EPFL

Thèse n° 7217

Hardware-Software Co-Design of an RPC Processor

Présentée le 11 novembre 2021

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

Arash POURHABIBI ZARANDI

Acceptée sur proposition du jury

Prof. P. Ienne, président du jury
Prof. B. Falsafi, directeur de thèse
Prof. C. Delimitrou, rapporteuse
Dr J. van Lunteren, rapporteur
Prof. C. Koch, rapporteur

École
polytechnique
fédérale
de Lausanne

2021

هر کس که داند و داند که داند
اسب خرد از گنبد گردون بجهاند
آن کس که داند و نداند که داند
آگاه نماید که بس خفته نماند
آن کس که نداند و داند که نداند
لنگان خرک خویش به منزل برساند
آن کس که نداند و نداند که نداند
در جهل مرکب ابدالدهر بماند

— منتسب به خواجه نصیرالدین طوسی

Anyone who knows, and knows that he knows,
makes the steed of intelligence leap over the vault of heaven.
Anyone who knows, but does not know that he knows,
should be informed and awakened from his sleep.
Anyone who does not know, but knows that he does not know,
can bring his lame little donkey to the destination nonetheless.
Anyone who does not know, and does not know that he does not know,
is stuck for ever in double ignorance.

— Attributed to Nasir al-Din al-Tusi

To Ardavan, Negar, and my parents.

Acknowledgements

My PhD journey has been the most challenging experience of my life. It transformed me into a better person and taught me the true value of perseverance, patience, empathy, and teamwork. I would have never made it to the finish line if it was not for the amazing people around me, who supported me and helped me push through the hardest times. I owe my most sincere gratitude to all of these people.

First, I am grateful to my advisor, Babak Falsafi. Babak pushed me out of my comfort zone right from the beginning and gave me the opportunity to become a better person. I learned a lot from him. By constantly asking "what are the questions?", he taught me how to think critically and the importance of asking the right questions; and by saying "Just make sure it's perfect.", he pushed me to go beyond what I would consider enough. Babak also taught me how to communicate my ideas effectively, both in writing and in speech. I am thankful for the group culture he has inculcated in PARSA, which forms strong ties among students.

Next, I would like to thank the member of my thesis committee, Paolo Ienne, Christoph Koch, Christina Delimitrou, and Jan Van Lunteren, for their guidance and feedback before, during, and after my thesis exam. Christoph has also been my mentor for the last few years of my PhD journey, and he even collaborated with us on a project related to my thesis. Alexandros Daglis and Mark Sutherland have been close collaborators in most of the work I did as part of my thesis. My other collaborators were Siddharth Gupta, Mario Drumond, Hussein Kassir, and Zilu Tian. Thank you for making my PhD journey more fruitful.

I have been fortunate enough to have several great teachers who greatly influenced my academic journey before I even started my PhD journey. I would like to thank Farshad Khunjush and Ali Hamzeh for their guidance and support throughout my Bachelor's and Master's studies

at Shiraz University. I would also like to thank Armando Fox for his generosity and support. I probably would not have started my PhD at EPFL without his help and support.

Next, I would like to extend my gratitude to the PARSA members, from whom I have probably learned the most during my PhD journey. First, I would like to thank Javier Picorel, who was the most senior student when I joined PARSA and was the person with whom I interacted the most regarding research in the first two years of my PhD. I learned a lot from him, particularly during my first semester that we worked together on a DATE submission. Next, I would like to thank the other two founding members of AlMaNeo, Alexandros Daglis and Mario Drumond. The three of us have been through a lot together, lots of laughter, jokes, tears, and burgers. Those were the good old days. I also learned a lot from them, particularly from Alex. I cannot even put into words how much I have learned from Alex (not limited to the academic world) and how much he has inspired me throughout all these years. He has been my go-to person for almost everything. He was also my bridge to the EPFL's Greek gang, and I am grateful for that. Mario and I spent most of our PhD life together; we lived together for almost four years and also shared an office for nearly five years. He has also been my fellow running enthusiast, and I have learned a lot from him about running. He has mentioned that I am probably the most patient person he has ever met; to that, I have to say he is probably the most impatient person I have ever met. Also, after all these years, I still do not understand how and why he loves hockey and not football as a Brazilian.

I am also grateful to Mark Sutherland, Siddharth Gupta, Ognjen Glamocanin, and Simla Harma for their support and friendship. Mark is one of the most knowledgeable and helpful people I have ever met, and he helped me a lot in my PhD journey. He has also been my go-to person for my English-related questions. Sid can do things in the shortest amount of time, but you should never ask him how he has done things because you may not like the answer. After Mario left, Oggy was my office mate and has been a great help acting as my private French translator. I also appreciate his understanding of my quotes and memes, particularly the ones related to The Office. Simla is very sweet and generous like a lady baker; I would not be surprised to find out that she had worked in a bakery before coming here, she has that kind of warmth. I want to thank Stéphanie Baillargues for being very helpful throughout all these

years with all the stuff related to administrative affairs and beyond. The next person whom I am very thankful for is Effi Georgala. She has been very supportive throughout my PhD journey, and we have had many interesting non-work-related conversations. Finally, I would like to thank Nooshin Mirzadeh for her support and encouragement, Rishabh Iyer—my FIFA buddy and the most competitive person I have ever met, Dmitrii Ustiugov, Ahmet Yüzügüler, Georgios Psaropoulos, Hussein Kassir, Atri Bhattacharyya, Dina Mahmoud, and Zilu Tian.

