC semantics removes protocol scalability bottlenecks *and* enables microservices to enjoy further performance benefits.

Keywords: datacenters, servers, microservices, remote procedure calls, hardware, load balancing, tail latency, network protocols, queueing theory, co-design

Résumé

Les services en ligne sont devenus omniprésents dans la société technologique forçant ainsi les grandes entreprises à construire des centres de données dédiés à ces services. Les tâches de développement, publication et gestion de logiciels à l'échelle d'un centre de données ont conduit à un paradigme divisant les services en plusieurs « microservices », chacun étant responsable d'un seul rôle et communiquant au moyen d'une interface applicative, la plus répandue prenant la forme d'appels de procédures à distance (RPC). Les microservices s'imposant aujourd'hui comme la norme pour le développement de logiciels à l'échelle d'un centre de données, les infrastructures doivent évoluer afin d'en favoriser l'exécution.

L'utilisation croissante de microservices augmente considérablement le trafic interne au centre de données. Chaque microservice reçoit et envoie plusieurs RPCs qui, pour la plupart, ne prennent que quelques microsecondes à s'exécuter. Par conséquent, il devient de plus en plus difficile de fournir aux utilisateurs un service en ligne interactif et rapide : chaque requête, aussi simple soit-elle, nécessite de nombreuses interactions sur le réseau. Il devient alors particulièrement difficile de garantir la latence des requêtes les plus lentes, aussi appelée « latence de queue ». La conception de systèmes pour les centres de données fait par conséquent l'objet d'une transformation visant à permettre le développement de microservices avec de bonnes performances et garanties de latence.

La gestion de RPCs à l'échelle des microsecondes sur des cartes réseau modernes - avec des débits allant jusqu'à 400 Go/s – est un véritable défi, qui ne peut être adressé qu'en exposant un support pour les RPCs à chaque couche de la pile de communication réseau. La mise en place de système de priorités associés aux RPCs permet déjà de meilleures performances. Cependant, le manque de support pour de tels mécanismes dans d'autres composants du centre de données empêche des gains encore plus grands. Nous suggérons qu'il est temps, pour l'architecture des serveurs, de s'adapter aux microservices et de libérer le plein potentiel

de certains de leurs composants.
Dans ce but, cette thèse présente la conception à la fois logiciel et hardware (matériel électronique) de serveur dédié à l'exécution de RPCs et favorisant les meilleures performances possibles. D'abord, nous montrons que l'implémentation en hardware de la terminaison de protocoles de communications approvisionnent excessivement les mémoires tampons dû à leur manque de compréhension des contraintes de latence de chaque RPC. Nous montrons qu'il suffit d'exposer ces informations de la couche RPC au hardware afin d'améliorer la performance par 1.25-2.2x. Ensuite, motivés par des études préexistantes montrant le coût considérable de la couche RPC, nous montrons comment un accélérateur RPC peut s'intégrer avec la mise en oeuvre de notre interface réseau. Enfin, nous proposons des techniques d'équilibrage de charge entre les cœurs d'un serveur, permettant aux microservices d'exécuter plus rapidement et simultanément, tout en maîtrisant la latence de queue. Ces propositions améliorent la latence de queue au 99eme centile par 2-5.5x pour les microservices qui stockent des données, et réduisent les « cache miss » d'instructions par 1.1-1.8x pour ceux qui sont « stateless ». En résumé, nous démontrons que la conception et l'implémentation de l'architecture des serveurs avec un support direct pour les RPCs supprime les goulots d'étranglement de ces protocoles, permettant ainsi de meilleures performances pour les microservices.

Mots-clefs : centres des données, serveurs, microservices, appels de procédure à distance, materiel, équilibrage de charge, latence de queue, protocoles de réseau, co-conception, théorie des files d'attente

Contents

List of Figures

List of Tables

List of Equations

1 Introduction

Online services have become a multi-trillion dollar industry, with e-commerce corporations boasting a combined market capitalization of 9.9 trillion USD in 2019 [89] and the single biggest player generating global sales of 386 billion USD [15]. These massive profits have led to tremendous investments in hyperscale datacenters, to a degree that a single facility often consumes more than 100MW of electricity [31, §4]. In the drive to attract more customers, service providers are continually enhancing their first-party services with personalized user recommendations, curated advertisements, social network connections, and logging each action taken for legal privacy compliance [238]. Offering such features requires not only development effort, but also the necessary staff to debug, deploy, and maintain the entire service *in situ* while it is live. It is an understatement to say that such a task is merely complex.

In addition to the first-party online services that gave rise to today's hyperscale datacenters, software vendors operating at smaller scales have begun to shift their services to the cloud. The benefits of migrating to cloud-native software for small- to medium-scale enterprises are numerous. Not only is acquiring and managing dedicated hardware now unnecessary, but programmers can now take advantage of low-cost system infrastructure such as Amazon's plethora of Software-as-a-Service offerings [13, 14, 262] or simplified programming models such as serverless computing [5, 124]. The end result of the move to cloud-native software is an ongoing confluence between traditional first-party datacenter workloads (e.g., web search) and private enterprise applications now running in datacenters. Both styles of applications are beginning to share the same underlying software services [248] and platforms [31, §2.7.1],

and experience the same scalability and cost-reduction benefits.

As first-party and cloud-native applications continue to grow in scale and complexity, software architecture best practices have changed to match. In particular, datacenter providers and third-party enterprises have both turned towards using a software architecture that employs many independent *microservices*, which are modular and interact with each other to communally implement the online service's functionality. Each microservice is traditionally developed by a small independent team to maximize programmer productivity. A microservices architecture provides significant benefits for fault tolerance, scalability, debuggability, and perhaps most importantly, deployment velocity [31, 127, 179]. As such architectures mandate strict modularity between each component, they rely on a shared communication API to interact, the most common of which is Remote Procedure Calls (RPCs). Therefore, it is becoming commonplace for online service providers to develop a custom RPC layer which is used across the entire organization and exposed to third-party tenants – examples include Google's gRPC [92] and Facebook's own fork of Apache Thrift [19, 72].

The ongoing growth of RPC-connected microservices implies drastic growth in communication between microservices, and hence the performance of each server's communication stack is absolutely paramount. In particular, as every single message between microservices must traverse the RPC and transport protocol layers, as well as the datacenter network itself, the performance of all three modules is directly manifested to the end user. Compounding the challenge is the well-known fact that users are sensitive to even slightly delayed responses. For example, Akamai reports that a 100ms delay in page load time reduces sales conversion rates by 7% [7] and Google executives report that 0.5-second delays in search results reduce user traffic by 20% [91]. The common practice of auto-scaling each microservice across many servers [190] further complicates the task of providing a seamlessly interactive user experience, due to a challenge dubbed the "Tail at Scale". The Tail at Scale principle dictates that as the number of servers involved in handling a user request increases, it becomes highly probable that the slowest individual servers will dictate the latency experienced by the user [59]. Tail-tolerant computing has become one of the key challenges in the datacenter space, and has created a significant research impetus to curtail sources of tail latency in all layers of the system stack [54, 98, 163, 193, 223].

Although cross-stack research and development has delivered substantial performance gains in response to the need for rapid network communication, such innovations have exposed server-side inefficiencies as performance bottlenecks for datacenter-wide microservice deployments. In particular, optimized topologies [90, 243] and production rollouts of multi-hundred gigabit NICs [153] have created a growing gap between server packet processing capabilities and what the network can supply. The fact that the roadmaps for future Ethernet and InfiniBand fabrics forecast bandwidth growth to 1Tbps [112, 252] in the face of slowing silicon density scaling [57] will only make the gap more cavernous, especially considering that future optical networks promise richer connectivity with round-trip times of just a few nanoseconds [28]. The need to bridge the gap between network and server capabilities has prompted the development of fully customized protocol stacks to replace legacy TCP/IP (e.g., Google's Snap [178]), or at-scale deployments of hardware terminated protocols [44, 93]. As systems aspire to allow communication-intensive microservices to operate as near to the hardware's raw capabilities as possible, server architecture itself will inevitably become an equally important optimization point.

In this thesis, we posit that in order for microservices to take advantage of current and future developments in networking technology, server architectures must also evolve to contain support for the underlying operations that comprise RPCs. We argue for such architectural adjustments on three grounds. Firstly, microservices are beginning to exhibit microsecond-scale runtimes; prominent applications such as data stores [54, 129, 176], software-defined network functions [274], state-machine replication [132], and user management [276] only occupy the CPU for a few µs at a time. This trend is not a problem by itself. However, recent work has shown that the computation cost of the software RPC protocols repeatedly invoked by such microservices is comparable to the runtime of the applications themselves [159, 276]. Therefore, it is logical to consider deploying specialized hardware accelerators for RPC processing [222]. Any RPC processing accelerator directly and repeatedly interacts with the server's NIC, memory hierarchy, and on-chip fabric, and therefore these components should be co-designed to function well together.

Secondly, the core counts of server processors have continued to grow, and therefore so does the importance of load balancing RPCs among those cores. Advances in chiplet manufacturing

technology have made it possible for CPU vendors to construct many-core CPUs without prohibitive cost or yield concerns [134], and therefore many processors already feature 64 or more physical cores [36, 209, 266]. It is rigorously established that minimizing tail latency requires aggressively limiting the queueing time of RPCs, and that any load imbalance between CPU cores worsens with increased total core count. Therefore, deploying microservices with strict tail-latency constraints on such many-core servers mandates RPC load balancing to be a primary design concern, particularly because there is growing evidence that simple shortest-queue load balancing policies are not optimal for all applications [61, 182]. In particular, the fact that common microservices often have µs-scale latencies creates an emerging opportunity – an enhanced load balancing policy with the capability to improve cache locality while simultaneously limiting queueing delays can significantly boost the performance of such short RPCs.

Thirdly, the fact that underlying datacenter transport protocols have begun to be designed around the specific semantics and constraints of RPCs [150, 153, 193] creates opportunities for drastic server-side performance optimizations. In particular, hardware-terminated protocols that are agnostic to such RPC semantics contain significant inefficiencies in buffer management and data placement, which can be remedied by natively offering RPC support and exposing the application's tail latency goals to the NIC. We therefore conclude that it is time for an RPC-centric server architecture to emerge to complement the combination of latency-critical µs-scale software, improved networks and protocols with native RPC support, and powerful many-core server hardware.

To that end, this thesis leverages the power of integrating network and compute logic, a trend that is gaining traction in both academia [108, 109, 137, 199] and industry [121, 204, 207]. Our work begins with a highly optimized, integrated network interface baseline that has already eliminated common bottlenecks such as the server's I/O interconnect [199] and core-to-NI interactions [53]. Such a baseline is the ideal substrate for realizing the full benefits of an RPC-centric server architecture.

1.1 Thesis Goals

Our primary goal is to design a bespoke server architecture which is purpose-built to support the needs of communication-intensive microservices relying on RPCs. We begin by investigating the tradeoffs inherent in deploying hardware-terminated protocol stacks at datacenter scale, because such protocols have begun to see large-scale deployments as a response to demands for increased bandwidth at lower latency [44, 93]. We find that the immense bandwidth of today's NICs can create memory bandwidth interference with the CPU cores when RPC payloads spill out of the server's caches and into its DRAM. Furthermore, we identify that the source of the problem is the buffer bloat created by scaling hardware-terminated protocols to many communicating nodes, and show that the resulting bandwidth contention can reduce the throughput of a server by 2×. Although it is possible to remove bandwidth interference with existing NIC technology, it comes at the cost of sacrificing inter-core load balancing. Therefore, we set out to remove bandwidth contention while maintaining the ability to balance load between a CPU's many cores.

We then turn our attention to the co-design of emerging RPC protocol accelerators with integrated network interfaces. Although multiple prior works have proposed hardware accelerator designs that remove the cost of production RPC layers [136, 276], we remark that such accelerators must explicitly be designed alongside the server's NIC or much of their potential performance will be lost due to pathological inefficiencies when interacting with the host server's memory hierarchy. Specifically, performance is lost due to excessive task offloading from the CPU to the accelerator, or the need to shuffle requests between CPU cores in software to implement RPC load balancing. In agreement with prior work that shows RPC accelerators must be directly interfaced with the NIC [276], the second goal of this thesis is to dovetail such support into the pipelines of an integrated network interface.

Finally, we study load balancing policies for RPC-centric servers, in the context of µs-scale applications where cache locality is critical and can drastically reduce application runtime. Although cache locality has already been discussed as a critical factor for software packet processing workloads [76], we look beyond such applications and demonstrate specifically how improved locality exists in stateless microservices comprised of many independent functions,

and well known data caching layers. However, identifying sources of locality is not enough to ensure improved performance under tail latency constraints. Our final goal is therefore to enhance load balancing policies with the aforementioned sources of locality in mind, while simultaneously maintaining the microservice's tail latency guarantees.

1.2 Thesis Statement

Judicious design of a server's NIC hardware around RPC semantics removes bottlenecks to deploying microservices at datacenter-scale, and facilitates load balancing enhancements that boost throughput under tight tail-latency guarantees.

1.3 Thesis Contributions

Our work proposes modifications to both NIC hardware and microservice software to form the first RPC-centric server architecture. The first contribution of our work is NEBULA, a set of NIC hardware and protocol extensions that unblock hardware-terminated protocols from incurring memory bandwidth interference while retaining the flexibility of inter-core load balancing. The key insight enabling NEBULA is that a given microservice's tail latency target is effectively an implicit constraint on the queue depth that a particular RPC can experience. Therefore, the buffer growth associated with scaling hardware-terminated protocol stacks is unnecessary and can be shrunk without introducing excess tail latency. Under heavy load, RPCs that would have arrived at the end of a long queue are eagerly rejected, and clients are informed that their tail latency would have been violated.

Our second contribution extends prior work that designs an accelerator for the de-serialization operations underlying production RPC layers [275] to implement CEREBROS, a NIC-integrated accelerator that is capable of executing the entire RPC stack. CEREBROS is specifically designed to dovetail into the pipelines of NEBULA and its state-of-the-art Manycore Network Interface [53], so that the raw performance potential of existing de-serialization accelerators can be preserved in the context of many-core chips and multi-hundred gigabit NICs. Specifically, we demonstrate how CEREBROS can be architected and implemented across many *distributed* NIC components. A critical feature of CEREBROS is that it provides full visibility into the protocol

headers of incoming RPCs, making it an ideal substrate for our final contribution: enhancing state-of-the-art load balancing policies to account for application-level cache locality based on RPC header fields.

Finally, our third contribution proposes architectural and software extensions to integrated NICs in order to facilitate load balancing decisions for increased application-level cache locality. We show that although existing NIC technologies can be employed to enhance the cache locality of microservices, current best-case designs can only employ static policies that sacrifice load balancing, and therefore tail latency guarantees, by design. In contrast, we contribute the insight that knowledge of the state of each thread's input queue of RPCs is sufficient to develop locality-aware load balancing policies that reduce the microservices' processing times while maintaining tail latency.

Concretely, we demonstrate the following two scenarios: firstly, for microservices that primarily execute stateless application logic, instruction cache (I$) locality is plentiful but hidden by current policies that interleave the executions of various RPCs on the cores. Making load balancing decisions based on the temporal state of each CPU core's input queue reveals the hidden I$ locality, reducing cycles stalled on I$ misses and improving RPC throughput. Secondly, for microservices that cache data in key-value stores, locality manifests itself through concurrency control. Current policies treat all write requests equally, regardless of whether they are truly conflicting or not; we show that this decision leads to unnecessarily inflated tail latency *and* reduced throughput depending on the workload. We therefore introduce C-4, a simple NIC extension and matching software optimization which allows independent write requests to be balanced across cores, and creates batches of dependent writes that are applied as one to reduce synchronization events.

We implement and evaluate all the above contributions on top of the Scale-Out NUMA network protocol stack [199], extending its Remote Memory Controller to become a fully-fledged, RPC stack-terminating endpoint. Our evaluation focuses on throughput under tight 99th% latency constraints, showing that judicious hardware and software support for RPC-centric server architecture indeed fulfills our thesis design goals. In particular, we show that RPC-centric server architectures reap the following benefits:

1. Evaluated on a simple key-value store microservice, NEBULA grants 1.9× improved throughput under 99th% tail latency constraints, which is within 12% of the theoretical maximum our software stack can reach.
2. The RPC protocol acceleration of CEREBROS allows load balancing decisions to be made on fields comprising an RPC's header at the line rate of the server's NIC. Setting instruction cache locality as the primary goal reduces I$ misses by $1.1-1.8\times$ in a representative microservice taken from the DeathStarBench suite [81].
3. When deployed with C-4's hardware support to balance independent writes and software support for write batching, it is possible for key-value store microservices to attain $2-5.5\times$ better 99th% latency and $1.3-1.7\times$ higher throughput.

1.4 Thesis Organization

The remainder of this thesis is organized as follows. First, Chapter 2 makes the case for why server architecture should evolve to become RPC-centric, focusing on five critical software and technology trends. Then, we present the design, implementation, and evaluation of our work in three parts:

- **Part I** presents the design of an RPC-centric server architecture. Chapter 3 covers an overview of the architecture, the role of each contribution, and its relevant motivating trend(s) covered in Chapter 2. Chapter 4 then explains the design of NEBULA, beginning with the quandary between memory bandwidth interference and load imbalance before explaining the mathematical insights underpinning our proposed set of NIC and protocol extensions. Finally, Chapter 5 shows how locality-aware load balancing policies can do much more than simply limit RPC queueing, namely reduce the processing times of a microservice's RPCs themselves.
- **Part II** presents our implementation and evaluation of the design contributions in Part I. Chapter 6 lays out how we build NEBULA on top of Scale-Out NUMA, enabling its integrated NIC and lean protocol stack to terminate traffic from thousands of communicating nodes without experiencing bandwidth interference. Chapter 7 demonstrates how existing accelerators for RPC de-serialization can be unified with NEBULA to form

CEREBROS, a full RPC accelerator. CEREBROS is a key enabler for locality-aware load balancing policies because it grants access to each RPC's header fields as they arrive at the NIC's load balancer. Finally, Chapter 8 concretely instantiates hardware support for the locality-aware load balancing policies in Chapter 5 and evaluates their effectiveness.

- **Part III** discusses the rich field of related work in Chapter 9, and presents future research directions in Chapter 10. We conclude the thesis in Chapter 11.

1.4.1 Bibliographic Notes

This thesis was conducted under the supervision of my advisors: Babak Falsafi (EPFL), and Alexandros Daglis (Georgia Institute of Technology). Portions of it are based on collaboration with multiple colleagues: Arash Pourhabibi, Siddharth Gupta, Virendra Marathe, and Dionisios Pnevmatikatos. Chapters 4 and 6 are based on a publication in the *Proceedings of the 47th International Symposium on Computer Architecture (ISCA)* in 2020 [250]. Chapter 7, and parts of Chapters 5 and 8 are based on a publication in the *Proceedings of the 54th International Symposium on Microarchitecture (MICRO)* in 2021 [276], which is shared between this thesis and "Hardware-Software Co-Design of an RPC Processor" [222]. The contributions of CEREBROS presented in this thesis are the microarchitectural integration with NEBULA, and locality-aware load balancing. Finally, the remaining parts of Chapters 5 and 8 are based on a publication which has been accepted and will appear in the *2023 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

2 Why Design Server Architecture for RPCs?

In this section, we provide a background on current trends in the datacenter hardware and software environment that form supporting pillars to our contribution of an RPC-centric server architecture. First, we discuss the trend of software decomposition, and explain how the choice to adopt a microservices architecture results in a single user request generating hundreds or thousands of fine-grained RPCs with only a few microseconds (µs) of runtime each. We then describe how datacenter networks and systems software have evolved to support such applications, by proposing latency-optimized topologies and protocols to reduce round-trip times. The confluence of these two trends implies that server-side latency will form the majority of the service's execution time, as traditional bottlenecks such as protocol processing and network traversal disappear. Systems software optimizations have exacerbated the increasing contributions of server-side latency by providing user-level network I/O and removing sources of latency jitter. Next, we identify that server architectures are increasingly characterized by high core counts, which increases the importance of request-level load balancing, and motivate further considerations for enhancing current policies with the notion of application-level locality. Finally, we note that the emerging industry and academic trend towards network-compute co-design is a powerful tool to enable specializing server architecture to handle challenging µs-scale RPCs at the bandwidth of today's NICs, with tight tail latency guarantees.

2.1 Microservices and µs-Scale RPCs

Online service providers are faced with the challenging task of architecting highly available, interactive services that scale to billions of users, with new features being added weekly or daily [31]. To simultaneously achieve these challenging goals, software engineers are increasingly turning towards decomposing their services into many independent software modules which interact with each other to synthesize responses [81, 127]. When taken to the extreme, such software decomposition results in a *microservice architecture*, where each module performs only a single task and is isolated from the rest of the surrounding services by means of enforced modularity. Although microservices architectures are drastically more scalable and simpler to maintain, they rely heavily on a lightweight and scalable communication API, the most common of which is the Remote Procedure Call (RPC); examples of company-wide libraries include Google's gRPC [92] and Facebook's own fork of Apache Thrift [19, 72].

Despite the benefits of microservice architectures for developers and cloud providers, deploying software in such a manner creates new performance challenges. The first challenge is that common microservice software tiers are becoming increasingly fine-grained, with runtimes as short as a few µs for applications such as data stores [54, 129, 176], network function virtualization [274], state-machine replication [132], and simple user management business logic [276]. Fine-grained applications like these increase the amount of time spent in the layers of the computing stack that *support* the microservices, rather than the microservices themselves, increasing the importance of rapid network communication.

Structuring applications as microservices also increases the difficulty of providing guarantees known as "Service Level Objectives" (SLOs) on the latency of the online service's *slowest* responses [59]. Each additional software layer in the microservice graph complicates the process of controlling the service's tail latency, because a single user interaction will have a higher chance of experiencing a transient event (e.g., a context switch) [81, §8]. The depth to which production services are layered is profound, with Amazon reporting that a single page load typically accesses more than 150 internal services [60], and the request tree that is generated on a single Google search reportedly spanning across 1000 servers [58, 227]. Each of these numerous inter-microservice RPCs must transit multiple software and hardware queues,

introducing additional latency and the potential for significant delays due to the well known non-linear behaviour of queueing systems.

Fine-grained applications also imply that traditionally non-consequential server-side events (e.g., context switches, TLB misses, interrupts) may now form tail events resulting in SLO violations at higher layers of the overall service. To illustrate, consider the Masstree in-memory data store, which employs a hybrid trie-tree as an index to all key-value pairs it contains [176]. The sequential nature of the tree traversal implies that a simple `get` RPC may encounter TLB misses for multiple tree nodes, each of which costs 200–400ns, depending on page table caching, DRAM latency, and the number of levels in the architecturally-defined page table. As the measured mean service time for `gets` is approximately 1.25µs [54], multiple TLB misses can easily result in response latencies falling in the latency tail; other events such as context switches or interrupts would have an even greater effect.

Summary. Although microservices and tiered architectures significantly improve ease of deployment, debuggability, and fault tolerance for software engineers, the commensurately decreasing software runtimes and growing network communication costs will inevitably result in worse average *and* tail latency. Therefore, systems architects have commenced a hunt for the "killer microseconds" [32] across the datacenter system stack, particularly focusing on the layers providing inter-microservice communication.

2.2 Datacenter Networks and RPC Transport Protocols

In order to provide the requisite capacity for the internet services they support, today's datacenter fabrics have been rapidly scaling their capacity, because bandwidth demands are doubling every 12-15 months [243]. Today's hyperscale datacenter owners architect their networks using custom topologies that all resemble a multi-stage Clos network with many layers of aggregation [8, 90, 229, 243]. This is done in order to provide enough path diversity in the network core so that all server-to-server communication patterns can attain uniform high capacity, i.e., the rate of transmission should only be dictated by the available bandwidth of the endpoint NICs. Server NICs themselves have also rapidly been scaling in bandwidth, with current products already shipping that offer up to 400Gbps Ethernet bandwidth [208]

and 220M IOPS [206], up from the 40Gbps products that saw deployment in Google datacenters in 2012 [243]. Furthermore, the bandwidths of future NICs is likely to continue steady growth, because both Ethernet and InfiniBand consortia have publicly released roadmaps towards achieving 1.2-1.6Tbps NICs [112, 252]. These tremendous line rates raise challenges for server and switch architects alike because the time interval between small packets is just a few nanoseconds, and the networking hardware must be specifically designed to keep pace in order to maintain non-blocking communication.

Although next-generation switch hardware will be able to accommodate increased endpoint bandwidth (e.g., ASICs such as the Broadcom Trident 4 providing 32 ports of 400Gb/s Ethernet) [41], the primary metric of interest for µs-scale software is round-trip time (RTT), because every µs spent traversing the network counts towards the overall service's latency. To that end, datacenter operators have already begun to deploy hardware-assisted solutions like RDMA Over Converged Ethernet (RoCE) [93] and its derivatives (e.g., OmniPath) [114], and custom protocols like Microsoft's Catapult [44]. Hardware protocol termination not only shrinks end-to-end latency by delivering packets directly to the application layer and bypassing protocol processing, but drastically reduces the impact of tail-inducing events such as timeouts or retransmissions that occur because of traffic bursts [93]. Instead of deploying RoCE or custom FPGA hardware at scale and bearing the associated cost, alternative protocols such as Homa [193] and NDP [96] have been proposed to provide single-µs RTTs with software protocol termination.[1]

Whether hardware- or software-terminated, the key feature of emerging latency-optimized transports which is synergistic with the needs of microservices is that they are explicitly designed around the abstraction of an RPC. Exposing the notion of an RPC to the transport layer allows the transport to optimize for latency in multiple ways: optimizing for short messages over long flows [193], dynamically allocating in-network priorities, or using the receivers to actively pull new packets from senders [96]. The combination of the aforementioned techniques, and others not mentioned, has allowed RPC-centric transports to deliver datacenter-wide RTTs of 2-15µs, 100x better than traditional TCP/IP at the 99th percentile [193, §5.1].

A network with µs-scale RTTs makes it much more attractive to deploy applications as mi-

[1] NDP has even been realized in hardware by the NanoPU project to further reduce latency [108].

croservices, because the network's increased uniformity provides greater RPC performance predictability. Hence, individual service tiers can make heavier use of distributed systems techniques such as replication, auto-scaling, and even memory disaggregation when the underlying system software and network's performance allows it [168, 239]. Foundational research has already proposed network requirements to allow applications to act as distributed, disaggregated systems [82], and prototypes have already been proposed that implement network support for foundational techniques such as fault-tolerant state-machine replication [148]. We argue that such developments will permit application developers to be less constrained by the latency of network traversals, and excess latencies at the server endpoints will become far more impactful.

Summary. The combination of escalating NIC bandwidths, latency optimized network topologies, and protocols that provide µs-scale message delivery times provides a substrate for multi-tiered microservices to be deployed on any of the hundreds of thousands of servers making up a datacenter. In such a context, the burden lies squarely on the hardware and software at the network endpoints (i.e., the individual servers) to handle incoming traffic at the rates sustainable by the network itself. We now shift to discussing the specific trends in systems software that directly target this challenge.

2.3 Optimized Systems Software

An ecosystem whose applications already generate µs-scale requests, using networking hardware that provides single-digit µs round-trip times, must inevitably embrace a redesign of legacy network software to provide performance commensurate with the surrounding layers. Recent industry and academic systems have already made this shift, launching solutions that range from user-space network stacks (e.g., DPDK [171]) to software dataplanes such as IX [33] and Arrakis [217]. Each of these software architectures adopts similar principles, the most important of which is the choice to *run requests to completion* before moving on to other work; in contrast, standard Linux runs network processing in a fashion decoupled from the application requests, because it is designed for multiple-flow processing where single-request latency is not the primary concern. In order to benefit from the insights originating from academic proposals of RPC-oriented transports [96, 193], current work has even begun to

implement such protocols in the Linux kernel [213].

Server-side latency overheads are not only contained to transport protocol termination. Prior work has identified operating system overheads, available NIC offloads, interrupt routing and delivery, CPU power modes, and multi-socket NUMA latencies as factors that can induce excess server-side latency [163, 167]. However, a survey of related work in this area shows that system designers have already eliminated the critical latency bottlenecks for µs-scale RPCs through heavy optimization. For example, kernel-bypass solutions such as IX [33] or Google's Snap microkernel [178] remove commodity operating systems from the critical path, enabling servers to run richly featured network stacks at network line rates. These designs make heavy use of the multi-queue features of modern NICs such as RSS or Flow Director [120], which are simple means to map incoming packets to network queues (and thus application threads which are often paired in a one-to-one fashion) without requiring kernel demultiplexing. Additionally, the impacts of interrupt affinity, NUMA latencies and CPU power-saving modes can be addressed with existing software for core and memory placement policies (e.g., `numactl`).

Although the above solutions have been developed in the context of traditional Ethernet NICs, we argue that the techniques are equally applicable to emerging network stacks featuring hardware termination, because the only fundamental difference between the two systems is the hardware-software interface. We classify the interfaces of hardware- and software-terminated network stacks into two families. Traditional interfaces use Tx/Rx queues in hardware and employ ring buffers (RingBufs) to specify DMAs between hardware and software. In contrast, interfaces derived from the Virtual Interface Architecture (VIA) [66], such as InfiniBand and RDMA, use sets of in-memory queue pairs (QPs) as the interface between the NIC and software threads. The VIA was originally designed to support protected, zero-copy user-level I/O between multiple closely-interconnected nodes, which are principles increasingly adopted by the kernel-bypass dataplanes previously mentioned. The only difference between the two families is that the VIA often imposes the restriction that each QP is connected in a one-to-one fashion to another QP [66] whereas RingBuf-style designs simply specify work-lists of DMAs posted by the user or kernel driver to be performed by the NIC hardware.[2] Regardless of

[2]This one-to-one connection restriction comes from the InfiniBand specification for *RC* and *UC* transports, but

interface, the only fundamental requirement for µs-scale remote communication is for each core to have synchronization-free, zero-copy access to the memory which contains incoming and outgoing packets. As this requirement is achievable regardless of whether multi-queue Ethernet [33] or InfiniBand/RoCE NICs [111] are used, we consider either style as a valid interface for a server architecture designed for µs-scale RPCs.

Summary. Prior research has already proposed a re-think of legacy network stacks to eliminate traditional operating system overheads. With many frameworks available that deliver RPC traffic directly to user-level threads, the remaining factors resulting in excess latency can fall into two categories: queueing created at the thread handling the RPC, or interactions between the NIC and the server's memory system itself. We next discuss the state-of-the-art in server-side load balancing and queue management.

2.4 Curtailing Queueing with Inter-Core Load Balancing

The rise of fine-grained microservice architectures (§2.1) implies further attention must be paid to the Tail-at-Scale problem [59], because more richly connected microservice graphs increase the probability of a long-tail event. In a microservice graph, the most latency-critical services are the leaves (which are often data stores), because every long-latency event propagates to all the tiers above them. Although improvements in datacenter transports (§2.2) and systems software (§2.3) have indeed eliminated many sources of tail latency, achieving tight guarantees on tail latency requires limiting server-side queueing time before an RPC is processed. In emerging contexts where the applications receive requests directly from a hardware-terminated protocol, the task of limiting queueing time resolves to a simple decision: to which application thread should an incoming RPC be assigned?

From a theoretical perspective, queueing theory rigorously demonstrates that for RPCs whose processing time distribution has low variance, achieving optimal tail latency requires implementing a single-queue, first-come-first-served policy, where all serving units (i.e., CPU cores) actively *pull* requests from that queue one at a time [267]. High-dispersion distributions require a degree of preemption to be added [61, 126, 267]. Although the theoretical answers

does not necessarily imply that each QP is assigned to a single thread only. We do not consider QP sharing as a use-case for our systems because of its well-documented effects on latency [65].

to tail-optimized scheduling are fairly well known, implementing such policies with existing hardware is a challenging task with a plethora of tradeoffs, particularly for today's server CPUs featuring many cores in a single socket. Representative products at the time of writing have roughly 64 cores, namely Intel's Sapphire Rapids [95, 170], AMD's Milan [94], Amazon's Graviton [266], and Huawei's Kunpeng [270]. NVIDIA's recently launched Grace CPU has the highest currently available single-socket core count at 144 [209]. As core counts grow, any load imbalance exposed by a suboptimal queueing model or its implementation will result in greater relative under-utilization.

The key tradeoff that is faced by all RPC load balancing frameworks is how to achieve single-queueing while simultaneously meeting the message arrival rates possible in today's NICs. Realizing single-queueing in software requires mutual exclusion to be enforced on a single centralized queue of RPCs *or* the use of a single software dispatch thread. Both choices drastically limit throughput and cannot scale to the bandwidths of emerging server NICs – when a centralized queue is used, contention on the queue grows proportionally to incoming load, and therefore the system is bound by synchronization overhead and not the CPU cores' maximum service rate. With a single dispatch thread, as used by RAMCloud [214] and designs that offload request assignment to SmartNICs [106, 172], the dispatcher becomes a bottleneck even at 10-20Gbps network rates.

In contrast, the multi-queue capabilities of modern hardware NICs allows eliminating any synchronization among CPU cores for incoming RPCs, allowing a server to drive its NIC at its line rate at the cost of giving up single-queue load balancing. Examples of such capabilities include Receive-Side Scaling (RSS) [189] and Flow Director [120], used by the optimized systems software previously discussed (§2.3) to apply static rules mapping RPCs to many incoming queues (e.g., using the TCP 5-tuple or UDP port number). As such rules are static and do not account for runtime information, they are fundamentally vulnerable to load imbalance by design – consider the case where a short RPC is blocked behind a long one because they both happen to map to the same queue. However, state-of-the-art proposals propose to break this tradeoff using hardware support to realize inter-core single-queue load balancing at the network's full throughput [54, 150]. Such proposals rely on the presence of a single hardware unit on the critical path of all incoming and outgoing requests, in order to

keep up-to-date load information pertaining to each CPU participating in the RPC context, and therefore make *dynamic* load-balancing decisions.

A growing body of evidence shows that single-queue balancing with added preemption is not optimal for every type of microservice, and bespoke per-application policies have the potential to deliver better performance. For example, Demoulin et al. show that it is actually beneficial for tail latency and throughput to allow some of a server's cores to remain idle and ready to accept new RPCs, rather than greedily starting RPCs and having to later preempt them [61]. Additionally, it is highly unlikely that single-queueing will be optimal in the case where incoming RPCs have data dependencies between them. In this thesis, our focus is on how load balancing policies should be enhanced to take into account *application-level locality*. Current load balancers act at the interface between the transport and application layer, which is agnostic to two critical server-side factors: inter-thread synchronization, and cache locality. However, these factors have the potential to drastically impact the performance of µs-scale RPCs both positively and negatively. Assigning an RPC to a core which has just accessed a piece of data, or executed a shared code library, will result in higher L1 cache hit rates and increased application-level throughput compared to a locality-agnostic policy. The inverse is true when considering synchronization – an RPC that blocks because it cannot acquire a lock temporarily wastes a CPU core that could handle other requests.

Summary. The increased importance of tail latency in microservice-oriented software architectures, combined with the steady growth of server core counts, means that inter-core load balancing will become an absolutely critical feature in next-generation server architectures. Although today's best load-balancing frameworks propose hardware-driven synchronization-free implementations of single-queue policies, they lack any knowledge of application-level locality – which we expect to have significant impacts for microservice software exhibiting µs-scale RPCs.

2.5 Network-Compute Co-Design

Due to the increased importance placed on network communication, both industry and academia have experienced a strong architectural trend shift towards co-designing network

and compute hardware. A well-established example of this trend is technology that enables I/O traffic to be served directly from server caches rather than passing through system memory, such as Intel's DCA/DDIO [105, 119] or ARM's Cache Stashing [21]. Both these techniques allow the host to achieve higher network bandwidths and lower per-packet processing latencies.

In addition to features in existing processors designed to support commodity NICs, significant prior work has already argued for the need to go further, and integrate the network interface directly with its codependent compute logic. By doing so, system architects generally target one of the following two goals: either reducing communication latency by eliminating the round-trip time of traditional I/O interconnects such as PCI-Express [199], or offloading functionality traditionally performed on the host CPU in order to free up cycles for useful application-level work [70]. Academic architectures advocating for integrated NICs include Scale-Out NUMA [53, 199], L-NIC [109], NanoPU [108], and the open-source FAME-1 Rocket Chip SoC [137, § 3.2] from UC Berkeley's RISC-V ecosystem. Industry examples of network interface integration include Mellanox/NVIDIA's BlueField SmartNIC [204], NVIDIA's NVLink technology for inter-GPU RDMA [207], and Intel's Xeon-D chips which already integrate the Ethernet L2 MAC logic on-die [121].

Network-compute integration is a key enabler for RPC-centric server architecture, because the layers surrounding a server's network interface (i.e., the network protocol beneath the NIC and the systems software above it) are capable of delivering RPCs to the server at latencies similar to the NIC's raw line rate itself. As §2.2 and §2.3 have already showed, these conditions are already fulfilled, and therefore we expect future servers to operate in a similar environment where NICs are designed to handle RPCs natively. Prior work in network-compute co-design illustrates this fact by showing that it is logical to now perform operations such as load balancing [54], de-serialization [275], and concurrency checks [55] at the NIC, because the transport and network layers are now fast enough to expose these operations as a bottleneck.

Tighter network integration also means that application-level metrics can be exposed to the server's NIC, allowing wiser load balancing decisions to be made by taking into account application locality. For example, emerging accelerators designed for common RPC layer operations expose the fields of an RPC's header directly to hardware [122, 136, 275, 276],

which can then be used as proxies for reasoning about application-level locality or potential synchronization. Although request headers have already been used to statically differentiate between request types (e.g., `GETs` and `SETs` in NetCache [123] and R2P2 [150], as well as point-queries vs long-running requests in Persephone [61]), we expect that fine-grained metrics such as synchronization and cache locality will require tightly integrated NICs, because disseminating information about the server's load to the NIC at µs-scale latencies is only possible with integration. We demonstrate the load balancing benefits of NIC integration in §5.5.1 by comparing the performance of an integrated load balancing unit versus one residing on a commodity PCIe-attached SmartNIC. Our results show that commodity devices can only expect to attain 50-80% of the performance of an integrated design due to reduced proximity to the CPU cores, further motivating the benefits of network-compute integration.

Finally, we expect the trend towards hardware component specialization and application-level disaggregation to create further challenges for server architectures not designed expressly for RPCs. The end of Dennard scaling and the rise of accelerator-centric computer architecture [100] will inevitably result in the widespread deployment of specialized chips for common workloads such as the Google Tensor Processing Unit (TPU) for matrix operations in machine learning [125]. Deploying these accelerators in a datacenter cost-effectively mandates that they be network-attached and utilizable by many clients; the alternative means that accelerators are rigidly coupled to the host CPU, meaning that they cannot be provisioned independently and thus will inevitably suffer from periods of low utilization. Microsoft cites this exact reason when describing the motivation for their Catapult 2.0 accelerator architecture [44], in addition to benefits in cabling and fault tolerance. Operating system models have already been designed to manage datacenters architected around *disaggregated, network-attached hardware components* [239], of which the TPU would be one of many. Both these trends would mean that a single network-attached accelerator can expect traffic arriving to it from many server endpoints – creating a similar trend to that exhibited by a single server which sends and receives RPCs to and from microservices on many other servers.

Conclusions. Servers must evolve if they are to support the demands of today's microservice software architectures, and tomorrow's accelerator-centric datacenters at the bandwidths of upcoming NICs. As datacenter topologies and transports, systems software, and load-

balancing frameworks have already drastically advanced and pushed the remaining bottlenecks to the servers themselves, in this thesis we target three critical remaining challenges: (1) a server architecture whose hardware-terminated protocol can receive RPC traffic from hundreds to thousands of endpoints, (2) a NIC-driven load balancing design which takes into account application-level locality, and (3) a NIC architecture and implementation that dovetails into existing accelerators for production RPC layers.

Designing an RPC-Centric Server Part I

3 System Overview

In this chapter, we present a high-level overview of our proposed RPC-centric server architecture, describing the key modifications to existing hardware and software and highlighting the outstanding trends that motivate our additions. We also describe the modelling and simulation techniques we developed to study RPC-centric server architecture, allowing rapid experimental result generation as well as detailed architectural simulation. Throughout the chapter we use Figure 3.1 as a guideline, which shows the three server subsystems in the scope of this thesis: the NIC, the memory hierarchy comprising L1 caches, a shared LLC, and DRAM, and the CPU cores that run RPC-dependent software. All unmodified subsystems are simple outlines and remain unshaded, and our proposed changes are colour-coded to show the components involved in accomplishing each addition.

3.1 Novel Design Features

In this work, we propose three key additions shown in Figure 3.1 to form an RPC-centric server architecture. Firstly, our proposed RPC-centric server architecture contains hardware components (red) and protocol extensions (green) that exist on the data plane of network traffic and handle data placement for incoming and outgoing RPCs. Secondly, we architect our design to permit tight integration of state-of-the-art RPC protocol accelerators (grey), which are key enablers to allow software to be deployed using fully-featured production RPC stacks (e.g., gRPC) [92] and operate at the full underlying rate of the NIC. Finally, we propose hardware

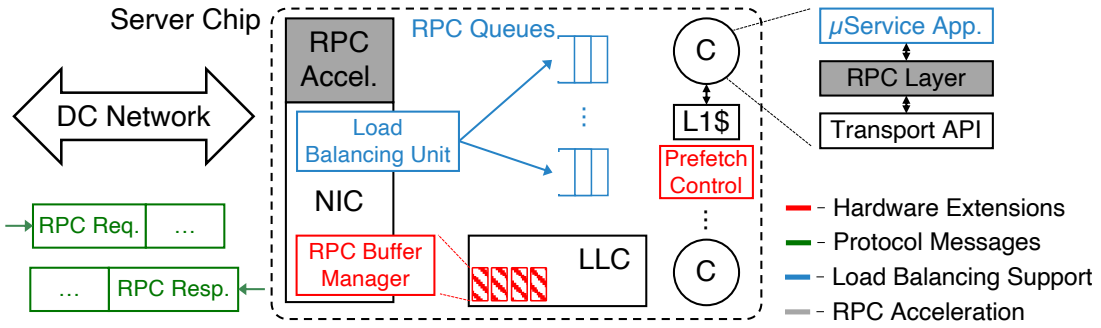


Figure 3.1: System overview of an RPC-centric server architecture, with colour coding to indicate logically grouped components that implement specific functionalities.

to support single-queue load balancing *with* application-level locality considerations (blue), and the matching software modifications (also blue) to inform the load balancing unit of RPC completion events. These components are decoupled from RPC payload data placement, and act as a "control plane" that assigns incoming RPCs to threads by means of in-memory queues. We now provide a conceptual overview of the functionality of each feature proposed by this thesis.

3.1.1 In-Cache RPC Placement

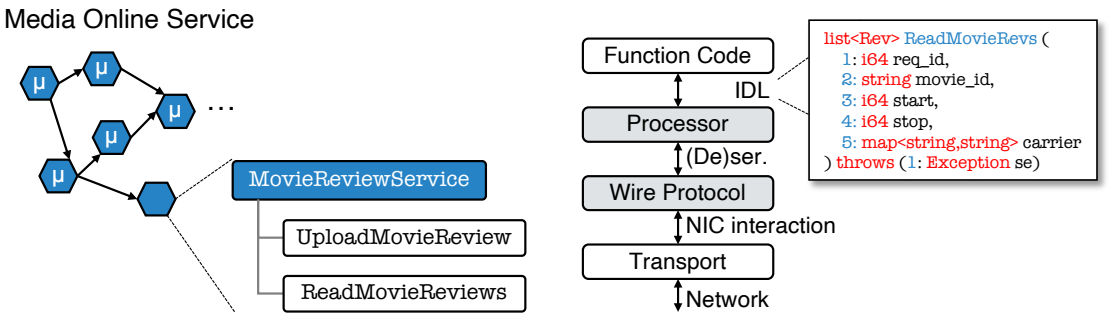
The combination of µs-scale software with high bandwidth NICs and optimized systems software (see Chapter 2) has resulted in at-scale deployments using hardware-offloaded protocol stacks for latency-critical datacenter services. Examples such as Microsoft's Catapult [44] and Google's 1RMA [244] are industry examples of how hardware-terminated stacks have already made their way into production datacenters. However, protocol offload is not a panacea towards attaining lower latency, as evidenced by the numerous prior works demonstrating the amount of careful engineering required to obtain acceptable performance on top of existing NICs (e.g., with RDMA) [47, 65, 129–132, 201, 257]. In particular, several such works identify the server's I/O interconnect (e.g., PCI-Express) as a key bottleneck that imposes constraints on how existing RDMA verbs must be used, and threads can be connected together [65, 129, 130, 257].

As network-compute integration sweeps over server architectures (see §2.5), ensuring that at-scale deployments of hardware-terminated protocol stacks can continue to keep pace with

the network and the plethora of serving units (i.e., cores or accelerators) will require ensuring that RPC traffic is judiciously handled in the server's memory hierarchy itself. Current trends towards larger, deeper, and more heterogeneous memory hierarchies that are connected by the server's on-chip network imply that previously neglected sources of RPC queueing may be exposed as previous bottlenecks such as protocol processing dissipate. In particular, we identify queueing at the server's memory controllers as an emerging performance quandary, that can result in a significant fraction of server-side RPC latency occurring before the RPC is even processed.

Queueing at the memory controllers occurs because hardware-terminated stacks use *per-endpoint buffer provisioning*, where every connected pair of servers allocates a dedicated set of buffers in which the NIC can place traffic. With high numbers of connected endpoints being the common case for today's online services (see §2.1), per-endpoint provisioning causes RPC buffers to overflow the server's LLC into main memory despite the existence of DDIO/DCA technology. Allowing network buffers large enough that they spill into DRAM is not only an unnecessary source of latency and excess DRAM bandwidth, but is problematic when the application competes for bandwidth with the NIC. Prior work has shown that bandwidth contention often leads to unacceptable performance degradation for server applications [173].

Our work solves the problem of memory bandwidth contention by proposing hardware and protocol extensions that eliminate RPC buffers spilling out of the server's LLC, ensuring that compute logic can access them from fast on-chip SRAM. The key insight that drives our work is that when a software architect places a tight SLO on a microservice, they are essentially stating that incoming RPCs must be handled from short queues, and therefore per-endpoint buffer provisioning is fundamentally excessive and should be relaxed. Relaxing per-endpoint buffer provisioning requires that the server NIC must dynamically assign RPCs a receive buffer at arrival time, and reject those which cannot be received. We perform such operations with a hardware module shown in Figure 3.1 as the "RPC Buffer Manager" (colour-coded red), whose protocol messages are shown in green. Furthermore, we advance the state-of-the-art in RPC data placement from simply loading data into the LLC (as is done by Intel DDIO [105, 119, 152] and ARM Cache Stashing [21]), to placing RPC buffers directly in the CPU's L1 caches just before the thread running on a core accesses them. We perform L1-cache data placement via



(a) A microservice and its functions.

(b) Intra-RPC stack layering, with generated code in grey.

Figure 3.2: The architecture of a microservice and a production RPC stack, and an example of a function's specification in an Interface Description Language.

a simple prefetch controller component at the back side of each L1 cache which generates prefetches for incoming RPC payloads, virtually ensuring the cores can access them in just a few cycles. Figure 3.1 also shows the "Prefetch Control" component attached to each core in red. Chapter 4 covers the design of these extensions in detail, and Chapter 6 describes their implementation and evaluation.

3.1.2 Fully-Featured RPC Layer with Hardware Acceleration

The rapid proliferation of microservices as the software architecture of choice for online service developers naturally implies that RPC-centric servers should be expressly designed to match, because custom RPC libraries are commonly used as a glue to connect microservices. Examples of company-wide libraries include Google's gRPC [92] and Facebook's own fork of Apache Thrift [19, 72]. Supporting such libraries with RPC-centric architecture provides a single common point for optimization and the addition of functionality, so that all the benefits provided by the improved architecture apply to every programming team. In this thesis, we therefore target support for fully-featured, production-ready RPC layers.

Figure 3.2 presents a visual representation of the architecture of a microservice and the underlying RPC API on which it depends.[1] The depicted microservice is called `MovieReviewService`,

[1] Our example uses terminology from the publicly available version of Thrift [19], but the same operations are present in gRPC [92].

and is taken from the "Media" application in the DeathStarBench suite of microservices [230].[2] Each microservice represents a single piece of the business logic of the full online service, which in this case represents a media streaming service like Netflix. `MovieReviewService` handles all movie reviews submissions and lookups. Going down one layer, this microservice consists of multiple underlying *functions*, which are specified by the programmer similarly to class method declarations in programming languages like `C++`. Figure 3.2a shows that `MovieReviewService` consists of two functions: `UploadMovieReview` and `ReadMovieRevs`.

Production RPC stacks are comprised of a multi-layered architecture, where the upper layers are auto-generated with programmer assistance, and the lower layers interact with the surrounding system (e.g., the network). When writing `MovieReviewService`, the programmer will use an *Interface Description Language* (IDL) to declare all the microservice's functions, and the desired interface for each function. The programmer needs to specify the name of the function, its arguments and their corresponding types, and the final return value. Figure 3.2b shows the Thrift IDL for the `ReadMovieRevs` function. Then, at compile time, the RPC framework generates all the code to expose a service with an API matching the IDL. The programmer can implement the function in any language as long as it adheres to the generated API.

The `Processor` layer is analogous to what is seminal RPC literature calls an RPC stub [35]. By generating a Processor instance which exports an interface compliant with the programmer-specified IDL, the programmer does not have to concern themselves with any specifics of threading, data framing, or network interactions. It also allows multiple `Processor` types to be implemented depending on the needs of the application itself: for example, choosing between a `Processor` with a thread pool or a single-threaded version. In between the `Processor` layer and the `Wire Protocol` layer, the RPC stack must take the RPC's arguments defined in the IDL and perform serialization (in case of sending a request), or de-serialization (in case of a new request arriving). We refer to this process hereafter as (de)serialization for brevity. After (de)serialization takes process, and the wire format of the RPC is generated, the next layer to be traversed is the `Transport` itself, which can interface with any protocol it wishes, from traditional TCP/IP or specialized stacks such as the custom hardware-terminated transport proposed in this thesis.

[2] This diagram matches commit hash `55f3696d2ee6f22b30bb3518d9c835eee65e511e`.

Although the many-layered architecture of production RPC stacks provides generality and convenience to users, that functionality comes at a high cost. At the breakneck speeds attainable by today's hardware-terminated network stacks, particularly with support for data placement in the cores' L1 caches, the µs-scale operations that make up production RPC layers are a crippling overhead for application throughput. Prior work has already observed that the functionality included in production-ready RPC layers costs between $20-80\%$ of total on-server time for an open-source suite of microservices [159, 275]. Fundamentally, this entire cost is paid because of the modularity demanded by the microservices architecture (see §2.1) and does not contribute to application goodput. Although a naïve solution may be to simply dedicate cores to RPC layer processing, this is an unacceptable solution for datacenter providers because silicon density scaling has reached a near-standstill [57, 71], and CPU cores are a key resource that must be monetized. Fortunately, prior works exist to accelerate production RPC layers by providing support for the most common operations in hardware [122, 268, 275, 276]. However, the integration of such designs with state-of-the-art network interfaces has not yet been sufficiently considered.

As prior work has already shown that integrated network interfaces need to be *distributed* to scale to the needs of many-core servers [53], we remark that any proposed RPC protocol accelerator also needs to be designed to match. There are two particular challenges involved in such a co-design: firstly, the RPC's payload processing components need to be distributed across the many nodes of the server's on chip network. Secondly, the order that operations are performed in the RPC stack needs to be changed in order to support NIC-driven load balancing. In this thesis, we demonstrate an implementation that allows a previously proposed accelerator for RPC layer operations [276] to dovetail into our hardware-terminated protocol stack, forming a full RPC protocol accelerator. In Figure 3.1, we graphically display the inclusion of an RPC protocol accelerator (RPA) by the additional NIC component shaded in grey. Chapter 7 discusses the implementation and microarchitecture of a NIC supporting RPC protocol acceleration.