I am very grateful to my friends from the good old days back in Shiraz, who are now spread out all over the world. These folks are like family to me, and they made Shiraz my second (or even debatably first) home during my six years of study. They have also been a constant source of inspiration, support, and encouragement ever since. I am forever in debt to all of these people because they changed my life and made me a better person. I want to thank all of these cherished friends: Siavash, Zohreh, Mahsa, Negar, Bahareh, Mohammad-Hossein, Afsaneh, Rojin, Saba, Afshin, Masih, Amir-Hossein, Parisa, Saeed, Abbas, Tina, Fatemeh, and Sara. I am looking forward to happy get-togethers, wherever it might be, for years to come. I also want to remember two special friends of mine, who unfortunately left us very early, Maysam and Ardavan. They were both very nice people and the kind of friends you could always count on. Losing Ardavan, in particular, has been probably the most painful incident of my life. He was always cheerful, full of energy and positive attitude, and he would transfer this positivity and energy to everyone around him. It was almost impossible to be sad around him. He was also perhaps the most persistent and most persevering person I have ever met. He was about to finish his PhD and wanted to become a professor. Thinking about him in the past year has been a major source of inspiration and encouragement for me. That is why I have decided to dedicate this thesis to him. May both of you rest in peace. You are both deeply missed.

Last but not least, I want to extend my gratitude to my family for their endless love, support, and encouragement throughout my whole life. I would like to thank my parents, Mohammad-Ali and Nasrin, for everything they have done and all the sacrifices they have made for me. I hope I have made them proud and happy. Next, I am thankful to my brother Nima and his lovely wife Parisa, who have also been very supportive and kind to me all these years. Nima has always been a source of inspiration in so many ways, and I am grateful to him for being by

my side growing up. I hope we all get together in one place very soon. I would like to thank my dear uncle Hossein, who has been like a godfather to me throughout all my life. Finally, I would like to thank my best friend, my partner in crime, and the love of my life, Negar. She is the main reason I am what I am now, and I could not have finished this journey without her. Before her, my whole world was in black and white, and she brought color to my world. I am so grateful to have had Negar in my life for the past twelve years, and I am looking forward to spending a lifetime with her.

I would also like to thank the people and organizations that financially supported my PhD journey. I am thankful to EPFL, the Swiss National Science Foundation, and also the Swiss people who supported me through their taxes.

Lausanne, September 2021 Arash Pourhabibi

Abstract

The booming popularity of online services has led to a major evolution in the way these services are built and deployed. To cope with such online data-intensive services, service providers deploy several massive-scale datacenters, also referred to as warehouse-scale computers, each populated with up to hundreds of thousands of servers. The services also follow the paradigm of microservices, which decomposes online services into fine-grained software modules frequently communicating over the datacenter network using Remote Procedure Calls (RPCs). Microservices simplify and accelerate software development and allow independent development and performance debugging of each microservice using the most suitable programming language and tools. Furthermore, microservices simplify software deployment and enable scaling and updating individual microservices independently. However, because services are deployed in a distributed fashion, frequent communication is needed to complete a request, putting pressure on the networking infrastructure of the datacenter.

As a result, networking technology has been evolving rapidly both in software and hardware to address this extra communication overhead, also referred to as the "RPC tax" in datacenters. High-performance network fabrics and new network protocols have been developed to address the performance and scalability issues associated with the increasing volume of communication between software components. Although the tax on inter-microservice communication includes both the RPC layer and the underlying network stack, ongoing advancements have mainly targeted the network stack, leading to a drastic reduction of the networking latency and exposing the RPC layer itself as a bottleneck. While modern fabrics continue improving network bandwidth, silicon's efficiency and density scaling met an abrupt slowdown with the end of Dennard scaling and the slowdown of Moore's law, putting more pressure on the RPC

layer running on the general-purpose CPUs. Overall, the RPC layer accounts for a significant fraction of both a single request's latency and the datacenter's total compute capacity; thus, optimizing the hardware-software stack for RPCs is of critical importance.

In this thesis, we break down the underlying modules that comprise production RPC layers and show that CPUs can only expect limited improvements for such tasks, mandating a shift to hardware to remove the RPC layer as a limiter of microservice performance. Motivated by the growing RPC tax in datacenters, we advocate for hardware-software co-design to evade the RPC tax. We present design principles guiding the architecture of an RPC processor and show that conclusively removing the RPC layer bottleneck requires all of the RPC layer's modules to be executed by a NIC-attached hardware accelerator. We propose a NIC-integrated RPC processor that runs production RPC layers and acts as an intermediary stage between the NIC and the microservice running on the CPU. Because such an RPC processor can peek into the request's data, it opens up further opportunities such as intelligent load balancing and request dispatch. We make the case that such an RPC processor is an ideal candidate for inclusion in future server chips to better support and run microservices as they decompose into even finer granularity.

Keywords: datacenters, servers, warehouse-scale computers, microservices, remote procedure calls (RPCs), datacenter tax, serialization, data transformation, co-design, hardware acceleration.