3.1.3 Enhanced Load Balancing

When handling µs-scale requests, cache locality matters, because the bandwidths of current NICs imply that small-sized RPCs can arrive faster than all but a server's smallest SRAM caches [254]. With already-available 400Gbps NICs [208], a 64-byte RPC can arrive every $\sim 1.5ns$, meaning that effort must be made to minimize memory accesses to enable a server's compute units to maximize their rate of RPC handling. In this thesis, we argue that an RPC-centric server architecture needs support to assign incoming requests weighing both load balancing and locality metrics. As previously discussed in §2.4, numerous prior works have studied transport-level load balancing, where incoming RPCs are assigned to threads based on policies opaque to cache locality. However, the fact that emerging applications are often characterized by runtimes ranging from less than one µs to a few µs, means that significant throughput improvements would be available if the unit(s) assigning RPCs to cores can enhance application-level locality. This claim about the importance of cache locality is also corroborated by prior work that investigates *nanosecond-level* savings for software packet processing frameworks [75, 76] and NIC-to-core data transfer [108].

We specifically target two types of application locality for RPC-centric server architectures. First, our work improves instruction-cache locality, which presents itself as various RPCs invoke different functionality on the server and leave their code working sets in the CPU's L1 instruction caches. Second, we improve the performance of concurrency control mechanisms in microservices that cache application data, by assigning requests to threads that already possess exclusive access rights for certain regions of memory. Integrated RPC accelerators are a strong enabler for locality-aware load balancing, because they provide the load balancing module knowledge of each RPC's headers and arguments (e.g., the `movie_id` in Figure 3.2b) which can be used to make code- or data-locality aware decisions.

Figure 3.1 shows the enhanced load balancing unit we propose for RPC-centric server architectures, which is co-designed along with the hardware units handling incoming RPC data (see §3.1.1). The additional hardware we propose in this thesis is roughly comparable to the size and complexity of a TLB, and has drastic positive impacts on application locality. Despite the simplicity of our design, we demonstrate up to an 18× reduction in cycles stalled on instruc-

tion cache misses, and up to 1.7× higher system throughput due to improved concurrency control. Chapter 5 covers our design for locality-aware load balancing, and our evaluation in Chapter 8 demonstrates its performance benefits.

3.2 Applicability of RPC-Centric Design Extensions

As the work in this thesis is cross-layer and involves many server system components, this section lays out the scope and server hardware prerequisites for each feature just introduced. In particular, we describe the deployment horizon for each feature, demarcated by the type of NIC that is available as a baseline in the host server. Shifting more functionality to the NIC from the host node will increase performance when designed well, but is also likely to increase capital cost to deploy more advanced network devices. Therefore, this section breaks down the contributions of this thesis by their ability to be deployed on the following three types of NIC architectures.

First, we consider traditional off-chip NICs that are attached over I/O interconnects like Peripheral Component Interconnect Express (PCIe). Except for limited functionality such as TCP Generic Segmentation Offload or support for packet checksums, such devices can be primarily viewed as high-bandwidth DMA engines, that move data to and from host memory and inform software of DMA completions. Due to the limited *application-level* information that can be taken into account on existing devices, the opportunities for RPC-centric enhancements are equally limited. The only contribution in this thesis that can apply to such devices is a concurrency control enhancement that only requires host-side software modifications (proposed in §5.4.2 and evaluated in §8.4.3).

With increased opportunity for network-compute co-design in the datacenter, multiple vendors have launched "SmartNIC" products that support more advanced functionality, the most well known being Mellanox/NVIDIA's BlueField [202, 204]. Any hardware support for RPC protocol operations integrated into traditional NICs would also fall into this category. These enhanced NICs often contain application-programmable logic (e.g., small FPGAs or full CPU cores), and are able to run entire application-layer protocols like NVMe and cooperate with virtualization software. The additional capabilities of such devices opens up many more

possibilities for the techniques we propose.

In particular, a subset of our proposed hardware and protocol extensions for in-cache RPC data placement (see Chapter 4) could be implemented on the programmable devices in SmartNICs, granting the host node the ability to serve requests from hardware-terminated protocols at large deployment scales. Additionally, our proposed policies for enhanced NIC-driven load balancing (see Chapter 5) could also be realized in SmartNIC programmable logic. The caveat of a SmartNIC-based design for load balancing is that the reduced proximity between the NIC and CPU cores will result in reduced performance for tail-latency critical software, compared to a cutting-edge design with full network-compute integration. §5.5.1 quantifies the degree of performance reduction when implementing load balancing on any off-chip device.

Finally, the most advanced (and highest cost) architectures are those that feature full network-compute integration. In these devices, which are beginning to become more commonplace in both academia and industry, the entire scope of work in this thesis would be possible, due to the ability to adapt the host's architecture for the specific network operations we propose. In addition to all the features above, the full scope of our proposal for in-cache data placement becomes possible. The NIC would then be able to place data into the CPU cores' private L1 caches (see §4.2.2). Importantly, due to the host node and NIC sharing the same memory hierarchy without an I/O interconnect dividing them, full RPC acceleration becomes feasible (see Chapter 7).

With full integration, locality-aware load balancing will now be able to reach its full potential, because the NIC and host can exchange fine-grained statistics and metadata about the application's real time state, which can be used to make improved load balancing decisions. In particular, tighter integration will specifically benefit our proposed techniques for instruction-cache aware load balancing (see §5.2) and NIC-assisted concurrency control (see §5.3.2), because they both target RPCs whose latencies are on the order of $1\mu s$ or less.

Summary. Table 3.1 displays a summary of which contributions of this thesis are possible and effective on various NIC architectures. Commodity NICs are available at the lowest price, but their relative lack of features at layers above the network transport means that they can only implement a single software-only change we propose. Although SmartNICs

	Contribution	Commodity NIC	SmartNIC	Integrated
Chapters 4 and 6	In-Cache RPC Placement	×	✓	✓
	+ L1 Prefetching	×	×	✓
Chapter 7	RPC Acceleration	×	✓**	✓
Chapters 5 and 8	NIC-Driven Load Bal.	×	✓**	✓
	+ Locality-Aware Policies	SW-only (§5.4.2)	✓**	✓

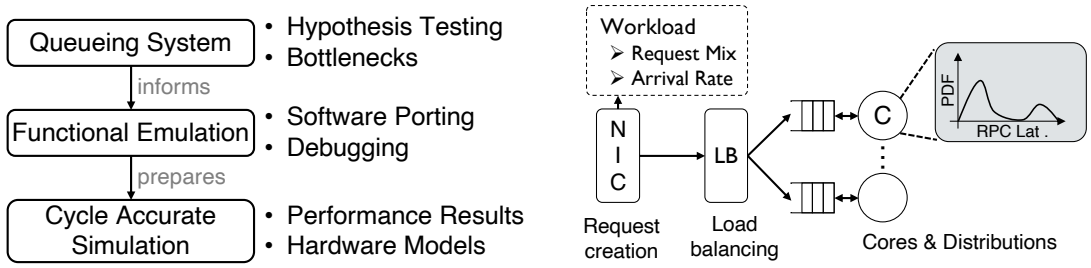
Table 3.1: Scope of applicability of contributions towards RPC-centric server architecture. Asterisks indicate that implementing a feature is possible, with reduced benefit.

bring multiple contributions into scope, their off-chip location reduces the performance of two key contributions, namely RPC protocol acceleration and NIC-driven locality-aware load balancing. Fully integrated designs are naturally the most feature rich and highest performance, which is logical as they require the highest expenditure and implementation effort.

3.3 System Modelling Techniques

Simulating a full RPC-optimized server architecture in complete cycle-accurate detail is not just a challenging implementation task, but also prohibitively slow in terms of results turn-around time. To quantify, full-system cycle-accurate simulation is known to proceed roughly six orders of magnitude slower than real hardware [265]. Although statistical sampling is a time-tested method for measuring the impacts of microarchitectural techniques on server workload throughput [269], it is unclear how sampling techniques can be applied when tail latency is the metric of interest, because accurately computing tail latency percentiles requires many full requests to be run to completion [149].

Therefore, we have developed a family of three techniques to model and study RPC-optimized server architectures, that provide different levels of detail depending on what metric is being studied. In Figure 3.3a, we show the techniques used in this thesis, and their respective use-cases and applicability. The highest level is a simple event-driven queueing simulator which allows reasoning about system bottlenecks and estimating the improvements of particular optimization techniques, without requiring any system implementation or application porting. We find that queueing systems are sufficient to model not just the impacts of load balancing policies (as seen in ZygOS [223], RPCValet [54], and Shinjuku [126]), but also bottlenecks in



(a) Techniques developed in this thesis.

(b) An abstract queueing system, further instantiated in each subsequent chapter.

Figure 3.3: Modelling techniques and their respective use-cases for studying RPC-centric server architectures.

lower-level cache and DRAM behaviour (seen in Chapter 6).

Our model is built in Python 3, and uses the SimPy discrete event simulation framework [242] as its underlying implementation for the events' priority queue and all interactions between system entities. We instantiate various system components like CPU cores, load balancers, or DRAM channels as simple event generators and consumers. To illustrate, in Figure 3.3b we show an abstract representation of a server CPU comprised of a NIC, a load balancer, and cores connected by a queueing network. Individual RPCs are passed between these components, and latencies are assigned based on statistical distributions, which we choose to model the desired system behaviour. For example, a CPU core's latency distribution can be shifted to model increased locality. We have extensively used this methodology in our work, and have open-sourced it for future research.[3]

The second modelling and simulation technique developed in this thesis and shown in Figure 3.3a is *functional protocol emulation*. After a reasonable system performance expectation is obtained by hypothesis testing with our queueing simulator, confirming the results with a real system simulation is the next step. However, before any results can be obtained, it is necessary to implement any hardware and protocol changes in the simulator, and simultaneously port the application software to run on top of the aforementioned protocol. In the early stages of our work, we found this task to be extremely error-prone, because it is difficult to isolate the source of correctness or performance problems when all layers from the application to the

[3]The repository can be found at https://github.com/parsa-epfl/queue_flex.

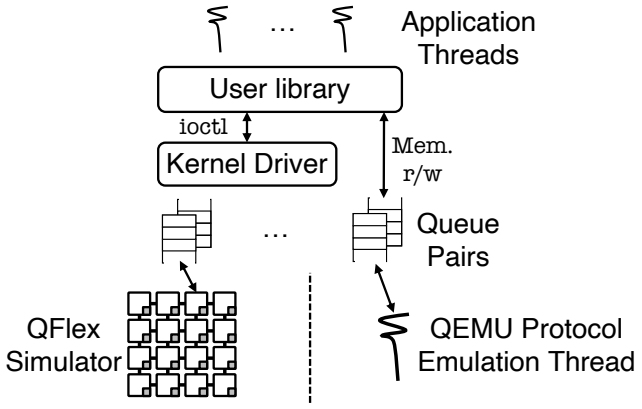


Figure 3.4: Unified emulation and simulation stack, using QEMU and QFlex.

hardware have been modified. Therefore, we sought a method that would allow us to develop protocol extensions and application software more rapidly, while maintaining compatibility with the cycle-accurate simulator and avoiding the need to support multiple platforms.

Our functional protocol emulator is inspired by the Scale-Out NUMA emulation platform [199, §7.1], but uses QEMU [51] instead of the Xen hypervisor. The critical benefit of our system over the emulation platform developed by Novaković et al. is that our cycle-accurate simulator uses the same QEMU emulator with an enhanced set of timing models, meaning that all software ported to the emulator functions equivalently on the simulator. Additionally, all guest checkpoints are usable by both the functional emulator and cycle-accurate simulator, and we only need to support a single version of the integrated NIC's kernel driver and RPC library.

Figure 3.4 shows the architecture of our unified emulation and simulation platform, where the host system is a commodity server and the guest is a virtual machine (VM) running in QEMU's full virtualization mode (i.e., KVM is not currently supported). We modify QEMU's `virt` machine type to expose our proposed NIC to the guest operating system as a special type of PCIe device, and wrote a loadable kernel driver which exposes that NIC to user-mode applications through `sysfs`. To send and receive RPC traffic to and from our NIC, an application first opens `/dev/sonuma_rmc`, then allocates memory for incoming and outgoing RPC buffers and its in-memory Queue Pairs, and finally informs the kernel driver of their respective locations in its address space using regular `ioctl` system calls. Once the application invokes `ioctl` with a specific "magic" value, the emulator then spawns a new thread which

implements the NIC's protocol by reading and writing the guest VM's memory accordingly. During this thesis we heavily leaned upon protocol emulation for application porting and solving correctness issues, because we are unaware of any debugging tools that are able to debug the guest VM's state when running in cycle-accurate level of detail. In contrast, protocol emulation runs quickly enough to allow the use of indispensable tools such as `ASAN` and QEMU's `gdbstub` to inspect guest memory, single-step the guest VM, and inject faults by writing to guest memory locations.

Finally, we relied on cycle-accurate full-system simulation for generating all the experimental evaluation results for our RPC-optimized server architecture. We use the QFlex simulator [216] which extends the QEMU emulator with a "timing-first" mode of execution whose base implementation is derived from the SimFlex simulator [265]. After the cycle-accurate simulator models the detailed execution of an instruction in its out-of-order (OoO) CPU cores, it instructs the emulator to advance by executing a single instruction, and then returns control to the timing model. QFlex also contains a detailed set of memory consistency, on-chip network, and cache coherence models, which allow us to study in detail the hardware extensions required to support our proposed architecture. All the baseline libraries and NIC timing models were derived from the Scale-Out NUMA project, which we extended and ported to QFlex. For any other details on Scale-Out NUMA, we refer the interested reader to [52].

Summary. Our proposed RPC-optimized server architecture contains three novel design features: 1) Hardware and protocol extensions for handling all RPC traffic from the server's existing memory hierarchy, 2) tight integration with previously proposed accelerators for RPC-layer operations, and 3) enhanced load balancing that improves application-level locality. In order to model, develop, and evaluate our proposed architecture, we used a variety of techniques at different layers of abstraction. Having summarized our proposed architecture's features, we now proceed to discuss the first addition – support for in-cache traffic management.

4 In-Cache RPC Traffic Management

The importance of limiting queueing delays to deliver minimal latency is an insight that spans many layers of the datacenter stack, and has resulted in proposals attempting to eradicate queueing in subsystems from network switches [96] to operating systems [150, 223]. Due to the impacts of queueing on the performance of microservice meshes (§2.1), and the body of work that reduces queueing in the network (§2.2) and systems software (§2.3), this chapter focuses on server-side queueing that forms as packet buffers transit the server's memory hierarchy during µs-scale RPCs.

Bandwidth contention arises when incoming network packets generate memory accesses that interfere with the accesses *from the application itself*, forcing them to queue up at the memory controllers and directly impacting RPC tail latency. This chapter studies the queueing network of a server's memory hierarchy and identifies a yet-unaddressed source of delays for servers handling µs-scale RPCs: bandwidth contention that occurs at the server's memory controllers. For a server running µs-scale microservices, we find that such queueing reduces throughput under strict SLOs by up to 2×.

We therefore argue that RPC-centric server architectures merit specific extensions to ensure that RPC traffic is does not overflow the server's SRAM caches, eliminating bandwidth contention and improving latency. To achieve this goal, we introduce one simple insight: network endpoints should provision only enough buffering to cover the number of requests which could theoretically meet the application's SLO, rather than traditional approaches like the

number of endpoints or the bandwidth-delay-product between endpoints. Using insights from queueing theory, we show that the number of requests that can be queued and still meet a given SLO is extremely limited, and therefore buffer provisioning for latency-critical µs-scale RPCs can be shrunk down to an amount which can easily reside in the server's SRAM (10s of KBs in our experiments). Furthermore, because the requisite buffering is so limited, simple architectural extensions can also accelerate RPC startup time by placing packets into the matching L1 cache of the core about to handle the incoming RPC; this capability is made possible by leveraging an on-chip integrated NIC (§2.5).

The rest of this chapter presents the motivation for such architectural extensions, the theoretical foundations for our idea to shrink buffer sizes for µs-scale RPCs, and the design of NEBULA, our proposed set of architectural extensions for in-cache management of RPC traffic.[1]

4.1 Current Obstacles: Load Imbalance and Bandwidth Interference

The future of network fabrics is certain to be characterized by tremendous bandwidth, as both the InfiniBand and Ethernet standards have released roadmaps to achieve link rates greater than 1Tbps [112,252]. Such abundant bandwidth will directly affect server design, as incoming network traffic will become a non-trivial fraction of a server's available bandwidth. For proof of this trend, consider the evolution in memory architecture of state-of-the-art switch ASICs (e.g., Broadcom's Jericho 2) that already have the capability to support ~ 10Tbps of I/O (12 ports each offering full-duplex 400Gbps). Providing 10 Tbps of I/O bandwidth requires Broadcom to use in-package High-Bandwidth Memory (HBM) modules [40] that have up to 1TB/s of bandwidth.

In contrast, server-side memory bandwidth has historically been so much greater than network bandwidth that network traffic has not impacted memory architecture; however, the explosive growth of network bandwidth (see §2.2) indicates that server memory architecture must adapt. To quantify, the original MICA key-value store requires eight 10Gbps NICs to saturate two octa-core Xeon E5-2680's [166], and follow-on work from the same authors argued that a 60-core server socket can support a 300Gbps NIC using six DDR4 memory controllers providing

[1] NEBULA is a portmanteau for NEtwork BUffer LAcerator.

115GB/s of total memory bandwidth [165]. In their final proposed system, full-duplex 300Gbps operation consumes up to 65% of the total memory bandwidth. Such memory bandwidth consumption from the NIC indicates that the CPU cores can perform a maximum of four memory accesses per RPC (on average) to not exhaust the available bandwidth. Despite the possibility of provisioning more memory bandwidth headroom by dedicating more chip pins to DRAM channels or using emerging technologies such as HBM, these decisions would drive up cost unnecessarily compared to a solution which handles the NIC's traffic from the server's existing SRAM.

Although current server products are equipped with Data-Direct I/O (DDIO) technology that presumably ameliorates the challenge of memory bandwidth consumption by allowing I/O devices to directly pass traffic through a server's LLC [119], today's user-level network stacks complicate the task of in-cache buffer management due to their buffer provisioning requirements. Providing zero-copy and synchronization-free message reception fundamentally requires network receive buffers to be provisioned on a per-endpoint basis (i.e., per connection), so that the server can receive a message from any other connected endpoint at any time. This requirement is more challenging in the case where dedicated hardware terminates the network protocol (e.g., in RDMA), because a buffer must be already be allocated to the receiving connection for the incoming packet's DMA to succeed.[2] Note that although support has been proposed for handling incoming DMA page faults [161], buffers still must be pre-allocated before being faulted in. Such connection-oriented provisioning creates two fundamental scalability problems, because it wastes precious DRAM resources and raises performance problems in existing RPC stacks when QP entries cannot be cached on the NIC [65, 131, 132, 201].

The buffering requirements of hardware-terminated stacks directly impacts application performance, because the multi-client interleaving of incoming requests results in unpredictable access patterns when incoming packets are written to network buffers. Effectively, the accesses for incoming requests are pseudo-random, minimizing the probability that they hit the server's LLC and thus inflating DRAM bandwidth consumption. This previously negligible

[2] Although we refer to connection-oriented transports, the same is true for unconnected ones like UDP or RDMA's UD transport.

effect significantly hurts the latency of µs-scale RPCs, particularly when incoming network traffic is a significant fraction of DRAM bandwidth. We hereafter refer to this behaviour as *memory bandwidth interference*.

Aiming to avoid the memory capacity waste of connection-oriented provisioning, InfiniBand (IB) offers an option called Shared Receive Queue (SRQ), which enables multiple network endpoints to share receive buffers posted to a common software structure. [3] The use of SRQ implicitly ameliorates memory bandwidth interference because its reduced buffer footprint would have increased LLC residency. Another approach to preventing buffers from overflowing into DRAM is to statically limit the quantity of buffers available for the network stack to allocate. This choice is made by prior work such as ResQ [256] and eRPC [132], which impose static limitations on system scale based on the fraction of the LLC that the NIC can use for its DMAs. However, SRQ, ResQ, and eRPC are vulnerable to load imbalance between the CPU cores, as they correspond to multi-queue systems *by design*: clients must specify the connection on which each RPC is sent and no server-side load balancing is possible. The same multi-queue limitation is inherent to Receive-Side Scaling (RSS) [189] support, often used for inter-core load distribution [33, 217, 223].

As discussed in §2.4, load balancing between a CPU's many cores is critical for tail-latency constrained systems, and therefore systems like SRQ are not a panacea because they cannot provide load balancing. However, recent proposals have demonstrated system support to allow the server's many cores to serve requests in a single-queue fashion [126, 223]. However, such approaches inevitably incur the cost of synchronization when re-balancing RPCs between cores - which can be comparable to the RPC service time itself for µs-scale tasks. For the most challenging RPCs with single-digit µs runtimes that we consider in this thesis, the throughput loss corresponding to unavoidable software synchronization motivates including hardware support for load balancing [54, 106]. Unfortunately, the buffer provisioning requirements of hardware-terminated stacks bring back the problem of memory bandwidth interference, exposing a fundamental tradeoff: either accepting load imbalance, or excess DRAM traffic that can degrade application performance.

Conclusions: Current RPC systems are amenable to either memory bandwidth interference or

[3] IB uses the VIA interface as in §2.3, and therefore the endpoints are Queue Pairs.

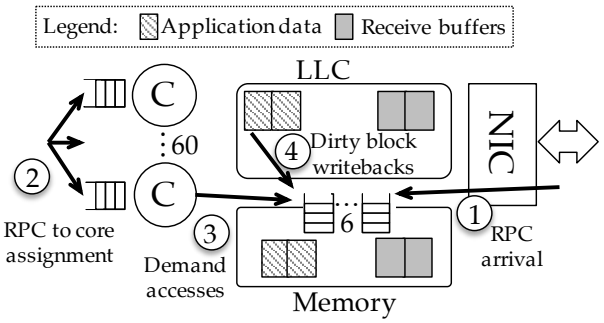


Figure 4.1: Queueing system model of the memory hierarchy interactions associated with handling incoming RPCs.

load imbalance, particularly when deployed at datacenter scale with thousands of communicating nodes. Eliminating memory bandwidth interference mandates limiting deployment scale or the number of outstanding RPCs per server, while eliminating load imbalance requires accepting the overheads of software synchronization. We now move to quantify the performance impacts of each factor, and then present the design of system extensions that simultaneously deliver both benefits.

4.1.1 Modelling Imbalance and Interference

To study the implications of the tradeoff between bandwidth interference and load imbalance, we use our discrete-event simulation methodology proposed in §3.3. Our work begins with well-known queueing systems used to study load balancing, and adds a simple model of a server's LLC and DRAM to allow us to take into account the impacts of memory bandwidth interference. We expect that such models are sufficient to capture the high-level bottlenecks, and as practical engineering overheads are systematically reduced, the accuracy of theoretical predictions should more closely track the behaviour of the real system.

In Figure 4.1, we show a concrete instantiation of our queueing simulator that models the following components of a server CPU: the NIC, LLC, system memory, multiple cores, and the RPC-to-core assignment policy. Without loss of generality, we assume a multi-core CPU and on-chip integrated NIC that places packets directly into the LLC, similar to existing DDIO NICs [119]. To model the steps involved as RPCs transit the server's memory hierarchy from the NIC to the cores, and responses being created, we take into account the following four

interactions: (1) traffic arriving from the NIC which is then placed into the LLC, (2) assigning RPCs to cores, (3) memory accesses created by the cores during the RPC itself, and (4) write-backs of dirty blocks from the LLC to memory. Step (2) pertains to load imbalance, while Steps (1),(3), and (4) pertain to memory bandwidth interference.

When a new RPC arrives in the system in Step (1), the NIC creates cache block requests matching the RPC's size that arrive at the LLC. Based on the relative size of the receive buffers compared to the size of the LLC, the RPC's buffers have a chance to hit in the LLC - if the access is deemed to be a miss, they are write-allocated from the DRAM, creating memory bandwidth. The RPC is then dispatched to one of the processing cores in Step (2), according to one of the load balancing or distribution policies we considered. During the RPC's service time in Step (3), it creates a number of memory requests which themselves reach the LLC, check for a hit, and generate memory bandwidth if they miss. Finally, at each of Steps (1) and (3), we model double the memory bandwidth to represent the newly allocated block's future eviction from the LLC (Step (4)) *if and only if* the access is both a write and determined to be a miss. We now explain how we model each of the above components.

NIC: The model's load is driven by the NIC component, which generates new RPC arrivals following a Poisson process. The rate parameter of the distribution, denoted λ_{arr} is the input variable to our model as it defines the average inter-arrival time between two requests. Each RPC is broken down into a number of cache-block writes N_w for the RPC's payload, which we assume can all proceed independently through the server's cache and memory hierarchy. The NIC component then creates an RPC dispatch message and places it into one of the CPU cores' input queues; our queueing model supports both single- and multi-queue models, following prior work that studies load imbalance between cores when handling RPCs. We do not penalize the single-queue model with any additional service time due to the cores accessing the single-queue, as multiple implementations have been published which achieve synchronization-free single-queueing [54, 150].

LLC: We model an LLC with a deterministic service time, μ_{LLC}. Each access to the LLC is modelled as an independent Bernoulli trial, with the "success" condition defined as an LLC miss that creates memory bandwidth. Depending on whether the access comes from the CPU

cores or NIC, we attach a corresponding miss probability. For accesses coming from the NIC, we calculate the miss probability as:

$$p_N = 1 - min\left(\frac{LLC\ capacity}{Net.buf.size}, 1\right) \tag{4.1}$$

Where both the LLC capacity and network buffer sizes are configurable characteristics of the modelled server. We assume that incoming requests are abundantly interleaved from all the clients communicating with this server, and thus there is negligible reuse - this leads to our estimation of p_N which scales inversely with the amount of provisioned network buffers. Also note that our model for p_N optimistically assumes the NIC can use the entire LLC, whereas Intel servers place a default (but configurable) limit at 10% of the LLC [119].

CPU cores: The CPU cores receive dispatch messages from the NIC, and model a generalized service time as follows. As the key variable we are interested in studying is the impact of memory bandwidth interference on the RPCs service times, we split the service time into two components. The first component takes place purely on-core, and the second depends on the memory access time that changes with incoming load and bandwidth interference. Therefore, the service time of each RPC can be modelled as

$$\bar{S} = t_c + (N_{acc} * AMAT)$$

where t_c is a fixed amount of on-core computation, N_{acc} is the number of serialized memory accesses per RPC, and $AMAT$ is the average memory access time. We set AMAT to 45ns at zero load, so it is only affected by input queueing due to bandwidth interference. Under this model, each type of RPC will have its own number for N_{acc}.

Memory channels: Under Kendall's notation, each memory controller in our model is modelled as an $G/D/k/FCFS$ system. This corresponds to a system with a generalized arrival distribution, deterministic service times, k workers/outstanding requests, and first-come, first-served memory scheduling. The arrival distribution is dependent on the system load, the LLC model, as well as the RPC service time model. We assume a deterministic service time of 45ns once a request is served by the memory controller, and first-come first-served memory scheduling which is common. Using a fixed service time for each memory access, we calculate

that each modelled memory controller must be able to serve $k = 15$ requests in parallel to match the desired bandwidth of a DDR4-2666 channel (~21GB/s). Each parallel request in principle represents an independent DRAM bank - as today's modules can reach 32 banks per DIMM [187], our model is conservative.

Experimental setup: With a model for each component as described above, we can answer the question: how does excess traffic at the memory controllers impact the latency percentiles of the RPCs? Due to our interest in latency-critical requests that are potentially impacted by memory bandwidth interference, we consider key-value requests as a basic class of RPCs that are both latency critical and generate significant memory bandwidth. Therefore, we parametrized our queueing simulator with values that reproduce the current best-performing server architecture, specifically architected for key-value serving: 60 CPU cores, 45MB of LLC, six DDR4 memory channels, and a 300Gbps NIC [165]. Today's high-end servers such as Intel Xeon Gold [170], Amazon Graviton2 [16], and Qualcomm Centriq [169] also have similar values for core count, LLC, and DRAM, and production NICs can already deliver bandwidths up to 400Gbps [208].

For the RPCs themselves, we model their execution following the MICA key-value store, as it is still the best-performing software system for in-memory key-value serving [166]. As all MICA accesses first perform a hash-index lookup (64B read) followed by reading/writing the payload from/to the data store, this gives us $N_{acc} = 2$. For this experiment, we use 512B values. We measured the average service time of MICA `SETs` to be $\bar{S} = 630ns$ on an Intel Xeon E5-2680v3, and additionally confirmed this value on our simulation platform. Therefore, we set $t_c = 540ns$ for the fixed on-core computation time.

4.1.2 System Configurations

Using the queueing simulator just described, we study the following four system configurations as a "spanning set" of the design space of highly optimized RPC systems:

1. *RSS.* A multi-queue system with uniform assignment of incoming RPCs to cores and 136MB of receive buffers (see §6.7 for sizing details). As discussed in §4.1, RSS suffers from load imbalance and memory bandwidth interference.

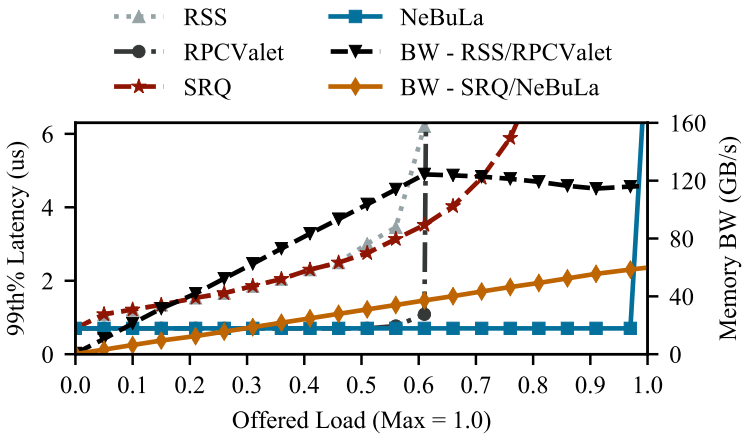


Figure 4.2: Discrete-event simulation results, showing the impact of load imbalance and memory bandwidth interference on the throughput and tail latency of µs-scale RPCs.

2. *RPCValet*. Single-queue load balancing of RPCs to cores with equivalent receive buffer provisioning to RSS. Although RPCValet solves load imbalance, it still suffers from bandwidth interference.
3. *SRQ*. Like RSS, a multi-queue system with uniform distribution of incoming RPCs to cores, but assumes *ideal* buffer management, where all network buffers are reused in the LLC, eliminating bandwidth interference.
4. NEBULA. Combines the best traits of *RPCValet* and *SRQ*: single-queue load balancing and ideal buffer management.

We emphasize that these configurations represent an ideal deployment of each respective system, and we do not penalize any of them for implementation-specific details. This is most applicable to *SRQ* - we do not model any latency penalty for synchronization between threads when posting buffers to the *SRQ* data structure, although a real system must incur this overhead. Our model therefore slightly over-estimates *SRQ*'s performance compared to a real deployment.

4.1.3 Results Discussion

Figure 4.2 shows the 99th% latency of the MICA RPCs and the total memory bandwidth utilization of each modelled system. We assume an SLO of 6.5µs ($\simeq 10 \times \bar{S}$), following prior

work that studies RPC load balancing in the relevant context of µs-scale computing [150, 223]. The systems with connection-oriented buffer bloat (RSS and RPCValet) saturate early at a load of 0.61, because they exhaust the server's memory bandwidth by generating $127GB/s$ of traffic. Although RPCValet attains sub-µs 99th% latency until saturation—an order of magnitude lower than RSS at a load of 0.56—thanks to improved load balancing, it only supports 61% of the maximum possible load because of its memory bandwidth bottleneck.

SRQ scales beyond RSS and RPCValet, meeting the SLO up to a load of 0.76. SRQ outperforms RPCValet despite load imbalance because its network buffers are always reused in the LLC, thus eliminating all traffic from Step (1) of Figure 4.1 and also writebacks of dirty network buffer blocks in Step (4). Effectively, network contention for memory bandwidth is removed from the accesss in Step (3).

Despite SRQ's improved performance, it still leaves 24% of the server's maximum throughput unprocured, due to its inability to balance load across cores. SRQ's lost throughput would increase proportionally to RPC service time variance. The parameterization of §4.1.2 and Figure 4.2 models the service times of the MICA in-memory data store, which we have observed have a narrow variance similar to an exponential distribution. NEBULA combines the best of RPCValet and SRQ, attaining near-optimal 99th% latency up to a load of 0.97, only saturating when it becomes CPU bound. Even at maximum load, NEBULA only consumes ~ 50% of the server's maximum memory bandwidth.

Conclusions: Our model shows that RPC-centric server architectures must address both load balancing and memory bandwidth interference. Although load balancing has been extensively addressed in prior work from both architectural [54] and operating system [150, 223] perspectives, our models demonstrate that memory bandwidth contention is also a primary performance determinant. Next, we present the principles guiding our design to address memory bandwidth interference and attain the performance of Figure 4.2's best-performing configuration.

4.2 Designing the NEBULA Architecture

To begin the NEBULA design process, we begin with a server and network architecture featuring a hardware-terminated, user-level network protocol, with an on-chip integrated NIC [53, 199]. The fundamental reason we chose to target hardware-terminated, user-level protocol stacks is their suitability for the most challenging, µs-scale RPCs (see §2.5). NEBULA's prerequisites from the underlying network protocol are the following two characteristics.

Prerequisite 1: RPC-oriented transport. Recent work advocates for RPC-oriented transports as a better fit than conventional byte-stream transports in datacenters, because they enable improved flow control [193], latency-aware routing [96], and inter-server load balancing [150]. As RPC transports expose the abstraction of an RPC in the transport layer, the NIC can make decisions pertinent to the application-level unit that an RPC represents, rather than the limited view of packets as mere chunks of bytes. With RPC as a native network abstraction, the NIC can multiplex the buffers of a single network endpoint among all its clients. Therefore, combining RPC-oriented transports with hardware-terminated stacks allows reducing the theoretical minimum number of buffers for N RPC clients from N to one.

Prerequisite 2: NIC-driven load balancing. NEBULA relies on a synchronization-free mechanism to load-balance RPCs between a server's many cores, allowing the cores to reach their highest possible throughput. Otherwise, in the presence of load imbalance, a server could begin violating the application's SLO due to increased application-layer queueing, hitting an earlier bottleneck than bandwidth interference, as shown by the behaviour of the *SRQ* system in our analysis in §4.1.

Design Insights. NEBULA's enabling insight to break the tradeoff between memory bandwidth interference and load imbalance is the realization that by placing tight constraints on the SLO of the system, the system designer is effectively stating that RPCs *must be served out of short queues*. Conversely, RPCs which are idling in deeper queues will almost surely violate the SLO; therefore, it is sufficient for a server to only provision enough space for queue depths not corresponding to SLO violation. In theory, such short queues would easily be accommodated in the SRAM resources already existing in a server chip. The next section shows that this exact scenario is attainable for a variety of request processing time distributions. We leverage this

insight to relax both buffer provisioning and hardware requirements, achieving a performance gain of up to ~ 2× on our evaluated system.

Given the insight that an SLO-abiding queue depth should be small enough to trivially reside in a server's LLC, we identify a further optimization to improve the performance of the RPCs themselves – an integrated NIC that handles RPC load balancing has the ability to place incoming RPC payloads directly into the various cores' L1 caches, because it knows the exact core that will process each RPC. Therefore, the RPC's payload will be accessible in just a few cycles when the core is ready to process it, rather than reading it from an LLC location which is potentially many network hops away. Our design agrees with prior motivation showing on-chip interactions can indeed matter when payloads are just a few cache blocks [53, 76].

Next, we present the theoretical underpinnings that inform NEBULA's critical design feature: an analysis that shows the queues for latency-critical RPCs can indeed be fully SRAM-resident, simultaneously eliminating bandwidth interference and enabling the performance benefits of RPC steering into L1 caches.

4.2.1 Achieving LLC-Resident Queues

Our work in this thesis targets latency-sensitive online services with a strict tail-latency SLO, and therefore responses that violate the SLO have limited value. We leverage this qualitative characteristic to relax buffer provisioning. Bounding queue depths by taking SLO constraints into account allows the network buffers to shrink and achieve greater LLC residency. This observation leads to the natural questions: how long *are* the queues of a server which is operating under its SLO target, and could those queues be made to reside in LLC, thus eliminating bandwidth interference? In order to answer these questions, we conduct the following theoretical queueing analysis.

Let a single server's RPC response time be $t_r = t_q + t_s$, where t_q and t_s represent an average RPC's queueing and service time, respectively. Assuming an SLO of $10 \times t_s$ as is common in the literature [54, 223] constrains queueing time to $t_q \leq 9t_s$. To respect the SLO, a hypothetical server with a single processing unit must be operating at a load point where the 99th% of the distribution of the number of waiting RPCs is ≤ 9.

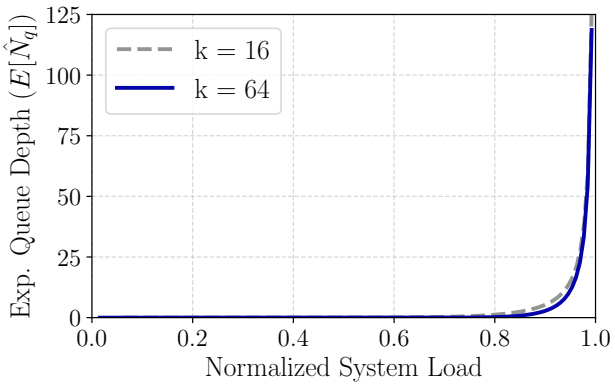


Figure 4.3: Relationship between offered load and $E\left[\hat{N}_q\right]$ for two different core counts (k).

To generalize this observation to a many-core server, consider a k-core server modelled after an `M/G/k` queueing system, assuming Poisson arrivals and a general service time distribution. The answer to our question about queue depth lies in the percentiles of the distribution $\hat{N}_q$ that represents the number of queued RPCs under a system load $A = \frac{\lambda}{\mu}$, where λ is the arrival rate of new requests, and μ is the per-core service rate (stability condition: $\lambda < k \times \mu$). When the service time distribution is exponential, $E[\hat{N}_q]$ is given by Equation (4.2) [279], where $C_k(A)$ is the Erlang-C formula:

$$E[\hat{N}_q] = C_k(A)\frac{A}{k-A} \tag{4.2}$$

In Figure 4.3, we plot $E[\hat{N}_q]$ for $k = 16$ and $k = 64$, to represent a server CPU with a medium- and high-core count. Note that although $E[\hat{N}_q] \xrightarrow{A \to k} \infty$, solving the equation for high system load results in $E[\hat{N}_q]$ values approximately equal to k. For example, for a 64-core CPU at an extreme load of $A = 63 (\lambda = 0.984)$, $E[\hat{N}_q] = 54$. This result shows that server queue depths at the 50th% are approximately equal to the number of cores k, and therefore under stable load will easily fit in a server's LLC.

Ideally, we would be able to analytically solve for the 99th% of the $\hat{N}_q$ distribution, but closed-form expressions for the various percentiles of $\hat{N}_q$ are not known, especially in the case of generalized arrival and service time distributions. Therefore, to study the queue depths at the 99th% of $\hat{N}_q$, we return to the queueing system previously described in §4.1.1 to collect values

Distribution	Max Load at SLO	99th% Queue Depth at SLO
Deterministic	0.999	54
Exponential	0.995	319
Bimodal	0.940	410

Table 4.1: 99th% of the number of queued requests, for a many-core server at maximum load under 10× SLO.

for $\hat{N}_q$ during the simulation's runtime. For this experiment, we use $k = 64$ and replace the service time distribution of MICA RPCs with the following distributions taken from ZygOS [223], to represent distributions with low and high dispersion:

- Deterministic: $P[X = \bar{S}] = 1$
- Exponential: $P[X] = \lambda e^{-\lambda x}$ ($\bar{S} = \frac{1}{\lambda} = 1$)
- Bimodal: $P[X = \frac{\bar{S}}{2}] = 0.9$, $P[X = 5.5 \times \bar{S}] = 0.1$

In Table 4.1 we show the maximum load meeting an SLO of $10 \times \bar{S}$, and the 99th% of $\hat{N}_q$ at that load. The results corroborate the intuition that as the service time dispersion grows (e.g., for bimodal), the peak load under SLO drops and the 99th% queue depth increases. In contrast, the deterministic distribution's 99th% is equal to $E[\hat{N}_q]$, because there is no variability in the rate at which cores drain requests from the queue. This analysis shows that even at extremely high loads, the number of RPCs waiting in the queue is small enough to easily fit inside a server's existing SRAM resources. Assuming each RPC is provisioned 1KB of payload buffering, a microservice exhibiting a bimodal service time distribution would require less than 500KB of buffering.[4] This amount is easily accommodated in modern server LLCs that often exceed 1-2MB of space per CPU core [16, 170]. Provisioning for deeper queueing is effectively useless, because RPCs landing in queues deeper than the upper bound demonstrated by our analysis will violate the SLO anyway.

Design Implications. Our above analysis shows that even at the 99th%, the number of queued RPCs is small enough to be resident in the server's LLC. However, if the buffers are full (e.g., during spikes of incoming load) the transport protocol can either signal a "failure to deliver" or negative acknowledgement (NACK) to the client, or silently refuse to receive the RPC. Silently

[4] 90% of all flows in Facebook's datacenters are less than 1KB [229].

dropping requests means that the clients must perform proactive load monitoring for requests that were rejected as expected SLO violations, because the transport protocol does not provide active feedback. We argue that exposing queueing failures through NACKs to the client is a superior choice, because it follows the end-to-end principle in systems design [234]: the client application is best equipped to handle such violations and should be informed immediately. Furthermore, actively "shedding load" is one of the critical principles in designing systems that rely on dynamic resource allocation [157, §5].

For the NEBULA architecture, we choose to give clients the responsibility to react to receiving NACKs from the servers; example policies might include retrying the RPC or sending it to a different server, as proposed by Kogias et al. [147]. This policy is also synergistic with existing tail-tolerant software techniques, which eagerly retry [59] or reject [97] requests predicted to be delayed. The use of an RPC-oriented transport (Prerequisite 1) strongly enables such policies.

Moving from connection- to RPC-oriented transports means that packets belonging to various multi-packet RPCs are intermingled in the network, and the protocol controllers at both endpoints must demultiplex the RPCs into destination buffers in the correct order. Given our basic assumption of a hardware-terminated protocol, such demultiplexing and reassembly needs to be handled by the NIC. Therefore, NEBULA requires the addition of NIC extensions to perform the following three tasks:

1. Assign buffer addresses dynamically to incoming RPCs, because the transport-level connection no longer dictates where the hardware controller writes the RPC into memory.
2. Reassemble fragmented messages into the buffers that each RPC is allocated.
3. Send extended protocol messages to clients indicating that an incoming RPC is NACKed, when sufficient buffers are not available on the server.

Server architectures featuring on-chip integrated NICs exacerbate the dynamic buffer assignment and reassembly challenges in two ways. First, the on-chip resources that can be dedicated to the NIC's buffer assignment and reassembly hardware are limited by tight power and area constraints, as well as by the use of existing interfaces to the server's memory system. Second, because architectures with integrated NICs often use small MTUs (e.g., 64B in

Scale-Out NUMA [199]), the fraction of RPCs that are fragmented and need reassembly is large.

Although most RPCs in the datacenter are small, many are still larger than 64B [26, 193]. Prior work circumvented these reassembly challenges by allocating dedicated buffers for messaging per node pair [54], leading to the buffer bloat implications detailed in §4.1. As NEBULA drastically reduces buffering requirements and shares buffers between endpoints, we employ a protocol-hardware co-design to support efficient packet reassembly, even at futuristic network bandwidths. Having described the insights and architectural additions required to virtually guarantee that NIC traffic is handled from the server's caches rather than system memory, we now shift to discuss how an integrated NIC can increase RPC throughput by judiciously placing RPC payloads into the server's L1 caches.

4.2.2 Steering RPC Payloads to L1 Caches

In a system where all network traffic can be handled from the server's cache hierarchy (as discussed in the prior section), the performance of that cache hierarchy lies on the critical path, and therefore the numerous on-chip interactions involved in the delivery of an RPC to a core directly impact the server's attainable throughput. Therefore, steering an incoming RPC from the network to a core's L1 cache, rather than having the core read it from memory or the LLC, can noticeably accelerate the RPC's startup. However, such an action has to be both *accurate* and *timely* to avoid adverse effects.

A key challenge of NIC-to-core RPC steering is that the NIC generally does not know a priori which core will handle a given incoming RPC, and inaccurate steering would be detrimental to performance. A second peril is potentially over-steering RPCs, because storing several incoming RPCs in a core's L1 cache before the core consumes them could lead to cache thrashing. DDIO avoids these complications by conservatively limiting network packet steering to a configurable fraction of the LLC [119] and not further up the cache hierarchy, but exposes the latency of the server's distributed LLC to each RPC.

The existence of NIC-driven load balancing mechanisms (Prerequisite 2) is actually an implicit solution to the complications of NIC-to-core steering, because the NIC is the module that

chooses which core will handle an RPC. Therefore, an RPC's payload can be steered to the core's L1 cache with no accuracy concerns, except in the rare case where the operating system migrates an RPC-handling thread to a core C_j *after* RPC payloads are steered to its previous core C_i and *before* it accesses those payloads. However, we do not consider this case as a primary design requirement because we expect it to be exceedingly rare. As threads handling latency-critical RPCs will likely be running on dedicated cores, such rescheduling events are limited by design. Furthermore, RPC payload steering does not introduce any correctness implications for the application or operating system scheduler, and is completely transparent to both layers.

Successful NIC-to-core RPC steering requires breaking RPC handling into two distinct steps: arrival and dispatch. The goal of the arrival step is payload placement in an LLC-resident set of buffers to mitigate memory bandwidth interference, which we have discussed in §4.2.1. The dispatch step's responsibility is to take this fully-reassembled RPC, and assign it to a core based on the NIC's particular load balancing policy. Steering that RPC's payload from the LLC to a core's L1 cache can be done in parallel with the dispatch step, sufficiently before that core starts processing the RPC; therefore, the challenge of NIC-to-core steering is no longer *which* core to dispatch to, but *when* to steer the payload.

Ensuring that payload steering happens in a timely fashion requires overlapping the execution of one RPC with the payload steering of another, and hence it is no longer possible to achieve true single-queueing because each core has multiple requests assigned to it at the same time. It is intuitive that the shorter the length of each core's private queue, the closer the overall system approximates single-queueing. Prior work has shown that allowing this limited multi-queue effect is actually mandatory to hide the communication latency between the CPU cores and the load balancing component [54, 150]. We therefore repurpose the design decision to allow two outstanding requests per core for the purposes of NIC-to-core steering – as the first request is executing, the second request's payload is being steered into the L1 cache. Next, we present an end-to-end overview of how RPC traffic flows through NEBULA's data plane.

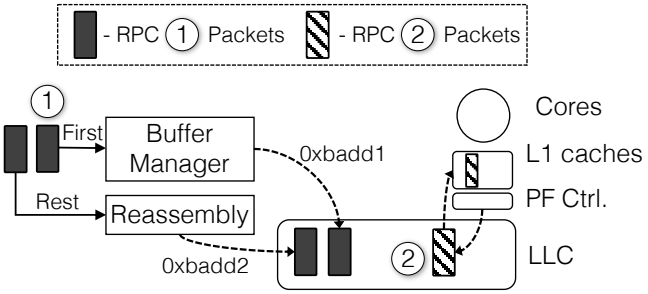


Figure 4.4: Walkthrough of the process of RPC reception in NEBULA.

4.3 NEBULA Operational Walkthrough

In Figure 4.4, we display the three additional hardware components previously discussed in this section, that collectively make up the NEBULA architecture. Note that the proposed design does not assume any specific network stack implementation, and its only two prerequisites are the two previously discussed in §4.2. Figure 4.4 shows the system in a state where it is currently processing two RPCs – RPC (1) that is undergoing arrival and reassembly, and RPC (2) that has already exited the transport layer and is in the process of having its payload steered to the CPU core.

When the first packet comprising RPC (1) arrives at the server, it interacts with the component which assigns the RPC a set of buffers from a pool exposed to the NIC – this component is labelled "Buffer Manager" in Figure 4.4. To know how large the RPC is and therefore how much space must be reserved for it, the buffer manager reads the transport protocol's headers which contain the RPC's size. We expect RPC sizes to be readily available at the transport layer, because all emerging RPC transports we are aware of contain the RPC's size as a header field. Knowledge of the RPC's size and number of fragmented packets enables features such as in-network prioritization in Homa [193], load balancing triggered by an RPC's first packet in R2P2 [150], and payload trimming in NDP [96]. In NEBULA, exposing the RPC's size and number of incoming packets at the transport layer has a similar benefit, allowing the NIC to trigger new buffer reservations on the first packet.

RPC (1)'s subsequent packets interact with the component labelled "Reassembler", whose responsibility is to ensure the RPC is arranged contiguously in the server's memory so that

it is readable by the software. The Reassembler component does this by calculating the correct cache block address for each RPC payload packet with a simple addition from the base address originally returned by the Buffer Manager. Once again, exposing the RPC's size and packetization allows the Reassembler to handle each packet independently, simplifying the underlying hardware implementation because no intermediary storage components are required (e.g., to re-shuffle the packet's ordering). All the RPC's packets are written into the server's memory hierarchy in the exact same way that the NIC would usually do so (e.g., with DDIO's write-allocate, write-update policy [105, 119]). NEBULA does not impose any specific requirements on this process, leaving the caches and coherence protocol unmodified for simplicity.

Finally, once an RPC is completely reassembled by the previous two components, NIC-to-core payload steering takes place. NEBULA performs payload steering by augmenting the pre-existing NIC-driven load balancer (Prerequisite 2). All such load balancers need to communicate the existence of new RPC arrivals with the CPU cores through some kind of metadata, whether through a dedicated interface as in NanoPU [108] or in-memory queues as in RPC-Valet [54]. Therefore, it is sufficient to achieve payload steering via a simple prefetch controller that parses these metadata messages and generates the appropriate cache coherence protocol requests that load the payload's data into the L1 cache. Figure 4.4 shows the prefetch controller component (PF Ctrl.) generating a prefetch request for RPC (2), that has one of its two payload packets already present in the displayed L1 cache.

4.4 Chapter Summary

In this chapter, we have presented the motivation and design of the NEBULA architecture, which breaks the tradeoff we identify between load imbalance and memory bandwidth interference in hardware-terminated network protocol stacks. The underlying insight that drives the design of NEBULA is that meeting strict SLOs requires shallow server-side RPC queues. Our queueing analysis and simulations show that under such SLO goals, server-side queues are shallow enough to easily fit inside a server's LLC. Thus, system designers can leverage this observation to provision drastically reduced buffering, eliminating excess memory footprint for RPC buffers, and by extension, the latency implications of memory bandwidth interference.

RPC buffers can also be accurately prefetched into the L1 caches of the server's CPU cores, boosting throughput even further. In the second part of this thesis (Chapter 6), we propose a concrete implementation of the NEBULA architecture and demonstrate that it yields up to 2× higher RPC throughput under SLO. NEBULA ensures that hardware-terminated protocol stacks can scale to the line rates of future NICs and thousands of communicating endpoints without incurring excess on-server latency, enabling µs-scale software deployments at full datacenter scale.

5 Locality-Aware Load Balancing

Designing a RPC-centric server architecture requires not just taking into account the unique constraints of µs-scale RPCs, but also the unique opportunities they expose. In Chapter 4, we discussed an emerging performance quandary in handling RPCs with µs-scale runtimes. In contrast, this chapter presents an emerging opportunity – enhancing load balancing policies to take into account locality available in the RPCs themselves, reducing latencies at both the average and tail.

We first study two classes of microservice applications, and identify which locality metrics each one benefits from. Specifically, for microservices that primarily execute stateless application logic, we show that the most salient form of locality presents itself in the CPU cores' instruction caches. In contrast, microservices that primarily store or cache data are more impacted by improved concurrency control because they are often bottlenecked by synchronization.

Based on modelling the load balancing tradeoffs inherent in both above metrics, we claim that existing systems are unable to effectively implement locality-aware load balancing because the relevant application-level metrics (e.g., the RPC's type that dictates instruction-cache locality) are exposed at the wrong layer. Clients themselves are aware of these RPC-level metrics, but are only able to implement simple static policies that sacrifice tail latency by design. In contrast, hardware components that can perform load balancing (e.g., SmartNICs or emerging integrated designs) are agnostic to RPC semantics and only use transport-level information, meaning that their shortest-queue load balancing policies cannot benefit from

the opportunities we identify.