Résumé

La popularité croissante des services en ligne a entraîné une évolution majeure dans la manière dont ces services sont construits et déployés. Pour soutenir ces services en ligne à forte intensité de données, les fournisseurs de services déploient plusieurs centres de données à grande échelle, également appelés ordinateurs à l'échelle d'un entrepôt, qui comptent chacun jusqu'à des centaines de milliers de serveurs. es services suivent également le paradigme des microservices, qui décompose les services en ligne en modules logiciels à grain fin communiquant fréquemment sur le réseau du centre de données au moyen d'appels de procédure à distance (RPC). Les microservices simplifient et accélèrent le développement de logiciels et permettent le développement indépendant et le débogage des performances de chaque microservice à l'aide du langage de programmation et des outils les plus appropriés. En outre, les microservices simplifient le déploiement des logiciels et permettent la mise à l'échelle et la mise à jour de chaque microservice de manière indépendante. Cependant, comme les services sont déployés de manière distribuée, des communications fréquentes sont nécessaires pour répondre à une demande, ce qui exerce une pression sur l'infrastructure réseau des centres de données.

En conséquence, la technologie des réseaux a évolué rapidement, tant au niveau des logiciels que du matériel, pour faire face à cette surcharge de communication, également appelée "taxe RPC" dans les centres de données. Des composants de réseau à haute performance et de nouveaux protocoles de réseau ont été développés pour résoudre les problèmes de performance et d'évolutivité associés au volume croissant de communication entre les composants logiciels. Bien que la taxe sur les communications inter-microservices englobe à la fois la couche RPC et la pile réseau sous-jacente, les progrès en cours ont principalement ciblé la pile réseau, ce qui

a entraîné une réduction drastique de la latence du réseau et exposé la couche RPC elle-même comme un goulot d'étranglement. Alors que les puces intégrés modernes continuent d'améliorer la bande passante du réseau, l'efficacité du silicium et l'augmentation de la densité ont connu un ralentissement brutal avec la fin de l'échelle de Dennard et le ralentissement de la loi de Moore, ce qui a accentué la pression sur la couche RPC fonctionnant sur les CPU polyvalents. Dans l'ensemble, la couche RPC représente une fraction significative de la latence d'une seule requête et de la capacité de calcul totale du centre de données ; par conséquent, l'optimisation de la pile matérielle-logicielle pour les RPC est d'une importance critique.
Dans cette thèse, nous décomposons les modules sous-jacents qui composent les couches de production RPC et montrons que les CPU ne peuvent s'attendre qu'à des améliorations limitées pour de telles tâches, ce qui rend obligatoire un changement de matériel pour supprimer la couche RPC en tant que limite de la performance des microservices. Motivés par la taxe RPC croissante dans les centres de données, nous plaidons pour une co-conception matériel-logiciel afin d'eviter la taxe RPC. Nous présentons les principes de conception guidant l'architecture d'un processeur RPC et montrons que pour éliminer de manière concluante le goulot d'étranglement de la couche RPC, il faut que tous les modules de la couche RPC soient exécutés par un accélérateur matériel relié à une carte réseau. Nous proposons un processeur RPC intégré à la NIC qui exécute les couches de production RPC et agit comme une étape intermédiaire entre la NIC et le microservice fonctionnant sur le CPU. Parce qu'un tel processeur RPC peut observer des données de la demande, il ouvre de nouvelles possibilités telles que l'équilibrage intelligent de la charge et la répartition des demandes. Nous démontrons qu'un tel processeur RPC est un candidat idéal à inclure dans les futures puces de serveur pour mieux supporter et exécuter les microservices au fur et à mesure qu'ils se décomposent en une granularité encore plus fine.

Mots clés : centres de données, serveurs, ordinateurs à l'échelle d'un entrepôt, microservices, appels de procédure à distance (RPC), taxe de centre de données, sérialisation, transformation de données, co-conception, accélération matérielle.

Contents

List of Figures

List of Tables

1 Introduction

Today's connected world is fundamentally enabled by the existence of datacenters; the online services they deliver are ubiquitous in the lives of billions of daily users [2]. Email, social networking, web search, and e-commerce are a few examples of such popular massive-scale services. Google now processes over 90,000 search queries every second on average, which translates to over six billion searches per day and over 2.2 trillion searches per year worldwide, showing a 2× rise in the past few years [3]. Similarly, Facebook reported over 2.8 billion active monthly users [4, 3], and Amazon has an active customer base of over 300 million people, resulting in over three million item shipments per day. With every user generating data and each user request requiring data traversal of this massive dataset, compute and storage demands are growing dramatically, and software has undergone a major evolution [2]. To cope with such online data-intensive services, providers deploy several massive-scale datacenters, also referred to as warehouse-scale computers, each populated with tens of thousands of servers [2].

Many of the applications hosted on these datacenters are interactive, latency-critical services with strict performance and availability constraints [2, 1, 5]. Moreover, these applications are frequently updated and need to have short release cycles in the order of a couple of weeks or even days [2]. To meet these requirements, modern online services are shifting away from complex monolithic services that encompass the entire application functionality in a single bi-

nary to graphs with tens or hundreds of single-purpose, loosely-coupled *microservices* [2, 1, 6]. The microservices architecture provides composable software design, with each microservice being responsible for a small subset of the application functionality. Hence, microservices not only simplify and accelerate software development, but also facilitate deployment, scaling and updating individual microservices independently [2, 1, 6]. Moreover, microservices allow independent development of each microservice using the most suitable programming language and tools and simplify correctness and performance debugging, as each microservice can be isolated easily [2, 1, 6].