We therefore argue that the use of RPC-oriented transport protocols (Chapter 4) and the inclusion of RPC protocol acceleration (Chapter 3) provide the ideal substrate for enabling locality-aware load balancing algorithms at individual servers. To achieve this goal for an RPC-centric server architecture, we introduce a simple insight – knowing the *content* of each application's RPC queue, in addition to its length, is sufficient to realize improved locality-aware load balancing. We use this insight to show that NIC-driven load balancing policies can simultaneously reduce RPC execution times while maintaining tail latency guarantees.

The remainder of this chapter first explains the motivation for improved load balancing for microservices, demonstrates the two specific opportunities we identify to improve microservice performance, and finally proposes a simple set of NIC extensions which fit well into the constraints of on-chip integrated designs. We demonstrate the implementation and evaluation of our set of extensions in Chapter 8.

5.1 How Can Load Balancing be Improved?

In a server ecosystem characterized by µs-scale RPCs, hardware-terminated protocols, and continually growing core counts (see Chapter 2), inter-core load balancing is of critical importance. Due to the cost of acquiring and maintaining server hardware, datacenter operators typically attempt to ensure that their systems are as highly utilized as possible without violating end-to-end Service Level Objectives (SLOs). A deployment that experiences significant load imbalance contradicts this objective by needing to be under-utilized to control tail latency, even with sufficient incoming load to theoretically utilize all the CPU cores. It is therefore important that RPC-centric server architectures can maximize the utilization of various classes of microservices while controlling tail latency via the use of state-of-the-art load balancing policies (see §2.4).

The critical shortcoming of state-of-the-art load balancing frameworks is that they are fundamentally application agnostic, and primarily deal with RPCs as opaque tasks with independent service time distributions. However, it is intuitive that application-level constraints also have a significant part to play in the performance of a load balancing policy. For example, assigning

two RPCs to different cores is counterproductive when they need to enter the same critical section – knowing this information at load balancing time would allow such pathological decisions to be avoided. Furthermore, assigning a RPC to a core that already has that request's instructions and/or data in its L1 cache can reduce service time. Implementing such policies may create scenarios where it is beneficial to artificially create queueing rather than to avoid it at all costs. In this chapter we therefore ask: how can a load balancing policy actually improve the service times of the RPCs themselves while simultaneously maintaining tail latency constraints?

Our answer to this question is that every application's constraints are unique, and therefore load balancers need to expose an interface to software that allows per-application policies to be expressed and cooperatively implemented. In this chapter, we identify and quantify two application-specific policies that can indeed improve RPC service times through improved locality, boosting microservice performance by $1.2 - 1.8\times$. First, we focus on microservices that primarily perform stateless computations of the application's business logic. We show that the instruction working sets of individual RPCs are small enough to be resident in the L1 instruction caches, but such near-ideal cache behaviour is *prevented* by a load balancing policy that is locality-agnostic and thus assigns multiple RPC types to the same CPU core, inducing instruction cache capacity misses. Next, we change our focus to microservices which store data (e.g., caching layers), and identify that improved concurrency control can drastically improve performance and data cache locality. In particular, we show that state-of-the-art concurrency control algorithms for in-memory data stores expose significant inefficiencies when handling write-heavy workloads, which result in up to $5\times$ worse 99th% latency or up to $2\times$ reduced throughput. Through the use of analytical modelling and event simulation, we show that *both* above opportunities result in the same fundamental tradeoff and the benefits of an improved policy.

Furthermore, we claim that current systems are unable to benefit from either above locality opportunity, because all currently available system support to improve locality results in RPCs being *permanently* partitioned among server cores. Such permanent partitioning directly contradicts the well-accepted goal of minimizing RPC queueing, and would result in a final deployment that sacrifices tail latency by design because it eliminates the possibility of load

balancing. For example, although it would be possible to improve instruction locality by partitioning RPC types across TCP connections and enforcing connection-to-core affinity, the server would be unable to re-balance RPCs to other cores in the case of long-latency events. Adding the functionality for cores to shuffle connections among themselves as in ZygOS [223] would improve tail latency in such cases, but the overheads of core-to-core communication quickly become prohibitive with small µs-scale tasks. ZygOS' authors report that with 5µs tasks, ZygOS can only achieve ~ 60% of the maximum theoretical load [223, Fig.7], which can be attributed to the cost of software work stealing.

The critical factor that enables us to enhance current load balancing policies with application-level metrics is the combined presence of hardware acceleration for the network protocol stack, RPC protocol unit, and NIC-CPU interface. We employ hardware termination at both the transport and RPC protocol layers to extract certain fields of the RPC's header at the NIC's line rate, and a hardware load balancing module to dispatch requests from the server's NIC. Microservice software running on the CPU cores sends feedback messages to the NIC to aid it to make load balancing decisions with knowledge of each core's real-time state. We now present the first opportunity that we identify to improve RPC throughput – increasing instruction-cache locality in microservices with many independent functions.

5.2 Temporal Instruction Locality in Microservices

Recall that in §3.1.2, we introduced the software architecture of a representative microservice, which exposes a set of user-visible *functions* that perform various tasks. Our new insight in this section is that significant temporal instruction locality exists among the functions making up a microservice, and that enhancing the system's load balancing policy with the notion of instruction locality can drastically reduce the runtimes of the functions themselves. Improved function performance translates to higher throughput under the programmer-specified SLO limit or equivalently, lower tail latency at the same load.

The instruction supply problem in servers is a well-known problem stretching back decades. Despite ongoing attempts from both the software and hardware fields to eliminate these issues, servers at Google still waste between 15 – 30% of their pipeline slots due to inefficiencies in

the CPU's front-end [133], reportedly forming a multi-million dollar problem [27]. We studied the microservices from DeathStarBench [81] and measured that their working sets ranged from 64 – 150KB, roughly 20× smaller than those of traditional cloud applications studied by seminal work from Ferdman et al. [78] and later corroborated by industry [133]. We expect the microservices' instructions to still spill out of the L1 instruction caches (I$s), because their working sets are roughly 2 – 4× larger than the I$s of today's server processors (e.g., Intel's Ice Lake [255] or AMD's EPYC v3 [36]). Current work confirms that the frontend inefficiencies remain considerable – the authors of DeathStarBench report that the pipeline slots wasted due to frontend stalls are between 20 – 40% [81, Fig. 10]. As a result, CPU front-end stalls remain an important bottleneck to address for microservices.

The inclusion of accelerators for RPC protocol operations [136, 275] that offload the `Wire Protocol` and `Transport` layers of the RPC stack implies means that all the code comprising those layers disappears completely, leaving only the actual function code behind. In our experiments, removing these two layers shrinks the code footprint of our studied microservices by 50 – 75%, meaning that the working sets of some microservices are small enough to be L1 I$-resident. However, for microservices consisting of numerous functions, we have observed that the aggregate working set again outstrips the L1 I$ and thus frontend stalls reappear. These stalls are a form of *function thrashing* in the L1 I$, which is attributable to the fact that the same core is handling all the microservice's functions.

We argue that a sufficient solution to this problem is to form "function partitions" whose aggregate resources are within the capacity of the CPU's frontend. Creating such partitions with NIC-driven load balancing means only assigning requests for a function to the cores that are part of its respective partition, minimizing function thrashing and increasing the CPU's throughput. We hereafter refer to this as *affinity-based load balancing*, following seminal literature in OS scheduling based on data cache footprints [245]. However, forming static partitions creates the potential for load imbalance to arise between them, as a result of the mixture of incoming requests sent by clients. The resulting tradeoff between minimizing function thrashing and preserving load balancing flexibility begs the question: despite the increased service rate from static affinity-based load balancing, is it truly effective when the tail latency of the microservice is the metric of interest?

Answering the above question requires estimating two factors for the microservice under consideration. First, one needs to determine the relative size of each function partition – the more functions comprising a partition before function thrashing occurs, the more load balancing is inherently preserved in a partitioned deployment. Second, given a set of function partitions, one needs to determine the per-core throughput recouped from various function partitioning strategies. In the next section, we propose a basic analytical model that estimates the size of the function partitions required for a microservice.

5.2.1 Boundary Conditions for Function Thrashing

The size of a microservice's function partition is primarily dependent on the frontend microarchitecture of the CPU on which it executes. We simplify the plethora of microarchitectural factors that can impact a given microservice's performance (e.g., I$ size and associativity, choice of prefetcher, and instruction issue width), by classifying the design space of CPU frontends into two categories based on whether they implement branch predictor-driven I$ prefetching or not. We choose this characteristic as the principal delimiter because it dictates which frontend resource comes under stress when many functions run on a single core.

Function thrashing should induce instruction stalls in basic frontends (e.g., those with simple next-line prefetching) mainly due to limited cache capacity, as the aggregate working set of the functions spills out of the cache, and limited misses can be covered. In contrast, frontends with state-of-the-art predictor-driven prefetching (e.g., designs such as Boomerang [155] or Shotgun [154]) can speculatively explore the program's future control flow, and cover misses *if and only if* the BTB attains a high hit rate. Such designs will shift the capacity limitations from the cache itself to the predictor's metadata stored in the BTB.

Knowledge of the functions' working sets makes it possible to calculate the boundary condition beyond which a particular grouping of functions will begin producing contention. For basic frontends, the threshold for producing function contention can be calculated as the number of functions F that satisfies the following inequality:

$$\sum_{i=0}^{F} (W_i - L) > S_{L1} - L \tag{5.1}$$

Frontend Type	L = 5KB	L = 10KB	L = 15KB
Basic	1	2	3
Predictor-Driven	3	5	9

Table 5.1: Number of functions that can be entirely contained in basic and predictor-driven CPU frontends without incurring I$ thrashing.

Where S_{L1} is the instruction cache capacity, W_i is the working set of function i, and L is the amount of shared code (e.g., due to libraries) between all functions. This analysis assumes *a priori* knowledge of L, which itself can be calculated as the intersection of all working sets for the functions i as follows: $L = \{W_0 \cap W_1 \cdots \cap W_i\}$. We choose the unit of each term to be the *number of instructions* and not bytes, to facilitate the following comparison.[1]

For frontends featuring predictor-driven prefetching, an expression similar to Equation (5.1) applies, replacing S_{L1} with S_{BTB}, the capacity of the BTB. Assuming the well known rule of thumb that $\frac{1}{5}$ of instructions are branches, both W_i and L are therefore scaled down by a factor of 5. The two expressions are equivalent when $S_{L1} = 5S_{BTB}$, meaning that a predictor-driven frontend has 5× higher function capacity than the instruction cache because it only stores branch instructions.

In Table 5.1, we report the values of F for both CPU frontend types, assuming homogeneous 20KB functions, a 32KB I$, and a 3K-entry BTB. Our model indicates that for CPUs with basic frontend designs, contention can occur with as few as two functions concurrently residing on the same core. In contrast, advanced frontends can support more functions due to the bottleneck shifting from the instruction cache to the more space-efficient BTB. Note that the values for the predictor-driven frontend are not simply equal to 5× the value of the basic frontend, because our assumptions of L1 I$ and BTB sizes give $S_{L1} \simeq 2.5S_{BTB}$. As the amount of library code grows, contention naturally drops and more functions can co-exist before contention occurs. These results are conservative, as we have observed that the working sets of some functions are much larger than 20KB and will lead to more contention.

Using the above model as a guide, frontend contention can be eliminated by ensuring that each core receives requests corresponding to no more than F unique function types. To form the

[1] NB: In modern ISAs such as ARM64 and RISC-V, instructions are fixed-length, so the two terms are related by dividing size by instruction width.

partitions, the system designer needs knowledge of the CPU microarchitecture, the functions' working sets, and the library code shared among them. While the former is simple to discover through architecturally-exposed registers, the latter requires some form of profiling. We expect that for datacenter software which is deployed at scale, the required profiling support could be built as an extension of production systems that aggregate performance counters across machines [27]. With the aid of our model allowing us to estimate the size of a microservice's function partitions to minimize function thrashing, we now can answer whether function partitioning is indeed effective under tail latency constraints.

5.2.2 Towards Dynamic Function Partitioning

To study whether the improved execution times from static function partitioning outweigh the sacrifice in load balancing flexibility, we return to our queueing simulator (§3.3). We conduct a simple experiment that models a 16-core server running a single microservice with 16 functions, each having exponentially distributed service times. Incoming RPCs are generated with exponentially distributed inter-arrival times, and we assume that functions are uniformly popular. Our experiment compares the following partitioning strategies:

- **None** - this design represents the baseline system where any function can be run on any core, and therefore represents single-queue load balancing. Therefore, we set the average service time of each function to be $\bar{S} = 1$.
- **Full** - in this design point, the function partition size is equal to one, thus each function must be assigned to a single core. This partitioning strategy corresponds to the case of a basic CPU frontend running functions with little common code (see Table 5.1).
  To model the increased instruction locality, we assume that recouping all the wasted frontend slots leads to a 20% performance increase, and thus we set $\bar{S} = 0.8$.
- **Partial** - this design creates four partitions of four functions each, which either corresponds to a basic frontend with significant code sharing, or an advanced frontend.
  We use $\bar{S} = 0.9$ for Partial to model less locality than Full partitioning.
- **Ideal** - this best-case design combines the superior load balancing of None, with the service time reduction of Full partitioning. For this system we set $\bar{S} = 0.8$ and allow single-queue load balancing.

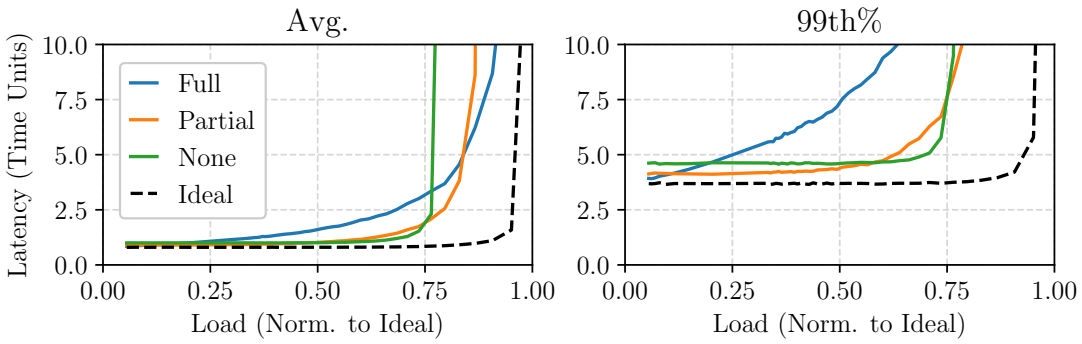


Figure 5.1: Load vs. latency curves for different function partitioning strategies, assuming an SLO of 10 units ($10 \times \bar{S}$).

Although other choices are possible (e.g., two partitions of eight functions each), we omit them for brevity as the results would fall between the above strategies and the theoretical optimum.

Figure 5.1 plots the average and 99th percentile RPC latency for each system. The results for average latency show the additional throughput granted by the Full and Partial function partitioning strategies, as both attain a higher load than the baseline policy (i.e., ≥ 0.76) without reaching instability. However, despite a $10-20\%$ reduction in service time, neither partitioning strategy provides significant benefits when throughput under 99th percentile SLO is concerned. The system with Full partitioning violates the microservice's SLO at a load of just 0.63 because of load imbalance when new requests must queue at occupied cores due to the strict partitioning policy. In contrast, adding a modicum of load balancing with Partial partitioning achieves nearly identical throughput under SLO as None, as they both violate the SLO at a load between 0.76 and 0.8. Partial partitioning does grant reduced 99th% latency in low-load regimes (i.e., when load is beneath 0.58). Naturally, the Ideal system provides both benefits, attaining 20% higher throughput while only violating the SLO at loads beyond 0.93.

Based on these results, we conclude that although static partitioning strategies do provide higher throughput, a 10-20% reduction in service time due to instruction locality does not significantly improve throughput under tight SLO constraints. Static function partitioning may be attractive for microservices which are not bound by tight SLOs, particularly because such partitioning can be completely realized in software. A sample design could be to re-organize the intermediary layers of the RPC protocol stack to create a unique `Transport`-level

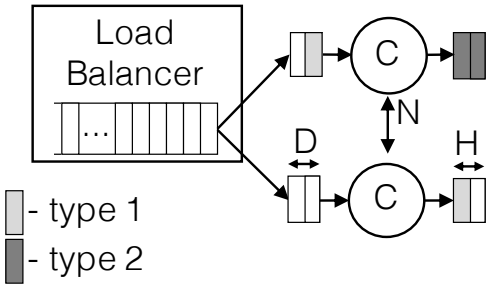


Figure 5.2: Simulated queueing network for `JBSQ(D)` with history.

connection for each function partition, and then using support from Intel's Flow Director [120] to ensure that all requests on this connection are processed by a certain set of cores. To make such as strategy valuable, the SLO either needs to be relaxed, or the instruction locality benefits need to be larger than what we assume. However, we set our goal to design a policy that attains improved I$ locality without sacrificing tail latency, like the Ideal policy shown in Figure 5.1.

The key deficiency of static partitioning is that it does not take advantage of all the information available to the NIC's load balancing stages. As the NIC has knowledge of which core every incoming request has been assigned to, in principle it can assign requests while taking into account load balancing *and* the I$ locality this function will experience on the selected core. At low load, it can prioritize sending incoming requests to cores with more I$ locality, and minimize tail latency at high load by choosing cores with the shortest queues of incoming requests. It is also possible to choose a core having affinity among multiple possibilities with the same queue length. We refer to this policy as *dynamic affinity*, and note that it is similar to the way `irqbalance` maps interrupts to cores in Linux [103].

Intuitively, no policy can ever attain perfect load balancing and perfect instruction locality – achieving one means yielding some of the other. Therefore, it is natural to ask how near a hypothetical load balancing policy can come to the performance of a system like the Ideal one modelled above. The critical variables dictating the performance of such a policy are (i) the fraction of decisions for which affinity is available, and (ii) the choices made by the load balancer to schedule requests based on affinity or limiting queue depth.

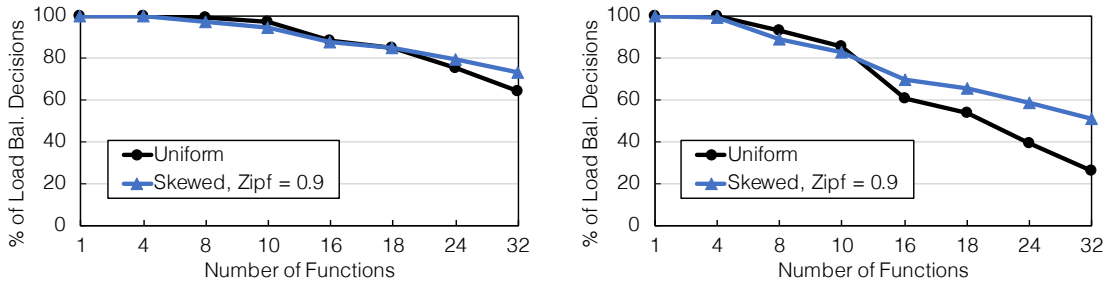
To study (i), we hypothesize that for common deployments, the fraction of decisions with available affinity is high. We base this hypothesis on the insight that the number of "recently

dispatched" functions is large enough to find a single core which has recently served this function, or is about to serve it from its input queue. Furthermore, the affinity potential of a particular deployment should be insensitive to the incoming load. To explain these insights, Figure 5.2 shows a queueing network representing a system with a centralized load balancer having an unbounded queue, and N cores each having a private queue of depth D and a "history" of the H most recently executed functions. This model represents a system using the state-of-the-art Join-Bounded Shortest Queue load balancing policy, with depth D (`JBSQ(D)`) [54, 150].

Recall that previously, we used a known result from queueing theory to show the expected number of queued RPCs is approximately equal to the number of cores at high but stable load (see Equation (4.2)). Therefore, it follows that with a shortest-queue load balancing policy, each core's input queue should be of length $\simeq 1$. It then follows that when the distribution of function popularity is uniform, the majority of load balancing decisions should encounter *at least* one core with potential affinity, under the condition that the number of functions is less than or equal to the number of cores. At low load when many of the private queues will be empty, the location where affinity will be found shifts from the cores' input queues to the function history.

To confirm this expectation, we simulated the queueing network in Figure 5.2 and collected statistics for the number of load balancing decisions where potential affinity was found. We consider that affinity is available when the ID of the function dispatched from the load balancer's unbounded queue matches one of the first F unique function IDs in any private queue, where each queue is processed starting from the tail. The parameter F can be selected in accordance with the analytic model in §5.2.1, but we use $F = 1$ because it corresponds to the worst-case for affinity. For example, in Figure 5.2, a new load balancing decision for function type 2 needs $F \geq 2$ to find affinity in the upper core, whereas a function of type 1 would find affinity in both cores. We simulated a host system with 16 cores at a load of $A = 12$, and do not modify the load balancer's decisions to avoid biasing the results.

Figure 5.3 shows the fraction of load balancing decisions with affinity, for both uniform and skewed function popularity distributions. In particular, Figure 5.3a shows that when the



(a) Affinity to one or more cores.

(b) Affinity to strictly more than one core.

Figure 5.3: Results demonstrating the availability of affinity at RPC dispatch time, for a 16-core system.

number of functions is less than or equal to the number of cores, over 80% of load balancing decisions have the potential to find affinity to at least one core. When the number of functions is smaller than the number of cores, function popularity does not significantly affect the results as expected. Furthermore, the existence of skew positively influences the available affinity where the number of functions is beyond the number of cores. Finally, we confirm that these results are insensitive to load – sweeping the load between 10–90% ($A \in [1.5, 14.4]$) results in less than a 1% change in available affinity.

We also display the fraction of decisions that have potential affinity to *strictly more than one core* in Figure 5.3b. This result is important as it indicates whether available affinity is plentiful enough so that load balancing among multiple affinity-having cores can also be considered. Similar to the results in Figure 5.3a, we see that the vast majority of load balancing decisions have the opportunity to choose among multiple affinity-having cores. In this experiment, popularity skew between the functions positively impacts the availability of affinity when the number of functions is larger than the number of cores (16). Figure 5.3b shows that decisions with potential affinity to more than one core increase by $9-25\%$ with function popularity skew, which is logical as the most popular functions will be more likely to be found in many cores' input queues. Also, the gap between the uniform and skewed distributions grows with function count, which is logical because the popularity relationship between items in a Zipf distribution increases with item count.

Given the abundance of affinity in the above experiment, we remark that a simple proof-

of-concept load balancing policy is sufficient to grant most functions affinity benefits while retaining significant load balancing flexibility. By retaining the function IDs that have recently been assigned to each core, the NIC's load balancer can select a subset of the cores at each new RPC arrival that have affinity to the function being requested in the RPC. Then, the NIC can select the core with the *shortest queue length* from among the affinity-having subset, falling back to the shortest *global* queue if the cores with affinity have queue depths exceeding a pre-defined maximum (e.g., $D-1$).

We remark that such a basic policy is by no means exhaustive, and microservices with specific characteristics may need other factors to be considered. In particular, microservices having orders-of-magnitude differences in function execution times would benefit from concrete policy enhancements to avoid queueing functions with "short" execution times behind those that are "long". A policy like DARC [61] would be sufficient for such cases, and we believe that hybridizing such designs with affinity-enhanced load balancing would be interesting for future work.

In this section, we have shown that despite the pervasiveness of instruction supply issues in microservice workloads, it is possible to use *function partitioning* to alleviate I$ contention. Using analytical modelling and queueing simulation, we conclude that software-driven function partitioning is only effective at relaxed SLOs, and that dynamic policies should be investigated for tight SLOs. Finally, we present evidence that a simple proof-of-concept load balancing policy is sufficient to allow RPCs to execute with improved I$ locality, while maintaining load balancing flexibility. We now turn to demonstrate the second technique with which we propose to enhance load balancing – improved data locality through enhanced concurrency control for microservices that serve data.

5.3 Load Balancing for Write-Intensive Key-Value Workloads

Key-Value Stores (KVS) are a backbone for online services, forming the leaf nodes of most microservice graphs and even serving as underlying infrastructure for software such as message queues [220] and publish-subscribe systems [4]. For proof of their continued importance, consider that four of the six microservices comprising DeathStarBench contain caching layers

in front of *nearly every single database* [81, Figs. 4-7], and industry papers are continually being published to gain more insight into the evolution of key-value workloads [43, 246, 271]. Not only are KVS ubiquitous in production, but they are extremely performance-critical because their response latency is on the critical path of user-facing queries that must meet real-time interactivity requirements [59]. This combination of commonality and stringent performance needs has driven major research and development efforts to optimize the layers underlying KVS [33, 150, 166, 193], or even employ dedicated hardware [123, 162] for the goal of boosting KVS performance.

The load balancing policy and concurrency control algorithm are intimately connected in a system running a KVS, because the load balancer must not create the possibility of threads executing in a manner disallowed by the concurrency control policy. To illustrate, a shortest-queue load balancing algorithm like `JBSQ` assigns requests to workers without regard for the synchronization events taking place among the workers themselves. It is trivial to see that balancing two requests to different workers may be a suboptimal decision if those requests need to serialize based on a lock. Therefore, it is common practice for KVS architects to design bespoke policies that use in-network support to enforce concurrency control invariants – for example, the state-of-the-art MICA KVS uses Flow Director to partition requests for certain objects among cores [166] and the R2P2 protocol employs a *sticky* mode to direct write requests to a master node while spreading reads across replicas [150].

We claim that existing contracts between load balancers and KVS concurrency control algorithms cannot effectively handle write-intensive workloads, *for the same underlying reason* that we saw in the case of permanent function partitioning. Partitioning writes in KVS is commonly done to attain higher throughput via reduced synchronization overheads, but gives up the flexibility of single-queue load balancing and thus sacrifices 99th% latency. As we have already shown in § 4.1.3 and 5.2.2, sacrificing load balancing flexibility introduces the need to run the KVS at lower load to control tail latency. It is therefore eminently desirable to be able to design a KVS that can utilize a server's many cores as much as possible, because handling a given load can be done with fewer total servers, minimizing consistency and replication concerns as well as inter-partition load balance [102, 104]. The trend towards better KVS load balancing is important enough that researchers have even begun to advocate for pooling memory across

traditional socket and server node boundaries [84, 200]. In the remainder of this section, we focus on the single-server case, and demonstrate how simple extensions to NIC-driven load balancing allow a practically implementable system to approach the performance of an ideal single-queue system with no synchronization overheads.

5.3.1 Why Target Write-Intensive Workloads?

In today's datacenters, KVS have transcended their traditional use-cases such as in-memory caching of disk-resident data [42, 81, 198]. Due to the extreme cost of keeping data in-memory [67, 259], system designers must maximize the performance of every provisioned CPU core. Therefore, it is logical for KVS to adopt state-of-the-art proposals promising reduced network delay, kernel-bypass protocols, and first-class load balancing [33, 54, 150, 193, 223].

State-of-the-art KVS systems are often expressly designed around read-mostly workloads with considerable item popularity skew [123, 162, 165, 166, 200]. Emerging studies of production KVS data have revealed the prevalence of workloads that are more write-intensive and/or more severely skewed than previously assumed [43, 271], leading to a natural question: what bottlenecks prevent today's KVS from performing ideally on such workloads? We answer this question with the aid of a taxonomy of KVS workloads, that allows us to intuitively explain the bottlenecks created by each one.

KVS Workload Taxonomy

A given KVS workload can be represented by its location on the cross-product of two axes representing its skewness and write fraction—knowing a workload's location is enough to describe the challenges it will face. Figure 5.4 shows these two axes and identifies four regions that serve as a "spanning set" of the space. Regions are identified using the nomenclature $\{R|RW|WI\}_{(sk|uni)}$, where the leading letter(s) represent(s) whether the workload is read-mostly, read-write, or write-intensive, and the subscript represents whether the data popularity distribution is skewed or pseudo-uniform.

The two regions lying beneath the dashed line are well-studied in KVS literature. R_{uni} workloads are the least challenging - providing a high-throughput KVS is possible with production

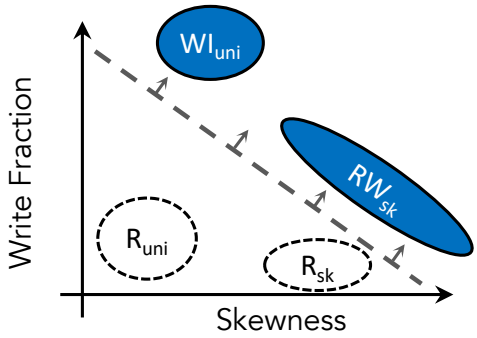


Figure 5.4: KVS workload taxonomy, showing regions targeted by this thesis in blue.

systems such as `memcached` [185]. Adding popularity skew moves the workload to the R_{sk} region, where performance is primarily determined by the maximum throughput for reading the hottest items. Absorbing the workload's skew requires KVS to use a concurrency control policy supporting *concurrent lock-free readers*, such as MICA [166], Masstree [176], RackOut [200], KV-Direct [162], and ccKVS [84].

In contrast, workloads above the dashed line are largely unaddressed by current research. The fundamental challenge in both workload regions is that the increased prevalence of writes creates a tradeoff between achieving high system throughput and creating load imbalance across threads. When writes are plentiful, achieving high throughput mandates a concurrency control policy that partitions writes across threads to avoid synchronization overheads (known as Concurrent Read, Exclusive Write—CREW [166]), but such partitioning necessarily creates load imbalance that leads to increased tail latency or throughput bottlenecks.

WI_{uni} workloads (i.e., those with ≥ 50% writes) will experience inflated tail latencies on today's KVS' simply because write partitioning precludes any load balancing framework [54, 150, 223] from acting on 50% or more of the requests. A recent characterization from Twitter has shown that more than 35% of their production workloads can be classified as WI_{uni} [271], motivating additional work to handle them with improved tail latency. Facebook also reports the prevalence of extremely write-dominated workloads (i.e., having 92% writes) that store ML statistics [43, §4.1].

RW_{sk} workloads also lack write load balancing, and face two additional challenges. First, the write load on a small group of items can be high enough to overwhelm the hottest thread.

Although this challenge has been identified in prior work, the issue has been understood to apply to RW_{sk} workloads with ~ 50% writes [123, 165, 200]. However, in light of recent work showing that skew coefficients are far higher than previously studied [271], static write imbalance becomes a far more common bottleneck which we demonstrate can occur even with single-digit write fractions. Second, the read-write nature of the workload creates high contention on the hottest partition's cache lines, further decreasing per-thread throughput.

Our insight is that concurrency control designs based around static write partitioning are the underlying cause of both problems. A theoretically optimal system would allow any thread to serve reads and writes, leading to completely balanced load. However, enabling concurrent writers has been shown to reduce KVS throughput by 2× due the need for writer-writer synchronization, and assumed costs in cache coherence [166].

Furthermore, although emerging frameworks exist that target writer concurrency and thus support write load balancing [146, 180], they are unlikely to yield benefits for in-memory KVS with µs-scale service times. Despite drastically improving scalability, the additional per-RPC overhead from these frameworks' use of software transactional memory techniques (e.g., version-chaining and garbage collection) would translate to limited performance gains. We confirmed this by porting a hash table-based KVS to the Multi-version Read-Log-Update (MV-RLU) framework [146], and observed that read and write latencies increased by 1.75× and $2-40\times$ respectively. Therefore, instead of pursuing true write concurrency, we contribute the following insights allowing write partitioning to serve as an enabling factor rather than an impediment.

Insight 1: Static write partitioning is overly conservative, and can be relaxed in the vast majority of cases. The only time write partitioning is strictly mandatory is when two writes to the same partition are outstanding at the same time—there is no reason for a write to partition P_X to queue behind one to P_Y when $X \neq Y$. Load balancing of independent writes would allow WI_{uni} workloads to have nearly identical tail latency to R_{uni} ones.

Insight 2: Write partitioning naturally creates batches of queued writes that can be compacted into a single update. In RW_{sk} workloads at high load, mandatory write partitioning is the common case, creating an optimization opportunity at the writer thread—software can com-

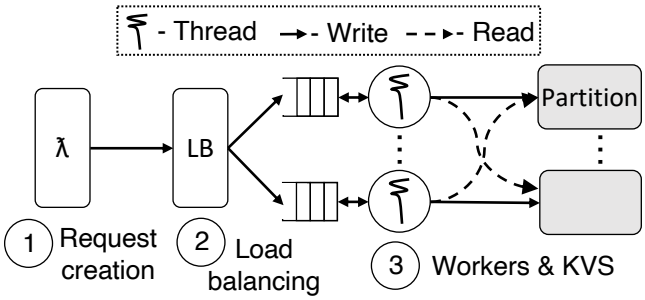


Figure 5.5: Queueing model for a server running a KVS.

pact all the writes into a single update, reducing reader-writer synchronization and cache line access latency. Write compaction has been previously applied to kernel data structures [39], LSM-based KVS [11], and commutative transactions [195], but never in the context of KVS with µs-scale tail-latency Service Level Objectives (SLOs). Having described the challenges created by common concurrency control choices, we now develop a high-level model to estimate the benefits of improved concurrency control for WI_{uni} and RW_{sk} workloads.

5.3.2 KVS Concurrency Control Challenges and Opportunities

To quantify the performance impacts of the aforementioned two insights, we once again use our discrete-event simulator, and construct a queueing system modelling a many-core server running a KVS, whose design is shown in Figure 5.5. In Step (1), our load generator component creates new requests according to a Poisson process, with configurable inter-arrival times. We model workloads from Figure 5.4's regions by varying the percentage of writes, and model the popularity skew according to a Zipfian distribution with configurable γ, as is standard in KVS literature [162, 165, 166, 200]. Step (2) models a load balancer (LB) component which assigns requests to threads. If the KVS' concurrency control policy allows load balancing for this type (e.g., for reads in CREW), the LB assigns the request to a thread according to the JBSQ(2) policy [150]. If not, the LB assigns the request by hashing the key.[2]

Step (3) concerns the worker threads and the KVS itself. We model the KVS as a set of abstract partitions, each containing a version number to allow us to model optimistic concurrency control – when readers fail versioning, they must re-run the entire request. Each request's

[2]We use random strings as keys, and the `sha256` hash algorithm inside Python3's `hashlib`.

service time is modelled as $\bar{S} = T_{kvs} + T_{fixed}$, where T_{kvs} represents the KVS' run-time, and T_{fixed} represents interacting with the load balancer and network stack. We choose T_{kvs} as uniformly distributed in the interval $[400, 800]ns$ to model in-memory data stores, and $T_{fixed} = 100ns$ to model a hardware-terminated protocol. More methodology details can be found in §8.4.1.

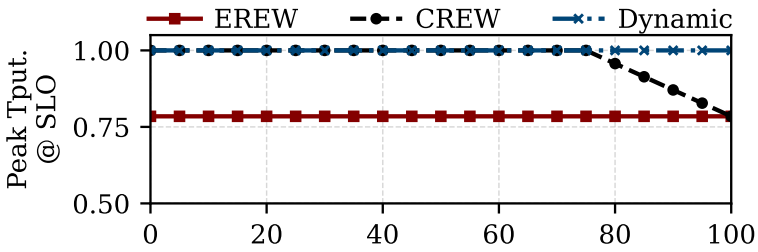
Dynamic Write Partitioning

To demonstrate the effects of *Insight 1* for WI_{uni} workloads, we parametrize the model with 64 worker threads, one million keys with a uniform popularity distribution, and vary the write fraction f_{wr} from 0 – 100%. Borrowing terminology from MICA [166], we study three concurrency control policies:

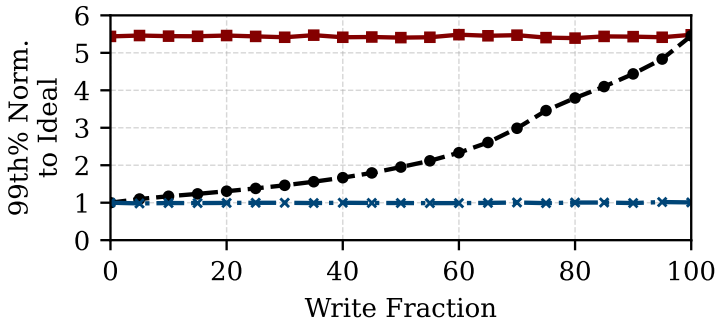
- *Exclusive Read, Exclusive Write (EREW)*: A fully partitioned system with no load balancing because each partition is exclusively assigned to a single core using static key-to-partition hashing for every incoming request.
- *Concurrent Read, Exclusive Write (CREW)*: This system allows concurrent readers and hence load balancing of reads, but uses partitioned writes. All state-of-the-art KVS we are aware of implement some variant of CREW [84, 162, 165, 166, 200].
- *Dynamic*: Maintains concurrent readers, and only treats a partition P_X as being in "exclusive write" mode as long as a write is still outstanding to it. After the write is completed, future writes to the same partition can be served by any thread. Allows load balancing of independent writes disallowed by EREW and CREW.

Figure 5.6a shows the maximum throughput each policy can attain under SLO, and Figure 5.6b shows the resulting excess 99% latency over an ideal system that allows both concurrent reads and writes without any synchronization overheads. In this experiment, we continue to use a 99th% latency target of $10 \times \bar{S}$. Each point in Figure 5.6b is calculated by taking the maximum throughput under SLO of each policy, and normalizing the 99th% latency at that load point to the ideal system. This methodology does not artificially penalize the non-ideal policies if they reach lower throughput than the ideal system (e.g., CREW for $f_{wr} \geq 80\%$).

The additional load balancing flexibility granted by the CREW and Dynamic policies impacts



(a) Throughput under SLO.

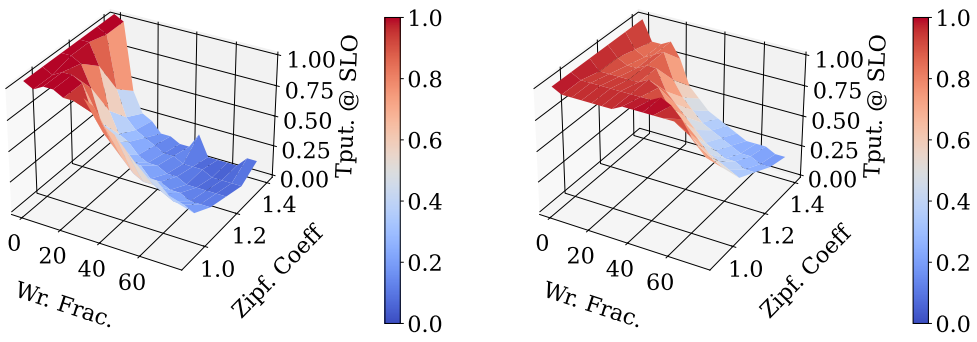


(b) Tail latency.

Figure 5.6: Throughput and tail latency of concurrency control policies compared to an ideal system with no synchronization overhead. Uniform key popularity distribution.

both achievable throughput and tail latency. EREW's inability to balance load leads to system saturation at 80% of the ideal system's throughput, regardless of f_{wr}. By allowing concurrent reads, CREW matches the ideal system's throughput for $f_{wr} < 80\%$. As f_{wr} approaches 100%, the performance of CREW converges to EREW due to queueing of write requests. Thus, even with CREW concurrency control, the KVS must be run at lower load to control tail latency. The Dynamic policy is not constrained by this tradeoff, and matches the performance of the ideal system regardless of f_{wr}.

Figure 5.6b shows that CREW's tail latency matches the ideal system for a read-only workload, and increases proportionally with f_{wr}. For WI_{uni} workloads with $\geq 50\%$ writes, the static write partitioning of CREW and EREW inflates tail latency by $2-5.5\times$. In contrast, the Dynamic policy delivers near-identical tail latency to the ideal system because of its strictly greater load balancing ability than CREW. Our model shows that a dynamic partitioning policy following *Insight 1* can deliver the tail latency of an unattainable ideal system with full concurrency and no synchronization overheads.



(a) CREW baseline.

(b) Compaction enabled.

Figure 5.7: Throughput of a CREW KVS with and without write compaction, normalized to an ideal system with no synchronization overhead.

Write Compaction

To show the impacts of static write imbalance for RW_{sk} workloads and the benefits of write compaction (*Insight 2*), we enhance Step (3) of our queueing system with the following process: i) upon receiving a new write request, a worker scans its request queue for writes to the same key, ii) if found, the writes are buffered into a private log, and iii) the worker initiates a timer as the beginning of a "compaction window". When the timer approaches the KVS' specified SLO, the worker closes the compaction window and sends replies to all requests collected in the log.[3] While a compaction window is open, any writes that are collected in its private log have service time $\bar{S}_{comp} = T_{fixed} + T_{comp}$, where $T_{comp} = 100ns$. We set T_{comp} by measuring the latency of adding a small `struct` representing the compacted request to a pre-sized `std::vector` on a Xeon E5-2680 CPU. An RW_{sk} workload can arise from various parameter combinations in Figure 5.4's design space, as long as the write load on the hottest partition(s) is approximately equal to $1/N$ where N is the number of workers. Due to the fact that higher write load implies greater compaction opportunity, we study the entire design space defined by $\gamma \in [0.9, 1.4]$ and $f_{wr} \in [0, 80]$.

Figure 5.7a shows the throughput under SLO of a CREW KVS, normalized to the same ideal system in §5.3.2. Our results show that write partitioning becomes a clear bottleneck for the vast majority of workloads making up the RW_{sk} region, as the heavy skew simply overwhelms

[3] We discuss consistency considerations in depth in §5.4.2.

the KVS' hottest worker with writes. For example, a workload with $(\gamma, f_{wr}) = (0.99, 35\%)$ only attains 56% of its ideal maximum throughput. Our model shows that write partitioning even leads to load imbalance and thus a throughput bottleneck with f_{wr} in the low single digits for the highest skews. For $(\gamma, f_{wr}) = (1.4, 5\%)$, our model shows that the KVS only attains 66% of its maximum.

In comparison, Figure 5.7b shows the throughput of a KVS with write compaction. Compaction allows a workload with $\gamma = 0.99$ to match the ideal system's performance up to $f_{wr} = 55\%$, and provides a 1.56× speedup for the extremely skewed workload with $\gamma = 1.4$ and $f_{wr} = 5\%$. The above opportunities are in fact conservative, because our queueing system does not consider cache or memory access times. In a real deployment, the CREW baseline would experience significant cache contention on the hottest partition(s), degrading performance faster than shown in Figure 5.7a and granting compaction even greater relative improvements.

In summary, our models show evidence that static write partitioning is not necessarily a bottleneck for KVS performance. Contrary to the inherent limitations of current systems, dynamic write partitioning makes it possible for WI_{uni} workloads to attain near-ideal tail latency. In addition to dynamic write partitioning, software write compaction can boost throughput under SLO by $1.5-2\times$ for RW_{sk} workloads. Following these key insights, we now introduce C-4, a **C**ooperative **C**oncurrency **C**ontrol **C**o-design of NIC-driven load balancing hardware and KVS software to break the tradeoff between static write partitioning and load imbalance.

5.4 Enhancing Load Balancing with C-4

5.4.1 Write Balancing for WI_{uni} Workloads

In §5.3.2, we demonstrated the tail latency benefits of a concurrency control policy that enables load balancing for writes, under the invariant that a partition can only be written by a single thread at a time. It is straightforward for a load balancer to maintain this invariant using a set of ephemeral partition-to-thread mappings and dispatch requests accordingly: partitions with an active mapping are in "exclusive mode", and all writes to them must be sent

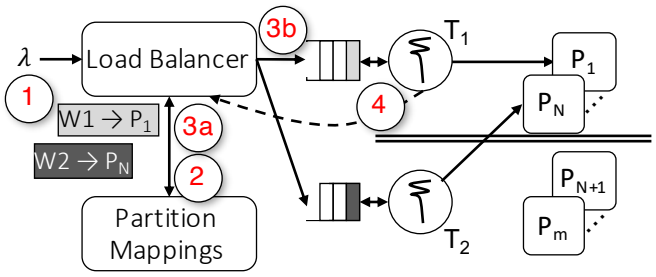


Figure 5.8: Load balancing additions required to realize Dynamic CREW (d-CREW).

to the mapped thread. Requests to other partitions can be freely load-balanced. We call this policy "dynamic Concurrent Read, Exclusive Write" (d-CREW).

Figure 5.8 displays our proposed design to realize d-CREW in today's NICs, by showing two writes W_1 and W_2 that target KVS partitions P_1 and P_N. We assume CREW's static partitioning maps both partitions to thread T_1, shown by double line separating the partitions, meaning the load balancer must assign both W_1 and W_2 to thread T_1. In contrast, d-CREW allows the requests to be balanced across threads T_1 and T_2, as long as the load balancer has knowledge of the partition being accessed by each write request.

With d-CREW, the NIC in Figure 5.8 unpacks each write's application-level header and determines the writes target partitions P_1 and P_N (Step (1)). The load balancer then looks into its active partition-to-thread mappings for the requested partitions, finds no active match (Step (2)), meaning that the NIC's load balancing policy can freely assign the requests to any thread. The NIC then creates two new partition-to-thread mappings for P_1 and P_N, placing them in exclusive mode (Step (3a)) and making them "sticky" to threads T_1 and T_2 as long as each write is outstanding. The writes are then sent to the threads (Step (3b)) for processing. When the threads send responses to the writes, the partitions are no longer in exclusive mode and the NIC can free the two respective partition mappings (Step (4)), allowing incoming writes to P_1 and P_N to again be assigned to any thread.

Introducing d-CREW into the NIC means that the state requirements for storing partition-to-thread mappings must be small enough to result in acceptable hardware provisioning, and low enough access times to meet the NIC's peak sustainable throughput. To illustrate, a KVS serving a workload with 75% writes at an aggressive $\sim 200 MRPS$ where each request takes $\sim 600 ns$

(see §5.3.2) has roughly 90 writes outstanding at any given time. Even with overprovisioning to absorb transient fluctuations, storing a few hundred mappings only requires a few KBs, which today's NICs could easily accommodate and access in nanoseconds.

Implementing "exact match" operations between incoming requests and existing mappings is more difficult than storing the mappings, because it traditionally requires Content Addressable Memories (CAMs) [261]. However, our basic estimate above (confirmed in §8.4.2) reveals the number of mappings to be similar to the size of a CPU TLB. Therefore, we believe it is feasible to implement partition-mapping hardware using existing CAM technology. In the worst case where the system exhausts all the hardware's partition mappings, the drastic mismatch in request arrival and processing rates indicates a load spike requiring flow control mechanisms to throttle or drop requests [49, 249, 250] as we demonstrated in Chapter 4.

5.4.2 Write Compaction for RW_{sk} Workloads

C-4's second mechanism, write compaction, addresses both bottlenecks in RW_{sk} workloads, namely: i) static load imbalance created by the write fraction and skewness that overwhelms a single writer, and ii) contention between writer and reader threads for the cache lines comprising the hottest partition(s). C-4 does so while maintaining the baseline KVS' consistency model.

To absorb the load created by highly skewed workloads, C-4 must achieve drastic speedups for the single thread serving the hottest partition. The authors of NetCache similarly observed that a single unit serving requests on behalf of many workers must provide orders of magnitude higher throughput to handle skewed workloads [123, §2]. However, in our context the hottest thread is not required to handle both read and write load, because the NIC balances reads across all other threads. Therefore, the requisite speedup for the hottest thread drops significantly. As our design takes advantage of the NIC's ability to load-balance reads and independent writes, the request queue of the overwhelmed thread will therefore be full of writes to *only* the hottest partition(s), creating opportunities for compacting these writes into a single batched update.

Intuitively, compacting a given write into a private batch is faster than writing into the KVS'

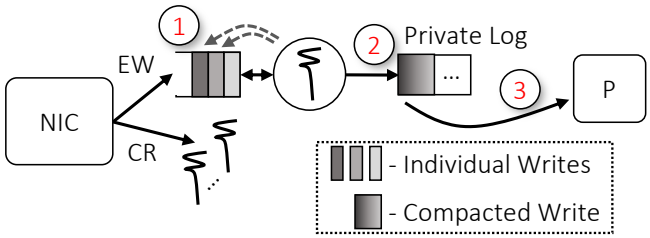


Figure 5.9: Design of write compaction in C-4.

underlying data store, because it is unnecessary to take any locks or write shared cache lines. We estimate that the possible acceleration A of such compaction is:

$$A \simeq \frac{T_b + T_f}{\frac{T_b}{N} + T_c + T_f} \tag{5.2}$$

Where T_b, T_f, T_c, and N represent the baseline service time, the fixed time to reply to a request (see §5.3.2), the time required to collect a single write into an ongoing batch, and the number of compacted requests, respectively. Using a software compaction microbenchmark, we measured values for each time parameter that indicate $A \simeq 3$ is attainable in an unloaded system. Furthermore, as read-write contention on hot partitions grows at high loads, the value of T_b will increase while T_c remains unaffected because the writer thread dirties no shared cache lines as it gathers writes into a batch. Our evaluation studies these parameters in §8.4.3.

Figure 5.9 shows the cooperation required between the NIC and the KVS threads to achieve write compaction. The NIC's responsibility is to implement the CREW (or d-CREW) policy, spreading read load among all the threads and creating opportunities to compact dependent writes. When the KVS thread pulls a write request from its private queue, it scans a limited number of extra queue slots to search for any matching writes to the same item (Step (1)). If found, the thread opens a new "compaction window", and collects all incoming writes to this item into a single update until the window closes (Step (2)). When the compaction window closes, the thread performs the single combined write to the KVS' partition (Step (3)) and sends all the responses to the compacted writes (not shown).

The length of the compaction window permitted defines the achievable acceleration—the higher the number of compacted writes N, the greater the value of A. We observe that KVS' tail latency constraint creates an opportunity to finish all the compacted writes "just in time"

before the SLO expires. Even with a stringent SLO of 10× the average service time, it is possible to compact upwards of 10 writes into a single window because the compacting thread is significantly accelerated. However, introducing compaction implies that all the writes are now invisible to the readers, until the window closes and the final update is applied. We now discuss how C-4 maintains strong consistency guarantees in the presence of write compaction.

Maintaining Consistency Under Compaction

We target a baseline KVS providing *linearizability*, a strong consistency guarantee that requires a total write order and that the system's behavior respects the real-time ordering of requests [101]. Note that linearizability is a *local* property, which only applies to single objects (in our context, keys) in isolation. We do not consider consistency across multiple keys in this work for two reasons. First, the vast majority of KVS do not support multi-key updates [84, 162, 165, 166, 176, 200, 272], following the system design principle to "make it fast, rather than general or powerful" [157]. Such tasks are better implemented in higher layers such as transaction lock managers, which have their own set of related but different consistency models. Indeed, attaining atomically visible multi-key updates requires either serializability or *strict* serializability. We argue that because C-4 maintains linearizability, it is an equivalent candidate to an existing KVS for inclusion in a transactional system.

Figure 5.10 displays three possible executions of a KVS with write compaction that demonstrate our key insight necessary to maintain linearizability. Each operation is expressed as $Op(Args)$: $Client$, where Op represents a KVS `get`, `set` or `response`, $Args$ contains the arguments passed to the KVS (e.g., the key being accessed, and its associated value for sets/responses), and $Client$ is the client thread requesting this operation from the KVS.

To argue that C-4 maintains linearizability, we focus on the specific scenario induced by the addition of write compaction—when a compaction window is opened for key K with simultaneous outstanding read requests to K. Informally, a system's execution E can be *linearized* if it can be transformed into E' where two criteria are respected: (1) E' is sequential, meaning that each invocation is immediately followed by its corresponding response, and (2) the order of operations in E' respects all partial orderings in E [101]. A partial ordering

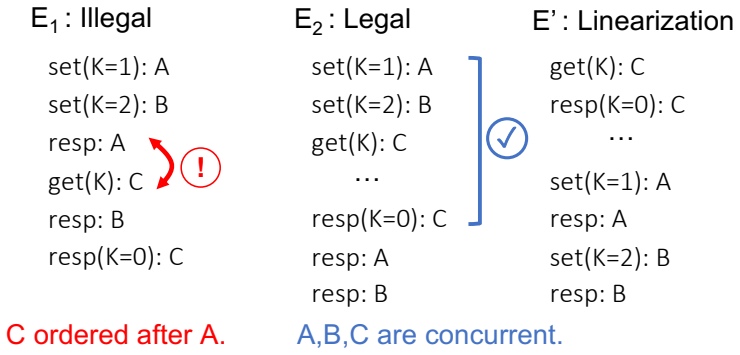


Figure 5.10: Executions showing linearizable compaction.

is created when the response to an operation R_1 appears in the execution E before another operation R_2's invocation – all operations that are not partially ordered are concurrent. Our key insight is criterion (2) allows such concurrent operations to be re-ordered in E', because they are not constrained by orderings emanating from E.

To demonstrate, consider the execution E_1 in Figure 5.10. In this execution, clients A and B both attempt to write key K with two different values, while client C reads K. E_1 is not linearizable because C's `get` could return the initial value $K = 0$ only if it executed *before* both A and B's `sets`; however, in E_1 C's `get` is partially ordered after A's `set`, therefore a linearizable system must return either $K = 1$ or $K = 2$ to C. A naive write compaction implementation can result in the exact violation in E_1. If a thread opens a compaction window for K after A's `set` and responds to A, any future read operations will return the value in the KVS' underlying data store, despite needing to return the value `set` by A.

To preserve linearizability, C-4 delays responding to the writes in a compaction window until it closes, and therefore reads during the compaction window are considered concurrent and can legally return the value in the KVS' data store. To illustrate, E_2 in Figure 5.10 contains the same operations as E_1 with the response to A deferred until the compaction window closes. Therefore, C's `get` is concurrent with both A and B's `sets` and is not required to be ordered after either one. Then E' is a sequential execution meeting both criteria for linearizability: C's `get` executes "first", before A and B's `sets`, which close the compaction window and set $K = 2$. For reads that are ordered before or after a compaction window, C-4's design is no different from the baseline KVS, so we omit these cases for brevity.

Conclusions. Write-intensive key-value workloads are ripe for improved load balancing. Our modelling results indicate that improving KVS throughput and tail latency rests on the ability of the load balancing unit to distinguish writes into those that are independent, and those that must be serialized. Independent writes do not have any fundamental reason preventing them from being balanced across cores, whereas dependent ones can expose additional software optimizations when they are executed on the same core. Following these two insights, we propose the C-4 policy, which addresses independent write balancing with d-CREW and dependent write load with compaction. In a sense, NEBULA can be viewed as a hardware-supported delegation system [175, 228]; the NIC's support for d-CREW determines which threads currently have exclusive access to partitions, and "delegates" writes to those threads via its load balancing mechanism. Software then combines the dependent writes without the overhead of shuffling requests between threads in traditional delegation systems.

5.5 Architectural Extensions for Improved Load Balancing

The fundamental commonality between the two use-cases we have considered and modelled is that static approaches to improving application-level performance (e.g., via permanent write or function partitioning) necessarily trade-off tail latency to achieve their goals. Evidence gathered from our discrete-event simulations indicates that it is indeed possible to break the tradeoff between increasing application locality and sacrificing load balancing flexibility using runtime information. Therefore, we argue that RPC-centric server architectures should contain the requisite NIC and software extensions to enable load balancing policies that collect such information and apply it to target both improved locality, and flexible load balancing. Integrated support for RPC protocol acceleration in hardware is a strong enabler for collecting such application-level information, because after the RPA finishes unpacking RPCs into application-readable format, it is able to extract specific fields of the RPC's arguments and pass them to the load balancing unit.

Both application-level load balancing enhancements we propose can be implemented with the same set of NIC and software extensions. First, both policies need a small amount of dedicated on-NIC storage to track the contents of the application's input queues, enabling the load balancing unit to make decisions based on queue state in addition to queue length.

For dynamic function affinity, the NIC would use this extra storage to contain the history of functions executed by each core. For dynamic write partitioning, the storage contains the list of partition IDs (or application-level keys at finer granularity) to which writes are currently outstanding, and which core is writing them.

Second, the API between the NIC and software needs to be enhanced to support explicit modification of the on-NIC load balancing state via control messages. Such control APIs can be used by applications to inform the NIC's load balancer to free certain entries (e.g., when a core completes a write), or direct the NIC to assign certain cores preferential treatment for certain RPC types. The operating system can also use such an API to indicate a thread has migrated, and existing mappings must be updated.

Unlike the static policies we studied in §5.2.2 and §5.3.2, dynamic function affinity and write partitioning are stateful and thus should be implemented server-side. Although static designs can be trivially implemented wherever the function ID or KVS partition is known (e.g., on the clients themselves [166] or in programmable networking hardware [150, 278]), rapid changes in the set of outstanding writes and per-core function histories would require constantly updating the static mappings at every participating entity. As servers often receive requests from thousands of concurrent clients [198, §3.1], broadcasting such updates is clearly not scalable.

In-network load balancers such as RackSched [278] are an attractive alternative to client-side solutions, because their centralized nature removes the need to broadcast updates. However, the increased distance between the load balancer and server threads implies the load balancer's information about server-side queueing is more stale, because application messages that modify entries in the load balancer's tables must traverse the network. As prior work has explicitly demonstrated that tail latencies increase with load balancer-to-server distance [150], we conclude that dynamic policies will yield superior performance when implemented server-side than in-network designs.

In Figure 5.11, we show the high level architecture of the NIC extensions and software API we propose to realize locality-enhanced load balancing policies. The left side of Figure 5.11 shows the layers an RPC passes through when it arrives from the network. After it exits NEBULA's

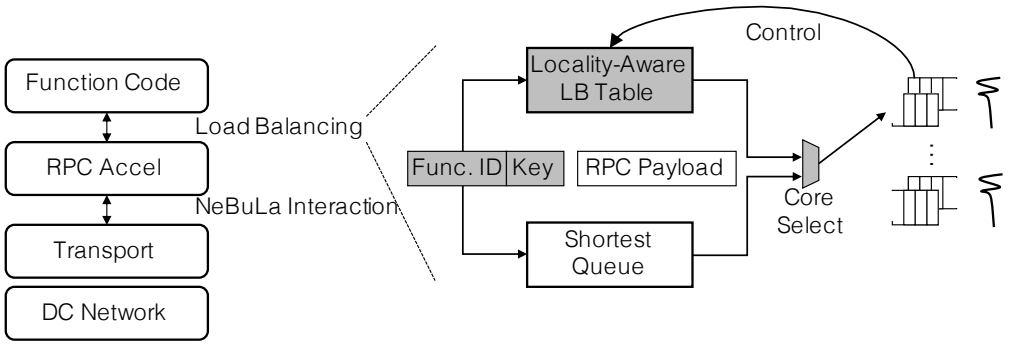


Figure 5.11: Design of NIC extensions and software API for locality-aware load balancing.

transport protocol, it passes through hardware RPC protocol acceleration, and then enters the load balancing stage(s) of the NIC which are expanded and shown in detail. We propose a small additional on-NIC component called the "Locality-Aware Load Balancing Table" (LLBT), which can be configured to store and look up any fields from the RPC's header and return a core (or list of cores) as potential dispatch candidates. These candidates are then filtered with queue length metrics (e.g., the shortest queue among them) before the final core selection is made. Application threads place control messages in their queue pairs to inform the NIC to modify entries in the LLBT (e.g., to free an entry), using the same queues for scheduling network transactions.

To describe how the extensions we propose are used to implement our enhanced load balancing policies, consider the d-CREW policy described in §5.4.1. The NIC stores its set of ephemeral KVS partition mappings in the LLBT, and all load balancing decisions for write requests look up the LLBT using the field in the RPC header corresponding to the KVS' hashed key. When KVS software completes the write, it places a feedback message in its Queue Pair which is processed by the NIC like any other request for a network transaction. This special feedback message indicates to the NIC that this core is relinquishing exclusive access to this partition, and that the mapping in the LLBT is now free to be re-used. A nearly identical process can be performed to implement dynamic function partitioning for multi-function microservices, with the main difference being that the LLBT now stores a FIFO list of function IDs per core, rather than the set of exclusive partitions to implement d-CREW.

We want to expressly mention that we are not the first to propose NIC support for making packet processing and scheduling operations. In particular, pioneering work from the

networking and systems communities has led to the wide adoption of P4-enabled network switches [38, 115], and research prototypes already exist that use such support for RPC load balancing [150, 278] as well as object steering in KVS [139]. If support for primitives like P4 materializes in future integrated NICs, our design could trivially be adapted to use a P4-based implementation. Our contribution is rather to demonstrate that server-side NIC load balancing can deliver increased RPC throughput without sacrificing 99th% latency, and identify two common use-cases in datacenter microservices (i.e., temporal instruction locality and dynamic write partitioning).

5.5.1 Locality-Aware Balancing on SmartNICs

Although we argue for an RPC-centric server with an integrated network interface (see §2.5), the architectural support we propose for locality-aware load balancing could also be implemented on commodity SmartNICs such as NVIDIA's BlueField [202, 204] and Innova [203], or Netronome's Agilio [196]. These products contain various compute units (e.g., ARM cores or programmable FPGA blocks) that can be used to implement JBSQ load balancing, as well as the LLBT and surrounding logic we propose. A successful load balancing implementation on a commodity NIC would be able to improve the throughput of latency-critical services on current servers at much lower deployment cost.

The architecture of today's SmartNICs creates two challenges to realize locality-aware load balancing, both because of the relatively large latency of the host's I/O interconnect when compared to µs-scale RPCs. First, the reduced proximity between the NIC's load balancing modules and the host's CPU cores limits the effectiveness of the load balancing policy itself, because a higher-latency I/O interconnect mandates that more requests must be queued at each core [150, §3.3]. Second, implementing RPC acceleration on a SmartNIC creates repeated and unnecessary µs-scale delays when (de)serializing RPC objects, because each pointer access in such objects incurs the full latency of the I/O interconnect [275, 276]. Emerging technologies such as Intel's Coherent Express Link (CXL) promise an order of magnitude lower latency than commodity PCIe, and would be sufficient to drastically shrink the performance detriment of both challenges.[4] However, at the time of writing, no combination of SmartNIC

[4]Although still PCIe-based, early CXL specifications reportedly require that pin-to-pin transfers for cache-

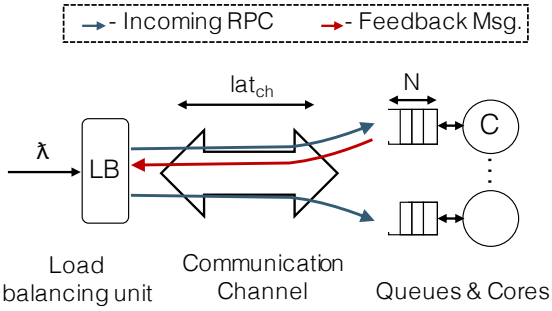


Figure 5.12: Queueing system model to study the impacts of load balancing unit proximity.

and host CPU hardware is available supporting CXL. Therefore, we conduct a study using our discrete-event simulator (see §3.3) to show the limitations and potential of off-chip implementations of locality-aware load balancing. This study *does not* take into account the limitations of performing dependent pointer accesses across an I/O interconnect, which we leave for future work.

Our simulation models a similar system to those in Figures 4.1 and 5.5, with the addition of a finite-latency communication channel between the load balancing module and the worker units. Changing the latency of the communication channel allows us to study the performance of locality-aware load balancing on current SmartNIC platforms as well as future technology developments. This enhanced queueing model is shown in Figure 5.12. Requests arrive according to a Poisson process with configurable arrival rate, and the load balancing unit implements the Join-Bounded-Shortest-Queue (JBSQ) policy [150]. The depth of each worker's private queue (shown as `N` in Figure 5.12) is chosen empirically. We begin with a heuristic value of N proposed by the creators of the JBSQ policy, namely $N = \lceil \frac{2 * lat_{ch}}{\bar{S}} \rceil$ where lat_{ch} represents the one-way channel latency, and $\bar{S}$ is the average service time of the requests. Then, we increase N to the minimum value required to reach the highest attainable system throughput.

We parameterize the model with various communication channel latencies and service time distributions. For channel latencies, we choose the following:

- **Current SmartNIC: 2.5µs.** This configuration represents work done by Humphries et al. that builds a load balancing unit on a Broadcom Stingray SmartNIC [106]. Section 5.1 of

coherence snoops and responses should be less than 50 nanoseconds [50, §13.0]. Such latency improvements are reportedly made possible by carefully engineering lower-overhead transaction and link layers [194].

System	Exponential	Bimodal
Current	15	18
PCIe Bound	8	8
Emerging	2	2
Integrated	2	2

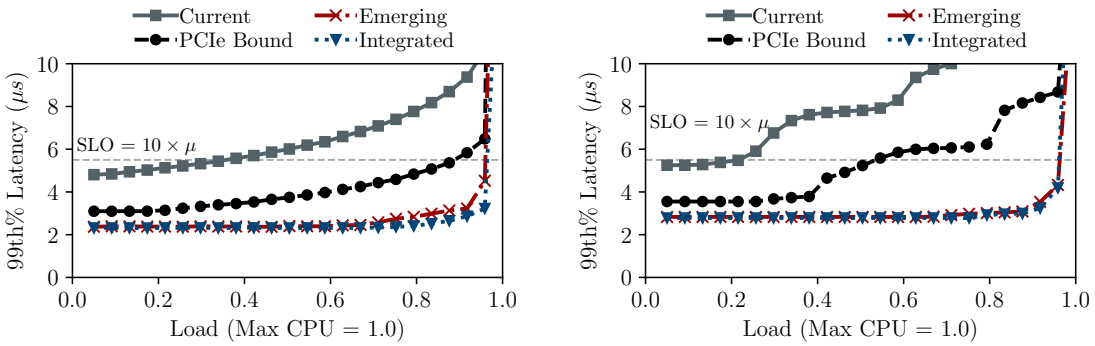
Table 5.2: Values of N in JBSQ(N) for various communication channel latencies and service time distributions.

their work reports the latency between the load balancing unit and workers.

- **PCIe SmartNIC Bound: 800ns.** This configuration represents our expected lower bound that could be attained by a PCIe-based SmartNIC with the required engineering effort to remove the overheads present in current designs. We use the latency reported by Neugebauer et al. in their in-depth study of PCIe-attached network devices [197].
- **Emerging: 20ns.** Represents emerging coherent interconnects (e.g., CXL) that attain drastically lower off-chip transfer latencies than current PCIe-based technology. We choose 20ns to represent a SmartNIC connected using `cxl.cache`, achieving 50ns pin-to-pin latency [50, §13.0], and assume 10ns of that latency is spent on the CPU host to access the coherence directory.
- **Integrated: 15ns.** Represents an integrated design, where the NIC and CPU are part of the same on-chip network. The latency of 15ns is calculated as the average latency of sending a single flit-sized message on a 64-node mesh network, operating at 1GHz.

For this experiment, we model exponential and bimodal service time distributions, both having $\bar{S} = 500ns$. The bimodal distribution is the same as used in §4.2.1, defined as $P[X = \frac{\bar{S}}{2}] = 0.9$, $P[X = 5.5 \times \bar{S}] = 0.1$.

In Table 5.2, we report the ideal value of N for JBSQ from our empirical event simulations, for both distributions. As expected, the integrated and emerging designs have the lowest value of N, where the designs with longer I/O interconnect latency require deeper private queues. Even with an idealized implementation, PCIe interconnect latencies require each worker's request input queue to be 4× longer than with an integrated design. Growing private queues in this manner will inevitably translate to reduced load balancing effectiveness, particularly for service time distributions with higher dispersion. In contrast, emerging technologies should be



(a) Exponential service times.

(b) Bimodal service times.

Figure 5.13: Load vs. latency curves showing performance attainable by load balancing units of various proximities to the workers they serve.

able to match the performance of integrated designs if production latencies meet expectations, because a 20ns interconnect is fast enough to make the load balancer seem "integrated". Such conclusions are in accordance with the emerging trend towards NIC-compute co-design, as significant engineering investment has gone into tighter integration between off-chip and host-side resources, to the point where cache coherence can be offered between off- and on-chip memory.

Figure 5.13 shows the throughput attainable under SLO for the various I/O interconnect technologies, for both service time distributions, using the values of N reported in Table 5.2. The tail latency of the "Current" configuration representing today's SmartNIC products degrades quickly because of the batching required to amortize the lengthy round-trip times between the load balancer and workers. Our model predicts that such devices could only attain 0.34 and 0.21 of the maximum theoretical load for exponential and bimodal service times under a 10× SLO. In contrast, the "PCIe Bound" system performs better, due to its roughly 3× improvement in load balancer-worker communication latency. The increased proximity roughly halves the queue depth needed for maximum throughput, allowing the PCIe Bound configuration to attain loads of 0.87 and 0.5, respectively.

For exponentially-distributed service times, Figure 5.13a shows that Emerging and Integrated designs provide only 9% higher load under SLO than the PCIe Bound system, because the exponential distribution does not have a heavy tail. The Emerging and Integrated designs are still able to provide 2.5× reduced 99th percentile latency at a load of 0.87 compared to

the PCIe Bound system. In the case of the bimodal distribution, which has a much heavier tail, the increased flexibility of the two designs with shorter input queues becomes apparent. Specifically, Figure 5.13b shows these two designs provide 46% higher throughput under SLO than PCIe Bound. Finally, the Emerging and Integrated designs deliver SLO-compliant loads of up to 0.96 for both service time distributions, because they only require a private queue of 2 slots per worker.

Our study shows the importance of designing an RPC-centric server architecture that can run the shortest input queues possible when SLO-constrained performance is of primary concern. Although integrated designs deliver the best performance, the latency reduction of emerging cache-coherent interconnects like CXL would make them performance-competitive for load balancing purposes, assuming the requisite engineering effort to attain the same level of flexibility as integrated designs. However, a lower-latency emerging interconnect would not transparently solve the throughput challenges inherent in (de)serializing pointer-based RPC objects, as each data access would still incur an overhead roughly comparable to a full off-chip memory read. The combination of these two challenges would render a SmartNIC-based implementation of RPC protocol acceleration and load balancing a very impactful result if attained. For this thesis, we chose to target an integrated load balancing system because of the clear limitations of today's designs shown in Figure 5.13, and because integration many synergies with the extensions for in-cache network traffic management in Chapter 4.

5.6 Chapter Summary

In this chapter, we have presented our case for enhancing RPC-centric server architectures with architectural support for realizing application locality-aware load balancing policies. We first showed two independent opportunities for improving RPC performance via improved load balancing – instruction cache locality in microservices with multiple underlying functionalities, and improved concurrency control in microservices storing data. We then argued that both scenarios in fact resolve to the same underlying tradeoff between improving throughput with permanent partitioning, or maintaining load balancing flexibility. Finally, we demonstrated that it is possible to preserve load balancing and improve throughput for both scenarios, with the addition of a single on-NIC component to track the state of each core's RPC

queues, and the software feedback messages to modify and free entries in this table. Using discrete-event simulations, we argued that an integrated load balancing module will provide significantly better throughput under tail latency constraints than today's SmartNIC-based implementations. Accordingly, Chapter 8 demonstrates a concrete implementation of the NIC extensions to enable locality-enhanced load balancing, and evaluates the performance of an RPC-centric server architecture in both scenarios that we identified in this chapter.

Part II Implementation and Evaluation of an RPC-Centric Server

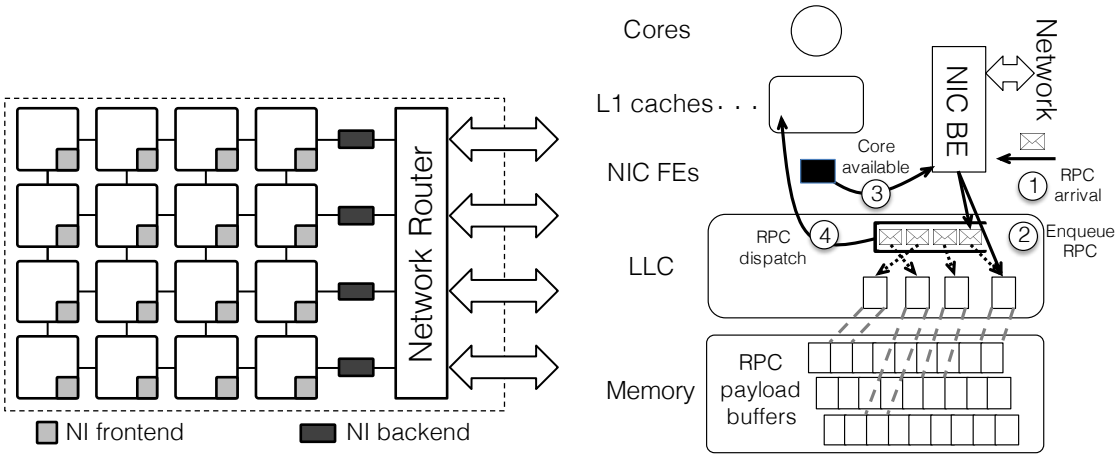
6 Building NeBuLa on Scale-Out NUMA

In Chapter 4, we introduced the NeBuLa architecture, a set of architectural extensions to a hardware-terminated protocol stack that ensure RPC traffic is served out of the CPU's caches rather than spilling into DRAM. This chapter describes a concrete implementation of NeBuLa, beginning with our architectural baseline and proceeding to describe our modifications to the software's RPC layer, and additions of a hardware buffer manager, RPC reassembler, and dispatch pipeline for L1 cache steering, respectively.

6.1 Baseline Architecture

We use Scale-Out NUMA (soNUMA) [199] as our baseline architecture. soNUMA combines a lean user-level, hardware-terminated protocol with on-chip NIC integration and a high-performance network fabric to provide rapid remote memory accesses. Applications schedule soNUMA operations (e.g., `send`) using a VIA-like memory-mapped Queue Pair (QP) interface. Although we target soNUMA as our baseline in this thesis, we emphasize that our set of design extensions are not directly tied to soNUMA, and could also be implemented on a variety of other architectures that feature tightly integrated network interfaces. Academic examples include the NanoPU [108] and the FAME-1 RISC-V RocketChip SoC [137]; commercial examples include Intel's Xeon-D servers with integrated Ethernet controllers [121].

To use soNUMA in the context of a modern server chip featuring many cores, we make use of the "Manycore Network Interface" NI_{split} implementation proposed by Daglis et al. [52, 53].



(a) A Manycore Network Interface-equipped server.

(b) Operation of the RPCValet system.

Figure 6.1: The baseline transport and load balancing architecture for NEBULA.

Figure 6.1a shows a server equipped with the NI_{split} architecture, which divides the server's NIC into multiple discrete entities, called NIC frontends (FEs) and backends (BEs). The former handles the control plane (i.e., QP interactions) and is co-located with each core; the latter handles the network packets and data transfers between the network and memory hierarchy and is located at the chip's edge. Being that soNUMA is an architecture expressly designed for remote memory accesses and not RPC-like communication, it does not meet either design prerequisite introduced in §4.2 – its transport protocol handles independent cache block read/writes, and it does not support load balancing.

The RPCValet [54] system is an extension of soNUMA that rectifies both issues – it enables native messaging support using the same integrated NIC as soNUMA, and also supports NIC-driven load balancing. RPCValet balances incoming RPCs across the cores of a multi-core server in a single-queue, synchronization-free fashion by coordinating the NIC FEs and BEs. Figure 6.1b provides a high-level demonstration of its operation. When a new RPC arrives (1) the NIC BE writes its payload in the LLC, creates an RPC arrival notification (2)—which contains a pointer to the RPC's payload—and stores it in a dedicated queue which is processed in FIFO order. As soon as a core becomes available to process a new RPC, its corresponding NIC FE notifies the NIC BE (3), which, in turn, dequeues the first entry from the arrival notification queue and writes it in the core's Completion Queue (CQ) (4). The core receives the RPC arrival

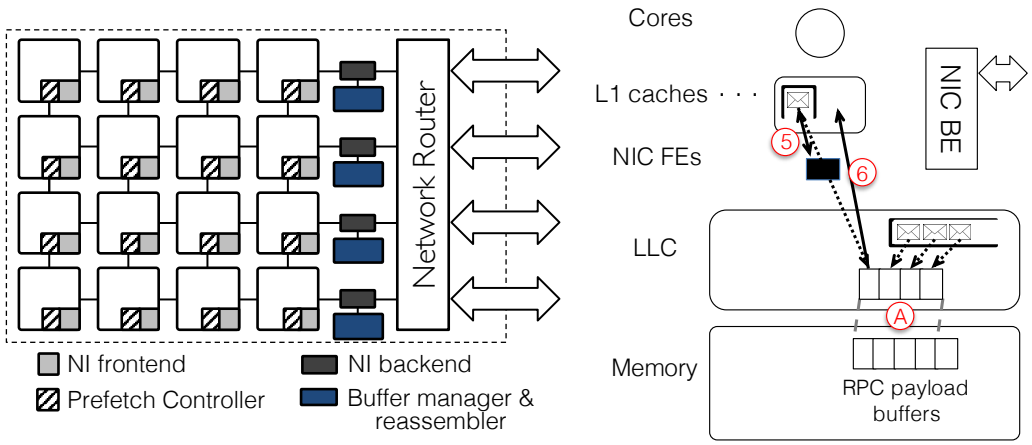
notification by polling its CQ, and then follows the pointer in the notification message to read the RPC's payload from the LLC. This greedy load assignment policy corresponds to single-queue behaviour.

As discussed in §4.1, the use of hardware-terminated stacks either implies memory bandwidth interference or load imbalance – and the RPCValet system shares this restriction. When RPCValet creates a software "messaging domain", the system provisions enough messaging slots to cover the NIC's full duplex bandwidth *for every other participating node*. Concretely, a 1024-node system with a maximum message size of 512B implies 136MB of buffer allocations, outstripping all but the largest LLCs at the time of writing (AMD's recently launched 3D V-Cache could contain buffers of this size, having an initial claimed capacity of 768MB) [3]. We now present our implementation of NEBULA's architectural extensions to shrink unnecessary server-side buffering, and break the tradeoff between bandwidth interference and load imbalance.

6.2 Architectural Additions

With the baseline NI_{split} architecture, NEBULA's Buffer Manager and Reassembler must be co-located with the NIC BEs because they are data plane components, involved with RPC payloads themselves. The Prefetch Controller for NIC-to-core payload steering is co-located with the FEs, which is logical because it interacts with each core's L1 cache to bring RPC payloads into it. In Figure 6.2a, we show our implementation of NEBULA's requisite NIC extensions on top of the NI_{split} baseline. It is immediately apparent that the logically centralized components previously introduced in §4.3 are now physically distributed due to the arrangement of the components making up the baseline NI_{split} architecture. We discuss the implications of distributing each component when their implementations are introduced in the upcoming sections.

Figure 6.2b highlights NEBULA's operational modifications in red. The first operational change, marked as (A) and detailed in §6.4 and §6.5, is NEBULA's in-LLC network buffer management. Steps (1)-(3) occur identically to the baseline shown in Figure 6.1b, with the modification that the RPC's buffer mapping is created by the combination of the Buffer Manager and



(a) Implementation of NEBULA on a Split-NI.

(b) NEBULA protocol operation.

Figure 6.2: Location of implemented NEBULA extensions, and updated system operation.

Reassembler. The assigned buffers are nearly guaranteed to reside in the LLC because of two factors. First is their drastic size reduction, which is visible in Figure 6.2b as a reduced number of in-memory buffers. Second is their increased reuse because an RPC's assigned buffer is no longer a function of the client's connection ID, leading to less interleaving – this factor is visible in Figure 6.2b as sequential RPC buffer mappings.

The second operational change is NEBULA's NIC-to-core RPC steering, which extends Step (4) in the baseline protocol's operation.[1] NEBULA extends the sequence of RPC arrival notification actions with payload steering, shown in Steps (5)-(6) and detailed in §6.6. Both steps are accomplished by the Prefetch Controller component, integrated with NI_{split}'s NIC FE.

6.3 RPC Software Interface

We implement an RPC layer on top of soNUMA's `send` operation, maintaining RPCValet's messaging interface [54]: all nodes of the same service join the same messaging domain, which includes the buffers and data structures defining where incoming RPCs are placed in memory. Every node participating in a messaging domain allocates a receive buffer in its memory to hold the payloads of incoming RPCs. We change RPCValet's underlying connection-oriented buffer management to achieve NEBULA's key goal of handling all traffic within the

[1] Numerical references in Figure 6.2b are extended from Figure 6.1b.

```
while (true):
    payload_ptr = wait_for_RPC(msg_domain)
    //process the received RPC and build response...

    free_buffer(buffer_ptr, buffer_size)

    RPC_send(resp_buffer_ptr, buffer_size,
                   target_node, msg_domain)
```

Figure 6.3: Pseudocode of an RPC-handling event loop.

server's caches. In NEBULA's case, after the receive buffers are allocated and initialized by software, they are managed by the NIC. We first focus on the software interface and detail hardware modifications later in this section.

Figure 6.3 demonstrates the three functions the NEBULA RPC interface exposes to applications within a sample RPC-handling loop. A server thread waits for incoming RPCs using the `wait_for_RPC` function, which can be blocking or non-blocking. The NIC sends RPC arrivals to this thread via the thread's CQ that is associated with the incoming RPC's messaging domain. After completing the RPC, the application invokes the `free_buffer` function to let the NIC reclaim the buffer. Finally, the application sends a response in the form of a new RPC, specifying the messaging domain, target node, and local memory buffer that contains the outgoing message. `RPC_send` has a return value indicating whether the outgoing RPC was accepted by the remote end or not, which only clients use to check whether their requests are NACKed by the server. In a well-designed system, the server does not use the return value, as clients should always provision sufficient buffering for responses to their own outstanding requests.

soNUMA acknowledges all messages in its hardware-terminated transport layer. NEBULA extends this mechanism with negative acknowledgements (NACKs), which are responses to `send` operations if the receiving end cannot accommodate the incoming RPC. In response to a NACK reception, the application layer receives an error code. The most appropriate reaction to an error code is application-specific, and can range from simple retry of the same `send` at a later time, to arbitrarily sophisticated policies.

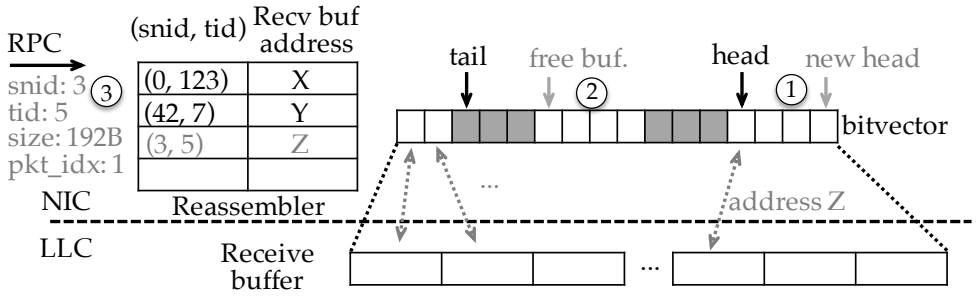


Figure 6.4: NIC RPC reassembly and buffer management.

6.4 NIC Extensions for Buffer Management

NEBULA's NIC manages a receive buffer per established messaging domain. After the software sets up and registers these buffers with the NIC, the NIC must dynamically allocate them to incoming RPCs, and place incoming packets appropriately. The buffer manager's role is to reserve enough space in the receive buffers for each RPC, ensuring each core has zero-copy access to the buffer until the application explicitly permits the NIC to reclaim it after the RPC is completed. As the allocator must operate at line rate, simplicity is paramount. The NIC therefore uses a chunk allocation algorithm that manages each receive buffer as an array of consecutive slots in host memory. In allocators of this style, performing a new allocation is as simple as returning the next available slot(s). Figure 6.4 shows the functionality and implementation of NEBULA's buffer manager component, which is replicated at each NIC BE.

The buffer manager uses a bit-vector built into the NIC, containing a bit for every cache block-sized (64B) slot in the receive buffer. Bits corresponding to allocated slots are set to 1 (shaded in Figure 6.4). Upon a new RPC arrival, the NIC advances the receive buffer's head pointer by $\lceil RPC\_size/64B \rceil$ slots and sets the corresponding bits in the bit-vector. In Step (1) of Figure 6.4, the arrival of a new RPC with $128B < size \leq 192B$ would trigger the `head` pointer to advance three slots to `new head`. As this RPC's size is greater than one cache block and will be fragmented into multiple packets, it also adds an entry to the reassembler. We defer description of the reassembler to later in §6.5.

Applications free slots by sending a `free_buffer` message specifying a buffer address and length to the NIC (see §6.3), which resets the bit-vector's corresponding bits (Step (2)). After

each `free_buffer` message, the buffer manager checks whether the tail pointer can advance, thus reclaiming receive buffer space. In Figure 6.4's example, the tail cannot advance because the slot(s) immediately in front of `tail` are still allocated. If the head pointer reaches the tail pointer, the receive buffer is full and the NIC NACKs any new RPCs.

As receive slots are freed out of order by the applications, a naive implementation can suffer from internal fragmentation and thus excess NACKs in the case of a rare long-latency event (e.g., the OS quantum expires while handling an RPC). We nevertheless expect chunk allocation to still function well in the common case, as queueing theory proves that the tail pointer (used to free slots) will advance at least as quickly *on average* as the head (used to allocate). This is because that in a stable system, the average arrival rate of new RPCs must be less than the maximum service rate of the cores. To cover transient events where that relationship temporarily reverses itself, we implement *scrubbing* logic, where the hardware pointers quickly skip through the bit-vector to find a contiguous available range of slots. Scrubbing is triggered when the head meets the tail pointer and is performed off the critical path of new allocation and freeing. If any allocation or free message operations modify the bit-vector during scrubbing, it is cancelled and restarted.

To size the receive buffer (and thus the bit-vector), we rely on our analysis in §4.2.1. We provision for $10 \times E[N_q]$ queued RPCs, which conveniently covers the 99% depth of even the bimodal distribution considered in Table 4.1. Factoring in the RPC size as well, we size the receive buffer to $10 \times E[N_q] \times avg\_RPC\_size$ bytes. As per §4.2.1's example, assuming a 64-core server at load $A = 63$ and an average RPC size of 1KB, our provisioning results in a 540KB buffer. This choice results in a 1KB bit-vector, a modest SRAM cost for a large server chip.

The above explanations have assumed a buffer manager which is logically centralized, and covers the entire set of receive buffers that are allocated by the software. However, because the buffer managers are distributed across the chip's many NIC BEs, it is necessary to divide the receive buffer space between them, and ensure that all packets making up an RPC arrive at the same NIC BE. Although it may appear that such division raises concerns about unequal distribution of incoming traffic across the BEs that could lead to excess NACKs, the underlying

soNUMA communication protocol allows us to alleviate such issues by ensuring a uniform spread of incoming RPCs across NIC BEs. Hence, the many distributed buffer managers and reassemblers will have uniform number of RPCs, ensuring that the allocated receive buffer space is well occupied *in the average case*.

In the original NI_{split} implementation of the soNUMA protocol, incoming transfers are unrolled into cache line-sized requests, and statically address-interleaved across the NIC BEs. This is done because the target of these transfers is *local memory* and hence the goal is to minimize the number of on-chip hops for each cache-block sized request to reach its home LLC tile [53]. The NI_{split} design accomplishes its interleaving by choosing a set of bits in the cache line address contained in the incoming request, and then steering the incoming request to the NIC BE which is nearby to the matching LLC tile. In contrast, the target of RPC traffic is a *CPU core*, and the primary role of the NIC BE is to reassemble the RPC so it can then be steered into that core's L1 cache.

To ensure that all packets of an RPC arrive at the same NIC BE, we use the RPC clients – when a client sends an RPC, it only specifies which *node* of the system the RPC is for, and the NIC BE can be selected by the protocol controller. Therefore, we implement a simple random-BE selection policy at all clients for all RPC traffic being sent over the soNUMA transport protocol. As RPC packets exit the network router, we modify the existing hardware logic that interleaves memory requests across the BEs, to steer all traffic identified as comprising an RPC to the specific BE indicated by the client.

6.5 NIC Extensions for RPC Reassembly

Incoming RPCs exceeding the network MTU in size are fragmented at the transport layer; thus, they must be reassembled before being handed to the application. In a hardware-terminated protocol, such reassembly has to be performed in the NIC's hardware. The specific challenges for designing such reassembly hardware for NEBULA are the baseline architecture's small MTU (64B) and the NIC being an on-chip component. The former implies high reassembly throughput requirements (e.g., every ~ 6ns for a 100Gbps endpoint), the latter implies tight area and power budgets.

In many emerging transport protocols for RPCs, all the packets of the same message carry a unique identifier (`tid`) assigned by the originating node [54, 150, 193]. Thus, an incoming packet's combination of `tid` and source node ID (`snid`) uniquely identifies the message the packet belongs to. The reassembly operation can be described as performing an exact match between the (`snid,tid`) pair found in each incoming packet's header, and an SRAM-resident "database" of all RPCs that are currently being reassembled. Returning to Figure 6.4's example, assume that the second packet of the RPC which previously arrived in Step (1) reaches the NIC in Step (3). The packet's header contains the pair (`3,5`), which is looked up in the reassembler and receive buffer address `Z` is returned. Being the second packet of this RPC (`pkt_idx=1`), the NIC writes the payload to address `Z+64`.

The most common solution for exact matching at high throughput is to use CAMs [261], which are power-hungry due to the large number of wires that must be charged and discharged each cycle. Contrary to our initial expectation, deploying a CAM is a feasible solution. Just as NEBULA's design decision to bound the queue depth of incoming RPCs shrinks receive buffer provisioning requirements, it also sets an upper limit for the number of incoming RPCs that may be under reassembly. Consequently, NEBULA sets an upper size limit on the CAM required for RPC reassembly purposes. With §6.4's 64-core configuration as an example, NEBULA requires deploying 8 CAMs (one at each NIC BE) with $\frac{10\times E[N_q]}{8} = 78$ entries each. Each entry contains a 58 bit receive buffer address and a single 8 bit counter to indicate the number of packets remaining to be reassembled in this RPC.[2] The tags are (`snid,tid`) pairs, which are 20 bits wide each.

To model the hardware cost of these CAMs, we use CACTI 6.5 [164] and configure it to model built-in ITRS-HP device projections, a 22nm process, and dynamic power optimization with the constraint of meeting a 2GHz cycle time. We choose such an aggressive cycle time to target a futuristic 1Tbps network endpoint—i.e., a packet arrival every ~ 0.5ns and therefore a potential reassembly every $1ns$. Each CAM has a reported dynamic power of 4.08mW, a leakage power of 0.05mW and an SRAM area of $3920\mu m^2$. In total, this translates to a power overhead of 0.012% for a 64-core server chip whose power draw reaches 280W, like AMD's EPYC 7763 [94]. We therefore conclude that the power of using CAMs does not seem prohibitive.

[2] We use 58 bits rather than 64 bits because the reassembler always returns cache-block aligned addresses.

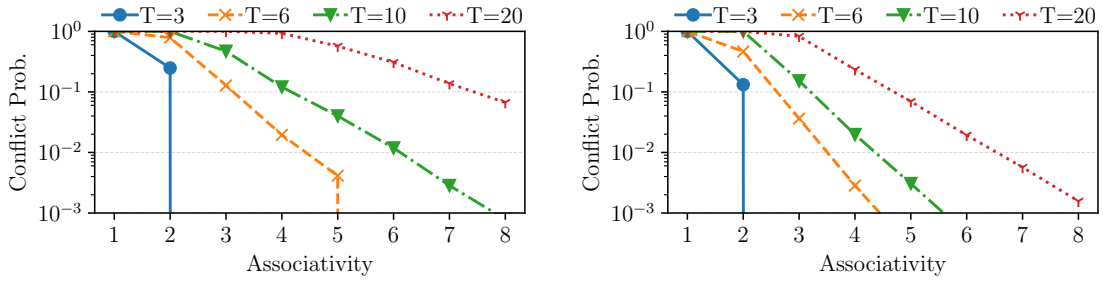
6.5.1 Reduced Associativity

Due to the end of Dennard scaling [62] circa 2005 [37, 71], and more recently the impending end of transistor density scaling [57], CMOS technology projections are an increasingly uncertain domain. It is therefore possible that the power drain of NEBULA's reassembly CAM would be significantly higher than forecasted by CACTI's technology assumptions. We therefore investigate the use of a set-associative design as an alternative design for NEBULA's reassembler.

The tradeoff of using a set-associative reassembler is that it introduces the potential for *conflict misses* when enough RPCs under reassembly map to the same set. When a set is full, RPCs whose `(snid,tid)` pair maps to that set cannot be accepted and must be NACKed. Evidently this is an undesirable scenario, because the server may now reject RPCs even in the absence of queueing. Despite this possibility, we argue it is still possible to attain the same performance as a CAM-based reassembler as long as the probability of a conflict falls below the application's desired SLO percentile.

We conduct a simple mathematical analysis to show the required associativity to bound the probability of reassembly conflict, given a set-associative structure with S sets, W ways, and T concurrently outstanding RPCs requiring reassembly. A conflict occurs when any one of the sets has more than W RPCs mapped to it. This problem statement is identical to a foundational question in computing, the "balls into bins" formulation, which has been extensively studied by theoreticians. To calculate the probability of a conflict, we must analyze the probability of any one of S total bins containing more than W balls, given T throws. Note that this analysis applies to all the reassemblers equally, because of the uniform distribution of requests between them.

First, let X_i be a random variable denoting the number of balls in a single bin i. It is evident that X has a binomial distribution, and so $X_i = Bin\left(T; \frac{1}{S}\right)$. Therefore, the probability of having *no conflict* in the ith bin is given by $P(X_i \leq W)$. To generalize $P(X_i \leq W)$ to the reassembler's many sets, we apply a known technique for "balls into bins" problems known as the Poisson approximation [192]. Under the Poisson approximation, the distribution of X_i is given by a Poisson distribution with $X_i \sim Pois(T/S)$, and each of the i bins are independent even though



(a) 64-entry reassemblers.

(b) 128-entry reassemblers.

Figure 6.5: Relationship between reassembler associativity and upper bound on RPC conflict probability.

the random variables $\{X_0, \ldots, X_S\}$ are stochastically related through the constraint that the sum of all balls must add up to T. With this assumption of independence, we have the probability of conflict P_c in any bin as unity minus the probability of all bins having no conflict:

$$P_c = 1 - \prod_{i=0}^{S} P\left(X_i \leq W\right) \tag{6.1}$$

The power of the Poisson approximation is that any event P_c in the case of independent bins has an upper-bound probability of $Q \leq 2 * P_c$ in the real stochastic system [192], when the event P_c has monotonically increasing probability with T. This approximation is extremely useful for designing NEBULA's reassembler, as we can calculate Q as an upper bound on the conflict probability and know that the real conflict probability will be smaller than Q in the deployed system.

In Figure 6.5a, we plot the value of Q for various values of W and T, with the invariant that each configuration has a total of $S \times W = 64$ entries. Note that we allow values of W that are not powers of two in this theoretical analysis for clarity of the results. Ensuring that reassembly conflicts do not impact an application-level SLO at the 99th percentile is equivalent to ensuring that the conflict probability is no greater than 10^{-2}. To select the values of T that are of interest, it is simple to apply Little's Law to determine the number of concurrently reassembled RPCs. Assuming a futuristic 1Tbps NIC, the RPC arrival rate at each of NEBULA's reassemblers ranges

from 15.6MRPS with 1KB RPCs, to 128MRPS with 128B RPCs.[3] Using a reassembly time of 100ns, this gives an approximate range for T from 2-20.

For small values of T, we note that the associativity requirements are modest. With $T = 6$, using a 5-way reassembler is sufficient to meet the application's SLO guarantee, whereas with $T = 10$, the associativity requirements grow to seven ways. Where all RPCs are minimum-sized giving $T = 20$, the reassembler needs more than eight ways to reach a conflict probability of 10^{-2}. As a real server will encounter a mixture of RPC sizes, we believe that the results with $T = 10$ are reasonable even for a 1Tbps network. Depending once again on technology projections and available manufacturing technologies, associativity may need to be reduced further. In this case, over-provisioning the reassembler by increasing S effectively trades off area for associativity requirements. Figure 6.5b plots Q for $S \times W = 128$, corresponding to a 2× overprovisioning. In this case, seven ways are required to grant a conflict probability upper bound of 10^{-2} in the most demanding case with $T = 20$, and only five ways for $T = 10$. Finally, we remind the reader that our analytical results are derived using an upper bound, and a real deployed system will have more relaxed requirements.

6.6 NIC-to-Core RPC Steering

RPCValet's mechanism to assign RPCs to cores involves metadata only: the NIC places a pointer to the buffer containing the next RPC's payload in the CQ of the thread that will service that RPC. NEBULA extends this mechanism to also trigger a dispatch of the RPC's payload to the target core's L1 cache. If this payload dispatch completes in a timely fashion, it reduces the RPC's execution time. The accuracy of such RPC payload prefetching is not probabilistic, as it is *based on prescience rather than prediction*: the hardware leverages the semantic information of an RPC arrival and assignment to a core to trigger payload prefetches in the correct L1 cache, accelerating RPC startup.

NEBULA's NIC-to-core steering mechanism is implemented as a sequence of additional steps after RPCValet's normal operation. First, we modify RPCValet's RPC arrival notifications that are shown in Figure 6.1b Step (4): in addition to the pointer to the RPC's payload, the

[3] NB: RPCs of sizes less than or equal to the MTU of 64B do not require reassembly.

notification also includes the payload's size. As soon as a new RPC is assigned to a core for processing, its NIC FE reads the payload buffer's base address and size from the notification message and forwards them to the core's L1 cache controller as a prefetch hint (Figure 6.2b, Step (5)). The cache controller uses this information to prefetch the whole RPC payload before the core starts processing the RPC (Figure 6.2b, Step (6)).

For RPC-to-core steering to successfully accelerate RPC startup, the payload prefetches must reach the correct L1 cache in a *timely* fashion. This timeliness requirement necessitates some overlap time between the process of assigning an RPC to a core, and the core beginning its processing. In order to ensure that L1 prefetches arrive in time, we configure NEBULA to allow two RPCs instead of one to be queued at the NIC FE, thus giving the cache controller ample time to prefetch the second RPC's payload while the first one is being processed by the core. Such slight deviation from true single-queue RPC balancing corresponds to the JBSQ(2) policy proposed by Kogias et al. [150], and has been shown to preserve tail latency in an on-chip setting [54].

Summary. Implementing NEBULA requires the following targeted changes to the RPCValet system: (i) software support for handling NACK'ed messages, (ii) a buffer manager which distributes incoming receive buffers to RPCs, and (iii) a packet reassembler which handles data placement for RPCs whose packets have been fragmented by the network layer. Our implementation realizes the buffer manager with a simple chunk allocator in hardware. For the reassembly unit, we use a CAM and surrounding hardware logic; our implementation has limited power drain thanks to our observation that shows the number of concurrent RPCs is bounded by the SLO (§4.2.1). Next, we evaluate the NEBULA architecture.

6.7 Methodology

System organization. To evaluate the NEBULA system, we simulate one ARMv8 server running Ubuntu Linux in full-system, cycle-level detail, using the QFlex cycle-accurate simulator [216]. As introduced in §3.3, QFlex is a fork of QEMU [34, 51], which enhances its emulation capabilities with the cycle-level timing models of the Flexus simulator [265]. Table 6.1 summarizes the simulation parameters we used. We chose our server's LLC, DRAM, and NIC parameters to

Cores	ARM Cortex-A57; 64-bit, 2GHz, OoO, TSO 3-wide dispatch/retirement, 128-entry ROB
L1 Caches	32KB 2-way L1d, 48KB 3-way L1i, 64B blocks 2 ports, 32 MSHRs, 3-cycle latency (tag+data)
LLC	Shared block-interleaved NUCA, 16MB total 16-way, 1 bank/tile, 6-cycle latency
Coherence	Directory-based Non-Inclusive MESI
Memory	45ns latency, 2×15.5GBps DDR4
Interconnect	2D mesh, 16B links, 3 cycles/hop

Table 6.1: Parameters used for cycle-accurate simulation of NEBULA on QFlex.

follow the state-of-the-art system tuned for in-memory key-value serving [165], which provisions 118GB/s of DRAM bandwidth, a 300Gbps NIC, and a 60MB LLC for 60 CPU cores. To make simulation practical, we scale the system down to a 16-core CPU, maintaining the same LLC size and DRAM bandwidth per core; thus, we provision 31GB/s of DRAM bandwidth and a 16MB LLC. Commensurately scaling the NIC bandwidth indicates provisioning an 80Gbps NIC. However, we found that under the most bandwidth-intensive workloads, NEBULA could saturate the full 80Gbps while still having ~15% idle CPU cycles; therefore, we increased NIC bandwidth to 120Gbps to permit NEBULA to reach the CPU cores' saturation point.

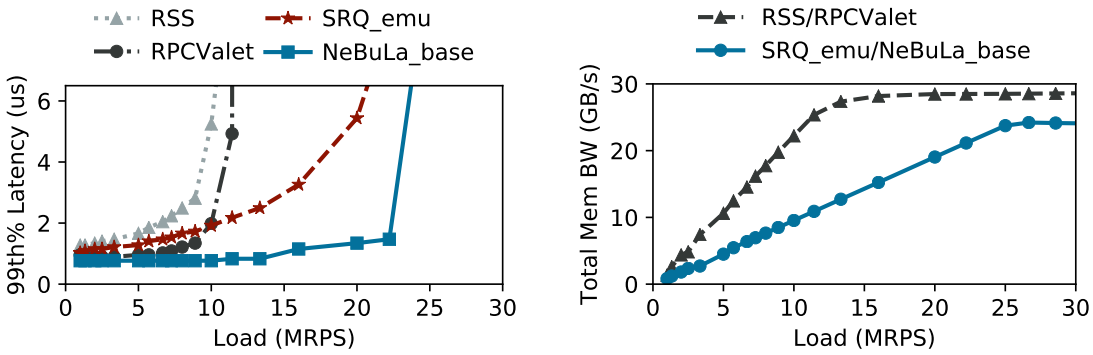
Application software. We use the MICA in-memory key-value store [166] with the following modifications: (i) we ported its networking layer to soNUMA, (ii) for compatibility reasons with our simulator, we ported the x86-optimized MICA to ARMv8. We deploy a 16-thread MICA instance with a 819MB dataset, comprising 1.6M 16B/512B key/value pairs. We use the default MICA hash bucket count (2M) and circular log size (4GB). Larger datasets would further reduce the LLC hit ratio, exacerbating the memory bandwidth interference problem that NEBULA alleviates.

We build a load generator into our simulator, which generates client requests at configurable rates using a Poisson arrival process, uniform data access popularity, and the following GET/SET query mixtures: 0/100, 50/50, 95/5. Unless explicitly mentioned, results are based on the 50/50 query mixture.

Evaluated configurations. We evaluate five different designs to dissect NEBULA's benefits:

- **RPCValet:** We use RPCValet [54] as a baseline architecture, which features NIC-driven single-queue load balancing, optimized for tail latency. RPCValet provisions its packet buffers in a static connection-oriented fashion, resulting in buffer bloat with high connection counts. Assuming a cluster size of 1024 servers, a maximum per-message size of 512B, and 256 outstanding messages per node pair, RPCValet's buffers consume 136MB, significantly exceeding the server's LLC size. This provisioning represents any RPC system that allocates buffer space per endpoint, such as those using RDMA [131] and UDP [132].
- **RSS:** A representative of the Receive Side Scaling (RSS) [189] mechanism available in modern NICs. Our implementation optimistically spreads requests to cores uniformly. Like RPCValet, RSS suffers from buffer bloat, and also suffers from load imbalance because it is a multi-queue system.
- **NEBULA $_{base}$:** This configuration retains RPCValet's load-balancing capability and adds NEBULA's protocol and hardware extensions for in-cache buffer management. Following our analysis in §6.4, NEBULA $_{base}$ allocates 81KB of buffering for incoming RPCs, corresponding to $10 \times E[N_q] = 160$ slots of 512B each.
- **SRQ$_{emu}$:** A proxy for InfiniBand's Shared Receive Queue (SRQ) mechanism, enabling buffer sharing among endpoints (threads) to tackle the buffer bloat problem. We optimistically assume the same hardware-based buffer manager as NEBULA $_{base}$ without any software overheads normally involved when the threads post free buffers to the SRQ. Therefore, our evaluation over-estimates the performance of SRQ. Unlike NEBULA $_{base}$, existing SRQ implementations do not feature hardware support for load balancing. Hence, SRQ$_{emu}$ represents an RSS system without buffer bloat, or, equivalently, a NEBULA $_{base}$ system without hardware support for single-queue load balancing.
- **NEBULA:** The full set of our proposed features, namely NEBULA $_{base}$ plus NIC-to-core RPC steering.

Evaluation metrics. We evaluate NEBULA's benefits in terms of throughput under SLO. Our SLO is a 99th% latency target of 10× the average RPC service time [223]. All of our measurements are server-side: each RPC's latency measurement begins as soon it is received by the NIC, and ends the moment its buffers are freed by a core after completing the RPC. As NEBULA



(a) 99th percentile latency.