Despite their benefits, microservices have broad implications ranging from cloud management and programming tools down to operating systems and datacenter hardware design, as they significantly depart from the way online services were traditionally designed [1]. In particular, while communication among software components occurs through simple function calls within a server in monolithic services, microservices require inter-server communication through a common API, such as Remote Procedure Calls (RPCs) or REST [2, 1, 6]. Although decomposing a monolith into microservices implies that each microservice does only a small fraction of the application-level work, the total time spent on inter-microservice RPCs increases in proportion to the number of microservices.

1.1 RPC Tax in Datacenters

Microservices are typically too simple to involve considerable processing; hence, the per-server amount of work required for a request is small, being comparable in terms of latency to the cost of inter-server communication. Consequently, microservices spend a considerable fraction of their execution time in communication, which gets exacerbated as service time shrinks due to higher degrees of service decomposition. In fact, a recent study has shown that the communication overhead can take up to 75% of a microservice's execution time [1]. The increase in communication to computation ratio creates a challenge to minimize the "tax" associated with each RPC. Therefore, inter-microservice communication within the datacenter becomes a first-order performance concern, leading to a "hunt for the killer microseconds"

across the entire datacenter system stack [7], particularly the parts exercised by RPCs.

The importance of communication has resulted in a recent wave of fast evolution in datacenter network infrastructure. As datacenter networks continue to scale in bandwidth, with speeds up to 1Tbps on the roadmap for both Infiniband and Ethernet [8, 9], there exists a large body of work to optimize both hardware and software [10, 11, 12, 13, 14, 15, 16] to operate at these rates. Modern datacenter network topologies [17, 18] and protocols for optimized congestion control [10, 12, 14] achieve network traversals of a few microseconds (μs) with high predictability. Furthermore, transport protocols in either user-space [19] or hardware [20, 21, 22] have drastically shrunk the cost of the transport layer from 10s of μs [23] to as low as sub-μsvalues [13].

Although the tax on inter-microservice communication includes both the RPC layer and the underlying network stack, recent research has targeted chiefly the network stack, leading to a drastic reduction of the networking latency. Hence, the time spent in the RPC layer is becoming a significant fraction of the end-to-end cost of invoking a microservice. Recent studies have reported that the overhead of the RPC software layer can be in the order of tens of microseconds, which is in the same ballpark of the service time of a simple microservice [7]. This cost is a direct consequence of a deep software stack with multiple layers of functionality that is executed even before the proper application-level computation to service the RPC request starts.

The RPC software overhead is not only on the critical path of every RPC request, but also accounts for a significant fraction of a datacenter's load. A recent study has shown that the RPC layer consumes about 12% of the total CPU cycles at Google's datacenters [24], which is not limited to running microservices. While modern fabrics continue improving network bandwidth, silicon's efficiency and density scaling met an abrupt slowdown with the end of Dennard scaling and the slowdown of Moore's law, putting more pressure on the RPC layer running on the general-purpose CPUs. Overall, the RPC layer accounts for a significant fraction of both a single request's latency and the datacenter's total compute capacity; thus, optimizing the hardware-software stack for RPCs is of critical importance.

Despite the critical nature of the RPC layer for microservice performance, CPUs are ill-suited to execute the RPC layer tasks. Even though the underlying tasks are inherently parallel, the software implementations are unable to extract said parallelism. The tasks are variable-sized and too fine-grained for thread-level parallelism to amortize synchronization costs. Additionally, it has already been reported that CPUs are plagued by instruction supply problems when executing microservices [1]. This problem will worsen with the number of functions, message types, and nested RPCs that make up a microservice. When the inefficiencies of CPU-centric RPC processing are combined with the instruction supply issues in microservices, using dedicated hardware for RPC tasks becomes an attractive solution.

The significant role of the RPC layer in the datacenter's total compute capacity and its relatively high latency overhead has motivated our effort to optimize and co-design the hardware and software stack exercised by RPCs. In order to justify the investment in dedicated hardware, it must be widely applicable and also contain some configurability for the sake of future software deployments. While there is a high diversity in applications running in datacenters and application code gets updated frequently [2], the APIs that applications expose to offer RPC functionality are much narrower and more stable [24]. We, therefore, believe these software layers are good candidates to inspect and accelerate. The commonality of RPC tasks justifies the investment in dedicated hardware for processing the RPC layer. Performing the RPC layer in a dedicated hardware unit integrated with the NIC opens up further opportunities for network-centric RPC request steering, such as a policy based on function affinity.

1.2 Thesis Goals

The primary goal of this thesis is the drastic acceleration of inter-microservice RPCs. As the microservices software architecture continues to proliferate, the common RPC layer gluing the microservices together is becoming a bottleneck. In this thesis, we claim that because CPUs are unable to perform the RPC layer's underlying functionality at rates matching commodity NICs, it is necessary to execute the RPC layer in hardware to ensure servers keep pace with improving network line rates. To that end, we present design principles and constraints

guiding the architecture of RPC processing hardware and propose a NIC-integrated RPC processor that executes the RPC layer. We show that RPC offload not only accelerates the inter-microservice communication, but also improves the CPU's performance when executing the microservice by improving its instruction supply. Moreover, it opens up new opportunities for more intelligent request dispatch policies that can further improve the performance and efficiency of server systems. The statement of this thesis is as follows:

Latency-critical microservice deployments require specialized hardware for RPC processing to match the performance of microsecond-scale software and network stacks.