(b) Total memory bandwidth.

Figure 6.6: Tail latency and bandwidth for all evaluated systems, using a 50/50 GET/SET query mixture.

can NACK incoming RPCs under high load, we conservatively count NACKs as ∞ latency in our final calculation.

6.8 Evaluation

6.8.1 Removing Bandwidth Interference

We start by evaluating §6.7's first four designs to quantify the impacts of load imbalance and memory bandwidth interference. Figure 6.6 shows 99th% latency and memory bandwidth as a function of load. Figure 6.6b groups the series for RSS/RPCValet and SRQ$_{emu}$/NEBULA$_{base}$ together, because they are effectively identical. RSS performs the worst because it suffers from both load imbalance and bandwidth contention. Although RPCValet delivers 2.6× lower 99th% latency than RSS at 10MRPS due to superior load balancing, both systems saturate beyond 11.4MRPS.

Figure 6.6b sheds light on the source of the performance gap between SRQ$_{emu}$ and the two systems suffering from buffer bloat. RSS and RPCValet utilize 25.3GB/s of memory bandwidth at 11.4MRPS, greater than 80% of the server's maximum of 31GB/s. In contrast, SRQ$_{emu}$ consumes less than 75% of that bandwidth at 20MRPS and therefore delivers 1.75× higher load than RSS/RPCValet, corroborating our claim that memory bandwidth contention can negatively impact latency. SRQ$_{emu}$'s performance, combined with the small difference be-

GET/SET Mix	RPCValet MRPS	RPCValet BW@SLO	NEBULA MRPS	NEBULA BW@SLO
0/100	11.4	24.5 GB/s	26.7	22.6 GB/s
50/50	11.4	25.3 GB/s	22.2	21.1 GB/s
95/5	11.4	24.8 GB/s	22.2	20.9 GB/s

Table 6.2: Sensitivity to various MICA GET/SET query mixtures.

tween RPCValet and RSS, may seem to suggest that load balancing is unimportant. However, we demonstrate next that as soon as the bandwidth bottleneck is removed, load balancing has a major performance impact.

NEBULA $_{base}$ is the only system of the four that avoids both destructive bandwidth contention and load imbalance, attaining a throughput under SLO of 22.2MRPS with a sub-2µs 99th% latency. We measured the mean zero-load service time of MICA RPCs using 512B payloads to be ~630ns, which corresponds to a throughput bound of 25.3MRPS. Thus, NEBULA $_{base}$ is within 12% of the theoretical maximum. Before saturation, NEBULA $_{base}$'s minimal 81KB of buffers are adequate and we observe no NACK generation, which concurs with our theoretical analysis suggesting that $10 \times E\left[N_q\right]$ receive buffer slots suffice for a server operating under strict SLO. Beyond ~22MRPS, the system quickly becomes unstable, server-side queues rapidly grow, and the number of NACKs escalates. We consider this to be a non-issue, because operating a server beyond its saturation point is not a realistic deployment scenario. Overall, NEBULA $_{base}$ outperforms SRQ$_{emu}$ by 1.2× in terms of throughput under SLO and by 2.2× in 99th% latency at a load of 16MRPS, due to improved load balancing.

6.8.2 Impact of Varying Workload Parameters

We now study the sensitivity of NEBULA $_{base}$ and RPCValet to different workload behaviors by varying the GET/SET query mix. Table 6.2 reports the maximum throughput under SLO and the memory bandwidth consumption at peak throughput for each query mix we experimented with. We find that both systems are largely insensitive to query mixture, as RPCValet reaches the same saturation point for all three workloads, remaining bottlenecked by memory bandwidth contention in all cases. As the fraction of GETs increases, the NIC-generated bandwidth drops because GET payloads only carry a key, as compared to SETs that carry a 512B value.

Despite less NIC-generated bandwidth, the cores' aggregate bandwidth commensurately rises, because they must copy 512B values out of the data store into the buffer holding the response to the GET. Ultimately, memory bandwidth usage per query remains roughly constant.

In contrast, NEBULA$_{base}$ experiences a 19% throughput increase for the 0/100 mix compared to 50/50, because the mean service time drops by 70ns. This improvement happens because of a change in the direction that MICA copies data. For a SET, the core loads the incoming value from the NIC's small on-chip cache or the LLC (where the payload buffer resides after the NIC writes it), and then must write it to the DRAM-resident MICA log. As the payload buffer is already on-chip, the time that the core's LSQ is blocked on the payload's data is a single remote cache read. In contrast, GETs first must fetch the value from the MICA log, and write it to the response buffer; thus, GETs block the core for a longer period of time.

Next, we evaluate the impact of varying MICA's item size. Figure 6.7 shows the maximum throughput under SLO and memory bandwidth of RPCValet and NEBULA $_{base}$ with 64B, 256B and 512B items. Items smaller than 64B (cache line size) result in the same memory bandwidth utilization as 64B items. As item size shrinks, RPCValet's throughput under SLO commensurately increases, reaching 16MRPS with 256B items and 26.7MRPS with 64B items: smaller items naturally result in less memory bandwidth generated from both the NIC and the CPU cores, alleviating memory bandwidth contention. For item sizes larger than 64B, RPCValet becomes bandwidth-bound, capping throughput under SLO at ~ 21MRPS.

NEBULA $_{base}$'s performance also improves with smaller items. Cores are less burdened with copying data to/from the MICA log, reducing the mean RPC service time by 9% and 19% with 256B and 64B items, respectively. The shorter service time results in a saturation point of 33.3MRPS with 64B items. This is 1.16× higher than RPCValet even when bandwidth contention is not a factor, because NEBULA $_{base}$ eliminates the costly step of write-allocating payloads into the LLC before an RPC can be dispatched to a core. Finally, we emphasize that due to efficient buffer management, NEBULA $_{base}$ attains equal throughput under SLO as RPCValet, handling items 4× larger (256B vs. 64B).

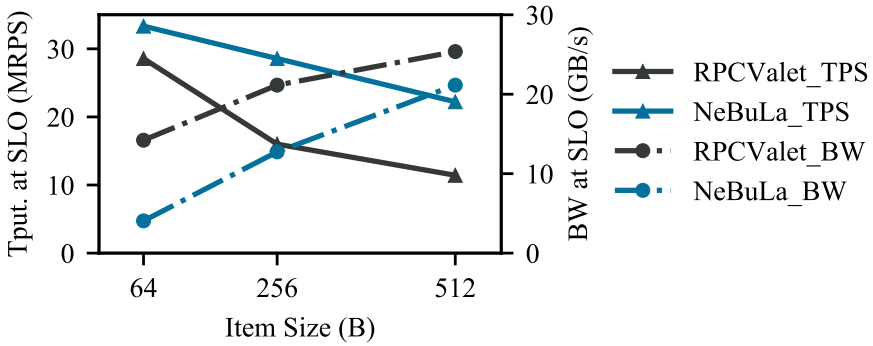


Figure 6.7: Performance of RPCValet and NEBULA varying the MICA value size, using a 50/50 GET/SET query mix.

6.8.3 Performance Benefits of NIC-to-Core Steering

Finally, we evaluate NEBULA's NIC-to-core steering mechanism. Our expectation is that NEBULA should trim ~ 50 cycles (25ns) from each RPC's runtime, hiding the latency of fetching the remotely cached RPC payload that is needed to access MICA's hash index. This improvement should only show at the tail of the latency distribution, because in the average case, the cores only have one RPC in their CQ without a "next RPC" to prefetch.

We additionally compare NEBULA to a software-prefetching solution, which uses a modified RPC-handling loop (Figure 6.3) to prefetch any future RPCs found waiting in the CQ. As the code needed to realize such software prefetching is on the critical path before the RPC's actual processing begins, we hand-optimized the software prefetch code to minimize its overhead. Our first prefetch implementations (e.g., a simple loop with compiler hints such as GCC's `__builtin__prefetch(...)`) were catastrophic for performance, reducing throughput by roughly 50% under MICA's tight SLO. Our final optimized configuration used handwritten assembly code to search 8 slots in the CQ (which fit in a single cache block and thus avoid incurring a cache miss) for a valid RPC to prefetch. As the NIC never dispatches more than two requests to a core at any time (§6.6), the prefetch degree is therefore one.[4] We measure the overhead of our optimized code as 60ns.

The latency improvements due to NIC-to-core steering are directly proportional to the average

[4]Note that although there are only two possible incoming RPCs, the CQ is also used for signalling to the core that outgoing requests have been served by the NIC, so we must scan more than two slots.

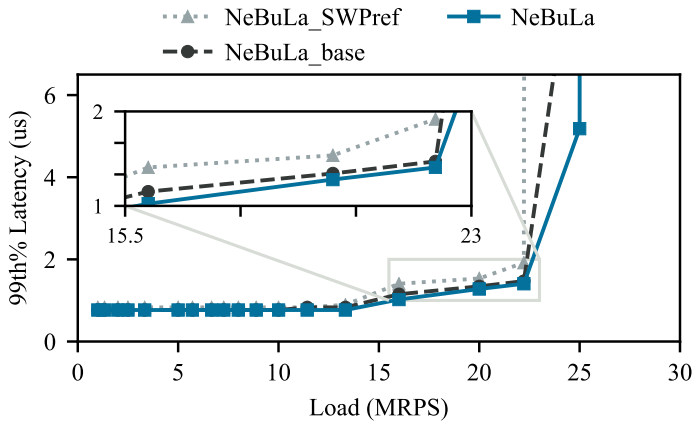


Figure 6.8: Comparison of NEBULA prefetching policies. MICA uses 512B values and a 50/50 GET/SET query mix.

remote cache access latency, because all incoming payloads will by default be fetched from the small cache which is co-located near the on-chip network interface. The most important factor controlling this latency is the average hop count of the server's network-on-chip (NoC). Prior work has shown that the impacts of on-chip latencies on the performance of cross-node memory accesses [53] are particularly costly in emerging CPUs with many cores. When designing hardware for latency-critical RPC services, NEBULA's NIC-to-core steering offers an alternative to deploying exotic NoC topologies to reduce hop count.

Figure 6.8 compares NEBULA $_{base}$ (no prefetching), NEBULA (NIC-to-core steering enabled) and NEBULA$_{SWPref}$, our configuration with software prefetching. Below 16MRPS, all configurations perform identically, because each core only has one RPC outstanding, and by definition there is no opportunity to prefetch incoming RPCs. Additionally, the overhead of our prefetching code is small enough to complete before the next RPC arrival, and thus has no negative impact. Above 16MRPS, the software overhead begins to manifest itself, causing a 31% increase in 99th% latency compared to NEBULA$_{base}$ at 22MRPS. As 60ns represents ~ 10% of MICA's service time, we conclude that prefetching at such high loads requires hardware support to eliminate the overhead of determining prefetch addresses in software.

Between 16 and 22MRPS, NEBULA improves the 99th% latency by 64ns, i.e., a 10% reduction in RPC service time. We attribute the roughly 2× difference with our expectation to the increased cache subsystem latencies in the loaded system. Therefore, the benefit of removing the longer

wait for the RPC's payload from the critical path via timely prefetching increases. As a result, NEBULA outperforms NEBULA $_{base}$ by 3MRPS. At high load, the fraction of RPCs dispatched to a core with CQ depth >= 1 grows to 75%, making NEBULA's 10% service time reduction the common case.

We also repeated the same experiment with 64B payloads, which have reduced on-core service times of 510ns. NEBULA $_{SWPref}$'s overhead grows to > 10% of the service time, and therefore NEBULA delivers 3× lower 99th% latency at 22.2MRPS. NEBULA delivers the same 3MRPS throughput benefit for both 64B and 512B payloads. Finally, we expect that repeating this experiment with the full-scale system, rather than our scaled-down version with only 16 cores, would result in greater performance benefits for NIC-to-core steering due to a larger NoC diameter.

6.9 Chapter Summary

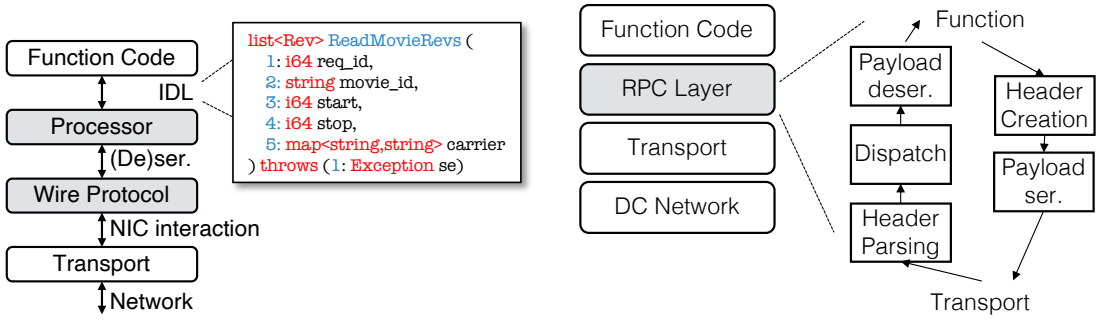
NEBULA is designed holistically to manage network traffic directly in the cache hierarchy of a many-core server and eliminate memory bandwidth interference. By observing that SLO constraints in principle declare that server-side queueing must be limited, NEBULA shrinks the amount of buffering provisioned to easily fit in server SRAM, and proposes a hardware/software co-design to eagerly report "queueing failures" to clients. With three non-intrusive network interface enhancements, NEBULA achieves the goal of in-SRAM handling of network traffic. Employing all of its features, NEBULA improves throughput under SLO over current multi-queue (SRQ) and single-queue (RPCValet) systems by 1.25× and 2.19× respectively. We show that for both 512B and 64B MICA RPCs, the system is CPU-bound and NEBULA does not leave significant performance headroom.

7 Microarchitecture for Integrated RPC Protocol Accelerators

In Chapters 2 and 3, we introduced the importance of Remote Procedure Calls for microservice architectures, and motivated the need to specifically target a bespoke server architecture for production-ready RPC layers that can provide cross-language interoperability. This chapter demonstrates our implementation of CEREBROS, an RPC accelerator capable of executing an entire production RPC layer autonomously, that is integrated directly into the NEBULA protocol stack which we described and implemented in Chapter 6. We begin by describing the need to co-design RPC protocol acceleration with the server's NIC architecture, proceed to describe the logical flow of RPC processing in our implemented system, and finally describe the microarchitecture of CEREBROS' pipelines.

7.1 Why Couple NICs and RPC Accelerators?

Recall that production RPC stacks are comprised of multiple layers and heavily employ generated interface code (§3.1.2). Figure 7.1a shows the software layering of production RPC stacks, with generated code layers shaded in grey, while Figure 7.1b shows the functional diagram of the three common operations that are exercised by these generated code layers at runtime. Recall that first, a programmer uses an *Interface Description Language* (IDL) to specify all the microservice's functions and their desired interfaces. After a new RPC is delivered by the `Transport`, the first task is to parse the application-level header, which indicates the function this RPC is requesting. Next, the RPC stack dispatches this new request to the



(a) Intra-RPC stack layering, with generated code in grey. (b) Runtime operations exercised when receiving RPC requests and sending responses.

Figure 7.1: RPC stack layers, and the underlying runtime hardware operations.

correct `Processor`, which contains all the logic to handle function requests of this type. The software then performs a de-serialization operation on the payload corresponding to the incoming request, converting the format generated by the `Wire Protocol` into an object which is parseable by the function. Finally, the `Function Code` itself runs and processes the request; any nested RPCs which this function calls to other microservices go through the same process in reverse, evidently without the need for dispatch.

Due to the drastic overheads associated with full RPC layers, multiple designs have already been proposed to either accomplish parts [136, 160, 225, 268] or the entire layer [276] in dedicated hardware. We note that two current trends in many-core server architecture dictate that such RPC accelerators (RPAs) need to be co-designed with the server's NIC itself. First, server CPU manufacturers have embraced designs which are comprised of many tiles (in the case of Intel's Xeon Scalable), or independent dies (for AMD's EPYC), connected by an on-chip fabric. As tile and die counts rise, on-chip network latencies between the server NIC and the CPUs will in all likelihood become a more significant factor of server-side RPC response time. Therefore, ensuring low RPC latency will inevitably require explicitly designing a server's network interface to minimize the time taken for all NIC-software interactions corresponding to an RPC to transit the server's on-chip fabric. As all RPCs processed by a server pass through the RPC accelerator on their way to/from the NIC, the location and design of said accelerator are key components in the latency-minimizing equation.

The second trend motivating NIC-RPC accelerator co-design is the sheer number of cores that

modern servers contain. As we have introduced in §2.4, load balancing among these cores is of primary concern because the amount of load imbalance in a queueing system scales proportionally to the number of workers. Given the existence and benefits of NIC-driven load balancing frameworks [54, 150, 182], it is logical that integrating an RPC accelerator should support NIC-driven load balancing policies, as well as the locality-aware enhancements which we introduced in Chapter 5.

All existing efforts to accelerate RPC stacks create one of two pitfalls: either by incurring excess cost in offloading individual operations to the RPA, or in re-shuffling RPCs among cores with software-based load balancing. In this thesis, we summarily mention that the cumulative costs of offloading individual header parsing, creation, and (de)serialization operations cripple the performance of an RPA that can accelerate all of those operations. Further argumentation and evidence to demonstrate this claim can be found in "Hardware-Software Co-Design of an RPC Processor" [222].

In contrast, RPA designs which are tightly coupled to the core's pipeline, such as the recent `protobuf` accelerator from Google [136], limit offload overheads to a minimum by offering an ISA-level interface to RPC software. However, such direct integration comes at the cost of obviating locality-aware load balancing policies, because the RPC's header fields are *not visible* to the NIC until the header parsing step takes place. With a tightly-coupled RPA, incoming messages first exit the `Transport` layer in Figure 7.1b, and are assigned to cores according to the NIC's implemented load distribution or balancing policy (e.g., RSS). After the core invokes its integrated RPA to parse the RPC's header, it then has the knowledge of the various header fields and could apply locality-aware load balancing. Such a system architecture incurs all the overheads that come with reshuffling requests between cores in software, such as synchronization on shared task queues and passing cache lines back and forth between L1 caches that are potentially far away in the on-chip fabric. Therefore, our goal in this thesis is to design a system that "does not hide the power" [157] of the NIC's presence on the path of incoming RPCs, and its capability to handle the RPC stack and apply load balancing without software involvement. We begin with a functional overview of the steps involved in CEREBROS' workflow when processing an RPC.

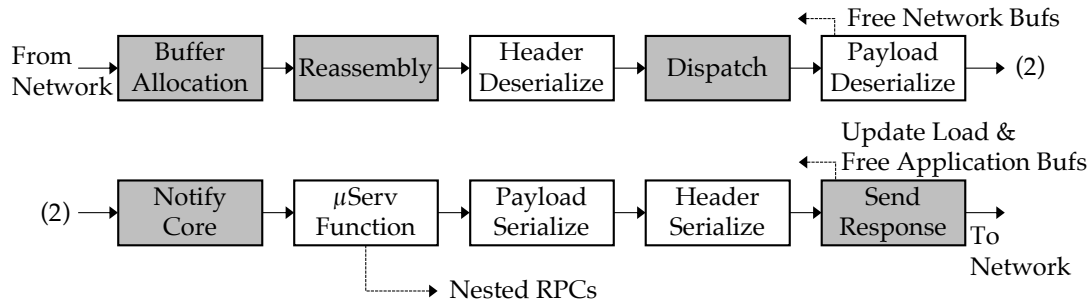


Figure 7.2: Block diagram depicting the stages of an RPC as it is processed by CEREBROS.

7.2 Functional Overview of RPC Processing in CEREBROS

Figure 7.2 shows the workflow of all operations executed by CEREBROS during the process of handling an incoming RPC. Stages which are shaded exist in NEBULA but have been modified, while unshaded stages represent unmodified stages from the RPC stack accelerator [222]. Upon receiving a new request from the network layer, NEBULA first performs buffer allocation for both the transport and application-level buffers, using the allocators previously described in §6.4. While new packets continue to arrive for this RPC, the Reassembly stage performs data placement (see §6.5), ensuring that the RPC is presented to the next stage as one contiguous and properly ordered buffer. After Reassembly completes, the modified NEBULA invokes the first RPA stage, providing it pointers to the transport and application buffers.[1]

The RPA receives both buffer pointers, and begins by de-serializing the RPC layer's header and writing it to the start of the application-layer buffer which will be presented to the core. Additionally, the header de-serialization stage extracts the function's ID from the header and provides it to the Dispatch stage as its input. CEREBROS' Dispatch stage uses the incoming RPC's function ID for two tasks: (i) to decide which core will serve the RPC according to the dynamic affinity policy of §5.2, and (ii) to determine the data layout of this RPC's payload, as specified by the programmer's IDL (see Figure 7.1a). Next, CEREBROS de-serializes the RPC payload in accordance with the data layout supplied to it by the Dispatch stage. Once the writes to the application-level buffer are completed, the RPA's payload de-serialization stage can then send a feedback message to the Buffer Allocation stage, which allows the

[1] We discuss how NEBULA calculates the size of the application buffer to allocate in §7.4.

transport-level buffer to be freed and re-used.

At this point in the flow of an incoming RPC, the selected core now needs to be notified of the new RPC arrival, so the software can begin processing it. CEREBROS uses a similar mechanism to NEBULA for this purpose, with slight modifications to account for the extra control-flow transfer that must take place on the core. Recall that when the RPC layer happens in software, there exists a control-flow transfer during the Dispatch stage to the `Processor` object that is responsible for the incoming function. However, with CEREBROS, that control transfer must happen after the notification is received. To facilitate this jump, CEREBROS provides an address to which control must be transferred along with each entry delivered to a core in its QP. Once the core receives an RPC arrival notification, it jumps to the address provided by CEREBROS, and begins executing the function code – note that all arguments can be guaranteed to be resident in the selected core's L1 cache by using NEBULA's L1 steering feature (see §6.6).

After the function's execution completes, it goes through a subset of the RPC reception process in reverse, to send the response value back to the client. The RPA is first invoked to serialize the response payload, then create a correctly formatted header, and finally NEBULA sends the response on the network. At this stage, NEBULA sends two feedback messages. The first goes to the Buffer Allocator, indicating that it can free the application-level buffer that it previously allocated for this RPC. The second goes to CEREBROS' Dispatch stage, indicating that this core has finished processing a request, so the Dispatch stage can update its snapshot of the system's queues. An additional possibility is that the actual function execution involves RPCs that are nested within each other, where the currently executing function must perform RPCs to another microservice to retrieve data. These RPCs would be handled in the same way as what is depicted in Figure 7.2 on the server which hosts the target microservice. Next, we discuss how we integrate CEREBROS into a server chip.

7.3 The Architecture of CEREBROS

To limit offload overheads and simultaneously enable locality-based load balancing, we adopt a NIC-interfaced RPA design with two critical features. First, the endpoint of the `Transport` layer exists at the RPA itself rather than at a CPU core, and second, the control transfer

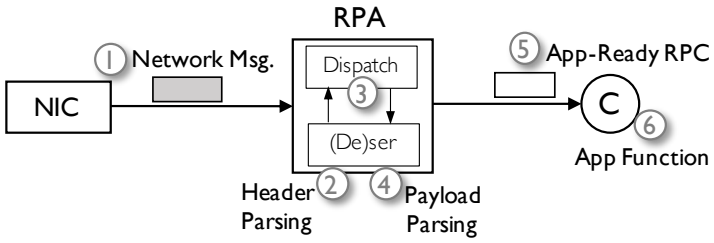


Figure 7.3: NIC-interfaced RPC accelerator design.

operation that occurs between the NIC's interface and `Function Code` is deferred until after the entire process of receiving the new message is complete. Figure 7.3 shows the design of CEREBROS we propose as part of an RPC-centric server architecture. The NIC directly interacts with the RPA and thus removes the excess data transfer and invocation costs from the CPUs, as well as enables enhanced load balancing: it receives incoming RPC requests directly from the NIC (①) and processes the RPC layer without the CPU's involvement, starting with header parsing (②). It then performs the dispatch module using hardware, by reading the function ID from the parsed header and looking it up in a dispatch table to find the metadata describing how to parse the corresponding payload type (③). Using this information, the accelerator parses the payload (④) and dispatches an application-readable RPC to the CPU (⑤) that executes the requested function (⑥). Additionally, as the RPA already has knowledge of the requested function ID from header parsing (②), its load balancing policy can use any fields that have been extracted from the incoming header to implement any policy (e.g., those we introduced in Chapter 5).

Pushing the dispatch layer's control transfer to the final step of the RPC handling process requires the addition of a single function pointer to the notification that the CPU receives when a new RPC is available. All that the core must do is perform an indirect jump to that function pointer, with the arguments provided by the RPA, to start executing the incoming function. Having presented our design to unify the NIC hardware with existing RPC accelerators, we now discuss the implications for the NIC's architecture as well as the RPA itself.

7.3.1 Multi-Buffer Management

Existing research on RPC systems generally assumes zero-copy network I/O, where the RPC payload written into memory is directly consumed by the application layer, without any intermediary operations being done on it [54,131,150,159]. However, fully-featured RPC stacks inherently require header and data manipulation, implying that the payload delivered by the transport layer is accessed and rewritten by the RPC stack itself. This functional difference mandates two changes to the buffer management structure of today's hardware-terminated network stacks:

1. Both the send and receive paths are no longer trivially zero-copy.
2. The combination of NIC and RPA needs to consume multiple variable-sized buffer locations per outstanding RPC (one for the wire format, and another for the application format).

We do not expect that the addition of a (de)serialization copy will form a significant performance problem for systems which provision buffers for LLC-residency such as ResQ [256] or NEBULA. First, the combination of a hardware-terminated transport with an RPA implies that the CPU is not involved in this copy, avoiding the well-known inefficiencies of software `memcpy` [246, Figs. 2-3]. Additionally, cache bandwidth is plentiful in today's servers; assuming 100Gbps of incoming RPC traffic which must be transformed and delivered to the cores, this translates to a cache bandwidth requirement of $\sim 25 GB/s$. This demand can easily be met by today's caches, which can reach bandwidths of $128 GB/s$ for an L1 that serves two independent 64B cache blocks every cycle, as demonstrated by current profiling and characterization work [17].

Even in a pathological case where the RPA's private cache is thrashed and begins to spill some blocks back to the server's shared caches, the multi-tile LLC should not be a bottleneck as each tile would have a bandwidth of $\sim 45 GB/s$ assuming 12ns latency and 8 MSHRs [17]. We calculate these values based on the private L2 cache latency, rather than the LLC, of the AMD EPYC 7571 because our evaluation assumes roughly 1MB/core of LLC, which is 2× the size of AMD's private L2. Therefore, we believe doubling the 6ns measured latency [17] of the EPYC 7571 is a reasonable proxy for the latency of a 1MB LLC tile. We conclude that cache

bandwidth is not an issue for the RPA's (de)serialization stages and therefore focus on the second challenge: allowing the NIC/RPA to consume multiple buffers for one incoming RPC.

The existence of two independent stages of buffering – in the `Transport` and `Wire Protocol` layers – implies that buffer exhaustion can happen in either disparate stage; creating additional hardware complexity to maintain strict layering when hardware controllers accomplish each stage. When software handles application-layer protocol processing, it can return a failure notification to the sender using the underlying transport protocol. Although hardware-terminated transports already contain support for handling cases where transport-layer operations fail [191], independently handling failure notifications with the RPA would require additional logic for the RPA to send unsolicited messages to clients and an extra interface in the NIC hardware that is distinct from its regular interface to the CPU cores.

We argue that a more efficient alternative is to unify the two stages of buffer management and make them "all-or-nothing" atomic. This decision to allocate both buffers together avoids the need for creating an additional interface in the NIC which is likely to be complex and costly. With unified buffer allocation and an RPC-oriented transport (§4.2), one request generates exactly one success/fail response using the NIC's existing pipelines which are already designed to handle RPCs natively.

Our design performs the following steps to guarantee that both the transport and application layers have enough buffer space for their respective operations. First, upon the arrival of a new RPC request, the NIC allocates a transport-layer buffer for it dictated by the "size" field carried in the transport header. If this request is of the type that requires RPA processing, the NIC further allocates an application-layer buffer for it. Once transport processing has completed, the NIC passes the address of the application-layer buffer down the pipeline to the RPA stages.

The two buffers can be allocated with a variety of hardware implementations. In a scenario where the NIC has bespoke buffer management logic, as in NEBULA, it can reserve two buffers with the exact sizes that are required. We defer discussion of the sizing of application-level buffers to §7.4. It is also possible to perform multi-buffer management in an architecture where the CPU makes memory available to the NIC through a shared ring buffer (e.g., in Ethernet). In this scenario, the NIC must consume entries from the ring until the total size

meets or exceeds that required for the transport and application layers.

7.3.2 Distributing the RPC Processing Components

In a server ecosystem characterized by continued network-compute integration (see §2.5) and growth in the number of components making up the server's on-chip fabric, it is critical to design the network interface to match the rest of the server's components. Multiple prior works have already shown that a failure to consider the specific characteristics of the server's memory hierarchy and on-chip fabric can lead to performance bottlenecks for operations such as remote memory accesses [53] and RPC (de)serialization [275, 276]. We therefore argue that it is logical to implement CEREBROS on top of the NEBULA transport protocol, because it already contains the requisite architecture to efficiently pass messages between the NIC and CPU cores. Although the use of NEBULA and its underlying Manycore Network Interface (mNI) [53] is *not* mandatory to implement CEREBROS, we choose it because of the performance benefits, particularly in view of future physical-layer bandwidths that will demand the NI to scale commensurately.

In a NEBULA-equipped server, it is immediately apparent that the logically centralized architecture of CEREBROS shown in Figure 7.3 needs to be implemented on a physically decentralized transport substrate – meaning that there need to be multiple disparate units for (de)serialization, dispatch, and core-NIC interaction. Our decentralized architecture of CEREBROS is enabled by the insight that the majority of operations comprising an RPC's data path do not require any coordination between the NIC's many components. The only operation that requires centralization is the RPC's dispatch stage, to avoid suboptimal decisions being made with respect to load balancing and locality. Prior work already has solved this issue by designating a single NIC component as the unique dispatcher, and enforcing that all other components must inform the dispatcher once the protocol processing component of the RPC has been completed [54].

Figure 7.4 shows the distributed, high-level architecture of CEREBROS, and its integration into a NEBULA-equipped server architecture. NEBULA's components are shown in maroon, with

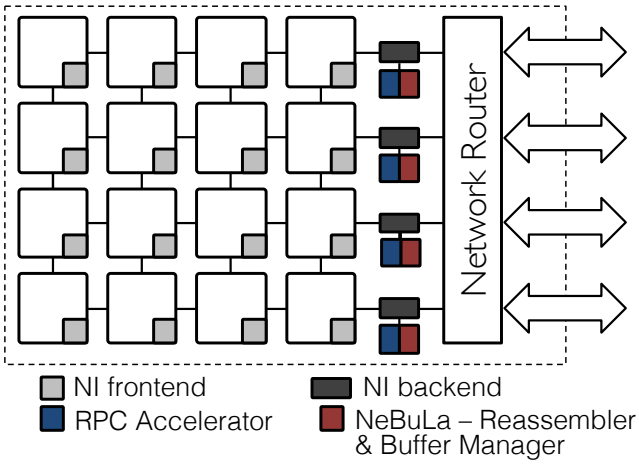


Figure 7.4: Chip-level integration of CEREBROS.

the additional architectural components for RPC acceleration in blue.[2] CEREBROS' design only requires modifications to NEBULA's NI backend components, as they are the ones that handle RPC data placement and load balancing. To keep payload de-serialization coordination free, CEREBROS implements a split dispatcher, where each of the RPAs can independently supply the correct payload format, and then core assignment is performed by the centralized dispatcher *after* payload de-serialization is completed. Replicating the payload formats across all of the NI backends is possible because the formats are read-only, and few in number as most microservices have only a few tens of functions at maximum.

CEREBROS shares a single MMU and small private cache with NEBULA, which is coherent with the rest of the server's on-chip cache hierarchy. Having both components share an MMU and cache is logical because once outgoing messages are written into NEBULA's transport-level buffers, CEREBROS' RPA components will immediately access that data for header parsing and de-serialization. These accesses will almost always hit the cache. For the inverse process of serializing an outgoing RPC, the same principle is true except the cache hits will occur when NEBULA reads the serialized payload data to encapsulate it into a transport buffer.

Each NI backend comprises two independent pipelines: the Request Generation Pipeline (RGP) and the Remote Request Processing Pipeline (RRPP). The RGP is responsible for creating new RPC requests or sending nested RPCs, while the RRPP handles new incoming requests.

[2]We omit NEBULA's prefetch controllers for clarity, but note that they are still part of the final implementation.

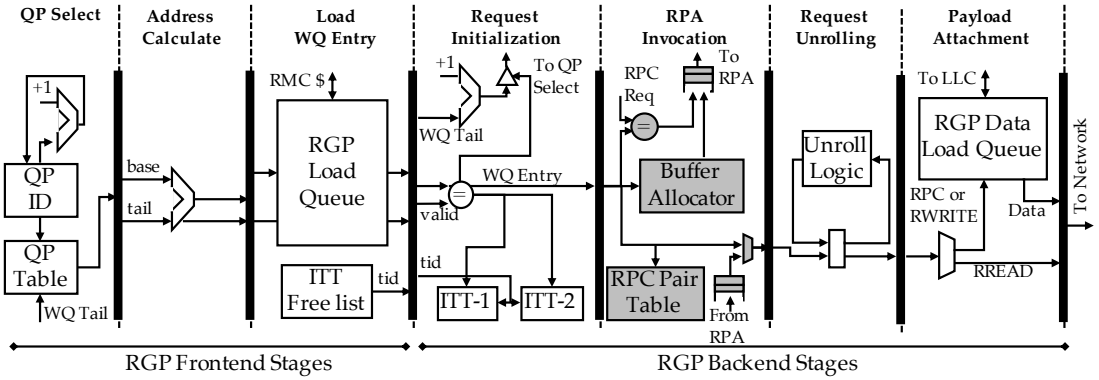


Figure 7.5: RGP microarchitecture. Shaded structures are new in CEREBROS.

As the RPA and its surrounding logic is involved with both incoming and outgoing RPCs, we add hardware queues at specific stages of the RGP and RRPP to create new operations for the RPA(s). We now present our modifications to each pipeline to build CEREBROS.

7.4 NIC Pipelines for CEREBROS

In this section, we walk through the microarchitectural changes required to NEBULA's NIC pipelines. The original pipeline implementations are based on the Scale-Out NUMA Remote Memory Controller (RMC) [199].

7.4.1 Request Generation Pipeline

Figure 7.5 shows the microarchitecture of CEREBROS' modified RGP. Beginning from the QP Selection stage, each RGP polls the WQ of its associated core, and sends new valid requests down the pipeline. Each protocol-level request is identified by a unique ID which comes from the structure called the Inflight Transaction Table (ITT). Once the WQ entry arrives at the RGP BE, the changes to the RGP are minimal, only involving the addition of an extra pipeline stage that invokes the RPA and receives responses from it. In addition to the SRAM structures used by the existing RGP, CEREBROS requires an extra buffer allocator and surrounding logic, to allocate space for the RPA to create the serialized version of the request. The serialized-buffer space is freed after the last stage of the RGP sends out each packet corresponding to this outgoing request, indicating that the data no longer must be held at the sending RGP.

In the original design of the NEBULA architecture in Chapter 6, the total amount of space available to the buffer allocators was sized by the number of outstanding RPCs at the 99th percentile of the latency distribution (see §4.2.1). The same insight applies when dedicating space for buffering in the RPA, with the exception that the RGP does not have the option to "NACK" a request based on an expected SLO violation. If the buffers become exhausted, the RGP will simply stall and cause backpressure in the WQ itself. Applying Equation (4.2) to an RPA which can support eight outstanding requests (giving $k = 8$) at an extreme load of $A = 7.5$ yields an expected queue depth of 12, and therefore a total buffering space of a few KBs would likely suffice. Note that this buffer space does not need to come from dedicated SRAM but can be mapped from the microservice's regular virtual memory space, and then registered with the RGP in the same fashion as receive buffers in NEBULA.

CEREBROS also relies on an additional structure called the RPC Pair Table (RPT). Compared to other transactions supported by the soNUMA communication protocol, RPCs contain the additional requirement that each application-level response must be paired with the exact matching request. Although the mNI architecture already tracks outstanding network transactions in the ITT, its intention is to match network-layer requests and replies as demanded by the soNUMA protocol [199]. For CEREBROS, we add the RPT to perform the same functionality but now corresponding to the RPC layer. When a new RPC request is passed to the RPA invocation stage, a unique identifier for it is stored in the RPT along with the QP id that is expecting a response. CEREBROS uses the same unique identifier for RPCs as NEBULA: the concatenation of the source node ID (`snid`) and the network transaction ID (`tid`) generated by the Request Initialization stage.

7.4.2 Remote Request Processing Pipeline

The majority of CEREBROS' hardware additions take place in the RRPP because it must handle new incoming requests, interact with the RPT, and also consume entries in the RPT created by the RGP. Figure 7.6 shows the microarchitecture of the modified RRPP in CEREBROS. Control logic for sending feedback messages to the buffer allocators is not shown.

The first modification needed for CEREBROS is the addition of an extra buffer allocator, visible

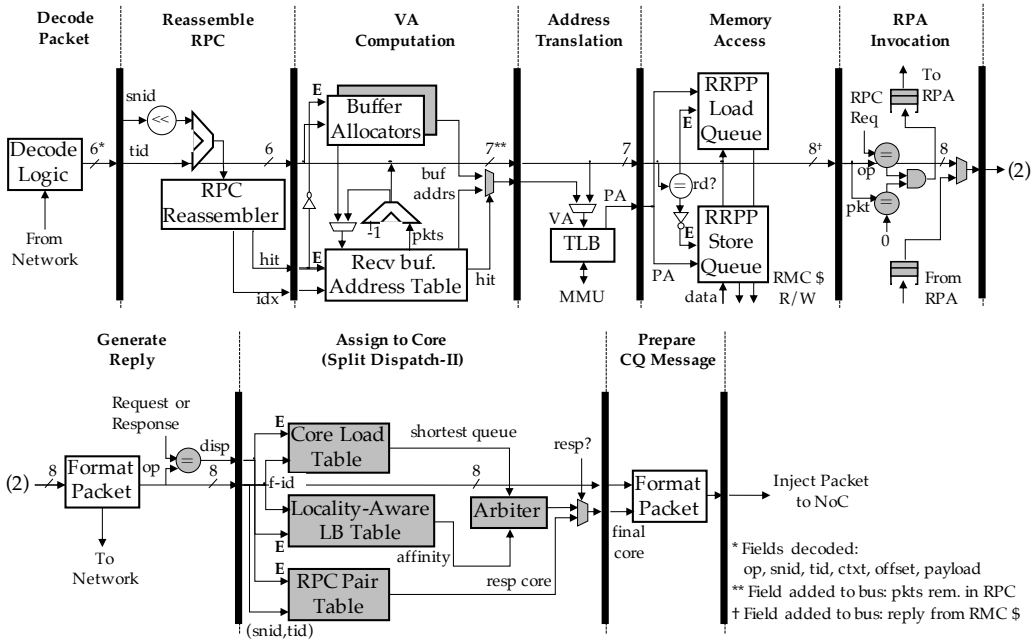


Figure 7.6: RRPP microarchitecture. Shaded structures are new in CEREBROS.

in the Virtual Address Computation stage. This allocator has the same function and structure as the existing one, but allocates memory for the de-serialized RPC as previously discussed. Both allocators are only enabled after an incoming RPC has incurred a miss in the reassembly table. The addresses of the newly allocated memory are written to the receive buffer address table (for transport layer buffers) or added to the control bus and sent down the pipeline to the RPA Invocation stage.

Unlike the transport layer where the message's size is carried along with it in the header (e.g., in UDP [221], R2P2 [150, Fig. 3], or NEBULA), the `Wire Protocol` of the RPC stack does not include the size of the incoming payload in its application-ready format. Even if whole-message compression is disabled, RPC arguments are actually shortened depending on their *values* – for example, Google Protobuf uses a type called `varint` where a full 8 byte integer can be compressed all the way down to 1 byte if its value is less than 128.[3] Furthermore, the RPC software must introduce metadata into the raw wire format in order to demarcate each field, and denote if the field itself is a complex data type such as the `Map` of Figure 3.2b. Therefore, it is impossible to determine *a priori* the exact size of the application-level data which is to be

[3] NB: The maximum value is 127 and not 255 because the 8th bit is needed to signify whether the receiver needs to continue reading if the integer is larger than 1 byte.

(de)serialized – however, to guarantee that network buffers are not over-written, the NIC must ensure that there is enough space for the RPA to write the message into.

To solve what seems like a circular dependence between the allocation stage and (de)serialization process, we apply a mathematical insight to bound the maximal application-level buffer size. In the scenario where a new message is received and must be de-serialized, the total message size is provided by the transport layer as mentioned above. The maximum application size can be calculated by assuming the message has the maximal number of fields, and each field is maximally compressed. We observe that the maximal compression for a field is a strict $4\times$, in the case where an 8B `varint` becomes 2B in the network format: one byte for the data itself, and one byte for a metadata tag used by the RPA to de-serialize the message. All other fields (e.g., strings, booleans, floats, and compound types such as `Map`) will incur a lower compression ratio, and therefore we believe that it is a sufficient solution to allocate an application-level buffer of $4\times$ the transport size.

This observation also can be used in the RGP, when RPC is being sent and must be serialized to its wire format. The RGP's allocator makes the conservative assumption that the transport size is no smaller than the application size, and allocates a buffer which is equal to the application size. The RPA can then inform the NIC of the final size to be used by the transport protocol to avoid bandwidth inflation.

Returning to the RRPP, the last added stage in the pipeline is the split dispatcher, whose job it is to assign incoming requests or responses to cores following the chosen load balancing policy. The first half of the split dispatcher resides in the RPA itself, which extracts the function ID from the parsed header and creates the correct de-serialization task. Choosing the core for an incoming request or response is based on three pieces of information: (i) the depth of each core's input queue, which dictates load balancing. (ii) The fields of the RPC's header that dictate potential locality and affinity (see Chapter 5). (iii) Whether this response must be paired exactly with a previously sent request.

The simplest dispatch tasks correspond to RPC responses. The RRPP accesses its RPT with the pair of `(snid,tid)` as an index, and returns the core which originally sent the request – this information was placed there by the RGP. However, if the incoming work corresponds to a

new RPC request, the dispatcher chooses a core based on factors (i) and (ii) above. First, it looks up the Core Load Table and Locality-Aware Load Balancing Table (LLBT) in parallel. The Core Load Table stores the depth of each core's input queue, and is unchanged from NEBULA. Depending on the exact policy which is desired by the software, the LLBT returns a set of one or more cores that can execute the incoming RPC with improved locality. For example, if I$ affinity is the policy of choice, the LLBT must return all the cores which have recently executed this function (see §5.2.2). Finally, the policy arbiter chooses the core which will execute the RPC, deciding between to the baseline `JBSQ` policy, or any of the locality-aware policies introduced in Chapter 5.

Locality-Aware Load Balancing Table Microarchitecture.

To implement both locality-aware load balancing policies we proposed in Chapter 5, the LLBT needs to support two fundamental operations. First, to support d-CREW it needs to provide the ability to match KVS keys of incoming requests against a set of partitions that are in exclusive mode (see §5.3.2). Second, for dynamic function affinity to increase I$ locality, it needs to support storage of a per-core history of executed RPC functions (see §5.2.2). Both modes require a different memory organization; supporting d-CREW requires an "exact match" operation that is trivial to achieve with a Content Addressable Memory (CAM), whereas the per-core history mode needs to be indexed by the core ID.

We claim that both memory organizations can actually be achieved with *the same* CAM structure by exploiting its internal microarchitecture. In Figure 7.7, we show the microarchitectural organization of the LLBT, following traditional CAM designs that are split into an associative portion which is the "tag array" and a direct-mapped RAM structure storing the data associated with each entry [215]. When a new lookup operation occurs, the value being searched for is placed on all the CAM's search lines, and then all matching entries are sent to a priority encoder at the output. The encoder then outputs the index of the chosen match (if there is more than one), which is then used to look up the direct-mapped RAM and output the matching data. It is trivial to implement d-CREW with such an organization by storing the KVS' partitions that are in exclusive mode in the CAM array, and the core which has "locked" them in the RAM.

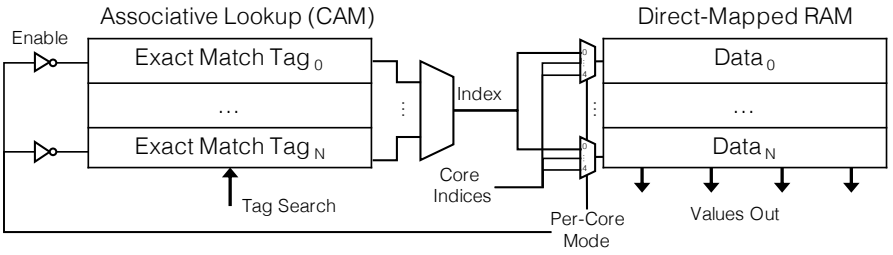


Figure 7.7: Microarchitecture of the Locality-Aware Load Balancing Table, showing associative lookup array and RAM storage.

To implement a per-core function history with the structure shown in Figure 7.7, we use only the RAM and clock-gate the associative tag storage section. Each entry in the RAM is used to store the function history of a single core, which can be directly accessed by presenting the core's ID as the index value to the RAM array. Although the simple design that we propose operates the LLBT in a binary fashion that uses either exact matching or per-core storage, the CAM organization itself does not prevent both modes from being simultaneously supported. For example, in a virtualized deployment where half of the server's cores wish to use dynamic function affinity and the other half are running a d-CREW KVS, the LLBT's RAM entries could be split accordingly.

Supporting dynamic function affinity with the aforementioned LLBT structure requires every row of the LLBT's RAM to be read out at each dispatch decision, which must be implemented at the rate of incoming requests. Assuming we target a 64 core CPU with RPCs requiring 500ns of computation, the LLBT has a limit of 16 cycles to read all its entries, assuming the server is running at its maximum sustainable load. Such a time limitation requires four reads per cycle, which can be implemented by adding multiple output buses to the LLBT's RAM stage at the cost of higher RAM array power. All other load balancing tables are accessed in parallel to the LLBT (as shown in Figure 7.6) to allow all outputs to be presented at the arbiter in time.

To model the area and power cost of the LLBT design we propose, we use CACTI 6.5 [164] configured to model a 22nm process with built-in ITRS-HP device projections and a 2GHz cycle time. We provision an LLBT with 128 entries to target a server with 64 cores, where each associative tag in the CAM portion is 30 bits wide, and each entry in the RAM portion is 64 bits. We choose a 64-bit data size because it is large enough to store the core ID for d-CREW, and a

history of function ID's for dynamic affinity load balancing (e.g., a function history of 8 entries with 8-bit IDs). The LLBT requires only $0.0073mm^2$ of chip area due to its limited amount of data stored (<1.5KB in total despite over-provisioning). Operating the full CAM structure with its associative lookup structure enabled requires $15.1mW$ of power, even pessimistically assuming it is accessed every single cycle. In the context of a 64 core server such as the EPYC 7763 [94] whose power is 280W, adding the LLBT incurs a 0.001% overhead, which we believe is an extremely modest cost for the performance gains it enables.

When operating in its per-core mode with the associative array clock-gated, the LLBT consumes less power than the CAM structure even accomplishing four reads per cycle. Our CACTI simulations show that only accessing a single row of the LLBT per cycle consumes $3.2mW$ – a conservative estimate of the requirements of multiple reads per cycle is to quadruple the power, leading to a maximum consumption of $12.8mW$. We believe such a cost is reasonable given that enabling the LLBT's CAM array actually results in a higher power consumption of $15.1mW$.

In conclusion, we reiterate that the pipeline modifications to NEBULA for implementing CEREBROS are limited. The sum of all the modifications can conceptually be described as a set of tables to reserve pre-allocated memory in which RPC (de)serialization takes place, and logic to assign RPCs to cores.

7.5 Chapter Summary

In this chapter, we architect and implement CEREBROS, a fully-integrated accelerator for production RPC stacks that fits directly into NEBULA. We argue that because current designs either incur excessive overheads to offload tasks to RPC accelerators [276], or prohibit NIC-driven load balancing, RPC accelerators must be co-designed with the server NICs. Our implementation handles RPC traffic in a fully distributed fashion, and only uses a centralized component for the load balancing stage, allowing CEREBROS to scale to the high bandwidths of future NICs. We rely on our implementation for the remainder of this thesis as a substrate for evaluating locality-aware load balancing.

8 The Benefits of Locality-Aware Load Balancing

This chapter demonstrates a concrete implementation of both locality-aware load balancing policies we motivated and designed in Chapter 5. We build our implementations on top of our hardware-terminated network and RPC protocol stacks which were introduced in Chapters 6 and 7. Although we use our own baseline to implement and evaluate locality-aware load balancing, the prerequisites to realize such policies are more general and allow implementations on any hardware-terminated protocol stack that has NIC-driven load balancing. The most plausible candidate systems that could integrate support for locality-aware load balancing would be the NanoPU [108], or SmartNICs such as NVIDIA/Mellanox' BlueField [204], or Netronome's Agilio LX 100 [196]. The remainder of this section is structured as follows: for each load balancing enhancement, we walk through our specific implementation of the hardware and software extensions required to realize it, introduce our simulation methodology, and then evaluate its performance.

8.1 Adding Affinity-Based Load Balancing to CEREBROS

In order to implement affinity-based load balancing as described in Chapter 5, we specifically configure the Locality-Aware Load Balancing Table (LLBT) so that it stores and returns the *function types* that have been assigned to each core.[1] We refer to this particular configuration of the LLBT as a "function map". The function map only uses the RAM portion of the LLBT, and

[1] The design and implementation of the LLBT were introduced in §5.5 and §7.4.2 respectively.

clock-gates the CAM portion of it to save power. Each entry in the function map corresponds to a single CPU core, and stores a FIFO list of recently executed function IDs for that core. When a new RPC is assigned to a core, the function ID is shifted in the head of the core's list, and the tail is shifted out. Our implementation only stores a single entry per core, so that a core is only considered as having affinity if it has just executed the exact same function – with the provisioning of the LLBT in §7.4.2, it is possible to support a history of up to eight functions per core. Further policy optimizations using deeper histories (e.g., in the case where multiple functions have constructive code sharing) are interesting extensions to the proof-of-concept policy we evaluate in this thesis.

Selecting a core for a new RPC involves the following steps: first, each entry in the function map must be read out of the LLBT's RAM and compared against the incoming function ID. Matching the function map's entries with the incoming function ID is implemented with a simple array of comparators, whose width must be equal to the product of: (1) the number of parallel function map entries read per cycle, and (2) the depth of each function history. Our configuration only requires 32 bits of comparisons per cycle (four function map entries where each is eight bits wide). Scaling the function map to deeper function histories or higher parallel read throughput would naturally require more comparators and increased hardware provisioning. In a particular deployment where such a cost becomes prohibitive, alternative storage methods can be considered – for example, using the function ID as the index to the function map, and storing a list of cores which have affinity in the RAM array. Such a design would trade off reduced hardware cost with an approximate notion of affinity; questions related to the performance of such an approximate policy may be interesting directions for future work.

At the same time as the function map and surrounding comparators are generating the set of cores with affinity to the incoming function, the Core Load Table outputs the core with the global shortest queue, which is used as a fallback option. Next, the function map and the Core Load Table forward their outputs to the arbiter (see Figure 7.6) to make the final core selection decision. The arbiter chooses the core with the shortest queue from all the affinity-having cores coming from the function map, or the fallback core if the shortest queue among the affinity-having cores is greater than two requests long.

After the load balancing policy selects the core, NEBULA's NIC pipelines create a metadata structure containing the corresponding request buffer, the incoming message's layout, and a function pointer that indicates the program counter where the core must begin executing. NEBULA notifies the selected core of a new incoming request, passing the metadata to it via a QP entry. Once the core receives the notification, it begins executing the function indicated in the metadata structure.