1.3 Thesis Contributions

This thesis identifies RPC processing as a common yet costly task in modern datacenters and motivates the need for hardware-software co-design for rapid and flexible RPC processing. We introduce guidelines for designing such an RPC processor that enables evading the RPC tax in datacenters. We then implement a proof-of-concept instance of such an RPC processor and demonstrate its benefits. Through a combination of real-hardware measurements, cycle-accurate simulation, and analytical and RTL modeling, we make the following contributions.

First, we demonstrate how increasing demands for fast inter-server communication and reduction in service time stemming from the proliferation of microservices, combined with continuous developments in networking and network bandwidth scaling and the slowdown in silicon density, necessitate hardware-software co-design to transfer functionality from the CPU to bespoke hardware accelerators designed to accelerate the communication software stack. We argue that despite the critical nature of the RPC layer for microservice performance, CPUs are ill-suited to execute the RPC layer tasks. We quantify the cost associated with the RPC layer, and after dissecting the RPC layer, we present insights on (i) why the RPC layer is costly and CPUs are ill-suited to perform the RPC layer functionalities, and (ii) why multi-threading or staging fails to improve the RPC layer's performance. We motivate the need for an RPC processor to address the growing cost of RPCs in datacenters and to shrink the gap between

the CPU and network processing rates.

Second, motivated by the growing cost of RPCs in datacenters, we present a set of guidelines and constraints for designing and architecting an RPC processor. We advocate for a design that (i) supports execution of all three modules comprising the RPC layer and leaves only the application business logic to CPU, (ii) resides logically between the server's NIC and its CPU cores to eliminate excessive offload overheads, (iii) is integrated with server's NIC to minimize silicon deployment costs and to enable affinity-based request steering, which improves instruction locality, (iv) employs a new abstraction called transformation schema that simply uses type identifiers and memory addresses, enabling parallelism and making the accelerator compatible with various frameworks, and (v) comprises specialized hardware converters which can perform costly data transformations in a handful of cycles and support a variety of operations defined by the software.

Third, following the aforementioned design principles, we present Cerebros, our implementation of a full RPC processor. Cerebros is integrated with the NIC and NEBULA architecture [25] and is able to execute the Apache Thrift RPC layer $37-64\times$ faster than CPU according to our experiments. Additionally, Cerebros also improves the CPU's performance when executing the application logic of microservice by shrinking the instruction working set and improving its instruction supply. Cerebros also features a novel affinity-based request steering policy, which provides further reduction in execution time for microservices whose functions contend for cache space. Our evaluation using the DeathStarBench microservice suite [1] shows Cerebros reduces the CPU cycles spent per microservice request by $1.8-14\times$. We believe Cerebros is an ideal candidate for inclusion in future server chips to support microservices as they decompose into even finer granularity.

1.4 Thesis Organization

This thesis is organized as follows. Chapter 2 provides background on key application and technology trends that necessitate a rethink in the way RPCs are handled. Chapter 3 motivates

the need for an RPC processor to address the growing RPC cost in datacenters. Chapter 4 presents key design principles of a specialized RPC processor that can remove the RPC layer's burdensome tasks. Chapter 5 introduces Cerebros, our implementation of a full RPC processor following the design principles of Chapter 4. Chapter 6 details our evaluation methodology, and Chapter 7 evaluates the performance impact of Cerebros on microservices and justifies the design choices we made in Chapter 4. Finally, Chapter 8 discusses related work, followed by Chapter 9, which discusses future research directions and concludes the thesis.

1.4.1 Bibliographic Notes

This thesis was conducted under the supervision of my advisor, Babak Falsafi. Portions of this document are based on the following publications: "Cerebros: Evading the RPC Tax in Datacenters", published in the Proceedings of the 54th IEEE/ACM International Symposium on Microarchitecture (MICRO'21) [26], and "Optimus Prime: Accelerating Data Transformation in Servers", published in the Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20) [27].

2 Application and Technology Trends

Online services are ubiquitous in the daily lives of billions of users. They need to operate at scale across geographical regions and offer fast content delivery as well as high resource utilization at low cost [6]. To cope with these needs, service providers have been evolving the way they develop, deploy and scale their services. This chapter provides an overview of the recent software and hardware technology trends in the context of online services and highlights the role of inter-server communication in datacenters. First, we discuss the trend toward software decomposition that led to the rise of microservice and the finer-grained RPCs, which have already reached the μs-scale (Section 2.1). Next, we look at the recent trends in server design; we discuss what a modern datacenter server system looks like and how networking technology in particular, has evolved to support the fine-grain communication needed by microservices (Section 2.2).

2.1 Datacenter Services

The rising popularity of cloud services, such as web search, social networking, e-commerce, and video streaming has urged service providers to rethink the way they develop, deploy and scale their services [2, 1]. Most modern cloud applications are interactive and have strict performance and availability constraints. Such services require tens of thousands of servers and petabytes of storage. Additionally, these services are under constant development and

have release cycles on the order of a couple of weeks or even days [2]. To meet these often contradicting needs, modern online services have experienced a major design shift from monoliths to microservices [1].

2.1.1 The Rise of Microservices

Traditionally, online services were developed and deployed as a monolith, meaning that the entire application was built as a single unit. The monolithic design approach for building such cloud services makes them hard to build, update, and scale [1, 6]. Over time the application grows into a monstrous monolith that is too large and complex to be fully understandable. Any change to the application requires the entire monolith to be rebuilt and deployed. In order to scale, the entire application, rather than parts of it that require more resources, has to be scaled.