In the load balancing policy we evaluate for I$ locality, there is no explicit need for software to explicitly control the function map's contents, because the microservices we study do not have inter-function dependencies or synchronization. Therefore, the only messages exchanged between the NIC and software occur when the two components notify each other when new RPCs arrive and complete processing so that their RPC buffers may be freed. The above exchanges are identical to what we have described in Chapter 6, so we do not discuss them further here.

8.2 Evaluating Affinity-Based Load Balancing

To evaluate the benefits of dynamic affinity-based load balancing, we first break down the instruction cache locality characteristics of a set of microservices. We then specifically evaluate the dynamic affinity-based load balancing policy proposed in §5.2.2. A detailed evaluation expressly showing the benefits of RPC protocol acceleration in isolation can be found in "Hardware-Software Co-Design of an RPC Processor" [222].

8.2.1 Methodology

Microservices and RPC Processing. We evaluated the following six microservices from DeathStarBench [81], chosen because they represent the various "classes" of microservices in the entire suite: UniqueId (UID), User (USR), URLShorten (URL), SocialGraph (SG), and ComposePost (CP). These microservices are comprised of one, six, one, seven, and six underlying functions, respectively. Many other microservices in this benchmark suite behave identically or similarly to those we evaluated. In Table 8.1, we further break down the six functions comprising the USR microservice.

Function ID	Name	Tasks
F0	RegisterUser	Create a new user in a social network.
F1	RegisterUserWithID	Create a new user with pre-defined ID.
F2	Login	Process user login requests.
F3	UploadCreatorWithUsername	Upload the creator of a post's username.
F4	UploadCreatorWithID	Upload the creator of a post's user ID.
F5	GetUserID	Lookup a user's ID.

Table 8.1: USR Microservice function IDs, names, and functionalities.

All microservices use Apache Thrift [19, 72] as their RPC layer, to which we have added a new hardware-terminated transport protocol based on NEBULA [250]. To measure the cache behaviour of each microservice in isolation, we create mock components for all other microservices surrounding the one being measured, which respond immediately with pre-constructed messages. To estimate instruction working set sizes, we apply the methodology used for profiling workloads in Google datacenters [133]: we collect the trace of executed instructions and measure how many unique cache lines cover 99.9% of the trace when ranked by popularity.

We use an RPC layer that implements a synchronous request processing model, where each microservice polls for incoming requests and executes them to completion. If a microservice performs nested RPCs as part of its function execution, it also synchronously polls for the results of those RPCs, which CEREBROS guarantees will be returned to the same thread via its RPC Pair Table (see Chapter 7). Running the microservices in this fashion is both an intentional optimization, and implementation necessity for the time being. First, it ensures that when a specific RPC is assigned to a core by the load balancer, it executes there to completion, ensuring that affinity-based load balancing's benefits are not hidden by an asynchronous request processing model. Second, support for asynchronous RPCs is still an open research and development problem [247], as using this pattern required a complete software rewrite at the time this work was completed. Our model for affinity-based load balancing still applies to an asynchronous request processing model, with the caveat that a function no longer remains in a core's input queue until it is completed. The software is responsible in this case to use the enhanced control messages we propose in §5.5 to inform the NIC that the core can handle another request.

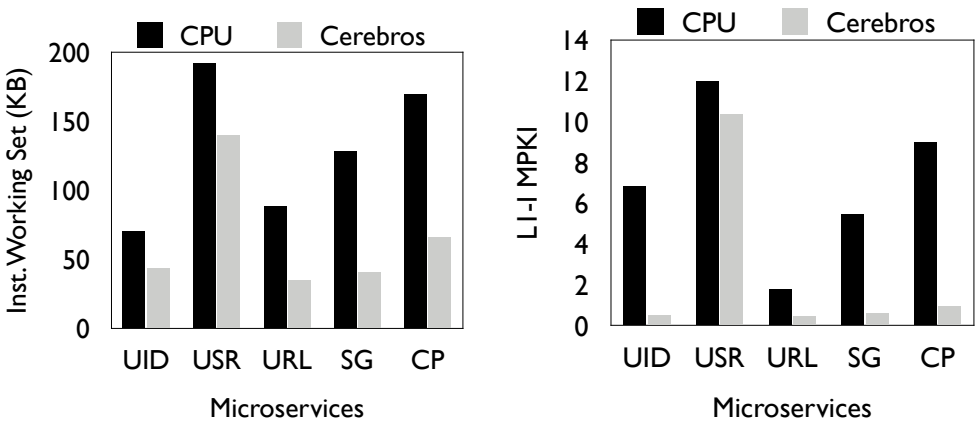
Cores	ARMv8, 4-wide issue OoO @ 2GHz 128-entry ROB, TSO, Next-line I$ Prefetcher
L1 Caches	64KB 4-way L1-D and L1-I, 64B blocks 2 ports, 32 MSHRs, 4-cycle lat. (tag+data)
LLC	Shared block-interleaved NUCA, 1MB/core 16-way, 1 bank/tile, 11-cycle lat. (tag+data)
Coherence	Directory-based Non-Inclusive MESI
Memory	45ns latency, 2×25.6 GB/s DDR4-3200
Interconnect	2D mesh, 16B links, 3 cycles/hop

Table 8.2: Parameters used for cycle-accurate simulation of locality-aware load balancing.

System Organization. We evaluate the performance of dynamic function affinity-based load balancing using cycle-accurate full-system simulation. We use the QFlex simulator [216] to simulate a 16-core ARMv8 CPU running Ubuntu Linux 18.04, whose parameters are summarized in Table 8.2. All workloads are pinned on 15 cores, leaving one core for system tasks and interrupt processing. We limit UID to four cores because lock contention limits its scalability. Our simulator includes a load generator that creates incoming requests based on a given popularity distribution, dictated by the structure of the microservice [222, §6.1], and delivers notifications to the CPU through the NEBULA transport stack.

8.2.2 Availability of Instruction Cache Locality

To clearly demonstrate the availability of instruction cache locality in our evaluated microservices, we measure their working set sizes and MPKI values in two configurations: when the RPC layer is performed by the CPU and when it is offloaded to CEREBROS. Figure 8.1a shows the instruction working sets of our evaluated microservices. In the baseline CPU system, the bloated RPC layer results in working sets that exceed the I$'s capacity by up to 3×. In contrast, CEREBROS' RPC layer offload reduces the working set by 27–68%, which naturally translates to a higher I$ hit rate. The working sets are most visibly reduced for SG and CP, because they have little business logic in their functions and their instruction footprints correspond mostly with RPC layer code due to their large number of nested RPCs and complex message types. Hence, when the RPC layer is offloaded to CEREBROS, we see an instruction working set reduction



(a) Instruction working set.

(b) I$ miss rates.

Figure 8.1: Frontend behavior of microservices.

of more than 60%. On the contrary, UID includes only one nested RPC and uses simpler messages, while the function itself is roughly 43KB in size. Even then, offloading UID's RPC layer to CEREBROS shrinks the instruction working set by 38%.

Figure 8.1b depicts the I$ Misses Per Kilo Instruction (MPKI) before and after RPC layer offload. The working set reduction achieved by CEREBROS directly affects the core's frontend performance, virtually eliminating instruction misses for four of the microservices and reducing CPU cycles wasted on instruction misses by 5–93%.

USR benefits the least among all microservices because it includes two functions with working sets larger than 90KB in size. Naturally, it also experiences the smallest reduction in the instruction working set, as shown in Figure 8.1a. In such cases where the aggregate working set of all the functions still outstrips the L1 I$, even fully offloading the RPC layer to CEREBROS provides limited benefits to the CPU's frontend. We now evaluate the performance of affinity-based load balancing that ameliorates CPU frontend inefficiencies in these exact cases.

8.2.3 Eliminating Further Instruction Cache Misses

Although the aggregate working set of the USR microservice when using CEREBROS is ~140KB, four of its six functions are small enough to fully reside in a 64KB instruction cache if running in isolation. However, the working sets of the other two functions are >90KB. When the NIC's

Figure 8.2: Breakdown of the USR microservice's functions into execution time and instruction cache misses.

core selection policy does not take into account function locality, all six functions will compete for I$ capacity, resulting in the high number of instruction misses visible in Figure 8.1b even after CEREBROS' RPC layer offload. This phenomenon particularly hurts the performance of functions for which I$ misses account for a large fraction of total execution time.

Figure 8.2 breaks down the CPU time of USR's six functions into execution time and time stalled on instruction misses and compares a CEREBROS baseline (C) against CEREBROS with affinity-based load balancing (CA). In the affinity-agnostic baseline, USR's functions are stalled on instruction misses for 15–44% of their total runtime. Function 2 is the only strongly compute-bound function, spending the majority of its time hashing strings after its working set is first loaded into the I$. All other functions have their CPU times divided roughly equally between execution and instruction stalls.

With affinity-based load balancing enabled, the fraction of time stalled on I$ misses drops by $1.05-18\times$, with the larger benefits being applicable to the most commonly executed functions, F3–F5. We measured that ~98% of requests were steered to a core that had just executed the same function type, demonstrating evidence for our hypothesis that function affinity is plentiful. For F3–F5, affinity-based balancing virtually eliminates I$ misses, leading to a $1.8-2\times$ reduction in CPU time. These functions benefit drastically because their instruction working sets are between 20–25KB, which are easily accommodated by our CPU's 64KB I$. Affinity-based balancing allows F3–F5 to execute with zero I$ misses for 94% of requests.

Despite their high number of I$ misses in the baseline, F0–F1 benefit only marginally from affinity-based steering because their I$ misses primarily come from limited cache capacity, not inter-function contention. We have verified this with an experiment enforcing that these two functions execute on dedicated cores to eliminate any contention from other functions. Even in this best-case scenario, F0–F1's CPU times are within 3% of what we observe with affinity-based request steering.

Aggregated across all the functions, affinity-based request steering reduces average CPU time for the USR microservice by 8.7%. The fact that USR's two largest functions (F0–F1) have execution times ~180× larger than its most popular functions (F3–F5) skews the average downwards. In contrast, the median CPU time drops by 33% because F3–F5 comprise 70% of total incoming requests and experience greater speedups.

Conclusions. Our evaluation above shows that even after CEREBROS shrinks a microservice's instruction working set via offloading the RPC stack to hardware, certain microservices still experience function thrashing in the I$, as we argued and studied analytically in §5.2.1. In these particular cases, we show that even a simple locality-aware load balancing policy is sufficient to execute RPCs whose instruction working set is small enough to be I$-resident, with zero instruction misses for the vast majority of invocations. As we have only studied a very simple microservice in detail whose functions are trivial to separate and have very little overlap, we believe that the applicability of affinity-based load balancing far exceeds what we have introduced in this thesis.

8.3 Implementing Cooperative Concurrency Control

Having finished evaluating our proposed load balancing optimizations for instruction cache locality, we turn to implement and evaluate the C-4 policy proposed in §5.4. C-4 consists of its d-CREW load balancing support as well as a single additional software module for write compaction (see § 5.4.1 and 5.4.2). Our implementation of C-4 requires modifications to CEREBROS' request assignment stage to implement enhanced load balancing, additional message types in the NIC-software interface used to control entries in the NIC's load balancing tables, and software modifications for C-4's write compaction optimization.

8.3.1 NIC-Software Interface Modifications

Both mechanisms comprising C-4 require the NIC-software interface to specify whether requests and their responses correspond to KVS reads or writes. The NIC requires such information for the d-CREW policy, to keep track of when a given partition enters and exits exclusive mode and update its set of partition-thread mappings. The KVS must communicate two additional pieces of information to the NIC to enable creation of exclusive mappings: first, how to identify the fields in the application header corresponding to the key and request type (i.e., read or write), and second, how to transform the application's key to a partition identifier.

The key and type field identification information (expressed as offsets and lengths within request packets' application-level headers), is communicated to the NIC during the NEBULA stack's "setup phase" where the application's threads inform the NIC of their active queues and packet buffers using `ioctl` system calls. For simplicity's sake, we implement a proof-of-concept system which assumes the KVS' headers are available at fixed offsets known *a priori* in the incoming RPC payload, similar to what prior work assumes [132, 139, 166]. However, if the KVS is ported to a production-ready RPC layer with variable wire formats (e.g., Protocol Buffers [87]), extracting the correct offsets could be done with CEREBROS' full RPC protocol accelerator to extract the desired fields in each RPC's header [275].

To convert the incoming request's key to a partition ID, C-4 assumes knowledge of the function $f()$ used by the KVS software to map keys into logical partitions. The granularity of partitions implicitly dictates C-4's load-balancing flexibility. Coarser-grained partitions create more load imbalance as more unrelated keys are conservatively considered to be in exclusive mode [63]. C-4 chooses the $f()$ used by the KVS software to select hash buckets, meaning that the minimal load-balancing unit is a few tens of keys. C-4 communicates the number of buckets to the NIC during the setup phase.

A partition mapped to a thread by C-4 can only be re-assigned after being released by the application. We therefore modify the RPC layer so that the function for sending an RPC response takes one more argument indicating whether the application is releasing exclusive access. Our implementation *always* releases exclusive mappings upon completing a write to maximize load-balancing opportunities; retaining mappings longer than strictly necessary to

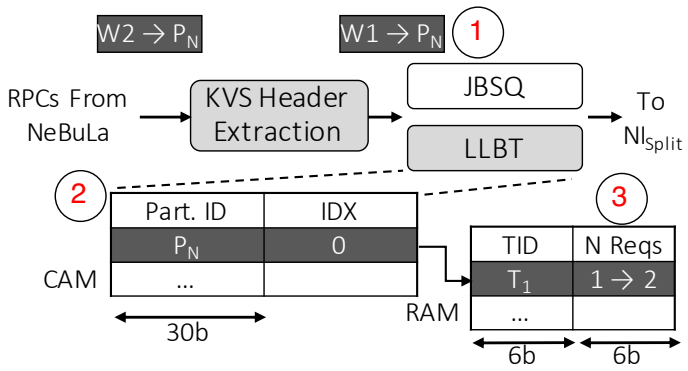


Figure 8.3: Modified NIC request assignment pipelines. The shaded portions are used to implement C-4.

potentially increase temporal item locality is an interesting future direction.

8.3.2 Instantiating Hardware Support for Write Balancing

Figure 8.3 shows how CEREBROS' existing pipelines implement d-CREW in C-4, using the example of two conflicting writes that pass through the load balancing stage. We implement C-4's set of outstanding active mappings, described in §5.3.2, with the LLBT. For C-4, we use the CAM-matching functionality of the LLBT, where the associative search array stores active partition IDs that are matched against incoming RPCs. In the data array, each active entry stores the ID of the thread holding this partition in exclusive mode, and a counter tracking how many writes are outstanding to it. When W_1 arrives at the load-balancing stage (Step (1)), dedicated logic extracts the KVS' headers to determine if this request is a write, and what partition must be looked up in the LLBT. As there is no active entry for P_N, the partition is unassigned and C-4 assigns the request to T_1 using JBSQ. C-4 then allocates a new entry in the LLBT for P_N, initializing its counter value to one outstanding write, and the owner thread to T_1 (Step (2)). When W_2 arrives, its lookup of P_N hits the LLBT and thus C-4 assigns W_2 to T_1 (Step (3)) and increments the outstanding write counter.

After T_1 processes both writes, it releases exclusive access to P_N using the RPC response interface of §8.3.1. Each message triggers an access to the LLBT for P_N to decrement the outstanding write counter (not shown). C-4 frees the LLBT entry after W_2's response, enabling writes to P_N to be load balanced again.

Figure 8.3 also shows the information stored in each field of the LLBT. We use the LLBT's full 30-bit tag width as the partition ID, and six bits of the data storage array for each of the outstanding request counter and thread identifier. Such provisioning allows C-4 to support up to 64 threads, with each partition supporting up to 64 concurrently outstanding writes. If desired, the request counter and thread identifier widths can be increased without any concerns for the LLBT's cycle time or power consumption, because the LLBT contains up to 64 bits of storage in each entry. Furthermore, it is simple to match the throughput of the NIC given that only one LLBT entry is read per cycle.

8.3.3 Software Support for Compaction

Our software implementation of write compaction is a single module residing between NEBULA's RPC layer and the KVS itself. To allow this layer to scan the incoming RPC queue for compaction opportunities, we add an interface to NEBULA's RPC layer allowing the caller to apply a lambda function on each valid request in the queue, similar to BPF [181]. C-4 scans for compactions on every incoming write request. When a compaction is detected, C-4 opens a compaction window, records the key being compacted, and begins buffering all incoming writes to it. Each compacted request also stores two pieces of information to allow C-4 to send a matching RPC response when the compaction window closes: i) the request's buffer address and size in memory, which NEBULA's transport layer later reclaims, and ii) the sender's node ID to whom the response will be sent.

To maintain the KVS' SLO guarantees, we use the abstraction of a monotonic hardware clock provided by most modern processors, that has a known frequency and is readable from user mode with limited overhead [138]. For example, `AMD64` provides the `tsc` instruction [113, §17.17] and `ARMv8` provides the architecturally visible Generic Timer [22, §D7.1.2]). Upon opening a compaction window, C-4 snapshots the current clock value and sets the window's expiration time as $T_{exp} = T_{cur} + (SLO - 1) \times \bar{S}$, where we assume the SLO and service time ($\bar{S}$) are known. C-4 checks the current clock value upon each request's completion to determine if the compaction window must be closed. When the expiration deadline is reached, C-4 applies the final compacted value to the data store and then creates an RPC response for each of the buffered writes. As this operation is on the critical path of future requests, we optimized it to

the point of being able to send a response in ~ 150 cycles. As closing compaction windows was not a bottleneck in our workloads, we did not need to further optimize it. We note that the task of creating many similar NIC-software descriptors in parallel is a perfect match for SIMD or vectorized instructions if further optimization would be needed.

8.4 Cooperative Concurrency Control in Action

We evaluate C-4 by focusing individually on the two challenging regions of the KVS workload space described in §5.3.1, and the C-4 mechanisms introduced for each one: d-CREW for write-dominated, uniform workloads (denoted WI_{uni}) and write compaction for read-write, heavily skewed ones (denoted RW_{sk}). Therefore, we present experiments showing the impact of each technique on its respective region of applicability, and show that both techniques have limited impact outside the target region.

8.4.1 Methodology

System Organization. To evaluate the impacts of C-4 on system performance, we use cycle-accurate full-system simulation of a 64-core processor running Ubuntu Linux 18.04, using the QFlex simulator [216]. We use the same simulation parameters as those in Table 8.2, except we increase the core count to 64, and the DRAM organization to eight DDR4-3200 channels to match. Our high-core count CPU is representative of today's server products such as Intel's Xeon Scalable [95, 170], AMD's EPYC [94], Amazon's Graviton [16, 266], and Huawei's Kunpeng [270]. C-4's performance benefits scale with core count, because load imbalance between workers in a queueing system increases with worker count. We expect core counts to modestly grow in the near term with developments in chiplet manufacturing and inter-socket interconnects.

We evaluate a NIC bandwidth up to 500Gbps, because our experiments showed C-4 was not able to reach its CPU IOPS bound before saturating NIC bandwidth for non-minimally sized KV items. Such per-server network bandwidth will be available in the near future, given the current availability of 400GigE products [208] and sufficient I/O interconnect bandwidth (e.g., an AMD EPYC 7763 has 128× PCIe 4.0 lanes, delivering 2Tbps of I/O bandwidth).

KVS Software. We use the MICA in-memory KVS [166], starting from the eRPC project's implementation [128, commit 1bfc7ec], porting it to NEBULA and ARMv8 for simulator compatibility, and adding C-4's compaction layer, requiring only 105 lines of code. All experiments use a 64-thread instance with a 819MB dataset of 1.6M items having 16B keys and 512B values in 1M hash buckets. We employ a workload generator which generates client requests at a configurable arrival rate, a Zipfian popularity skew with exponent γ, and write fraction f_{wr}, using a Poisson arrival process. We measure all latencies server-side using cycle-accurate timestamps inside the simulator. Latency measurement begins when the request's first packet arrives at the NIC, and ends when the corresponding response's last packet leaves the NIC.

Evaluated Configurations. We study six different system configurations to dissect C-4's benefits:

1. **Baseline**: The unmodified MICA KVS, running on the NEBULA stack and using CREW concurrency control.
2. **EREW**: Same as Baseline but using EREW.
3. **RLU**: A modified version of MICA that uses the RLU framework [180] to provide concurrent readers *and* writers without read-side locking, thus representing an improved version of CRCW. RLU uses commit-deferral with degree 16.
4. **MV-RLU**: An improved version of RLU that uses true multi-versioning rather RLU's restricted dual-versioning, granting further writer concurrency [146]. MV-RLU also adopts the `ORDO` primitive to remove global logical clock contention [138] for improved multi-core scalability.
5. **d-CREW**: Uses the same KVS as Baseline, and adds C-4's requisite extensions to support d-CREW. Targeted towards WI_{uni} workloads.
6. **Comp**: Enables C-4's software compaction support, while keeping the static CREW concurrency policy. Targeted towards RW_{sk} workloads.

Performance Metrics. We evaluate C-4's performance in terms of maximum throughput under a 99th percentile latency SLO. We set a target tail latency of 10× the average request service time [54, 126, 150, 223, 250]. As write compaction exposes a tradeoff between the duration of a compaction window and the performance benefit of C-4, we also show a slightly

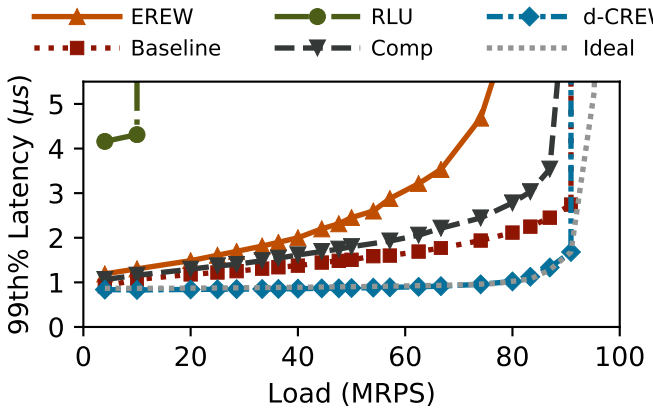


Figure 8.4: KVS throughput under SLO with uniform key popularity and f_{wr} = 50%. The vertical axis terminates at the KVS' SLO.

relaxed 20× SLO in the results for write compaction (see §8.4.3).

8.4.2 Dynamic Write Partitioning

We begin by evaluating the tail latency benefits of C-4's d-CREW policy. Figure 8.4 compares all of §8.4.1's system configurations, except for MV-RLU, using a WI_{uni} workload with uniform key popularity and f_{wr} = 50%. We also plot baseline CREW with a read-only workload, labelled "Ideal" due to its maximal load balancing flexibility and zero software synchronization.

Only d-CREW closely tracks Ideal up to 91MRPS, because it provides the greatest load-balancing flexibility, only falling back to the baseline's behaviour if there is a true write-write conflict. In contrast, all other systems incur excess queueing as load increases. RLU can only support 10MRPS under a tight 10× SLO, due to its costly `commit` process to write-back logged values – writes that promote logs run for 10–20µs despite RLU's commit deferral, inducing deep queues that lead to SLO violations. EREW fares better, delivering 76MRPS under SLO, but only reaches 80% of Ideal's throughput because it lacks load-balancing support. Although EREW eschews all software synchronization, static partitioning results in excess queueing and suboptimal performance compared to the systems supporting load balancing.

Our results confirm the intuition that write compaction is ineffective for WI_{uni} workloads. In fact, Comp performs 4MRPS worse than the baseline, because each thread incurs the additional cost of scanning its request queue for potential writes to compact, which is usually

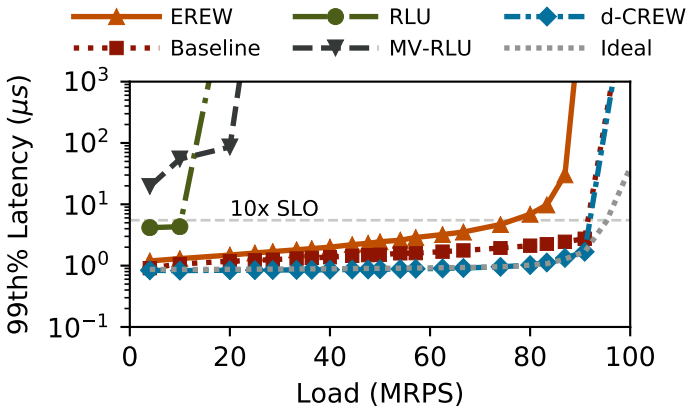


Figure 8.5: KVS throughput under relaxed SLO, using the same workload as Figure 8.4.

fruitless in WI_{uni} workloads. Such overheads grow at high loads—seen as a widening gap between Comp and Baseline (CREW)—due to more entries to scan in deeper queues. The best-performing systems, CREW and d-CREW, achieve the same throughput under SLO (91MRPS), with d-CREW achieving a 1.3× 99th% latency reduction compared to CREW, by relaxing unnecessary queueing—essentially approaching the superior queueing model of an Ideal system without the associated costs of software synchronization. Our simulations closely track the expectations set by our queuing model in §5.3.2.

To better compare the systems using multi-versioning to those that do not, Figure 8.5 shows the same results as before, substituting MV-RLU for Comp. We run the same workload as Figure 8.4, and show both a tight SLO of 10× MICA's average service time, and a relaxed SLO of 100µs. Note the logarithmic y-axis increases the highest latency shown to two orders of magnitude larger than Figure 8.4. MV-RLU cannot even meet the tight SLO at 4MRPS because its process to clean up versions involves long version chain traversals, and complex garbage collection algorithms using watermarks. However, the true multi-versioning in MV-RLU shows benefits when a relaxed SLO is acceptable, providing a 2.5× performance increase over RLU.

We measured that the KVS deployments using the RLU and MV-RLU versioning techniques exhibit mean latencies that are 1.36× and 1.58× higher than the non-versioned baselines. Therefore, the best performance that can be expected for software optimizations that address the long-tail events during version cleanup would be between 56-66MRPS. Our experience porting and studying a KVS with these frameworks indicates that the common impediment for

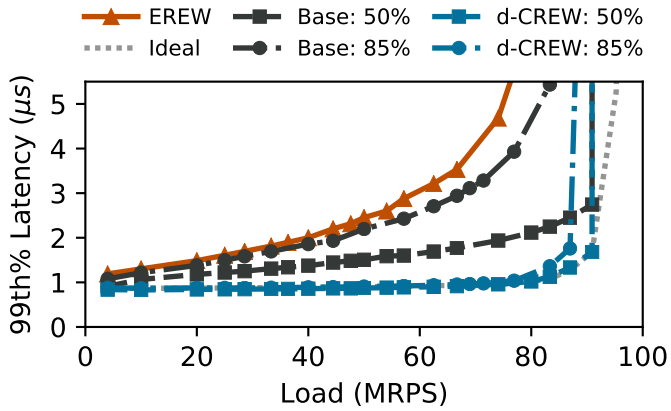
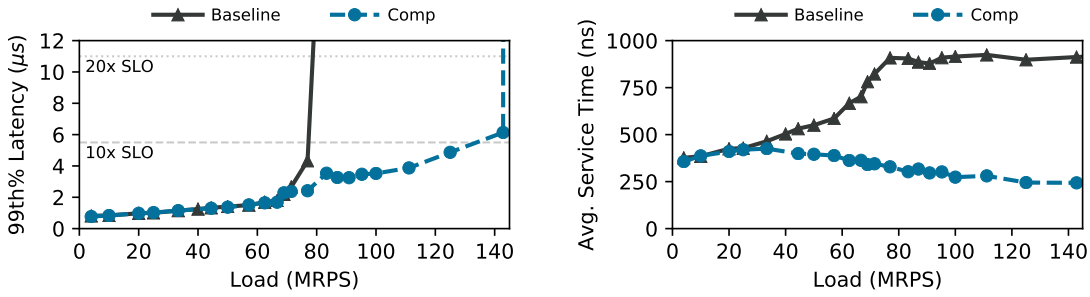


Figure 8.6: Comparison of KVS throughput under SLO with varied f_{wr}, using uniform key popularity.

both techniques is simply the software overhead of version management, which is also in line with observations from the original authors of these frameworks. For example, RLU's designers show that the software overheads of object duplication upon writing to a hash table leads to a 2× performance overhead compared to a non-duplicating implementation [180, §4.4], and therefore such techniques primarily grant benefit when it is possible to increase concurrency for read-mostly workloads. We conclude that for write-intensive KV workloads, it is unlikely that such techniques can close the 3× performance gap even to the EREW baseline, which widens to 3.6× for d-CREW.

Increased Write Fractions. Figure 8.6 shows the throughput under SLO of the baseline (CREW) and d-CREW systems, with a WI_{uni} workload as f_{wr} increases from 50% to 85%. We show a single line for EREW because it is insensitive to the workload's f_{wr}. As we predicted using our queueing simulator (see §5.3.2), the baseline policy's load-balancing potential diminishes with greater f_{wr}, approaching EREW and translating to increased 99th% latency and reduced throughput under SLO.

In contrast, d-CREW's benefits increase with higher f_{wr}: for 85% writes, baseline CREW cannot balance the vast majority of the requests, leading to a reduced peak throughput of 83MRPS and 5× higher 99th% latency compared to Ideal. These results also closely track our queueing model, which predicts 4.5× excess 99th% compared to Ideal, and a corresponding throughput loss of 9%. In contrast, d-CREW nearly matches Ideal's performance until its CPU saturation



(a) Throughput under SLO.

(b) Hottest thread's average service times.

Figure 8.7: System performance comparison for a RW_{sk} workload with $\gamma = 1.25$ and $f_{wr} = 5\%$.

point. Overall, d-CREW provides 87MRPS under SLO compared to CREW's 83MRPS, and 3.1× lower 99th% latency at a higher absolute load. Under a load of 90MRPS, the Locality-Aware Load Balancing Table (LLBT) has an average of 30 active entries with $f_{wr} = 50\%$ and 52 with $f_{wr} = 85\%$, confirming our simple estimate in §8.3.2 for the LLBT's limited size. We measured a maximum LLBT size of 64 and 90 for $f_{wr} = 50\%$ and 85%, respectively.

8.4.3 Software Write Compaction

We now evaluate C-4's write compaction support, targeting RW_{sk} workloads. Figure 8.7 compares baseline CREW and C-4 with write compaction enabled, for a highly skewed workload with $\gamma = 1.25$ and 5% writes. Despite the workload being read-dominated, Figure 8.7a shows that the baseline saturates at 76MRPS, because a single thread is overloaded by writes to a single partition. In contrast, with write compaction, C-4 scales to 125MRPS under a 10× SLO, and 142MRPS with a relaxed 20× SLO.

To further explain these performance gains, Figure 8.7b plots the average on-core service time of requests assigned to the hottest thread. As load increases, the baseline's service time grows exponentially because of increased time to access cache lines for the partition's version number and corresponding data. At 76MRPS, the hottest thread's service time increases by 2.4× to 908ns, and remains roughly constant beyond 80MRPS because the network stack's flow control mechanism begins rejecting incoming requests as the KVS saturates. The same trend also occurs for the rest of the threads, whose average service time grows by 1.6× because the KVS' hottest items are repeatedly invalidated from their L1 caches due to frequent writes.

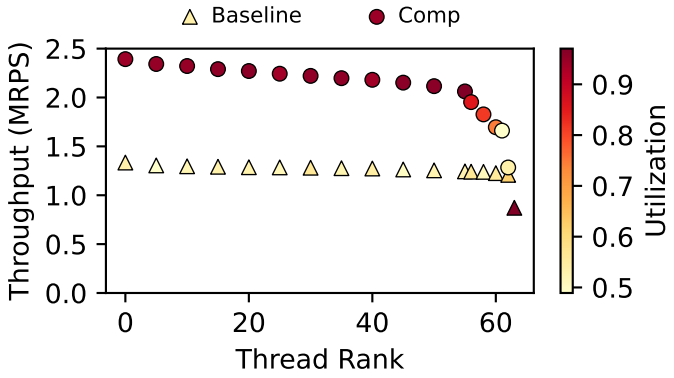


Figure 8.8: Per-thread throughput, using the same RW_{sk} workload as previously ($\gamma = 1.25$, $f_{wr} = 5\%$).

In contrast, C-4's write compaction policy inverts the aforementioned trend: as load increases, the hottest thread's service time decreases because it directly mitigates the overheads associated with contention between the single writer and multiple readers. Beyond 40MRPS, Figure 8.7b shows that the hottest thread begins to open compaction windows and effectively batch writes, resulting in an average service time reduction. Therefore, lock and cache line contention does not increase despite higher offered load, reaching a minimum service time of 243ns. To compare these results to our compaction performance model in Equation (5.2), we measured a 3.7× service time reduction for the hottest thread with write compaction, where our model predicts 3.9×. We attribute the difference to software overheads in managing compaction window metadata, such as reading the hardware system timer and checking on each request for an impending SLO violation.

Finally, Figure 8.8 compares the throughput and core utilization for a subset of the KVS threads, sorted by decreasing throughput, taken at 76MRPS for the baseline and 125MRPS for C-4.[2] In the baseline system, both throughput and utilization are roughly uniform across threads, with all threads except the overloaded writer attaining ~ 1.28MRPS. The overloaded writer (Figure 8.8's rightmost data point) attains < 1MRPS at near-maximal utilization due to its inflated service times (see Figure 8.7b).

With C-4, both reader and writer threads attain drastically higher throughput. The 3.7× service time reduction for the hottest thread translates to lower relative utilization, in contrast to

[2]We show a subset of the threads for readability. Omitted threads fall between the displayed points.

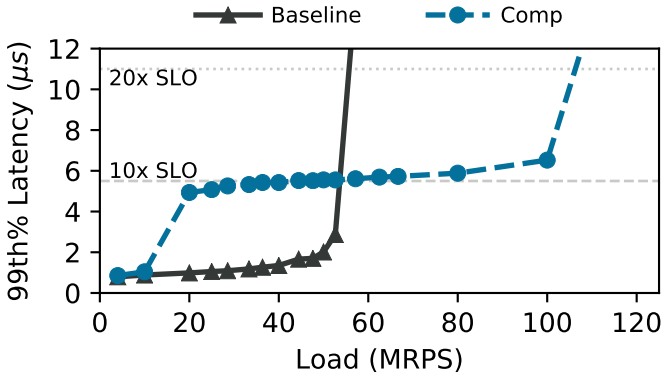


Figure 8.9: Performance for RW_{sk} with $\gamma = 0.99$, $f_{wr} = 50\%$.

the baseline where only the hottest thread utilizes its core while others are idle. This trend is visible by comparing the utilization of the hottest writer thread in Figure 8.8. With compaction enabled, the hottest writer's throughput improves from 0.92MRPS to 1.66MRPS and sees its utilization fall to 47% even at higher system load, because it handles writes more efficiently. Other threads with ranks between 55–63 show the same trend in C-4: rapid write compaction creates idle periods where other requests could be handled.

We verified that C-4 saturates not because of write overload, but due to reads saturating the threads in Figure 8.8, which are already serving > 2.3MRPS and operating near 100% utilization. Therefore, our implementation could scale to modestly higher throughput by harvesting the extra cycles made available by C-4's write skew absorption via compaction. Modifications to the underlying NEBULA stack for such purposes are possible, but we leave such optimizations for future work.

Figure 8.9 evaluates a workload with $\gamma = 0.99$ and 50% writes, used by prior work [123, 165, 166] due to its challenging nature. Due to less cache-resident hot data, the baseline only attains 56MRPS under SLO, compared to 76MRPS in Figure 8.7a. However, the single overloaded writer thread is the bottleneck for both workloads. In contrast, C-4 attains 58MRPS under an identical 10× SLO, and 100MRPS under a 20× SLO. C-4's drastic jump in tail latency beyond 10MRPS is because compaction events immediately form the 99th% of this workload. However, increasing load from 20MRPS to 80MRPS only causes the 99th% to increase by $\sim 300ns$, indicating that the compaction windows at lower load are often opened but collect

very few writes. This behavior is an artifact of our implementation, which only closes a window upon an impending SLO violation; a software modification to close compaction windows early under low load would resolve the early jump in 99th% latency.

8.4.4 Sensitivity Analysis

Finally, we evaluate the impact of varying MICA's key and value sizes on the performance of C-4's write compaction. We compare our previous results using the same workload in Figures 8.7 and 8.8, to a workload with the same (γ, f_{wr}) using 8B/8B KV pairs (called Tiny) and one using 16B/128B pairs (Medium). In all workloads, we scaled the number of KV pairs to keep the same total dataset size, so that the same overall data fraction is cache-resident. Table 8.3 compares the CREW baseline and C-4 in terms of maximum throughput under 10× SLO, as well as the speedups of the KVS' hottest compacting thread and the remaining threads.

As the item size shrinks, the baseline system's performance increases by 1.8× for Medium and 3.5× for Tiny items, because threads spend less time per request accessing data. C-4 benefits from these service time reductions to a lesser degree (e.g., by 1.5× for Medium items) because threads only write the underlying data structure once per compaction window, and compact all other requests locally, which is less sensitive to item size.

The hottest-thread speedup granted by C-4 drops with smaller KV pairs, as the baseline system's service times approach the latency of appending writes to an open compaction window. As the speedup granted to the hottest thread reduces, the impact on readers becomes the critical performance driver. The hottest thread only attains 1.1× speedup with Tiny items because our compaction implementation trades off increased compute latency for reduced coherence activity. Compaction has higher compute requirements because it requires many instructions to create the requisite metadata and compact its private logs, whereas the baseline can write a single 8B KV item with a single `store` instruction. In summary, C-4 provides robust throughput benefits across item sizes, improving throughput under SLO by 1.4× and 1.33× for Tiny and Medium items, respectively.

Conclusions. Our evaluation shows that despite the necessity of limiting synchronization events through partitioning writes to KVS, it is possible to attain *nearly optimal* tail latency by

	Tput. @ SLO (MRPS)		Thread Speedup	
Key/Value Sizes (B)	Base	Comp	Hottest	Others
8/8 (Tiny)	266	363	1.1×	1.3×
16/128 (Med)	142	190	1.3×	1.3×
16/512 (Lg)	76	125	1.6×	1.6×

Table 8.3: Impact of item size on write compaction.

leveraging runtime knowledge of which KVS partitions are being actively updated. Furthermore, even when true write conflicts are frequent, simple software optimizations are possible to drastically boost throughput while maintaining strong consistency guarantees.

8.5 Chapter Summary

Enhancing NIC-driven load balancing algorithms with application-level metrics has proved to be a fruitful direction. In this chapter, we have shown that adding a TLB-sized table to a NIC's load balancing stages is sufficient to realize dramatic improvements in server throughput while maintaining tail latency. Specifically, when a microservice's functions result in I$ thrashing, affinity-based load balancing reduces instruction stalls by up to 18×, improving the performance of the function's µs-scale RPCs by up to 2×. Similarly, when a KVS microservice is exposed to a write-heavy workload, C-4 reduces 99th% latency by up to 5× and improves throughput by up to 1.7×. We believe that such policies are ideal for inclusion in next-generation server NICs to increase their effectiveness in handling emerging microservice workloads.

Part III Related and Future Work

9 Related Work

In this chapter, we discuss the rich field of related work that this thesis either improves upon, or is inspired by. The chapter is organized by contribution: §9.1 discusses other works that progress towards cache-resident RPC traffic, §9.2 touches on design and implementation tradeoffs for RPC protocol acceleration, and §9.3 covers locality-based load balancing.

9.1 Towards Cache-Resident Network Traffic

9.1.1 Leaky DMA and Bandwidth Interference

ResQ [256] encounters the "leaky DMA" problem, which is similar to the bandwidth interference problem we study. ResQ's authors observe that when deploying co-located NFV workloads using DPDK, the LLC space required by DDIO exceeds its default limit, and memory traffic multiplies. ResQ ameliorates memory traffic by statically limiting DPDK's buffer allocation to 10% of the LLC. In contrast, our work establishes a mathematical bound on the number of required buffers, based on queueing theory. While we share the intuition that limiting outstanding buffers can reduce excessive memory traffic, NEBULA targets hardware-terminated transports and demonstrates that mere KBs of buffering space are sufficient to handle µs-scale RPCs. NEBULA also maintains inter-core load balancing, a factor beyond ResQ's scope.

The leaky DMA problem is also observed in the context of the Shenango runtime scheduler for latency-sensitive datacenter workloads [212]. Shenango foregoes zero-copy I/O to hand

buffers back to the NIC as soon as possible, increasing LLC reuse by DDIO. In contrast, NEBULA maintains zero-copy I/O, because we find that all useful packet buffers can be accommodated in the LLC when operating under a tight SLO.

Disk|Crypt|Net [177] makes the observation that the asynchronous design of Netflix' specialized I/O stack results in file pages and encryption/socket buffers thrashing a server's LLC, leading to excessive memory bandwidth and high CPU utilization. Their solution, Atlas, is a software pipeline which improves LLC reuse by performing disk I/O, encryption, and network I/O synchronously. Our work operates in a similar regime where network I/O activity creates excessive memory bandwidth, particularly with high connection counts, but we focus on latency-critical µs-scale RPCs rather than streaming traffic that is primarily concerned with throughput.

Characterizing and Optimizing Intel DDIO

The rapid growth of NIC bandwidth has resulted in many research works that characterize, reverse-engineer, and propose optimizations to Intel's DDIO technology. Multiple published works from Farshin et al. have studied Intel DDIO in depth in the context of optimizing low-latency networked applications (e.g., software NFV) [75–77]. CacheDirector [76] improves RPC latency by modifying DDIO to steer the header of each network packet into the LLC tile closest to the core that will process the packet. We go further by steering the whole packet all the way into the core's L1 cache. In the past, placing network data in L1 caches has been avoided due to pollution concerns, which NEBULA addresses by steering only a single RPC at a time.

Additionally, they closely study DDIO platforms and identify a similar problem to NEBULA – that application code and/or data can conflict with incoming network traffic placed in the server LLC by DDIO [77].[1] We distinguish our ideas in three ways. First, their discussion is primarily in the context of minimizing dropped packets, and maximizing achievable network throughput. Second, we remark that the two works differ in their system design assumptions. NEBULA considers a system unrestricted by existing DDIO implementation details, where the

[1] We note that NEBULA and their work appearing in the 2020 Usenix Annual Technical Conference are concurrent publications, with NEBULA appearing just one month earlier.

NIC can make use of the entire LLC in a general fashion, where their design targets improved performance on today's platforms.

Third, their work argues that statically limiting the amount of network buffering state is *not* a scalable solution to future NIC rates, because the amount of buffering needs to cover delays in incoming packet processing. The authors of IAT arrive at the same conclusion after their study of DDIO, attributing the throughput losses observed when shrinking NIC RX buffers to application-level tail events and load imbalance between cores [273]. Although our insight that an application's SLO allows queue depths to be statically limited seems to be in direct contradiction with their work, the reconciling factor becomes clear when considering that our work is concerned with hardware-terminated protocols. Using a hardware-terminated stack implies that packet processing delays are *extremely* limited in duration (e.g., a few tens of ns if RPC buffers happen to experience a longer-latency cache access), and our work does indeed provision enough buffer state to cover not just protocol delays, but also application tail-events (see §4.2.1).

A follow-up work from Farshin et al. proposes PacketMill, a deeply optimized software packet processing stack that identifies packet copying and type conversion in between the DPDK and user-level packet representations (e.g., in the Click framework) as a critical source of inefficiency [75]. Their proposed solution, *X-Change*, embeds packet modification and type conversion functions directly into the DPDK stack, eliminating copies and thus increasing application-level cache locality. X-Change is largely orthogonal to our work, because it operates at the API between the software-driven transport protocol (e.g., DPDK or Linux) and the application-layer framework for software NFV. A combination of NEBULA and X-Change would simultaneously allow L1 cache residency and the capability to deliver packets directly to NFV applications with no copying or conversion.

9.1.2 RPC Scalability Over Connected Transports

HERD [129], FaSST [131] and FaRM [65] all propose optimizations in order to alleviate the scalability issues of InfiniBand's connected transports. FaSST uses solely unconnected datagram transports to reduce the number of Queue Pairs that must be cached in the NIC [131], while

FaRM accepts the inherent performance loss of some connection sharing [65]. Storm [201] shows that at rack scale (i.e., up to 64 nodes), the newest generation of ConnectX-5 NICs have significantly improved in scalability, and hence designs a transaction API prioritizing RDMA operations which require connected transport. Our work targets datacenter-scale deployments and applies to any software with µs-scale RPCs.

eRPC [132] improves buffer scalability by using multi-packet receive queue descriptors introduced in Mellanox ConnectX-4 NICs. These descriptors keep NIC-resident state constant with the number of connected nodes. However, eRPC's server-side buffering state still scales with the number of connections, as it allocates memory for each pair of connected threads [132, §3.1]. We show that although server memory is plentiful, µs-scale RPCs require buffers to be kept to LLC-resident sizes in order to avoid memory bandwidth interference and meet tight SLOs.

The problem of transport protocol state overflowing beyond a particular memory size has also been observed by Eran et al. in the process of building a driver for network-attached accelerators [69]. Limited on-chip storage state for accelerators is particularly problematic, because provisioning more memory requires the architect to trade off chip space that would be used for enhancing the accelerator's compute capability. FlexDriver specifically targets the amount of on-chip storage required to store transmit/receive buffers and metadata rings, and leverages the insight that the accelerator's network driver can store that metadata in a compressed, internal representation which is different from what the NIC's vendor has defined [69, §5.2]. The insights presented in FlexDriver are largely orthogonal to NEBULA, because our baseline NIC model shares its virtual memory with the host server and virtualizes all its transport protocol queues in that memory. On the contrary, the FlexDriver model operates in a regime where each accelerator possesses its own NIC-invisible SRAM. One could in principle design a server where NEBULA operates between the NIC and CPU cores, and FlexDriver operates between the same NIC and PCIe-attached accelerators.

9.1.3 Network-Compute Co-designs

The importance of KVS in datacenters has resulted in multiple proposals for KVS-optimized hardware [165, 167]. SABRes [55] leverages coherent on-chip NIC integration to introduce a NIC extension for atomic remote object access, alleviating versioning overheads using one-sided RDMA reads for low latency. Li et al. [165] study MICA's performance characteristics and propose a bespoke CPU for KVS whose parameters we adopt as our simulation baseline. Their work also observes the primacy of in-LLC buffer management, and empirically sizes the LLC based on simulation to minimize miss rate. They do not observe the same DRAM bandwidth contention as we do, because their use of UDP means the amount of network state in their system would not scale with the number of communicating servers [166]. Achieving similar throughput with an order of magnitude lower latency, as we attempt to do, requires hardware-terminated network stacks and brings back the challenge of scaling dedicated per-endpoint state. NEBULA therefore begins with a hardware-terminated protocol, and demonstrates that packet buffering state should be sized based on SLO rather than system scale.

At the time of writing, Google's 1RMA [244] is the most similar published work to NEBULA. The designers of 1RMA wish to deploy their datacenter-scale storage and data serving systems on top of a hardware-terminated transport protocol like RDMA, and realize as we do that scalability to thousands of connected nodes is a critical problem for such transports. Among other design features (e.g., encryption and co-designed congestion control), 1RMA ensures $< 50\mu s$ transfer latency using a nearly identical fail-fast approach to NEBULA. Both systems use server-generated NACKs that are returned to the client in the case of excess queueing, allowing them to react via arbitrary policies (e.g., retry after exponential backoff) [244, §4.1]. We differentiate our contributions by noting that 1RMA focuses on remote memory reads/writes rather than RPCs, meaning that the concept of an SLO is not a primary concern in their work. Therefore, the concept of an acceptable queue length is not explicitly defined – the authors of 1RMA focus on network fabric RTT rather than end-to-end application time [244, §6]. In NEBULA, we define a server's acceptable queue length based on its application-level SLO, because we focus on RPCs rather than one-sided RDMA-like remote memory operations. We note that 1RMA and NEBULA are concurrent publications, with NEBULA appearing for publication in June 2020 and 1RMA in August 2020.

NanoTransport [23] implements a transport protocol for low-latency message-based RPCs, using programmable NICs supporting the P4 language [38]. Their implementation of the Homa and NDP RPC transports require reassembly and buffer management modules, which are achieved in extremely similar fashion to NEBULA. We take their work as further evidence that implementations of RPC reassembly and buffer management *are* feasible to accomplish in dedicated hardware.

Daglis et al. [53] studied the on-chip latency implications of the VIA/RDMA network programming model [66], and proposed NIC decomposition and passing messages between NIC components to avoid multi-hop coherence interactions. We have similar insights regarding the on-chip data path of RPC payloads under µs-scale SLOs, and propose a solution that uses the NIC's role in RPC dispatch to accurately prefetch payloads without cache coherence modifications.

9.1.4 Latency-optimized Systems Software

The need for low latency has led systems designers to aggressively limit queueing in the transport and RPC protocol stacks themselves [10, 147]. Kogias et al. [147] also observe that limiting server-side queueing is critical for µs-scale RPCs, and use TCP flow control to limit the number of requests per connection based on the application's SLO. Breakwater [49] shares much of its motivation and design with NEBULA, because it targets overload control for systems handling µs-scale RPCs with thousands of communicating clients, and employs Active Queue Management (AQM) to drop requests that would incur excessive server-side queueing.[2] The critical difference from NEBULA is that Breakwater employs a credit-based RPC flow control scheme where the server explicitly issues grants to RPC clients, similar to emerging schemes for congestion control [193, 211], and dropping requests based on AQM is the fallback rather than the primary mechanism. Breakwater's authors primarily avoid AQM because of the resulting server-side overheads in software-terminated protocols [49, §2.2]. Since NEBULA performs buffer management for hardware- rather than software-terminated protocols, it is feasible to use AQM as a primary mechanism for admission control, eliminating

[2] As before with 1RMA, we note that NEBULA and Breakwater were concurrent publications, with NEBULA appearing publicly in June 2020 and Breakwater in November 2020.

the need for per-endpoint RPC flow control as a side effect.

Thomas et al. [254] observe that at 100Gbps+, packet inter-arrival times drop below DRAM latency, and that applications performing memory accesses will inevitably backpressure the NIC and lead to dropped packets. They propose CacheBuilder, a slab allocator using Intel cache partitioning technology [145] to guarantee the application's dataset is LLC- rather than memory-resident. CacheBuilder therefore only benefits applications with LLC-resident datasets, whereas NEBULA considers SLO-constrained applications with memory-resident datasets.

9.1.5 Hardware Support for Packet Placement

Other Approaches to Steer RPC Payloads

NEBULA employs a prefetch hint mechanism to steer the payload of an incoming RPC to its target core. Payload prefetches differ from regular cache accesses that are triggered by front-side demand accesses, because they are initiated by the cache itself. Our work chooses to use prefetch hints because payload prefetches can be transformed into regular front-side initiated operations, keeping both the cache controllers and coherence protocol unmodified. Intel's DCA mechanism also leverages prefetch hints to load data into the LLC [151]. Other mechanisms such as direct injection (which is used by DDIO's write-allocate/update policy [119]) and Curious Caching [48] have been proposed to support back-side initiated operations. NEBULA could be trivially adapted to use these mechanisms if supported by the cache controller, because every step after NEBULA communicates the incoming RPC's payload address and size to the NIC FE uses the existing cache hierarchy.

The NanoPU is a recent full-system proposal which goes one step further than NEBULA, arguing that RPC traffic should transit dedicated on-chip wires into the CPU cores' register files, rather than the existing on-chip network and the L1 caches [108]. Assuming a system where such radical changes to CPU microarchitecture are desirable, the authors themselves write that such changes are most beneficial to applications with nanoseconds of on-CPU time [108, §4]. NEBULA instead chooses to maintain traditional CPU and cache microarchitecture, re-using existing resources and lowering implementation complexity. Under the assumption of further

software decomposition to nanosecond-level RPCs, we believe the destination of RPC traffic is an interesting question to revisit, particularly in the context of large server chips with hundreds of cores where on-chip network diameter is of primary importance [174].

Network Packet Decomposition

In our work, we only consider RPCs whose payloads will be either partially or fully accessed by the CPU cores that run them. However, multiple concurrent works have observed that a second class of applications exists that only transform a packet's *header metadata* and leave the payload untouched [88, 219, 235]. Therefore, it is possible to decompose an RPC into its header and data, and have the NIC treat them differently. For example, NFSlicer [235] and nmNFV [219] both propose packet decomposition for NFV workloads which exclusively operate on headers. In their designs, the NIC forwards the packet's header to the CPU cores and keeps the payload in its own dedicated off-chip memory, which is then re-spliced with the modified header after the CPU completes the NFV task. PayloadPark proposes a similar approach but uses programmable switch logic instead of SmartNICs [88].

Our work is orthogonal to such approaches, and we expect the two would have an additive effect. In particular, NEBULA's NIC-to-core steering would still reduce the latency of header processing by virtually ensuring an NFV applications' memory accesses hit the L1 cache. Such approaches based on "packet slicing and splicing" [235] would most clearly benefit microservices in the business logic tier of an online service, because we have observed that many of these microservices simply receive and create new RPCs, placing them in the category of applications that only process RPC headers.

9.2 RPC Framework and Hardware Enhancements

9.2.1 Other Systems Targeting RPC Acceleration

The prevalence of RPC-connected microservices in the datacenter has led to a plethora of proposals to use production NICs, equipped with FPGAs or their own CPU cores, to accelerate application-level tasks such as the RPC layer. Dagger [160] is the only other work we are aware of which proposes to offload the full RPC layer to hardware. They target the integrated FPGA

in an Intel Broadwell platform and build a customized RPC layer inspired by the Thrift [19] software stack. Similarly, NICA proposes a programming model and runtime to accelerate application-level tasks on FPGA-enhanced NICs [70], citing message (de)serialization as a potential application. Neither Dagger nor NICA currently supports the underlying modules of production RPC layers we describe in §7.2, because they lack support for the header and payload manipulation modules in production Thrift [160, §4.5]. Dagger and NICA share our use of hardware-terminated transport protocols, but still leave header/payload manipulation (and therefore RPC dispatch) to the host CPU.

Adding support for manipulation tasks to designs based on commodity NICs would require co-design with the host's DMA engine, because common-case objects cannot be deserialized on the NIC and then DMA'ed to the host without rewriting all the object's pointers. Although it may be possible to realize such a design in the future, as argued by Wolnikowski et al. [268] and Raghavan et al. [225], the hardware required will likely be similar to CEREBROS, and therefore we believe an on-chip design is more logical. Additionally, executing the RPC stack on FPGA-enhanced NICs will inevitably incur the overhead of transferring objects between the FPGA accelerator and the host's CPU cores. Dagger and NICA contain a highly customized communication stack to transfer objects while minimizing the latency of the FPGA-to-host interconnect—another form of offload overhead that CEREBROS does not incur due to its use of an on-chip NIC.

Due to the prevalence of data (de)serialization frameworks in hyperscale datacenters [87, 133], Google has recently open-sourced the implementation of a hardware accelerator for Protocol Buffers [136].[3] There are two key differences between their design proposal and what we achieve with CEREBROS. First, their design specifically targets the header and payload manipulation modules of the RPC stack, similar to the RPA design of Optimus Prime [275], leaving the dispatch module to the CPU. They justify their work by claiming that in Google's entire server fleet, "we find that only 16.3% of deserialization cycles are from the RPC stack and only 35.2% of serialization cycles are from the RPC stack. This challenges the common assumption that a `protobuf` accelerator should be placed on a PCIe-attached NIC. Instead,

[3] Note that their work and CEREBROS were concurrently published papers, appearing in the exact same conference (MICRO'21).

it is clear that other serialization and deserialization users (e.g. storage users) must be accounted for when deciding where to place a `protobuf` accelerator in the system" [136, §3.4]. However, such a system design creates inefficiencies for RPC use-cases, because it assumes that the server's transport protocol endpoints are at the CPU cores, regardless of whether the transport is software- or hardware-terminated. Therefore, locality-aware load balancing becomes implausible for µs-scale RPCs.

Although purely transport-level load balancing can be implemented over RPC-oriented protocols such as Homa [193, 213] or Snap [178], locality-aware load balancing can only take action *after* the RPC's de-serialized fields are already known. Hence, the workflow of incoming RPCs would proceed as follows: (1) RPCs arrive at the NIC, and are load-balanced at the transport level. (2) After the chosen CPU core receives the RPC and de-serializes the header, it can choose to implement locality-aware load balancing in software or with further hardware. (3) Potentially, the RPC must now be assigned to a different core based on the chosen policy. Whenever Step (3) takes place, the RPC is handled inefficiently because of the additional overhead required to shuffle requests between cores, as well as the fact that the core which received the transport-level message has its L1 cache polluted by RPC payload data which it will never actually consume. In contrast, we argue that RPC-centric server architectures should handle such tasks transparently, with a fully NIC-interfaced RPA. Our design supports both locality-aware load balancing *and* NEBULA's support for steering RPC payloads to the L1 caches of the cores which will actually process the RPCs, rather than relying on request reassignment between cores.

Customized RPC frameworks such as Cap'n Proto [143] or FlatBuffers [86] use a pre-flattened RPC message format, which inherently removes the aforementioned challenges with pointer-based objects. However, such frameworks sacrifice object mutability, generate larger wire-format objects, and experience higher latencies during costly object creation [225]. We believe these trade-offs are particularly burdensome for microservices with many nested RPCs because message objects are often modified on every nested call. CEREBROS can even benefit such frameworks by accelerating header parsing and with locality-aware load balancing.

Optimus Prime [275] introduces a data transformation accelerator (DTA) to perform RPC

payload manipulation at rates matching a high-end server NIC. CEREBROS builds on Optimus Prime's DTA architecture and transformation schema to perform both payload and header manipulation, and shows that a further 23× reduction in RPC processing time is possible compared to merely targeting payload manipulation. CEREBROS also performs dispatch in hardware, thus eliminating CPU involvement altogether.

9.2.2 Reducing CPU-Accelerator Control Overhead

The work which is most closely related to CEREBROS' need to limit RPC offload overheads is a recent proposal for "autonomous NIC offloads" by Pismenny et al. [218]. Their work is motivated by the fact that even though today's NICs contain support to offload specific transport processing operations (e.g., Large Segment and Large Receive Offloads for TCP), it is not possible to offload L5 protocol operations to NICs without performing the entire L4 transport protocol on the NIC as well. Implementing commodity protocol stacks like TCP or UDP on fixed-function hardware has generally been rejected for security and debuggability reasons [253], and therefore NICs are prohibited by design from integrating offloads for L5 protocol functionality. Our work in this thesis targets a different deployment scope – where the latency-critical nature of datacenter applications makes it feasible to offload the transport protocol to dedicated hardware. We note that such hardware-offloaded protocols have already reached production deployments [44, 93, 244]. Additionally, we note that the production RPC stacks we studied do not meet the requirements for autonomous offload. In particular, they require that L5 protocol operations *never add or remove bytes from the transport stream* [218, §3.1], which is not true for RPC stacks that use variable sized integer coding, field repetition, and optional values.

Shao et al. have observed that data movement between CPUs and accelerators limits performance, and propose optimizations to pipeline data transfers with computation [240]. Although such techniques could reduce some of the offload overhead we identify, they cannot eliminate it due to the complex pointer dependencies inherent to the messages in production RPC formats. M3-X provides support for accelerators to access OS services and communicate with rescheduled threads [24]. Systems such as Morpheus-SSD [258], GPUfs [241], and NVIDIA GPUDirect [205] address performance losses arising from CPU-mediated transfers between

peripherals, and all use peer-to-peer DMA to eliminate CPU time spent moving data through system memory. CEREBROS shares the motivation of removing the CPU from the accelerator's path of work, but operates at nanosecond timescales instead of milliseconds.

9.3 Application-Aware Load Balancing

9.3.1 Instruction Supply in Servers

The instruction supply problem is both well known and deeply studied for server workloads [6, 142]. Due to the drastic performance improvements available, a plethora of microarchitectural solutions exist to address instruction supply bottlenecks [18, 79, 140, 141, 154, 155, 226]. All of these purely hardware-based solutions share a common mechanism: storing and accessing prefetching metadata. For microservices with many functions or large working sets, the required metadata to cover their misses will outgrow the CPU's storage capacity and reduce coverage.

In §5.2.1, we argued that function state can thrash more than just the core's L1 I$, but also the prefetching metadata (e.g., the BTB in case of Boomerang and Shotgun) [154, 155]. Deploying affinity-based load balancing in the presence of an advanced prefetching proposal would actually increase the prefetcher's effectiveness when the CPU handles microservice deployments whose characteristics are sufficient to cause function thrashing. For example, Schall et al. propose Jukebox, a record-and-replay prefetcher for the instructions of lukewarm serverless functions [236]. Affinity-based load balancing would increase Jukebox' effectiveness because the prefetcher needs to fetch fewer overall cache lines, and thus can look further ahead into the anticipated future instruction fetch stream.

In contrast to purely hardware-oriented solutions for improved instruction supply, profile-guided solutions such as AsmDB [27] and I-SPY [144] trade off changing server microarchitecture with the need to collect and analyze datacenter-wide traces. Our work on affinity-based load balancing is done completely online and does not require datacenter-wide profiling or application recompilation.

9.3.2 Task Scheduling for Cache Affinity

Significant prior work has studied the possibility of scheduling system tasks based on their cache footprints. For example, Squillante and Lazowska's seminal work proposes six different practical scheduling policies to increase cache affinity [245]. Their Fixed Processor (FP) and Last Processor (LP) policies are in spirit very similar to what we have proposed and evaluated as static and dynamic affinity-based load balancing. However, our work considers a fundamentally different processor execution model – we consider a server running RPC tasks to completion, not a processor executing a multiprogrammed workload. The analytical model developed in their work would be a highly useful building block to estimate the overheads of reloading the I$ if an asynchronous RPC processing model is used, because such deployments imply that other RPCs can execute on a given core while a specific RPC is still processing.

Salehi et al. further formulated the problem of affinity scheduling to consider network protocol processing, and mentioned code affinity as a primary design concern [231–233]. In contrast to their work which targets improving the execution of parallel network protocol processing tasks with pseudo-uniform compute requirements, we focus on RPCs that have varying functionalities. Additionally, we make 99th% latency a prime goal for our affinity-based load balancing solution due to the SLO-bound nature of microservices, whereas their work focuses primarily on mean latency. Our conclusions about the prime importance of load balancing to minimize queueing delay agree with their empirical results showing that packet-stream affinity (equivalent to hard function partitioning) yields higher latency due to queueing, but improved maximum throughput (see Figure 5.1).

Cohort scheduling is a software-only technique that proposes to execute multiple similar *stages* of a server workload's computation together, improving cache locality for each stage [158]. In the context of RPC-dependent microservices, affinity-based load balancing is orthogonal to cohort scheduling for two reasons. First, forming a cohort fundamentally requires that certain operations are forced to block until the cohort is ready to execute, introducing application-level queueing delays. The type of queueing that we consider in this thesis is restricted to the input queues holding RPCs that are ready to execute. Therefore, it would be possible to deploy affinity-based load balancing to minimize input queueing as well as minimize cache misses

for the software layer forming cohorts. Second, cohort scheduling requires replacing a thread-based programming model of a system with a network of connected, asynchronous stages *which have scheduling autonomy*. Such a requirement implies that user-level scheduling support needs to be added, which has implications for RPC-to-thread mappings and thus the hardware-software interface in CEREBROS. We believe that adding such support on top of production RPC layers would be possible, with the requirement that a lightweight threading package such as Arachne [224] is used to limit the cost of context switching. Finally, as the compute time of each stage shrinks, as would be the case for µs-scale microservices, the overheads of creating the underlying closures and continuations used to represent stages will inevitably grow as a fraction of useful execution time, potentially necessitating the inclusion of hardware-support for task scheduling such as Carbon [251].

9.3.3 Application-Driven Scheduling Policies

The shrinking runtimes of traditional server applications into µs-scale tasks has begat a significant shift in design focus for systems scheduling research. Datacenter providers have already begun to specialize their network and scheduler implementations for latency-critical µs-scale requests. For example, Google's *Snap* microkernel provides user-level networking and a custom scheduling design that allows various packet processing tasks to be grouped into one of three scheduling modes [178]. Emerging studies have gone further, arguing for exposing application-driven scheduling hints to kernel space, thus allowing various policies to be implemented without requiring multiple kernel versions or fleet-wide upgrades [107]. Such works are also orthogonal to our proposal for locality-aware load balancing, because our goal is to improve a given server's theoretical optimum throughput via improved RPC locality rather than pursue system integration of existing load balancing policies into production systems.

iPipe proposes a scheduling algorithm to minimize tail latency for microservices offloaded to SmartNICs [172]. Their work assumes that applications are constructed according to the actor model, and proposes a formalized hybrid scheduling policy to determine whether to run incoming actors on the SmartNIC itself, or on the host. In a hybrid SmartNIC-host system, combining locality-aware load balancing with a scheduler such as iPipe's would provide performance benefits to actors.

Finally, recent work from McClure et al. has demonstrated a comparative analysis of the wide variety of scheduling policies that are implementable on existing and emerging NICs [182]. They conclude the following regarding the optimality of load balancing techniques: "In general, work stealing was consistently the best performing load balancing policy when given the same number of cycles as other approaches even as overheads and workload parameters vary... Ultimately, absent new hardware, work stealing is the best option for load balancing approaches among those we evaluated" [182, §4.2.1]. Therefore, we believe that the use of new hardware that supports locality-aware load balancing policies, such as the ones we have studied in this thesis, is the next logical step for systems architects to take to further improve server throughput for µs-scale RPCs. We also believe that an RPC-centric server architecture with support for locality-aware load balancing is a perfect platform for further scheduling policy research.

9.3.4 Improved KVS Designs

We group KVS systems into two categories: software-only or NIC-assisted. Representatives of the former include `memcached` [185], MemC3 [73], Masstree [176], and MICA [165, 166]. All of these systems reach their peak performance with R_{uni} or R_{sk} workloads—in fact, MICA's authors [165] propose using frequency scaling on the overloaded core to alleviate the static write imbalance bottleneck in RW_{sk} workloads. Frequency scaling alone is insufficient to absorb RW_{sk}'s load imbalance, but could be added to C-4 for even greater performance gains.

The Minos KVS targets improved assignment of items to cores based on their sizes, observing that requests for large items can consume orders of magnitude more CPU time than small ones [64]. Their insight is that queueing induced by large items can be avoided by reserving a specific core (or set of cores) to handle processing long-running requests, leaving the rest of the CPU cores to handle shorter requests for smaller items. Minos' solution can be viewed as a special-case predecessor of the Perséphone scheduler. C-4 solves an orthogonal problem to Minos, and we note that our baseline implementation of load balancing in hardware would implicitly solve load imbalance due to item size concerns – short requests are balanced around long requests regardless of the reason for the queueing delay. Additionally, in order to implement Minos, the authors need to give up the MICA's lightweight optimistic concurrency

control, and revert to spin-locks on each hash bucket [64, §4.2]. In contrast, C-4's d-CREW policy retains both lightweight concurrency control and load balancing even for write-intensive workloads.

NIC-assisted designs enhance the KVS with RDMA, FPGA, or programmable switch support. However, the vast majority of such designs do not perform well on write-intensive workloads. The two best performers in this space are NetCache [123] and KV-Direct [162] because dedicated hardware performs the entire KVS' functionality instead of using CPU cores. RDMA-enabled designs include FaRM [65], HERD [129], ccKVS [84], RackOut [200], and DrTM [45, 46]. The write imbalances we identify in §5.3.1 in a single-node context would be strictly worse in the multi-node distributed KVS settings targeted by these RDMA-assisted designs.

Emerging SmartNIC frameworks propose a split-KVS design, where a small amount of NIC memory stores the KVS' hottest items [70, 219], embodying the "small cache, big effect" principle [74]. Like NetCache [123], such designs focus on reads and are inefficient for the write-intensive workloads that C-4 targets (as shown by [70, Table 3] and [219, Fig. 16]), because writes are handled on a slow path to keep the CPU's main memory up-to-date.

LSM-based KVS like LevelDB [11] and RocksDB [186] optimize for write performance by buffering writes in memory before they are later flushed to storage. In contrast, our work targets in-memory KVS, whose throughputs are often orders of magnitude greater. Nevertheless, some similarities exist between state-of-the-art LSM-based KVS and C-4, particularly in our proposed design for in-memory write compaction of dependent writes to the same item(s). The authors of FloDB observe that despite the presence of an in-memory component to mask long storage latencies, the *data structures and algorithms* used to provide concurrent writes to that in-memory component simply do not scale to high degrees of concurrency [30]. Therefore, FloDB adds another level to the LSM data structure hierarchy – an even smaller in-memory buffer which can provide better update performance by supporting rapid random accesses. In C-4, we discuss an orthogonal concurrency bottleneck that occurs in systems where the in-memory buffer *is effectively the entire KVS*. Put another way, a hypothetical configuration of FloDB where the storage component is removed and the entire KVS operates with a single in-memory component would experience both bottlenecks we identify for write-intensive

workloads in §5.3.1, and could benefit from the approaches we propose in C-4. TRIAD proposes to solve the limitations in handling skewed workloads by preventing popular keys from being compacted and flushed back to the storage layer [29]. Such an approach would push the overall KVS system further into the regime we study where concurrency in the in-memory data structures is the bottleneck.

9.3.5 Advanced Concurrency Control and Synchronization

The most applicable techniques for the KVS workloads we target are those that can enable write load balancing (for WI_{uni}), and lock-free readers in the presence of frequent writes (for RW_{sk}). For WI_{uni} workloads, switching the KVS to a Concurrent Read, Concurrent Write policy and rewriting the software for Hardware Transactional Memory [117] or hardware-supported synchronization such as QOLB [156] should deliver similar performance to C-4 because true data conflicts are rare. However, both such designs come with drastically higher hardware implementation complexity, as evidenced by the fact that RTM is still disabled on Intel processors due to ordering violations [116], and is planned to be removed from most future Intel products [83]. In contrast, C-4 does not require changes to the caches or memory ordering hardware, only needing a small set of NIC extensions. Additionally, HTM or QOLB would not benefit RW_{sk} workloads, where true read-write conflicts between concurrent RPCs are the common case. For RW_{sk} workloads, C-4's combination of d-CREW and compaction is necessary for boosting performance.

RCU [183], RLU [180] and MV-RLU [146] enable both write balancing and lock-free reads by forcing writes to create new copies of data and make them visible to readers after a quiescent period. Our evaluation shows that the cost of versioning in RLU and MV-RLU versioning is prohibitive for µs-scale KVS, even with very few true conflicts. Delegation-based approaches such as `ffwd` [228], Flat Combining [99], and Remote Core Locking [175] could enable NIC-driven write balancing, but software threads now must re-shuffle requests so that critical sections execute on one core—essentially implementing CREW in software with extra shuffling overheads.

Both OpLog [39] and Doppel [195] propose similar designs to C-4's write compaction. OpLog

targets update-heavy kernel data structures by creating per-core update logs, which readers scan and apply lazily. Similarly, Doppel executes conflicting transactions by creating per-core values that are later merged upon commit. Applying OpLog to a KVS would simply move synchronization bottlenecks from the KVS' data structure to the per-core logs, and Doppel would incur similar overheads to MV-RLU at commit time (see §8.4.2). In contrast, C-4's compaction logs are invisible until the compaction window closes, and avoids costly copies and reconciliations.

10 Future Research Directions

10.1 Virtualized and Multi-Tenant Deployments

Virtualization and workload co-location is the common case for applications in today's datacenters, for both public-cloud serverless functions [5, 260] and first-party workloads developed by the datacenter owner [133, 263]. Not only does virtualization help datacenter service providers monetize their expensive hardware, but it also is a foundational underlying technology that helps enable microservices to be fungible through encapsulation and live migration. In contrast, our work in this thesis is limited to considering a simple deployment model where a server runs a single microservice whose threads are statically pinned to CPU cores. We believe this model is quite reasonable for deployments that are latency-critical and where µs-scale events matter, as prior work has shown that such applications benefit drastically from dedicated cores [163]. However, support for multiple co-located workloads is critical to usher in the adoption of RPC-centric architectures in real production systems.

Supporting multi-tenancy raises interesting challenges in both systems layering and server architecture. First, the existence of a hypervisor layer providing resource multiplexing and isolation between microservices implies that this layer must maintain full control of all additional features we propose in this thesis, allowing resource bindings to be created and broken as microservices are brought up and taken offline. This requirement raises challenges for designs which rely heavily on application-level information to implement certain policies at the underlying layers, as our work does. For example, we make the case that it is unnecessary

to receive any RPCs that would experience queues deeper than an application's SLO-abiding queue depth in Chapter 4. When multiple applications are co-located, multiple SLOs must be simultaneously expressed to the NIC by the respective microservices, multiple RPC buffers provisioned, and protocol messages demultiplexed into different sets of Queue Pairs. The same principle applies for locality-aware load balancing policies. We believe that an exokernel-like architecture [68] is a promising approach to allow each application to express its respective constraints and optimization metrics.

Secondly, the underlying NIC architecture must be able to export the underlying design extensions to the aforementioned hypervisor, as well as implement all of the application-defined policies simultaneously. Today's server NICs already contain hardware-assisted virtualization techniques like SR-IOV [118] as standard for this exact reason – they expose multiple virtual copies of themselves that can be used by software as if they were separate physical cards, while the hypervisor maintains control of the bindings. However, the current set of abstractions exported by NICs are primarily based on the PCIe standard, containing things like Base Address Registers, descriptor queues, doorbells, and interrupt lines. Revisiting the abstractions exposed by the NICs of RPC-centric server architectures is an interesting question for further work, which we believe would yield interesting design and implementation insights.

10.2 Towards Asynchronous Microservices

In this thesis, we have exclusively considered a synchronous, run-to-completion RPC execution model. This is logical for microservices that exist as leaf nodes in the overall service's dependence graph, and also allows the execution flow of interior nodes to be simplified, at the cost of imposing restrictions on concurrency. We believe that it is very logical for microservices that are placed in the interior "business logic tier" of the service graph to use an asynchronous execution model as there is already evidence to show that such models provide greater throughput. In particular, μ*Tune* shows that asynchronous threading models provide a 42% improvement in saturation throughput and a 12% reduction in 99th% tail latency across four microservices [247]. Moving in this direction creates two research opportunities.

First, asynchronous programming is notoriously difficult. Barroso et. al report that the

simplicity of synchronous programming outweighs the potential benefits of asynchronous models [32], and the authors of μ*Tune* mention that implementing their asynchronous microservices required four months of work, compared to three weeks for the synchronous ones [247, §3.1]. However, we have noted that in certain circles of the software community, there already exist open source frameworks to ease the process of expressing the dependencies and message types being passed among multiple microservices of asynchronous, event-driven software architectures, such as `AsyncAPI` [25]. Although such frameworks are most commonly proposed for services built around message queues rather than request-response RPC semantics, we believe that their approach to automatic code generation is a logical first step to reduce the programming complexity inherent in manually defining state machines for asynchronous microservices.

Secondly, services with asynchronous execution models allow interesting questions to be asked regarding the optimality of load balancing policies for RPCs that can yield their execution voluntarily (e.g., when generating their own nested RPCs). The existence of asynchronicity in RPC executions means that existing metrics such as queue depth are not necessarily the best predictor of queueing time. For example, assigning an RPC to an application thread that has an empty input queue but is currently running a non-yielding compute task may actually result in worse tail latency than assigning it to a thread whose input queue is full of RPCs that yield. We believe that asynchronous execution models may be able to be studied using queueing theoretic insights related to policies employing processor sharing, because in a sense an RPC that yields its execution can be seen as a form of voluntary preemption. Our event simulator is an ideal platform for such future work, as it is simple to create asynchronous execution models without the complexity of implementing the full service. However, evaluating any load balancing policy for asynchronous services would require an improved simulation infrastructure, leading to our next future direction.

10.3 Improved Server Simulation Methodologies

Simulating and modelling client-server workloads is a challenging task to say the least. Not only does it require deep knowledge of all levels of the server stack, but it also presents a complex set of methodological choices to the researcher whose answers are non-obvious,

particular when tail latency is concerned. For example, even when measuring the performance of real server hardware, one must answer various questions about the workload being generated and if the measured latency percentiles are statistically sound [149, 237]. This challenge becomes more complex and more time consuming in cycle-accurate simulation, because running enough full requests to reach a statistically sound final conclusion may require days or weeks of simulation time. For throughput-oriented applications, statistical techniques such as systematic sampling have reduced simulation time by orders of magnitude, and have been proven to accurately reflect server performance [265]. However, it is not clear how such statistical techniques can be applied to server architectural optimizations intended to benefit tail latency.

Furthermore, microservices are fundamentally distributed systems, where a single service is interconnected with many others through a complex dependency graph [81, 264]. The challenge in studying such architectures in cycle-accurate detail is that to study one microservice in isolation, the characteristics of the surrounding microservices must also be modelled in order to preserve the characteristics of the system as a whole. Consider the simple example of a microservice using the `nginx` web server, running PHP scripts which are JIT-compiled to machine code using a framework such as HHVM [2, 210]. A common deployment pattern is for the PHP scripts to first access a caching layer such as `memcached` or `MICA` to request data, and only access a backend relational database if the cache misses [42, 198]. If one is interested in architectural-level research or characterization of *any comprising microservice*, which we hereafter refer to as the "microservice under test" (MuT), maintaining simulation fidelity requires either simulating the entire microservice graph in cycle-level detail, or creating a higher-level abstraction for all microservices that surround the MuT.

The former approach seems to be more straightforward, because it does not require reducing complex service graphs to higher-level models, and creating alternative abstractions for peer microservices inside the simulator. However, as one of the fundamental tenets of datacenter software architecture is for services to be scalable, fault-tolerant, and fungible [248], we expect microservices to be deployed on many server nodes. Hence, the resulting simulation speeds will either slow down linearly with the number of microservices surrounding the MuT if a single host is used, or require use of large clusters to simulate all the nodes in parallel on

multiple hosts, while synchronizing time among them. Tools such as `dist-gem5` [9] and FireSim [137] take the latter approach, performing detailed simulation of all server nodes in the system.

We argue that a combination of high-level modelling and cycle-accurate simulation is a promising direction to enable researchers to simulate microservices without the need for large clusters. In our proposed approach, the MuT is isolated in cycle-accurate simulation, while all surrounding microservices are modelled as simple latency distributions connected by queues, similar to how we modelled the internal components of a server node in this thesis (see §3.3). Although previous systems have employed the use of queueing simulators to study system-level tradeoffs at scale, they do not consider the use of hybrid designs where a particular service is simulated at a higher level of fidelity [184, 277]. Therefore, these systems are not appropriate for modelling architectural or micro-architectural optimizations because they simply do not capture the required level of detail (e.g., by modelling cache accesses) [184, §1]. In contrast, a hybrid approach would enable a single server node to be simulated in full detail while maintaining coarse-grained modelling of the surrounding microservices to ensure the applicability of the simulation results in the entire system context. Successfully building such a model would allow drastically higher simulation density than existing approaches that use giant parallel simulations for a single experiment, and the ability to perform architectural studies of individual server nodes in a distributed system.

The use of hybrid cycle-accurate and queueing simulations raises many interesting questions from both methodological and systems performance perspectives. For example, when replacing a surrounding microservice's cycle-accurate execution with a simple latency distribution, how constant will that distribution remain when the MuT's performance is improved due to a research technique? Additionally, what level of network modelling should be done in our proposed hybrid approach? Prior works tend to either completely remove the network from consideration altogether [184, 277], or simulate the entire network itself in painstaking cycle-accurate detail [9]. However, as significant work in the networking community has called for a rethink of datacenter topologies [90] and protocols [193, 244] in order to support latency-critical microservices, we argue that such protocol optimizations certainly need to be included in microservice simulations. Fortunately, state-of-the-art protocols can readily be

simulated at packet- or flow-level using open source tools, and could be integrated into our hybrid methodology if desired. One may also ask whether the network itself can be replaced with its own statistical distribution, obviating the need for detailed protocol simulation at the cost of some fidelity.

Overall, we argue for a forward-looking, cross-disciplinary approach to server systems modelling that takes into account the rapid pace of improvements at each system layer. A critical open question is the level of simulation fidelity which is acceptable for the system layers *surrounding* the one being tested, which we believe will be a fruitful area of research and tool development. We believe that designing and building an open-source simulation methodology that allows the user to substitute various levels of detail for each system component will have dramatic impact to improve the pace of systems research that includes hardware modifications, and fit in well with current community initiatives to enhance the reproduceability of research results [1].

10.4 RPC Scheduling Based on End-to-End Latencies

As we have mentioned frequently in this thesis, the continual push to deliver low latency has led to a plethora of research contributions to optimize various modules in the operating system [126, 150, 212, 223], network stack [56, 80, 96, 153, 178, 193], and NIC hardware [20]. Despite the continued improvements granted by current research proposals, a common pattern is to study a particular module of the datacenter stack in isolation, and optimize it under an SLO corresponding to the time a request spends *in that module*. Datacenter operators use such quantifiable SLOs in order to set a service-level agreement (SLA), which is a legal contract concluded with their customers. The difference between SLOs and SLAs is that SLAs are expressed as *end-to-end* guarantees that are measured at the client's interface to the service [12, 188], whereas SLOs are often internal metrics such as network latency or storage availability. For online services deployed across many servers, there is currently a dilemma between setting a local SLO on each underlying module (which must always be conservative), versus a dynamic SLO that evolves on a per-request basis based on that request's client-visible latency (directly linked to the SLA).

Due to the importance of tail-tolerant computing, there already exist research works that attack this problem through a *proactive* approach. Both GrandSLAm [135] and Parslo [190] seek to derive local SLOs for the various microservices comprising a larger-scale online service according to a combination of theoretical system models and empirical measurements. In the case where user requests begin to experience SLA violations, Parslo proposes to make use of automatic auto-scaler frameworks that can adjust the number of instances of a microservice, or its number of allocated resources, to decrease its latency. However, such current work has two important limitations: first, the models they propose only apply to a restricted class of microservice applications – Parslo's authors write that their techniques only apply to stateless microservices which are unconcerned with factors like data consistency, replication, and load balancing. Second, neither system can *react* to requests that are approaching their end-to-end SLA deadlines and treat them differently than those which are less time-critical.

We believe that there is room for an orthogonal and also important set of research contributions, towards the goal of reactively prioritizing requests which are approaching their SLA in an attempt to reduce the number of end-to-end tail latency events. Such work would make use of open questions such as: which "deadline-approaching requests" are actually worth prioritizing, and which requests are simply a lost cause and should be allowed to fail? Additionally, prioritizing certain requests over others implies that the FIFO request processing model of most current load balancing frameworks needs to be reconsidered. Finally, any solution that utilizes reactive request prioritization has need of a system that allows each request to track its cumulative processing delay across microservices and the datacenter network. Contributions leveraging NIC timestamping (e.g., PTP) [110] or low-overhead software clock synchronization [85] seem to be ideal candidates to enable reactive request prioritization based on cumulative latency.

11 Conclusions

Decomposing software stacks into datacenter-deployed microservices is leading to an ever-increasing demand for network communication through RPCs. Furthermore, the continuous performance improvements delivered by network fabrics, systems software, and protocol termination innovations have collectively brought the individual server nodes in a datacenter closer together, meaning that overheads and inefficiencies on each server are continually becoming more problematic. Therefore, in order for RPC-dependent microservices to take advantage of the raw performance made available by the underlying network stack, server architecture must evolve in a matching fashion to support the underlying operations comprising RPCs. To meet this need, this thesis introduces the design and implementation of an RPC-centric server architecture, containing cross-stack innovations to improve the throughput and tail latency of multiple types of microservices. As hyperscale datacenter providers continue to enter the hardware business, as seen with Google's TPU, Amazon's Graviton CPU, and Microsoft's Catapult project, we believe that RPC-centric architectural patterns are ideal candidates for inclusion in future server processors.

Bibliography

[1] ACM. (2020) Artifact Review and Badging – Current. (Version 1.1, retrieved on May 13, 2022.). [Online]. Available: https://www.acm.org/publications/policies/artifact-review-and-badging-current Cited on page 184.

[2] K. Adams, J. Evans, B. Maher, G. Ottoni, A. Paroski, B. Simmers, E. Smith, and O. Yamauchi, "The hiphop virtual machine." in *Proceedings of the 29th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2014, pp. 777–790. Cited on page 182.

[3] Advanced Micro Devices, Inc. (2022) AMD 3D V-Cache Technology. [Online]. Available: https://www.amd.com/en/campaigns/3d-v-cache Cited on page 99.

[4] A. Adya, G. Cooper, D. Myers, and M. Piatek, "Thialfi: a client notification service for internet-scale applications." in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011, pp. 129–142. Cited on page 71.

[5] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight Virtualization for Serverless Applications." in *Proceedings of the 17th Symposium on Networked Systems Design and Implementation (NSDI)*, 2020, pp. 419–434. Cited on pages 1 and 179.

[6] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood, "DBMSs on a Modern Processor: Where Does Time Go?" in *Proceedings of the 25th International Conference on Very Large Databases (VLDB)*, 1999, pp. 266–277. Cited on page 172.

[7] Akamai Technologies. (2018) Akamai Online Retail Performance Report: Milliseconds Are Critical. [Online]. Available: https://www.akamai.com/newsroom/press-release/

akamai-releases-spring-2017-state-of-online-retail-performance-report Cited on page 2.

[8] Alexey Andreyev. (2014) Introducing data center fabric, the next-generation Facebook data center network. [Online]. Available: https://engineering.fb.com/production-engineering/introducing-data-center-fabric-the-next-generation-facebook-data-center-network Cited on page 13.

[9] M. Alian, U. Darbaz, G. Dózsa, S. Diestelhorst, D. Kim, and N. S. Kim, "dist-gem5: Distributed simulation of computer clusters." in *Proceedings of the 2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2017, pp. 153–162. Cited on page 183.

[10] 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*, 2010, pp. 63–74. Cited on page 166.

[11] Alphabet Inc., "LevelDB," March 2022. [Online]. Available: https://github.com/google/leveldb Cited on pages 76 and 176.

[12] Amazon Web Services Inc. (2019, July) Amazon S3 Service Level Agreement. [Online]. Available: https://aws.amazon.com/s3/sla/ Cited on page 184.

[13] ——. (2022) Amazon ElastiCache. [Online]. Available: https://aws.amazon.com/elasticache/?did=ap_card&trk=ap_card Cited on page 1.

[14] ——. (2022) Amazon Simple Storage Service. [Online]. Available: https://aws.amazon.com/s3/?did=ap_card&trk=ap_card Cited on page 1.

[15] Amazon.com Inc. (2021) Amazon.com Announces Financial Results and CEO Transition. [Online]. Available: https://ir.aboutamazon.com/news-release/news-release-details/2021/Amazon.com-Announces-Fourth-Quarter-Results/ Cited on page 1.

[16] Amazon Announces Graviton2 SoC Along With New AWS Instances. AnandTech. [Online]. Available: https://www.anandtech.com/show/15189/amazon-announces-

graviton2-soc-along-with-new-aws-instances-64core-arm-with-large-performance-uplifts Cited on pages 46, 52, and 148.

[17] Amazon's ARM-Based Graviton2 Against AMD and Intel: Comparing Cloud Compute. AnandTech. [Online]. Available: https://www.anandtech.com/show/15578/cloud-clash-amazon-graviton2-arm-against-intel-and-amd Cited on page 125.

[18] A. Ansari, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Divide and Conquer Frontend Bottleneck." in *Proceedings of the 47th International Symposium on Computer Architecture (ISCA)*, 2020, pp. 65–78. Cited on page 172.

[19] Apache Software Foundation. Thrift. [Online]. Available: https://thrift.apache.org/ Cited on pages 2, 12, 28, 140, and 169.

[20] M. T. Arashloo, A. Lavrov, M. Ghobadi, J. Rexford, D. Walker, and D. Wentzlaff, "Enabling Programmable Transport Protocols in High-Speed NICs." in *Proceedings of the 17th Symposium on Networked Systems Design and Implementation (NSDI)*, 2020, pp. 93–109. Cited on page 184.

[21] (2021) Arm DynamIQ Shared Unit Technical Reference Manual. Arm Limited. Rev. r3p0. [Online]. Available: https://developer.arm.com/documentation/100453/0300/functional-description/l3-cache/cache-stashing Cited on pages 20 and 27.

[22] *ARM Architecture Reference Manual for A-profile architecture. Version H.a,* ARM Limited, February 2022. Cited on page 147.

[23] S. Arslan, S. Ibanez, A. Mallery, C. Kim, and N. McKeown, "NanoTransport: A Low-Latency, Programmable Transport Layer for NICs." in *Proceedings of the Symposium on SDN Research (SOSR)*, 2021, pp. 13–26. Cited on page 166.

[24] N. Asmussen, M. Roitzsch, and H. Härtig, "M³x: Autonomous Accelerators via Context-Enabled Fast-Path Communication." in *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, 2019, pp. 617–632. Cited on page 171.

[25] AsyncAPI. (2022) AsyncAPI Generator Tool. [Online]. Available: https://www.asyncapi.com/tools/generator Cited on page 181.

[26] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store." in *Proceedings of the 2012 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2012, pp. 53–64. Cited on page 54.

[27] G. Ayers, N. P. Nagendra, D. I. August, H. K. Cho, S. Kanev, C. Kozyrakis, T. Krishnamurthy, H. Litz, T. Moseley, and P. Ranganathan, "AsmDB: understanding and mitigating front-end stalls in warehouse-scale computers." in *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*, 2019, pp. 462–473. Cited on pages 63, 66, and 172.

[28] H. Ballani, P. Costa, R. Behrendt, D. Cletheroe, I. Haller, K. Jozwik, F. Karinou, S. Lange, K. Shi, B. Thomsen, and H. Williams, "Sirius: A Flat Datacenter Network with Nanosecond Optical Switching." in *Proceedings of the ACM SIGCOMM 2020 Conference*, 2020, pp. 782–797. Cited on page 3.

[29] O. Balmau, D. Didona, R. Guerraoui, W. Zwaenepoel, H. Yuan, A. Arora, K. Gupta, and P. Konka, "TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores." in *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, 2017, pp. 363–375. Cited on page 177.

[30] O. Balmau, R. Guerraoui, V. Trigonakis, and I. Zablotchi, "FloDB: Unlocking Memory in Persistent Key-Value Stores." in *Proceedings of the 2017 EuroSys Conference*, 2017, pp. 80–94. Cited on page 176.

[31] L. A. Barroso, U. Hölzle, and P. Ranganathan, *The Datacenter as a Computer: Designing Warehouse-Scale Machines, Third Edition*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2018. Cited on pages 1, 2, and 12.

[32] L. A. Barroso, M. Marty, D. A. Patterson, and P. Ranganathan, "Attack of the killer microseconds." *Commun. ACM*, vol. 60, no. 4, pp. 48–54, 2017. Cited on pages 13 and 181.

[33] 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.*, vol. 34, no. 4, pp. 11:1–11:39, 2017. Cited on pages 15, 16, 17, 42, 72, and 73.

[34] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator." in *USENIX Annual Technical Conference, FREENIX Track*, 2005, pp. 41–46. Cited on page 109.

[35] A. Birrell and B. J. Nelson, "Implementing Remote Procedure Calls," *ACM Trans. Comput. Syst.*, vol. 2, no. 1, pp. 39–59, 1984. Cited on page 29.

[36] Bob Wheeler, "Ryzen 5000 Rides Zen 3 to the Top," *Linley Group Microprocessor Report*, November 2020. Cited on pages 4 and 63.

[37] M. Bohr, "A 30 Year Retrospective on Dennard's MOSFET Scaling Paper," *IEEE Solid-State Circuits Society Newsletter*, vol. 12, no. 1, pp. 11–13, 2007. Cited on page 106.

[38] 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." *Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, 2014. Cited on pages 89 and 166.

[39] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich, "OpLog: a library for scaling update-heavy data structures," Massachussetts Institute of Technology, Tech. Rep., 2014. [Online]. Available: https://dspace.mit.edu/handle/1721.1/89653 Cited on pages 76 and 177.

[40] Broadcom Corporation, "StrataDNX 10 Tb/s Scalable Switching Device Product Brief," https://docs.broadcom.com/docs/88690-PB100, 2018, (Date retrieved: 10 March 2020). Cited on page 40.

[41] ——, "High-Capacity StrataXGS Trident 4 Ethernet Switch Series," https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56880-series, 2019. Cited on page 14.

[42] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. C. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani, "TAO: Facebook's Distributed Data Store for the Social Graph." in *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)*, 2013, pp. 49–60. Cited on pages 73 and 182.

[43] Z. Cao, S. Dong, S. Vemuri, and D. H. C. Du, "Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook." in *Proceedings of the 18th USENIX*

Conference on File and Storage Technologies (FAST), 2020, pp. 209–223. Cited on pages 72, 73, and 74.

[44] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, "A cloud-scale acceleration architecture." in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 7:1–7:13. Cited on pages 3, 5, 14, 21, 26, and 171.

[45] H. Chen, R. Chen, X. Wei, J. Shi, Y. Chen, Z. Wang, B. Zang, and H. Guan, "Fast In-Memory Transaction Processing Using RDMA and HTM." *ACM Trans. Comput. Syst.*, vol. 35, no. 1, pp. 3:1–3:37, 2017. Cited on page 176.

[46] Y. Chen, X. Wei, J. Shi, R. Chen, and H. Chen, "Fast and general distributed transactions using RDMA and HTM." in *Proceedings of the 2016 EuroSys Conference*, 2016, pp. 26:1–26:17. Cited on page 176.

[47] Y. Chen, Y. Lu, and J. Shu, "Scalable RDMA RPC on Reliable Connection with Efficient Resource Sharing." in *Proceedings of the 2019 EuroSys Conference*, 2019, pp. 19:1–19:14. Cited on page 26.

[48] D. Chiou, "Extending the reach of microprocessors: column and curious caching." Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA, USA, 1999. Cited on page 167.

[49] I. Cho, A. Saeed, J. Fried, S. J. Park, M. Alizadeh, and A. Belay, "Overload Control for µs-scale RPCs with Breakwater." in *Proceedings of the 14th Symposium on Operating Systems Design and Implementation (OSDI)*, 2020, pp. 299–314. Cited on pages 82 and 166.

[50] *Compute Express Link (CXL) Specification*, Compute Express Link Consortium Inc., October 2020, protocol Revision 2.0. Cited on pages 90 and 91.

[51] S. F. Conservancy and the QEMU Leadership Committee. QEMU: the FAST! processor emulator. [Online]. Available: https://gitlab.com/qemu-project/qemu Cited on pages 36 and 109.

[52] A. Daglis, "Network-compute co-design for distributed in-memory computing," Ph.D. dissertation, EPFL, Lausanne, 2018. [Online]. Available: http://infoscience.epfl.ch/record/256864 Cited on pages 37 and 97.

[53] A. Daglis, S. Novakovic, E. Bugnion, B. Falsafi, and B. Grot, "Manycore network interfaces for in-memory rack-scale computing." in *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA)*, 2015, pp. 567–579. Cited on pages 4, 6, 20, 30, 49, 50, 97, 104, 116, 127, and 166.

[54] A. Daglis, M. Sutherland, and B. Falsafi, "RPCValet: NI-Driven Tail-Aware Balancing of µs-Scale RPCs." in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019, pp. 35–48. Cited on pages 2, 3, 12, 13, 18, 20, 34, 42, 44, 48, 50, 54, 55, 57, 69, 73, 74, 98, 100, 105, 109, 111, 121, 125, 127, and 149.

[55] A. Daglis, D. Ustiugov, S. Novakovic, E. Bugnion, B. Falsafi, and B. Grot, "SABRes: Atomic object reads for in-memory rack-scale computing." in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 6:1–6:13. Cited on pages 20 and 165.

[56] M. Dalton, D. Schultz, J. Adriaens, A. Arefin, A. Gupta, B. Fahs, D. Rubinstein, E. C. Zermeno, E. Rubow, J. A. Docauer, J. Alpert, J. Ai, J. Olson, K. DeCabooter, M. de Kruijf, N. Hua, N. Lewis, N. Kasinadhuni, R. Crepaldi, S. Krishnan, S. Venkata, Y. Richter, U. Naik, and A. Vahdat, "Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization." in *Proceedings of the 15th Symposium on Networked Systems Design and Implementation (NSDI)*, 2018, pp. 373–387. Cited on page 184.

[57] David Brooks. (2018) What's the future of technology scaling? [Online]. Available: https://www.sigarch.org/whats-the-future-of-technology-scaling/ Cited on pages 3, 30, and 106.

[58] J. Dean, "Building Software Systems at Google and Lessons Learned," 2010, Talk at Stanford University. [Online]. Available: https://www.youtube.com/watch?v=modXC5IWTJI Cited on page 12.

[59] J. Dean and L. A. Barroso, "The tail at scale." *Commun. ACM*, vol. 56, no. 2, pp. 74–80, 2013. Cited on pages 2, 12, 17, 53, and 72.

[60] 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)*, 2007, pp. 205–220. Cited on page 12.

[61] H. M. Demoulin, J. Fried, I. Pedisich, M. Kogias, B. T. Loo, L. T. X. Phan, and I. Zhang, "When Idling is Ideal: Optimizing Tail-Latency for Heavy-Tailed Datacenter Workloads with Perséphone." in *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, 2021, pp. 621–637. Cited on pages 4, 17, 19, 21, and 71.

[62] R. H. Dennard, F. H. Gaensslen, H. Yu, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted mosfet's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974. Cited on page 106.

[63] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri, "Practical Skew Handling in Parallel Joins." in *Proceedings of the 18th International Conference on Very Large Databases (VLDB)*, 1992, pp. 27–40. Cited on page 145.

[64] D. Didona and W. Zwaenepoel, "Size-aware Sharding For Improving Tail Latencies in In-memory Key-value Stores." in *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI)*, 2019, pp. 79–94. Cited on pages 175 and 176.

[65] A. Dragojevic, D. Narayanan, M. Castro, and O. Hodson, "FaRM: Fast Remote Memory." in *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, 2014, pp. 401–414. Cited on pages 17, 26, 41, 163, 164, and 176.

[66] D. Dunning, G. J. Regnier, G. L. McAlpine, D. Cameron, B. Shubert, F. Berry, A. M. Merritt, E. Gronke, and C. Dodd, "The Virtual Interface Architecture." *IEEE Micro*, vol. 18, no. 2, pp. 66–76, 1998. Cited on pages 16 and 166.

[67] A. Eisenman, D. Gardner, I. AbdelRahman, J. Axboe, S. Dong, K. M. Hazelwood, C. Petersen, A. Cidon, and S. Katti, "Reducing DRAM footprint with NVM in facebook." in *Proceedings of the 2018 EuroSys Conference*, 2018, pp. 42:1–42:13. Cited on page 73.

[68] D. R. Engler, M. F. Kaashoek, and J. W. O. Jr., "Exokernel: An Operating System Architecture for Application-Level Resource Management." in *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, 1995, pp. 251–266. Cited on page 180.

[69] H. Eran, M. Fudim, G. Malka, G. Shalom, N. Cohen, A. Hermony, D. Levi, L. Liss, and M. Silberstein, "FlexDriver: a network driver for your accelerator." in *Proceedings of the 27th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022, pp. 1115–1129. Cited on page 164.

[70] H. Eran, L. Zeno, M. Tork, G. Malka, and M. Silberstein, "NICA: An Infrastructure for Inline Acceleration of Network Applications." in *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, 2019, pp. 345–362. Cited on pages 20, 169, and 176.

[71] H. Esmaeilzadeh, E. R. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling." in *Proceedings of the 38th International Symposium on Computer Architecture (ISCA)*, 2011, pp. 365–376. Cited on pages 30 and 106.

[72] Facebook Inc. Facebook Thrift. [Online]. Available: https://github.com/facebook/fbthrift Cited on pages 2, 12, 28, and 140.

[73] B. Fan, D. G. Andersen, and M. Kaminsky, "MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing." in *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI)*, 2013, pp. 371–384. Cited on page 175.

[74] B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky, "Small cache, big effect: provable load balancing for randomly partitioned cluster services." in *Proceedings of the 2011 ACM Symposium on Cloud Computing (SOCC)*, 2011, p. 23. Cited on page 176.

[75] A. Farshin, T. Barbette, A. Roozbeh, G. Q. M. Jr., and D. Kostic, "PacketMill: toward per-Core 100-Gbps networking." in *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021, pp. 1–17. Cited on pages 31, 162, and 163.

[76] A. Farshin, A. Roozbeh, G. Q. M. Jr., and D. Kostic, "Make the Most out of Last Level Cache in Intel Processors." in *Proceedings of the 2019 EuroSys Conference*, 2019, pp. 8:1–8:17. Cited on pages 5, 31, 50, and 162.

[77] ——, "Reexamining Direct Cache Access to Optimize I/O Intensive Applications for Multi-hundred-gigabit Networks." in *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*, 2020, pp. 673–689. Cited on page 162.

[78] M. Ferdman, A. Adileh, Y. O. Koçberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware." in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012, pp. 37–48. Cited on page 63.

[79] M. Ferdman, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Temporal instruction fetch streaming." in *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2008, pp. 1–10. Cited on page 172.

[80] 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)*, 2018, pp. 51–66. Cited on page 184.

[81] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems." in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019, pp. 3–18. Cited on pages 8, 12, 63, 72, 73, 139, and 182.

[82] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker, "Network Requirements for Resource Disaggregation." in *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI)*, 2016, pp. 249–264. Cited on page 15.

[83] Gareth Halfacree. The Register: Intel sticks another nail in the coffin of TSX with feature-disabling microcode update. [Online]. Available: https://www.theregister.com/2021/06/29/intel_tsx_disabled/ Cited on page 177.

[84] V. Gavrielatos, A. Katsarakis, A. Joshi, N. Oswald, B. Grot, and V. Nagarajan, "Scale-out ccNUMA: exploiting skew with strongly consistent caching." in *Proceedings of the 2018 EuroSys Conference*, 2018, pp. 21:1–21:15. Cited on pages 73, 74, 77, 84, and 176.

[85] Y. Geng, S. Liu, Z. Yin, A. Naik, B. Prabhakar, M. Rosenblum, and A. Vahdat, "Exploiting a Natural Network Effect for Scalable, Fine-grained Clock Synchronization." in *Proceedings of the 15th Symposium on Networked Systems Design and Implementation (NSDI)*, 2018, pp. 81–94. Cited on page 185.

[86] Google. FlatBuffers. [Online]. Available: https://google.github.io/flatbuffers/ Cited on page 170.

[87] ——. Protocol Buffers. [Online]. Available: https://developers.google.com/protocol-buffers/ Cited on pages 145 and 169.

[88] S. Goswami, N. Kodirov, C. Mustard, I. Beschastnikh, and M. I. Seltzer, "Parking packet payload with P4." in *Proceedings of the 2020 ACM Conference on Emerging Networking Experiments and Technology (CoNEXT)*, 2020, pp. 274–281. Cited on page 168.

[89] Grandview Research. (2021) E-commerce Market Size, Share and Trends Analysis Report By Model Type. [Online]. Available: https://www.grandviewresearch.com/industry-analysis/e-commerce-market Cited on page 1.

[90] 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." in *Proceedings of the ACM SIGCOMM 2009 Conference*, 2009, pp. 51–62. Cited on pages 3, 13, and 183.

[91] Greg Linden. (2006) Marissa Mayer at Web 2.0. [Online]. Available: http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html Cited on page 2.

[92] P. Grosu, M. Rehman, E. Anderson, V. Pai, and H. Miller. gRPC. [Online]. Available: https://github.com/heathermiller/dist-prog-book/blob/master/chapter/1/gRPC.md Cited on pages 2, 12, 25, and 28.

[93] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn, "RDMA over Commodity Ethernet at Scale." in *Proceedings of the ACM SIGCOMM 2016 Conference*, 2016, pp. 202–215. Cited on pages 3, 5, 14, and 171.

[94] L. Gwennap, "AMD Milan Extends Server Lead," *Linley Group Microprocessor Report*, March 2021. Cited on pages 18, 105, 135, and 148.

[95] ——, "Sapphire Rapids Spans Tiles," *Linley Group Microprocessor Report*, September 2021. Cited on pages 18 and 148.

[96] 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*, 2017, pp. 29–42. Cited on pages 14, 15, 39, 49, 56, and 184.

[97] 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)*, 2017, pp. 168–183. Cited on page 53.

[98] 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)*, 2015, pp. 161–175. Cited on page 2.

[99] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir, "Flat combining and the synchronization-parallelism tradeoff." in *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2010, pp. 355–364. Cited on page 177.

[100] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture." *Commun. ACM*, vol. 62, no. 2, pp. 48–60, 2019. Cited on page 21.

[101] M. Herlihy and J. M. Wing, "Linearizability: A Correctness Condition for Concurrent Objects." *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, 1990. Cited on page 84.