As a result, online service providers, such as Twitter, Uber, Netflix, eBay, and Amazon, are increasingly building their services using the microservices architecture. This philosophy revolves around building smaller and modular components, the microservices, connected via clean APIs, such as remote procedure calls (RPCs) or RESTful APIs [1, 6]. A service typically implements a set of distinct features or functionality, and exposes an API that is consumed by other microservices or by the application's clients.

Each microservice may be developed separately by a different group of developers using different languages and tools. Components can be replaced or added seamlessly later as the business evolves and grows, making application lifecycle management both agile and scalable [6]. Hence, it is not uncommon to have release cycles on the order of a couple of weeks or even days [2]. The microservices architecture also enables developers to isolate the effect of a failure to individual components, thus making it easier to build fault-tolerant and reliable software systems. Similarly, it provides better scalability by enabling service providers to scale individual microservices based on the current demand, rather than replicating the whole monolith system [6].

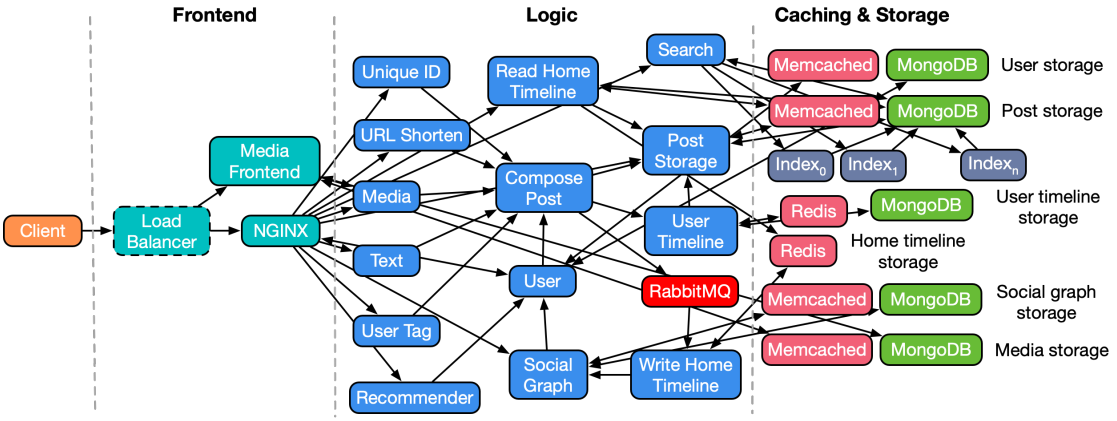


Figure 2.1: The architecture of the social network application from DeathStarBench [1].

Figure 2.1 illustrates a simple application that is created using the microservices architecture. It presents the architecture of the social network application of the DeathStarBench benchmark suite [1]. The blue boxes represent the mid-tier microservices, each performing a small subset of the overall business logic of the application. A client request reaches a web server (e.g., NGINX) after going through a load balancer. Depending on the type of user request, the web server then creates several requests to the microservices in charge of preparing the response to that request. For example, if the request is to create a new post, various post contents go through various microservices designed to handle each of the parts (i.e., text, user tags, media, or URLs). Each microservice may also invoke other microservices to complete the request. As the mid-tier microservices are stateless, the processing flow eventually reaches the data layer in charge of caching and storing the data. The invocations will eventually return one by one, and the final response is prepared to be sent back to the client.

While the microservices architecture brings many advantages in the form of scalability, reliability, programmability, and deployment, this architecture has drawbacks like every other technology. One of these drawbacks is the complexity that arises from the fact that a microservices-based application is in fact a distributed system running on hundreds or thousands of machines. Hence the services must interact using an inter-microservice communication mechanism over the network. Because microservices are typically simple and small (i.e., the per-service amount of work required for a request is small), the communication cost

can be a limiting factor for performance scalability.

2.1.2 Inter-Microservice Communication

In a monolithic application, components invoke one another using language-level methods or function calls. However, in the case of microservices architecture, the microservices composing an end-to-end application use an inter-microservice communication mechanism to interact with one another. These interactions can be either one-to-one, where each client request is processed by exactly one service instance; or one-to-many, where each request is processed by multiple service instances [28]. While one-to-one interactions can be synchronous or asynchronous, one-to-many interactions are always asynchronous.

Request/response is the most common type of one-to-one interaction, and is usually preferred as a synchronous operation. In this case, microservices typically communicate over Remote Procedure Calls (RPCs) or a RESTful API. REST is an inter-process communication (IPC) mechanism that (almost always) uses HTTP verbs for manipulating resources, which are referenced using a URL. For example, a GET request returns the representation of a resource, which might be in the form of an XML document or JSON object, a POST request creates a new resource, and a PUT request updates a resource [28]. While REST provides a simple IPC mechanism, it is typically only used as a gateway API for the clients, such as browsers or mobile apps, to access a service, rather than inter-microservice communication.

Alternatively, services can communicate using RPCs for synchronous request/response interactions. The RPC model is well-known and has been used for several decades in distributed computing. At the high level, the RPC model allows a service to expose an interface similar to a local function (or procedure) to other services over the network. A service that wants to invoke an RPC exposed by another service prepares an RPC request message indicating which service it is requesting and specifying the RPC's arguments. That information is then encapsulated in a network packet and delivered to the target server, which extracts the request message from the network packet, executes the invoked function locally, and responds to the requester with the result of the invoked function via the network.