[102] Y.-J. Hong and M. Thottethodi, "Understanding and mitigating the impact of load imbalance in the memory caching tier." in *Proceedings of the 2013 ACM Symposium on Cloud Computing (SOCC)*, 2013, pp. 13:1–13:17. Cited on page 72.

[103] N. Horman, P. Waskiewicz, A. Arapov, and et al. IRQBalance Linux Daemon. [Online]. Available: http://irqbalance.github.io/irqbalance/ Cited on page 68.

[104] Q. Huang, H. Gudmundsdottir, Y. Vigfusson, D. A. Freedman, K. Birman, and R. van Renesse, "Characterizing Load Imbalance in Real-World Networked Caches." in *Proceedings of the 13th ACM Workshop on Hot Topics in Networks (HotNets-XIII)*, 2014, pp. 8:1–8:7. Cited on page 72.

[105] R. Huggahalli, R. R. Iyer, and S. Tetrick, "Direct Cache Access for High Bandwidth Network I/O." in *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA)*, 2005, pp. 50–59. Cited on pages 20, 27, and 57.

[106] J. T. Humphries, K. Kaffes, D. Mazières, and C. Kozyrakis, "Mind the Gap: A Case for Informed Request Scheduling at the NIC." in *Proceedings of the 18th ACM Workshop on Hot Topics in Networks (HotNets-XVIII)*, 2019, pp. 60–68. Cited on pages 18, 42, and 90.

[107] J. T. Humphries, N. Natu, A. Chaugule, O. Weisse, B. Rhoden, J. Don, L. Rizzo, O. Rombakh, P. Turner, and C. Kozyrakis, "ghOSt: Fast & Flexible User-Space Delegation of Linux Scheduling." in *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, 2021, pp. 588–604. Cited on page 174.

[108] S. Ibanez, A. Mallery, S. Arslan, T. Jepsen, M. Shahbaz, C. Kim, and N. McKeown, "The nanoPU: A Nanosecond Network Stack for Datacenters." in *Proceedings of the 15th Symposium on Operating Systems Design and Implementation (OSDI)*, 2021, pp. 239–256. Cited on pages 4, 14, 20, 31, 57, 97, 137, and 167.

[109] S. Ibanez, M. Shahbaz, and N. McKeown, "The Case for a Network Fast Path to the CPU." in *Proceedings of the 18th ACM Workshop on Hot Topics in Networks (HotNets-XVIII)*, 2019, pp. 52–59. Cited on pages 4 and 20.

[110] IEEE Standards Association. (2019) IEEE Standard 1588-2019. (Date retrieved: 18 May 2022). [Online]. Available: https://standards.ieee.org/ieee/1588/6825/ Cited on page 185.

[111] Infiniband Trade Association, "InfiniBand Architecture Release 1.3.1," https://www.infinibandta.org/ibta-specifications-download, 2016, (Date retrieved: 6 March 2020). Cited on page 17.

[112] InfiniBand Trade Association, "InfiniBand Roadmap," https://www.infinibandta.org/infiniband-roadmap/, 2018. Cited on pages 3, 14, and 40.

[113] Intel Corp. Intel 64 and IA-32 Architectures Software Developer's Manuals. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html Cited on page 147.

[114] Intel Omni-Path Architecture Driving Exascale Computing and HPC. https://www.intel.com/content/www/us/en/high-performance-computing-fabrics/omni-path-driving-exascale-computing.html. Intel Corp. Cited on page 14.

[115] Intel Corp. Intel tofino 2 p4 programmable ethernet switch. [Online]. Available: https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html Cited on page 89.

[116] ——. Performance Monitoring Impact of Intel® Transactional Synchronization Extension Memory Ordering Issue. [Online]. Available: https://www.intel.com/content/dam/support/us/en/documents/processors/Performance-Monitoring-Impact-of-TSX-Memory-Ordering-Issue-604224.pdf Cited on page 177.

[117] ——. RTM Overview. [Online]. Available: https://www.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/intrinsics/intrinsics-for-avx2/intrinsics-for-tsx/intrinsics-for-

restrict-transactional-mem-ops/restricted-transactional-memory-overview.html Cited on page 177.

[118] (2011) PCI-SIG SR-IOV Primer. Intel Corp. (Date retrieved: 18 March 2022). Cited on page 180.

[119] (2012) Intel Data Direct I/O Technology. Intel Corp. [Online]. Available: https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology-brief.html Cited on pages 20, 27, 41, 43, 45, 54, 57, and 167.

[120] (2014) Introduction to Intel Ethernet Flow Director and Memcached Performance. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-ethernet-flow-director.pdf. Intel Corp. (Date retrieved: 6 March 2020). Cited on pages 16, 18, and 68.

[121] (2016) Intel Xeon Processor D-1500 Product Family. https://cdrdv2.intel.com/v1/dl/getcontent/333423. Intel Corp. (Date retrieved: 6 March 2020). Cited on pages 4, 20, and 97.

[122] J. Jang, S. Jung, S. Jeong, J. Heo, H. Shin, T. J. Ham, and J. W. Lee, "A Specialized Architecture for Object Serialization with Applications to Big Data Analytics." in *Proceedings of the 47th International Symposium on Computer Architecture (ISCA)*, 2020, pp. 322–334. Cited on pages 20 and 30.

[123] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "NetCache: Balancing Key-Value Stores with Fast In-Network Caching." in *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017, pp. 121–136. Cited on pages 21, 72, 73, 75, 82, 155, and 176.

[124] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson. (2019) Cloud programming simplified: A berkeley view on serverless computing. [Online]. Available: https://arxiv.org/abs/1902.03383 Cited on page 1.

[125] N. P. Jouppi, C. Young, N. Patil, D. A. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. luc Cantin, C. Chao, C. Clark, J. Coriell, M. Daley,

M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-Datacenter Performance Analysis of a Tensor Processing Unit." in *Proceedings of the 44th International Symposium on Computer Architecture (ISCA)*, 2017, pp. 1–12. Cited on page 21.

[126] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis, "Shinjuku: Preemptive Scheduling for usecond-scale Tail Latency." in *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI)*, 2019, pp. 345–360. Cited on pages 17, 34, 42, 149, and 184.

[127] G. Kakivaya, L. Xun, R. Hasha, S. B. Ahsan, T. Pfleiger, R. Sinha, A. Gupta, M. Tarta, M. Fussell, V. Modi, M. Mohsin, R. Kong, A. Ahuja, O. Platon, A. Wun, M. Snider, C. Daniel, D. Mastrian, Y. Li, A. Rao, V. Kidambi, R. Wang, A. Ram, S. Shivaprakash, R. Nair, A. Warwick, B. S. Narasimman, M. Lin, J. Chen, A. B. Mhatre, P. Subbarayalu, M. Coskun, and I. Gupta, "Service fabric: a distributed platform for building microservices in the cloud." in *Proceedings of the 2018 EuroSys Conference*, 2018, pp. 33:1–33:15. Cited on pages 2 and 12.

[128] eRPC Repository. [Online]. Available: https://github.com/erpc-io/eRPC Cited on page 149.

[129] A. Kalia, M. Kaminsky, and D. G. Andersen, "Using RDMA efficiently for key-value services." in *Proceedings of the ACM SIGCOMM 2014 Conference*, 2014, pp. 295–306. Cited on pages 3, 12, 26, 163, and 176.

[130] ——, "Design Guidelines for High Performance RDMA Systems." in *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*, 2016, pp. 437–450. Cited on page 26.

[131] ——, "FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs." in *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI)*, 2016, pp. 185–201. Cited on pages 26, 41, 111, 125, and 163.

[132] ——, "Datacenter RPCs can be General and Fast." in *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI)*, 2019, pp. 1–16. Cited on pages 3, 12, 26, 41, 42, 111, 145, and 164.

[133] S. Kanev, J. P. Darago, K. M. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. M. Brooks, "Profiling a warehouse-scale computer." in *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA)*, 2015, pp. 158–169. Cited on pages 63, 140, 169, and 179.

[134] A. Kannan, N. D. E. Jerger, and G. H. Loh, "Enabling interposer-based disintegration of multi-core processors." in *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015, pp. 546–558. Cited on page 4.

[135] R. S. Kannan, L. Subramanian, A. Raju, J. Ahn, J. Mars, and L. Tang, "GrandSLAm: Guaranteeing SLAs for Jobs in Microservices Execution Frameworks." in *Proceedings of the 2019 EuroSys Conference*, 2019, pp. 34:1–34:16. Cited on page 185.

[136] S. Karandikar, C. Leary, C. Kennelly, J. Zhao, D. Parimi, B. Nikolic, K. Asanovic, and P. Ranganathan, "A Hardware Accelerator for Protocol Buffers." in *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2021, pp. 462–478. Cited on pages 5, 20, 63, 120, 121, 169, and 170.

[137] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra, Q. Huang, K. Kovacs, B. Nikolic, R. H. Katz, J. Bachrach, and K. Asanovic, "FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud." in *Proceedings of the 45th International Symposium on Computer Architecture (ISCA)*, 2018, pp. 29–42. Cited on pages 4, 20, 97, and 183.

[138] S. Kashyap, C. Min, K. Kim, and T. Kim, "A scalable ordering primitive for multicore machines." in *Proceedings of the 2018 EuroSys Conference*, 2018, pp. 34:1–34:15. Cited on pages 147 and 149.

[139] A. Kaufmann, S. Peter, N. K. Sharma, T. E. Anderson, and A. Krishnamurthy, "High Performance Packet Processing with FlexNIC." in *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016, pp. 67–81. Cited on pages 89 and 145.

[140] C. Kaynak, B. Grot, and B. Falsafi, "SHIFT: shared history instruction fetch for lean-core server processors." in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013, pp. 272–283. Cited on page 172.

[141] ——, "Confluence: unified instruction supply for scale-out servers." in *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015, pp. 166–177. Cited on page 172.

[142] K. Keeton, D. A. Patterson, Y. Q. He, R. C. Raphael, and W. E. Baker, "Performance Characterization of a Quad Pentium Pro SMP using OLTP Workloads." in *Proceedings of the 25th International Symposium on Computer Architecture (ISCA)*, 1998, pp. 15–26. Cited on page 172.

[143] Kenton Varda, Sandstorm.io. Cap'n Proto. [Online]. Available: https://capnproto.org Cited on page 170.

[144] T. A. Khan, A. Sriraman, J. Devietti, G. Pokam, H. Litz, and B. Kasikci, "I-SPY: Context-Driven Conditional Instruction Prefetching with Coalescing." in *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 146–159. Cited on page 172.

[145] Khang Nguyen. Usage Models for Cache Allocation Technology in the Intel Xeon Processor E5 v4 Family. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/cache-allocation-technology-usage-models.html Cited on page 167.

[146] J. Kim, A. Mathew, S. Kashyap, M. K. Ramanathan, and C. Min, "MV-RLU: Scaling Read-Log-Update with Multi-Versioning." in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019, pp. 779–792. Cited on pages 75, 149, and 177.

[147] M. Kogias and E. Bugnion, "Flow control for Latency-Critical RPCs." in *Proceedings of the 2018 Workshop on Kernel-Bypass Networks (KBNETS@SIGCOMM)*, 2018, pp. 15–21. Cited on pages 53 and 166.

[148] ——, "HovercRaft: achieving scalability and fault-tolerance for microsecond-scale datacenter services." in *Proceedings of the 2020 EuroSys Conference*, 2020, pp. 25:1–25:17. Cited on page 15.

[149] M. Kogias, S. Mallon, and E. Bugnion, "Lancet: A self-correcting Latency Measuring Tool." in *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, 2019, pp. 881–896. Cited on pages 34 and 182.

[150] 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, pp. 863–880. Cited on pages 4, 18, 21, 39, 44, 48, 49, 55, 56, 69, 72, 73, 74, 76, 87, 89, 90, 105, 109, 121, 125, 131, 149, and 184.

[151] A. Kumar and R. Huggahalli, "Impact of Cache Coherence Protocols on the Processing of Network Traffic." in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2007, pp. 161–171. Cited on page 167.

[152] A. Kumar, R. Huggahalli, and S. Makineni, "Characterization of Direct Cache Access on multi-core systems and 10GbE." in *Proceedings of the 15th IEEE Symposium on High-Performance Computer Architecture (HPCA)*, 2009, pp. 341–352. Cited on page 27.

[153] G. Kumar, N. Dukkipati, K. Jang, H. M. G. Wassel, X. Wu, B. Montazeri, Y. Wang, K. Springborn, C. Alfeld, M. Ryan, D. Wetherall, and A. Vahdat, "Swift: Delay is Simple and Effective for Congestion Control in the Datacenter." in *Proceedings of the ACM SIGCOMM 2020 Conference*, 2020, pp. 514–528. Cited on pages 3, 4, and 184.

[154] R. Kumar, B. Grot, and V. Nagarajan, "Blasting through the Front-End Bottleneck with Shotgun." in *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018, pp. 30–42. Cited on pages 64 and 172.

[155] R. Kumar, C.-C. Huang, B. Grot, and V. Nagarajan, "Boomerang: A Metadata-Free Architecture for Control Flow Delivery." in *Proceedings of the 23rd IEEE Symposium on High-Performance Computer Architecture (HPCA)*, 2017, pp. 493–504. Cited on pages 64 and 172.

[156] A. Kägi, D. Burger, and J. R. Goodman, "Efficient Synchronization: Let Them Eat QOLB." in *Proceedings of the 24th International Symposium on Computer Architecture (ISCA)*, 1997, pp. 170–180. Cited on page 177.

[157] B. Lampson, "Hints and principles for computer system design," 2020, retrieved May 3, 2022. [Online]. Available: https://arxiv.org/abs/2011.02455 Cited on pages 53, 84, and 121.

[158] J. R. Larus and M. Parkes, "Using Cohort-Scheduling to Enhance Server Performance." in *USENIX Annual Technical Conference*, 2002, pp. 103–114. Cited on page 173.

[159] N. Lazarev, N. Adit, S. Xiang, Z. Zhang, and C. Delimitrou, "Dagger: Towards Efficient RPCs in Cloud Microservices With Near-Memory Reconfigurable NICs." *IEEE Comput. Archit. Lett.*, vol. 19, no. 2, pp. 134–138, 2020. Cited on pages 3, 30, and 125.

[160] N. Lazarev, S. Xiang, N. Adit, Z. Zhang, and C. Delimitrou, "Dagger: efficient and fast RPCs in cloud microservices with near-memory reconfigurable NICs." in *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021, pp. 36–51. Cited on pages 120, 168, and 169.

[161] I. Lesokhin, H. Eran, S. Raindel, G. Shapiro, S. Grimberg, L. Liss, M. Ben-Yehuda, N. Amit, and D. Tsafrir, "Page Fault Support for Network Controllers." in *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017, pp. 449–466. Cited on page 41.

[162] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang, "KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC." in *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017, pp. 137–152. Cited on pages 72, 73, 74, 76, 77, 84, and 176.

[163] 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)*, 2014, pp. 9:1–9:14. Cited on pages 2, 16, and 179.

[164] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "CACTI-P: Architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques." in *Proceedings of the 2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2011, pp. 694–701. Cited on pages 105 and 134.

[165] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, S. O, S. Lee, and P. Dubey, "Architecting to achieve a billion requests per second throughput on a single key-value store server platform." in *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA)*, 2015, pp. 476–488. Cited on pages 41, 46, 73, 75, 76, 77, 84, 110, 155, 165, and 175.

[166] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "MICA: A Holistic Approach to Fast In-Memory Key-Value Storage." in *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, 2014, pp. 429–444. Cited on pages 40, 46, 72, 73, 74, 75, 76, 77, 84, 87, 110, 145, 149, 155, 165, and 175.

[167] K. T. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch, "Thin servers with smart pipes: designing SoC accelerators for memcached." in *Proceedings of the 40th International Symposium on Computer Architecture (ISCA)*, 2013, pp. 36–47. Cited on pages 16 and 165.

[168] K. T. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch, "System-level implications of disaggregated memory." in *Proceedings of the 18th IEEE Symposium on High-Performance Computer Architecture (HPCA)*, 2012, pp. 189–200. Cited on page 15.

[169] Linley Group, "Centriq Aces Scale-Out Performance," *Microprocessor Report*, November 2017. Cited on page 46.

[170] ——, "Xeon Scalable Reshapes Server Line," *Microprocessor Report*, July 2017. Cited on pages 18, 46, 52, and 148.

[171] "Data Plane Development Kit," https://www.dpdk.org/, The Linux Foundation Projects. Cited on page 15.

[172] M. Liu, T. Cui, H. Schuh, A. Krishnamurthy, S. Peter, and K. Gupta, "Offloading distributed applications onto smartNICs using iPipe." in *Proceedings of the ACM SIGCOMM 2019 Conference*, 2019, pp. 318–333. Cited on pages 18 and 174.

[173] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Improving Resource Efficiency at Scale with Heracles." *ACM Trans. Comput. Syst.*, vol. 34, no. 2, pp. 6:1–6:33, 2016. Cited on page 27.

[174] P. Lotfi-Kamran, B. Grot, and B. Falsafi, "NOC-Out: Microarchitecting a Scale-Out Processor." in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012, pp. 177–187. Cited on page 168.

[175] J.-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller, "Remote Core Locking: Migrating Critical-Section Execution to Improve the Performance of Multithreaded Applications." in *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, 2012, pp. 65–76. Cited on pages 86 and 177.

[176] Y. Mao, E. Kohler, and R. T. Morris, "Cache craftiness for fast multicore key-value storage." in *Proceedings of the 2012 EuroSys Conference*, 2012, pp. 183–196. Cited on pages 3, 12, 13, 74, 84, and 175.

[177] I. Marinos, R. N. M. Watson, M. Handley, and R. R. Stewart, "Disk|Crypt|Net: rethinking the stack for high-performance video streaming." in *Proceedings of the ACM SIGCOMM 2017 Conference*, 2017, pp. 211–224. Cited on page 162.

[178] M. Marty, M. de Kruijf, J. Adriaens, C. Alfeld, S. Bauer, C. Contavalli, M. Dalton, N. Dukkipati, W. C. Evans, S. Gribble, N. Kidd, R. Kononov, G. Kumar, C. Mauer, E. Musick, L. E. Olson, E. Rubow, M. Ryan, K. Springborn, P. Turner, V. Valancius, X. Wang, and A. Vahdat, "Snap: a microkernel approach to host networking." in *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019, pp. 399–413. Cited on pages 3, 16, 170, 174, and 184.

[179] Matt Ranney. (2021) Lessons Learned From Scaling Uber To 2000 Engineers, 1000 Services, And 8000 Git Repositories. [Online]. Available: http://highscalability.com/blog/2016/10/12/lessons-learned-from-scaling-uber-to-2000-engineers-1000-ser.html Cited on page 2.

[180] A. Matveev, N. Shavit, P. Felber, and P. Marlier, "Read-log-update: a lightweight synchronization mechanism for concurrent programming." in *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, 2015, pp. 168–183. Cited on pages 75, 149, 152, and 177.

[181] S. McCanne and V. Jacobson, "The bsd packet filter: A new architecture for user-level packet capture." in *USENIX Winter*, vol. 46, 1993. Cited on page 147.

[182] S. McClure, A. Ousterhout, S. Shenker, and S. Ratnasamy, "Efficient scheduling policies for Microsecond-Scale tasks," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. Renton, WA: USENIX Association, Apr. 2022, pp. 1–18. [Online]. Available: https://www.usenix.org/conference/nsdi22/presentation/mcclure Cited on pages 4, 121, and 175.

[183] P. E. McKenney and J. D. Slingwine, "Read-copy update: Using execution history to solve concurrency problems," in *Parallel and Distributed Computing and Systems*, vol. 509518, 1998. Cited on page 177.

[184] D. Meisner, J. Wu, and T. F. Wenisch, "BigHouse: A simulation infrastructure for data center systems." in *Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2012, pp. 35–45. Cited on page 183.

[185] Memcached: A Distributed Memory Object Caching System. [Online]. Available: https://memcached.org/ Cited on pages 74 and 175.

[186] Meta Inc. RocksDB GitHub. [Online]. Available: https://github.com/facebook/rocksdb Cited on page 176.

[187] (2018) MT40A1G8 DDR4 SDRAM Datasheet. Micron Technology Inc. (Date retrieved: 26 July 2020). [Online]. Available: https://media-www.micron.com/-/media/client/global/documents/products/data-

sheet/dram/ddr4/16gb_ddr4_sdram.pdf?rev=ad85f83238d84f2d920b4ae0d3892638 Cited on page 46.

[188] Microsoft Azure. (2020) SLA for Azure Cosmos DB. [Online]. Available: https://azure.microsoft.com/en-ca/support/legal/sla/cosmos-db/v1_3/ Cited on page 184.

[189] Microsoft Corp., "Receive Side Scaling," http://msdn.microsoft.com/library/windows/hardware/ff556942.aspx. Cited on pages 18, 42, and 111.

[190] A. Mirhosseini, S. Elnikety, and T. F. Wenisch, "Parslo: A Gradient Descent-based Approach for Near-optimal Partial SLO Allotment in Microservices." in *Proceedings of the 2021 ACM Symposium on Cloud Computing (SOCC)*, 2021, pp. 442–457. Cited on pages 2 and 185.

[191] R. Mittal, A. Shpiner, A. Panda, E. Zahavi, A. Krishnamurthy, S. Ratnasamy, and S. Shenker, "Revisiting network support for RDMA." in *Proceedings of the ACM SIGCOMM 2018 Conference*, 2018, pp. 313–326. Cited on page 126.

[192] M. Mitzenmacher and E. Upfal, *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005. Cited on pages 106 and 107.

[193] B. Montazeri, Y. Li, M. Alizadeh, and J. K. Ousterhout, "Homa: a receiver-driven low-latency transport protocol using network priorities." in *Proceedings of the ACM SIGCOMM 2018 Conference*, 2018, pp. 221–235. Cited on pages 2, 4, 14, 15, 49, 54, 56, 72, 73, 105, 166, 170, 183, and 184.

[194] Morgan, Timothy Prickett. The CXL Roadmap Opens Up The Memory Hierarchy. [Online]. Available: https://www.nextplatform.com/2021/09/07/the-cxl-roadmap-opens-up-the-memory-hierarchy/ Cited on page 90.

[195] N. Narula, C. Cutler, E. Kohler, and R. T. Morris, "Phase Reconciliation for Contended In-Memory Transactions." in *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, 2014, pp. 511–524. Cited on pages 76 and 177.

[196] "Netronome NFP-6000 Flow Processor," https://www.netronome.com/media/documents/PB_NFP-6000-7-20.pdf, Netronome, 2022, (Date retrieved: 16 May 2022). Cited on pages 89 and 137.

[197] R. Neugebauer, G. Antichi, J. F. Zazo, Y. Audzevich, S. López-Buedo, and A. W. Moore, "Understanding PCIe performance for end host networking." in *Proceedings of the ACM SIGCOMM 2018 Conference*, 2018, pp. 327–341. Cited on page 91.

[198] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling Memcache at Facebook." in *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI)*, 2013, pp. 385–398. Cited on pages 73, 87, and 182.

[199] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot, "Scale-out NUMA." in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014, pp. 3–18. Cited on pages 4, 7, 20, 36, 49, 54, 97, 129, and 130.

[200] ——, "The Case for RackOut: Scalable Data Serving Using Rack-Scale Systems." in *Proceedings of the 2016 ACM Symposium on Cloud Computing (SOCC)*, 2016, pp. 182–195. Cited on pages 73, 74, 75, 76, 77, 84, and 176.

[201] S. Novakovic, Y. Shan, A. Kolli, M. Cui, Y. Zhang, H. Eran, B. Pismenny, L. Liss, M. Wei, D. Tsafrir, and M. K. Aguilera, "Storm: a fast transactional dataplane for remote data structures." in *Proceedings of the 12th ACM International Conference on Systems and Storage (SYSTOR)*, 2019, pp. 97–108. Cited on pages 26, 41, and 164.

[202] NVIDIA BlueField-3 DPU. NVIDIA Corp. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-3-dpu.pdf Cited on pages 32 and 89.

[203] NVIDIA/Mellanox Innova-2 Flex SmartNIC. NVIDIA Corp. [Online]. Available: https://network.nvidia.com/files/doc-2020/pb-innova-2-flex.pdf Cited on page 89.

[204] (2020) BlueField-2 I/O Processing Unit Product Brief. https://www.mellanox.com/related-docs/prod_adapter_cards/PB_BlueField-2_IPU.pdf. NVIDIA Corp. (Date retrieved: 6 March 2020). Cited on pages 4, 20, 32, 89, and 137.

[205] (2020) Developing a Linux Kernel Module using GPUDirect RDMA. NVIDIA Corp. [Online]. Available: https://docs.nvidia.com/cuda/gpudirect-rdma/index.html Cited on page 171.

[206] (2020) NVIDIA ConnectX-6 DX SmartNIC. https://www.nvidia.com/en-us/networking/ethernet/connectx-6-dx/. NVIDIA Corp. Cited on page 14.

[207] (2020) NVLink and NVSwitch. https://www.nvidia.com/en-us/data-center/nvlink/. NVIDIA Corp. (Date retrieved: 6 March 2020). Cited on pages 4 and 20.

[208] (2022) NVIDIA ConnectX-7 400G Ethernet. https://nvdam.widen.net/s/srdqzxgdr5/connectx-7-datasheet. NVIDIA Corp. (Date retrieved: 19 April 2022). Cited on pages 13, 31, 46, and 148.

[209] (2022) NVIDIA Grace CPU SuperChip. https://www.nvidia.com/en-us/data-center/grace-cpu/. NVIDIA Corp. (Date retrieved: 19 April 2022). Cited on pages 4 and 18.

[210] G. Ottoni, "HHVM JIT: a profile-guided, region-based compiler for PHP and Hack." in *Proceedings of the ACM SIGPLAN 2018 Conference on Programming Language Design and Implementation (PLDI)*, 2018, pp. 151–165. Cited on page 182.

[211] A. Ousterhout, A. Belay, and I. Zhang, "Just In Time Delivery: Leveraging Operating Systems Knowledge for Better Datacenter Congestion Control." in *Proceedings of the 11th workshop on Hot topics in Cloud Computing (HotCloud)*, 2019. Cited on page 166.

[212] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan, "Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads." in *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI)*, 2019, pp. 361–378. Cited on pages 161 and 184.

[213] J. K. Ousterhout, "A Linux Kernel Implementation of the Homa Transport Protocol." in *Proceedings of the 2021 USENIX Annual Technical Conference (ATC)*, 2021, pp. 99–115. Cited on pages 16 and 170.

[214] J. K. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. M. Rumble, R. Stutsman, and S. Yang, "The RAMCloud Storage System." *ACM Trans. Comput. Syst.*, vol. 33, no. 3, pp. 7:1–7:55, 2015. Cited on page 18.

[215] K. Pagiamtzis and A. Sheikholeslami, "Content-addressable memory (CAM) circuits and architectures: a tutorial and survey." *IEEE J. Solid State Circuits*, vol. 41, no. 3, pp. 712–727, 2006. Cited on page 133.

[216] Parallel Systems Architecture Lab (PARSA), "QFlex," March 2020. [Online]. Available: https://qflex.epfl.ch Cited on pages 37, 109, 141, and 148.

[217] 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.*, vol. 33, no. 4, pp. 11:1–11:30, 2016. Cited on pages 15 and 42.

[218] B. Pismenny, H. Eran, A. Yehezkel, L. Liss, A. Morrison, and D. Tsafrir, "Autonomous NIC offloads." in *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021, pp. 18–35. Cited on page 171.

[219] B. Pismenny, L. Liss, A. Morrison, and D. Tsafrir, "The benefits of general-purpose on-NIC memory." in *Proceedings of the 27th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022, pp. 1130–1147. Cited on pages 168 and 176.

[220] Pivotal Software. RabbitMQ: Persistence Configuration. [Online]. Available: https://www.rabbitmq.com/persistence-conf.html Cited on page 71.

[221] J. Postel. RFC768: User Datagram Protocol. [Online]. Available: https://tools.ietf.org/html/rfc768 Cited on page 131.

[222] A. Pourhabibi Zarandi, "Hardware-software co-design of an rpc processor," Ph.D. dissertation, EPFL, Lausanne, 2021. [Online]. Available: http://infoscience.epfl.ch/record/289984 Cited on pages 3, 9, 121, 122, 139, and 141.

[223] 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)*, 2017, pp. 325–341. Cited on pages 2, 34, 39, 42, 48, 50, 52, 62, 73, 74, 111, 149, and 184.

[224] H. Qin, Q. Li, J. Speiser, P. Kraft, and J. K. Ousterhout, "Arachne: Core-Aware Thread Management." in *Proceedings of the 13th Symposium on Operating Systems Design and Implementation (OSDI)*, 2018, pp. 145–160. Cited on page 174.

[225] D. Raghavan, P. A. Levis, M. Zaharia, and I. Zhang, "Breakfast of champions: towards zero-copy serialization with NIC scatter-gather." in *Proceedings of the 18th Workshop on Hot Topics in Operating Systems (HotOS-XVIII)*, 2021, pp. 199–205. Cited on pages 120, 169, and 170.

[226] G. Reinman, B. Calder, and T. M. Austin, "Fetch Directed Instruction Prefetching." in *Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 1999, pp. 16–27. Cited on page 172.

[227] Richard L. Sites, "IC Colloquium : Finding Datacenter Software Tail Latency," 2017, dept. of Computer Science and Communications Colloquium at EPFL. [Online]. Available: https://memento.epfl.ch/event/ic-colloquium-finding-datacenter-software-tail-lat/ Cited on page 12.

[228] S. Roghanchi, J. Eriksson, and N. Basu, "ffwd: delegation is (much) faster than you think." in *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017, pp. 342–358. Cited on pages 86 and 177.

[229] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the Social Network's (Datacenter) Network." in *Proceedings of the ACM SIGCOMM 2015 Conference*, 2015, pp. 123–137. Cited on pages 13 and 52.

[230] SAIL Group, Cornell University. DeathStarBench GitHub. [Online]. Available: https://github.com/delimitrou/DeathStarBench Cited on page 29.

[231] J. D. Salehi, J. F. Kurose, and D. F. Towsley, "Scheduling for Cache Affinity in Parallelized Communication Protocols." in *Proceedings of the 1995 ACM SIGMETRICS International*

Conference on Measurement and Modeling of Computer Systems, 1995, pp. 311–312. Cited on page 173.

[232] ——, "The Performance Impact of Scheduling for Cache Affinity in Parallel Network Processing." in *Proceedings of the Fourth IEEE International Symposium on High Performance Distributed Computing*, 1995, pp. 66–77. Cited on page 173.

[233] ——, "The effectiveness of affinity-based scheduling in multiprocessor network protocol processing (extended version)." *IEEE/ACM Trans. Netw.*, vol. 4, no. 4, pp. 516–530, 1996. Cited on page 173.

[234] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-To-End Arguments in System Design," *ACM Trans. Comput. Syst.*, vol. 2, no. 4, pp. 277–288, 1984. Cited on page 53.

[235] A. Sarma, H. Seyedroudbari, H. Gupta, U. Ramachandran, and A. Daglis. (2022) NFSlicer: Data Movement Optimization for Shallow Network Functions. [Online]. Available: https://arxiv.org/abs/2203.02585 Cited on page 168.

[236] D. Schall, A. Margaritov, D. Ustiugov, A. Sandberg, and B. Grot, "Lukewarm serverless functions: characterization and optimization." in *Proceedings of the 49th International Symposium on Computer Architecture (ISCA)*, 2022, pp. 757–770. Cited on page 172.

[237] B. Schroeder, A. Wierman, and M. Harchol-Balter, "Open Versus Closed: A Cautionary Tale." in *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI)*, 2006. Cited on page 182.

[238] A. Shah, V. Banakar, S. Shastri, M. Wasserman, and V. Chidambaram, "Analyzing the Impact of GDPR on Storage Systems." in *Proceedings of the 11st USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage-XI)*, 2019. Cited on page 1.

[239] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang, "LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation." in *Proceedings of the 13th Symposium on Operating Systems Design and Implementation (OSDI)*, 2018, pp. 69–87. Cited on pages 15 and 21.

[240] Y. S. Shao, S. L. Xi, V. Srinivasan, G.-Y. Wei, and D. M. Brooks, "Co-designing accelerators and SoC interfaces using gem5-Aladdin." in *Proceedings of the 49th Annual IEEE/ACM*

International Symposium on Microarchitecture (MICRO), 2016, pp. 48:1–48:12. Cited on page 171.

[241] M. Silberstein, B. Ford, I. Keidar, and E. Witchel, "GPUfs: integrating a file system with GPUs." in *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013, pp. 485–498. Cited on page 171.

[242] Simpy repostory. [Online]. Available: https://gitlab.com/team-simpy/simpy Cited on page 35.

[243] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, H. Liu, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat, "Jupiter rising: a decade of clos topologies and centralized control in Google's datacenter network." *Commun. ACM*, vol. 59, no. 9, pp. 88–97, 2016. Cited on pages 3, 13, and 14.

[244] A. Singhvi, A. Akella, D. Gibson, T. F. Wenisch, M. Wong-Chan, S. Clark, M. M. K. Martin, M. McLaren, P. Chandra, R. Cauble, H. M. G. Wassel, B. Montazeri, S. L. Sabato, J. Scherpelz, and A. Vahdat, "1RMA: Re-envisioning Remote Memory Access for Multi-tenant Datacenters." in *Proceedings of the ACM SIGCOMM 2020 Conference*, 2020, pp. 708–721. Cited on pages 26, 165, 171, and 183.

[245] M. S. Squillante and E. D. Lazowska, "Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling." *IEEE Trans. Parallel Distributed Syst.*, vol. 4, no. 2, pp. 131–143, 1993. Cited on pages 63 and 173.

[246] A. Sriraman and A. Dhanotia, "Accelerometer: Understanding Acceleration Opportunities for Data Center Overheads at Hyperscale." in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020, pp. 733–750. Cited on pages 72 and 125.

[247] A. Sriraman and T. F. Wenisch, "µTune: Auto-Tuned Threading for OLDI Microservices." in *Proceedings of the 13th Symposium on Operating Systems Design and Implementation (OSDI)*, 2018, pp. 177–194. Cited on pages 140, 180, and 181.

[248] C. Staff, "A second conversation with Werner Vogels." *Commun. ACM*, vol. 64, no. 3, pp. 50–57, 2021. Cited on pages 1 and 182.

[249] L. Suresh, P. Bodík, I. Menache, M. Canini, and F. Ciucu, "Distributed resource management across process boundaries." in *Proceedings of the 2017 ACM Symposium on Cloud Computing (SOCC)*, 2017, pp. 611–623. Cited on page 82.

[250] M. Sutherland, S. Gupta, B. Falsafi, V. Marathe, D. N. Pnevmatikatos, and A. Daglis, "The NEBULA RPC-Optimized Architecture." in *Proceedings of the 47th International Symposium on Computer Architecture (ISCA)*, 2020, pp. 199–212. Cited on pages 9, 82, 140, and 149.

[251] D. Sánchez, R. M. Yoo, and C. Kozyrakis, "Flexible architectural support for fine-grain scheduling." in *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010, pp. 311–322. Cited on page 174.

[252] The Ethernet Alliance, "The 2018 Ethernet Alliance Roadmap," https://ethernetalliance.org/the-2018-ethernet-roadmap/, 2018. Cited on pages 3, 14, and 40.

[253] The Linux Kernel Foundation. (2016) TOE. (Date retrieved: 25 May 2022). [Online]. Available: https://wiki.linuxfoundation.org/networking/toe Cited on page 171.

[254] S. Thomas, R. McGuinness, G. M. Voelker, and G. Porter, "Dark packets and the end of network scaling." in *Proceedings of the 2018 Symposium on Architectures for Networking and Communication Systems (ANCS)*, 2018, pp. 1–14. Cited on pages 31 and 167.

[255] Tom R. Halfhill, "Ice Lake Debuts 10nm, New Cores," *Linley Group Microprocessor Report*, August 2019. Cited on page 63.

[256] A. Tootoonchian, A. Panda, C. Lan, M. Walls, K. J. Argyraki, S. Ratnasamy, and S. Shenker, "ResQ: Enabling SLOs in Network Function Virtualization." in *Proceedings of the 15th Symposium on Networked Systems Design and Implementation (NSDI)*, 2018, pp. 283–297. Cited on pages 42, 125, and 161.

[257] S.-Y. Tsai and Y. Zhang, "LITE Kernel RDMA Support for Datacenter Applications." in *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017, pp. 306–324. Cited on page 26.

[258] H.-W. Tseng, Q. Zhao, Y. Zhou, M. Gahagan, and S. Swanson, "Morpheus: Creating Application Objects Efficiently for Heterogeneous Computing." in *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*, 2016, pp. 53–65. Cited on page 171.

[259] D. Ustiugov, A. Daglis, J. Picorel, M. Sutherland, E. Bugnion, B. Falsafi, and D. N. Pnevmatikatos, "Design guidelines for high-performance SCM hierarchies." in *Proceedings of the 2018 International Symposium on Memory Systems (MEMSYS)*, 2018, pp. 3–16. Cited on page 73.

[260] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot, "Benchmarking, analysis, and optimization of serverless function snapshots." in *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021, pp. 559–572. Cited on page 179.

[261] G. Varghese, *Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004. Cited on pages 82 and 105.

[262] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao, "Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases." in *SIGMOD Conference*, 2017, pp. 1041–1052. Cited on page 1.

[263] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg." in *Proceedings of the 2015 EuroSys Conference*, 2015, pp. 18:1–18:17. Cited on page 179.

[264] W. Vogels. (2019) Modern applications at AWS. [Online]. Available: https://www.allthingsdistributed.com/2019/08/modern-applications-at-aws.html Cited on page 182.

[265] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe, "SimFlex: Statistical Sampling of Computer System Simulation." *IEEE Micro*, vol. 26, no. 4, pp. 18–31, 2006. Cited on pages 34, 37, 109, and 182.

[266] B. Wheeler, "Graviton3 Debuts Neoverse V1," *Linley Group Microprocessor Report*, January 2022. Cited on pages 4, 18, and 148.

[267] A. Wierman and B. Zwart, "Is Tail-Optimal Scheduling Possible?" *Oper. Res.*, vol. 60, no. 5, pp. 1249–1257, 2012. Cited on page 17.

[268] A. Wolnikowski, S. Ibanez, J. Stone, C. Kim, R. Manohar, and R. Soulé, "Zerializer: towards zero-copy serialization." in *Proceedings of the 18th Workshop on Hot Topics in Operating Systems (HotOS-XVIII)*, 2021, pp. 206–212. Cited on pages 30, 120, and 169.

[269] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling." in *Proceedings of the 30th International Symposium on Computer Architecture (ISCA)*, 2003, pp. 84–95. Cited on page 34.

[270] J. Xia, C. Cheng, X. Zhou, Y. Hu, and P. Chun, "Kunpeng 920: The First 7-nm Chiplet-Based 64-Core ARM SoC for Cloud Services." *IEEE Micro*, vol. 41, no. 5, pp. 67–75, 2021. Cited on pages 18 and 148.

[271] J. Yang, Y. Yue, and K. V. Rashmi, "A large scale analysis of hundreds of in-memory cache clusters at Twitter." in *Proceedings of the 14th Symposium on Operating Systems Design and Implementation (OSDI)*, 2020, pp. 191–208. Cited on pages 72, 73, 74, and 75.

[272] J. Yang, Y. Yue, and R. Vinayak, "Segcache: a memory-efficient and scalable in-memory key-value cache for small objects." in *Proceedings of the 18th Symposium on Networked Systems Design and Implementation (NSDI)*, 2021, pp. 503–518. Cited on page 84.

[273] Y. Yuan, M. Alian, Y. Wang, R. Wang, I. Kurakin, C. Tai, and N. S. Kim, "Don't Forget the I/O When Allocating Your LLC." in *Proceedings of the 48th International Symposium on Computer Architecture (ISCA)*, 2021, pp. 112–125. Cited on page 163.

[274] Y. Yuan, Y. Wang, R. Wang, and J. Huang, "HALO: accelerating flow classification for scalable packet processing in NFV." in *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*, 2019, pp. 601–614. Cited on pages 3 and 12.

[275] A. P. Zarandi, S. Gupta, H. Kassir, M. Sutherland, Z. Tian, M. P. Drumond, B. Falsafi, and C. Koch, "Optimus Prime: Accelerating Data Transformation in Servers." in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020, pp. 1203–1216. Cited on pages 6, 20, 30, 63, 89, 127, 145, 169, and 170.

[276] A. P. Zarandi, M. Sutherland, A. Daglis, and B. Falsafi, "Cerebros: Evading the RPC Tax in Datacenters." in *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2021, pp. 407–420. Cited on pages 3, 5, 9, 12, 20, 30, 89, 120, 127, and 135.

[277] Y. Zhang, Y. Gan, and C. Delimitrou, "μqSim: Enabling Accurate and Scalable Simulation for Interactive Microservices." in *Proceedings of the 2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019, pp. 212–222. Cited on page 183.

[278] H. Zhu, K. Kaffes, Z. Chen, Z. Liu, C. Kozyrakis, I. Stoica, and X. Jin, "RackSched: A Microsecond-Scale Scheduler for Rack-Scale Computers." in *Proceedings of the 14th Symposium on Operating Systems Design and Implementation (OSDI)*, 2020, pp. 1225–1240. Cited on pages 87 and 89.

[279] M. Zukerman. (2018, Dec.) Introduction to Queueing Theory and Stochastic Teletraffic Models. arXiv. ArXiv:1307.2968. Cited on page 51.

MARK JOHNATHON SUTHERLAND

Parallel Systems Architecture Lab, INJ 239 (Bâtiment INJ), Station 14, 1015 Lausanne | +41 78 685 18 95 | mark.sutherland@epfl.ch

EDUCATION

Doctor of Philosophy (PhD) in Computer Science Sept. 2016 – Aug. 2022
École Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland
Thesis: Hardware and Software Support for RPC-Centric Server Architecture
Advisors: Profs. Babak Falsafi (EPFL) and Alexandros Daglis (Georgia Institute of Technology)
Research Interests:
- Server architecture
- Datacentre systems and networking
- Concurrency and parallelism

MASc in Computer and Electrical Engineering Sept. 2014 – Jul. 2016
University of Toronto, Toronto, Ontario
Thesis: *Co-Locating Code and Data for Energy Efficient CPUs*
Advisor: Prof. Natalie Enright Jerger
GPA: 4.0/4.0

BASc in Engineering Science Sept. 2009 – May 2014
University of Toronto, Toronto, Ontario
Thesis*: Security At What Price: Investigating the Hardware Cost of Physical Layer Security in MIMO Wireless Communication Systems*
Advisors: Profs. Vaughn Betz *(ECE)* & Ashish Khisti *(Communications)*

EMPLOYMENT

Research Assistant, EPFL September 2016 – present
Lausanne, Switzerland
- Contributed to several projects on hardware-software co-design for datacenter server architecture.
- Supervised younger students, course projects, and interns.
- Core contributor and maintainer of the open-source QFlex project.

IPTV Applications Architect, Intern May 2012 – August 2013
Bell Canada Video Innovation Centre, Ottawa, Ontario
- Created Python scripts that reduced the firmware testing cycle from 2-3 weeks to 1-2 days.
- Verified proper functionality of Bell DSL/Fiber CPE, as well as various components of Bell's backend video network at multiple layers of the TCP/IP stack.

PEER-REVIEWED PUBLICATIONS

Cooperative Concurrency Control for Write-Intensive Key-Value Workloads
Mark Sutherland, Babak Falsafi, and Alexandros Daglis. In Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23). Vancouver, Canada. 2023. (Paper accepted and will appear.)

Cerebros: Evading the RPC Tax in Datacenters
Arash Pourhabibi, Mark Sutherland, Alexandros Daglis, Babak Falsafi. In Proceedings of the 54th Annual International Symposium on Microarchitecture (MICRO'21). Athens, Greece. 2021.
DOI: https://doi.org/10.1145/3466752.3480055

The NeBuLa RPC-Optimized Architecture
Mark Sutherland, Siddharth Gupta, Babak Falsafi, Virendra J. Marathe, Dionisios Pnevmatikatos, Alexandros Daglis. In Proceedings of the 47th Annual International Symposium on Computer Architecture (ISCA'20). Barcelona, Spain. 2020.
DOI: https://doi.org/10.1109/ISCA45697.2020.00027

Optimus Prime: Accelerating Data Transformation in Servers
Arash Pourhabibi, Siddharth Gupta, Hussein Kassir, Mark Sutherland, Zilu Tian, Mario Drumond, Babak Falsafi and Christoph Koch. In Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20). Lausanne, Switzerland. 2019.
DOI: https://doi.org/10.1145/3373376.3378501

RPCValet: NI-Driven, Tail-Aware Balancing of µs-Scale RPCs
Alexandros Daglis, Mark Sutherland, and Babak Falsafi. 2019. In Proceedings of the 2019 Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), April 13–17 2019, Providence, RI, USA.
DOI: https://doi.org/10.1145/3297858.3304070

Design Guidelines for High-Performance SCM Hierarchies
Dmitrii Ustiugov, Alexandros Daglis, Javier Picorel, Mark Sutherland, Edouard Bugnion, Babak Falsafi, and Dionisios Pnevmatikatos. 2018. In Proceedings of the International Symposium on Memory Systems (MEMSYS '18). Washington, DC, USA.
DOI: https://doi.org/10.1145/3240302.3240310

Near Data Processing at Runtime
Mark Sutherland and Natalie Enright Jerger
The 1st International Workshop on Architecture for Graph Processing (AGP-1)
As Part of: ISCA 2017

Not Quite My Tempo: Matching Prefetches to Memory Access Times
Mark Sutherland, Ajaykumar Kannan, and Natalie Enright Jerger
The 2nd Data Prefetching Championship (DPC-2)
As Part of: ISCA 2015

Texture Cache Approximation on GPUs
Mark Sutherland, Joshua San Miguel, and Natalie Enright Jerger
The Workshop on Approximate Computing Across the Stack (WAX 2015)
As Part of: PLDI 2015

PATENTS

Data Transformer Apparatus. (US Patent PCT/EP2019/072783 pending). A. Pourhabibi, S. Gupta, H. Kassir, M. Sutherland, Z. Tian, M. Drumond, B. Falsafi, C. Koch.

RESEARCH GRANTS

Virtual Memory for Post-Moore Servers 2021

- PIs: Babak Falsafi, David Atienza, Abhishek Bhattacharjee, Boris Grot, Mathias Payer
- Proposal ranked #1 by Intel Corp. globally.

TEACHING

(EPFL) CS471 – Advanced Multiprocessor Architecture	September – December 2021
(EPFL) CS212 – Systems Programming Project	February – June 2021
(EPFL) CS522 – Principles of Computer Systems	September – December 2020
(EPFL) CS322 – Introduction to Database Systems	February – June 2020
(EPFL) CS323 – Operating Systems	September – December 2019
(EPFL) EE490 – Lab in Data Science	February – June 2019
(EPFL) CS522 – Principles of Computer Systems	September – December 2018
(EPFL) CS112 – Programming II	February – June 2018
(EPFL) CS307 – Introduction to Multiprocessor Architecture	September – December 2017
(EPFL) MAT232 – Probability and Statistics	January – June 2017
(Toronto) ECE552 – Computer Architecture	September – December 2015
(Toronto) ECE243 – Computer Organization	January – April 2015

STUDENT SUPERVISION

Leyi Sun (Master's student at EPFL) February - June 2022

- I supervised Leyi on a project studying the instruction cache behaviour of web serving software, with the goal of comparing current software stacks to previous research results.

Henry Tseng (Master's student at EPFL) February - June 2022

- I supervised Henry on a project comparing the performance of cutting-edge ARM and x86 servers for a web search workload.

Setu Gupta (Summer@EPFL Intern) May – August 2021

- I supervised Setu during the duration of his internship, to build and integrate a unit testing framework for the QFlex simulation project.
- Performed design document assistance, quality control, and final code review.

CS471 Research Project Mentoring September – December 2021

- Over the course of the semester, CS471 students are required to conduct a research project.
- I proposed and supervised approximately fifteen projects over three months, on a broad range of topics. Sample projects include: analyzing Cloud Gaming as a workload for modern GPUs, evaluating the performance impacts of architectural (ARM Memory Tagging) and software framework (serverless programming) changes on microarchitectural behaviour, and applying gradient compression to the Hybrid Block Floating Point encoding method for AI training.

PROFESSIONAL SERVICE

Web Submissions Co-Chair – 2018 International Symposium on Computer Architecture

Web link: http://iscaconf.org/isca2018/
I served as the Submissions Co-Chair for ISCA'18, alongside my colleague Mario Drumond. We were responsible for the functionality and integrity of the HotCRP conference review system, as well as providing feedback and input on the overall review distribution and paper ranking process. You can read our post-mortem on the entire conference process here: https://bit.ly/2Xeo2tP

QFlex – A Quick and Flexible Rack-Scale Simulator

Web link: https://parsa-epfl.github.io/qflex/
QFlex is an open-source simulator that is specifically designed to accurately model the execution of a modern rack-mounted server. Other open-source simulators lack the vertical scope of QFlex, as it provides the ability for researchers to perform full-device modelling, detailed network simulation, and micro-architectural level studies. To this end, QFlex uses the following open source tools with active communities to provide the modelling fidelity lacked by other simulators: QEMU for device

emulation, and NS-3 for network modelling. QFlex also integrates the SMARTS sampling framework, to drastically reduce simulation turnaround times by using statistical sampling theory to simulate a subset of the program while remaining highly accurate in its measurements. I am the ongoing code maintainer for our modified version of QEMU, which contains extensions to make it suitable for full-system simulation at the cycle-accurate level.

CloudSuite – A Benchmark Suite For Online Services

Web link: http://cloudsuite.ch/
Cloud computing has essentially replaced private enterprise servers for the vast majority of data-intensive online applications. Due to that trend, computer systems researchers have launched many efforts to model realistic scale-out deployments in their publications and research efforts, rather than using traditional benchmarks such as SPEC. CloudSuite is an open source, community supported collection of eight applications which have been specifically selected due to their popularity in today's datacentres; it contains well known applications like Web Search and Data Caching, as well as emerging services such as In-Memory Data Analytics using Apache Spark. CloudSuite is packaged and deployed using Docker containers, an increasingly popular choice for modern datacentre providers.

Organizing Committee - The 2nd Cache Replacement Championship (CRC-2)

Workshop link: http://crc2.ece.tamu.edu/
CRC-2 was a workshop co-located with the 2017 International Symposium on Computer Architecture, and was a competition for researchers to develop the best last-level cache replacement policy under a set of constraints. I served as part of the technical committee that organized the simulation infrastructure and workload trace generation process, focusing on the addition of CloudSuite, an open-source set of benchmarks that is representative of modern datacentre services. Additionally, I applied the SMARTS sampling methodology to demonstrate that the supplied traces operated with statistically bounded errors of less than 5% (at 2σ).

TECHNICAL SKILLS

Programming

- Proficient languages: C/C++ (10 years experience), Python (5 years experience)
- Other languages: x86/ARM Assembly, Rust, Java, bash/zsh (elementary proficiency)
- Development and deployment of multi-container services with Docker & Kubernetes (2 years experience).
- Continuous integration and testing frameworks: TravisCI, GitHub Actions (2 years experience).

Hardware Design

- Full system simulation with various infrastructures (QFlex, SimFlex, gem5).
- Experience designing FPGA hardware using Verilog inside Altera Quartus II/ModelSim environment.
- Utilized SUE/MAX/Cadence First Encounter CAD tools to perform full custom ASIC layout.
- Layout modeling and power analysis with CACTI.

Computer Networking

- Practical experience testing and debugging large TCP/IP networks.
- WireShark and AirPcap for packet capture and analysis.

AWARDS AND HONOURS

Teaching Excellence Award – EPFL School of Computer Sciences Course: Advanced Multiprocessor Architecture	2021
Teaching Excellence Award – EPFL School of Computer Sciences Course: Introduction to Multiprocessor Architecture	2017
Winner, Orbis Entrepreneurship Challenge	2014
Shaw Engineering Scholarship	2009 – 2012
Alberta Premier's Citizenship Award	2009

Valedictorian – Cochrane High School 2009

LANGUAGES

English – Native language.
French – Basic working proficiency, Level B2.
Attained and passed the *DELF tout public* examination, Level B1/B2, Spring 2016.