RPCs are typically used for one-to-one, synchronous request/response interactions, where the client expects the response to arrive in a timely fashion. However, there are cases where asynchronous communication is preferred, or a one-to-many interaction is needed, following the Publish/Subscribe (Pub/Sub) pattern. In such cases, microservices interact via message queues or Pub/Sub systems, such as Amazon SQS [29] and SNS [30], and RabbitMQ [31]. At the high level, Pub/Sub systems include intermediary channels, knows as topics, and for each topic, they maintain a list of subscribers to relay messages to. To broadcast a message, the publisher simply pushes a message to a topic. Pub/Sub systems can be used to enable event-driven architectures, or to decouple applications in order to increase performance, reliability, and scalability. While such IPC mechanisms are used in datacenters, RPCs are still by far the most common communication mechanism used by microservices. Thus we limit our focus to this type of inter-process communication.

RPC frameworks like Apache Thrift [32, 33], or Google's gRPC [34], usually have a layered architecture, with each layer providing a unique functionality. Common functionalities related to RPC frameworks include data (un)marshaling or data transformation, dispatch and load balancing, encryption and authentication, and compression. The dispatch layer's main responsibility is detecting the service type requested by the incoming message and invoking the respective service function to process the request. Additionally, it can also be responsible for load balancing the incoming requests. Based on the service type and definition, the data transformation layer prepares the input arguments for the service function by parsing the payload. The same data transformation layer is later used to serialize the response message. While these layers are required for every RPC, other layers (e.g., encryption and compression) are optional.

Microservices are typically too simple, and the per-service amount of work required for a request is small, being comparable in terms of latency to the communication cost. The RPC stack overhead is not only on the critical path of every RPC, but also accounts for a significant fraction of a datacenter's load: in Google's datacenters (which are not limited to running microservices), the RPC software stack, including the management of protocol

buffers, consumes about 12% of the datacenter's total CPU cycles [24]. The growth in the inter-microservice communication overhead has led to a hunt for the "killer microseconds" across all layers of the datacenter system stack [7].

2.2 Datacenter Building Blocks

The rapid growth of cloud services not only has radically changed the way services are built, but it has also created a new model for how servers are built and connected together. Modern datacenters are treated as one massive computer—a warehouse-scale computer (WSC)—, where the massive amounts of well-connected storage and processing resources are amortized across many ubiquitous workloads and a large number of users [2]. The building blocks of modern datacenters are commodity server machines connected via commodity networking equipment, which provide better performance per cost ratios than high-end components due to economies of scale [2, 35, 36].

2.2.1 Server Architecture

At the core of modern datacenters or WSCs, there are mid-range server machines that are organized in racks interconnected by hierarchies of networks. Such racks include tens of servers that all share the same power and cooling infrastructure. Figure 2.2 shows a high-level architecture of a canonical Intel-based server node in modern datacenters. Each server blade typically incorporates two CPU sockets filled with mid-range server-class CPUs, each with around 20 cores [2]. More recently, servers also feature additional compute hardware, such as GPUs or custom accelerators like TPUs [37]. While a decade ago, the deployment of specialized accelerators in WSCs was limited, the slowdown of Moore's law and the wide adoption of deep learning models have led to a rise in the adoption of specialized hardware in datacenters [2]. Examples of such deployments include the wide usage of GPUs and TPUs for deep learning workloads in Google datacenters [37] and Microsoft's usage of FPGA-based accelerators in their datacenter fleet [38, 22].

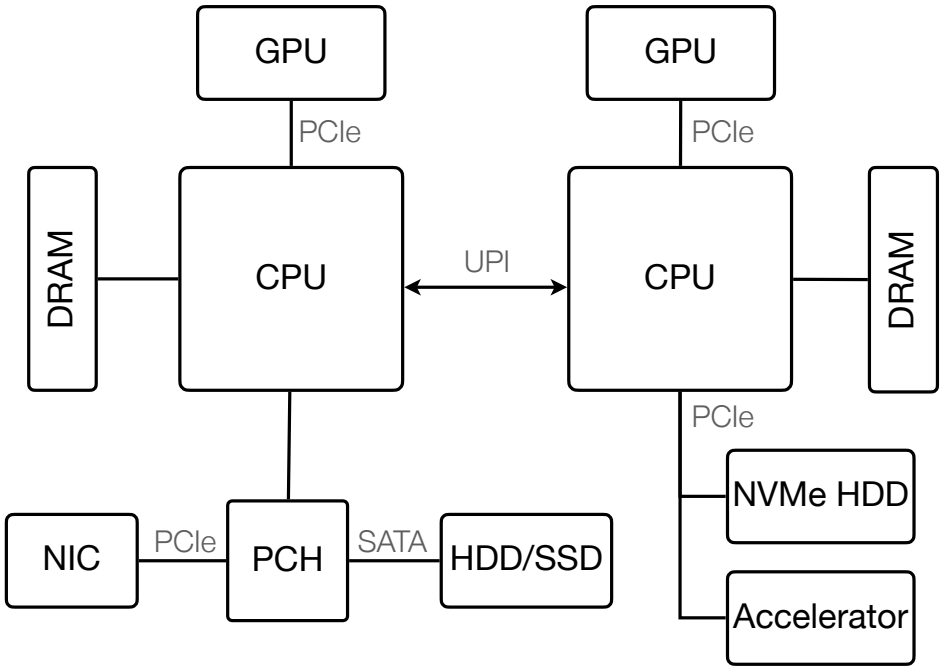


Figure 2.2: The high-level architecture of an Intel-based Server.

As for storage and memory, WSCs incorporate a combination of DRAM, disks, and flash SSDs. Due to the rapid evolution of datacenter networking technologies and the increasing demand for fast access to data, more and more data is being kept in faster storage/memory devices (i.e., SSDs and DRAM). Hence, servers can have 100s of GBs of DRAM and TBs of SSD. Emerging technologies such as non-volatile memory [39, 40] provide another tier between today's DRAM and storage hierarchy [2]. Lastly, one of the most important building blocks of WSCs is the networking hardware. In the following section, we focus on the networking technologies used in modern datacenters, and we look at the recent advancements (both in hardware and software) in this domain.

2.2.2 Datacenter Networking Technology

In order to cope with the increase in networking demand and to provide the requisite capacity for online services, today's datacenters have observed focused evolution efforts in the networking and systems domains. These efforts have led to an overall increase of network bandwidth and a drastic reduction in the latency of both transport protocols and datacenter network

traversals. Datacenter network fabrics have been scaling their capacity because bandwidth demands are doubling every 12-15 months [18]. Custom network topologies that resemble a multi-stage Clos network with many layers of aggregation are used by datacenter operators to provide enough path diversity that allows all inter-server communications to attain uniform high capacity [18, 41, 42, 17].

Server NICs have also seen a rapid increase in their bandwidth in the past decade, from the 40Gbps NICs deployed in Google datacenters in 2012 [18], to current products that offer 200Gbps bandwidth and 220M IOPS [43]. Future NIC generations are expected to have even higher capacities as both Ethernet and InfiniBand have forecasted to have bandwidths as high as $1 Tbps$ soon. Such high speeds demand optimized transport protocols that allow server systems to keep up with this tremendous growth.

Latency-optimized transport protocols such as Homa [14] and NDP [12] have been proposed to provide μs-scale round-trip times, 100× better than traditional TCP/IP at the 99th percentile [14, §5.1]. These newly proposed transport protocols are designed explicitly for the inter-microservice RPCs. 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, dynamically allocating in-network priorities, and using the receivers to actively pull new packets from senders [14]. The combination of massive NIC bandwidth, optimized network topologies and protocols that provide μs-scale message delivery times has shifted the burden to the hardware and software on the servers that are network endpoints.

Recent industry and academic systems have started to redesign legacy network software to provide commensurate performance. These solutions range from user-space network stacks (e.g., DPDK [19]) to software dataplanes such as IX [44] and Arrakis [45]. Hardware-assisted solutions such as at-scale RDMA over Converged Ethernet (RoCE) deployments [20] and their derivatives (e.g., OmniPath [46]), and custom hardware solutions like Microsoft's Catapult [22] all perform protocol termination in hardware, shrinking the latency overheads of network stack processing even further by delivering packets directly to the application layer.

Moreover, we have observed a recent trend toward network-compute integration in both academia and industry. This type of integration not only reduces the communication latency by eliminating the costly IO interconnects such as PCIe, but also frees up CPU cycles by offloading functionality traditionally performed on the CPU. Academic examples include Scale-Out NUMA [21], the FAME-1 RISC-V RocketChip SoC [47], and the NanoPU [48]. Commercial examples include Oracle's Sonoma [49], Calxeda's ARM SoC [50], and integrated Ethernet MACs in Intel's Xeon-D line [51]. Such integrated network interfaces can also perform server-side load balancing, further enhancing the performance [52, 25]. In future chapters, we show how such designs can be extended to also perform the RPC layer's functionality and use the information inside the message header to perform a more intelligent request steering.

We conclude that due to the concerted efforts of system architects and their recent hardware and software proposals, network protocol processing and software overheads have largely moved off the critical path for handling μs-scale RPCs. In the following chapter, we show that once the transport and network layers are out of the way, the RPC layer itself is the main performance limiter.

3 The Need for an RPC Processor

The popularity of online services has urged service providers to rethink the way they develop and scale their services. As a result, modern online services are built using the microservices architecture, where the whole service is decomposed into tens or hundreds of small modular components connected via a clean API, such as RPCs [1, 6, 7, 2, 53]. While the microservices architecture brings many advantages in the form of scalability, reliability, programmability, and deployment, this architecture has drawbacks like every other technology. A single incoming user request may require hundreds or even thousands of servers communicating with each other to process the request [7, 24]. Typically, the per-server amount of work required per request is small, being comparable in terms of latency to the cost of inter-server communication [1]. Therefore, efficient network communication is a first-order performance concern.

While modern fabrics continue improving network bandwidth, silicon's density scaling met an abrupt slowdown [54, 55], putting more pressure on the system stack running on the general-purpose CPUs. To quantify, running Microsoft Azure's 40GbE network stack on commodity servers already requires the use of two full CPU cores, and dedicating more cores is an untenable solution at future network bandwidths [56]. The signaling rates of InfiniBand have been doubling every three years for the past decade, resulting in commensurate bandwidth improvements, and such an improvement rate is projected to continue in the near

future [8]. A recent InfiniBand NIC from Mellanox delivers 200$Gbps$ and 200M IOPS, which means the server CPUs have fewer than ~ 1000 cycles to complete a request associated with a single network packet [35]. Hence, utilizing all the available network bandwidth has become increasingly challenging, especially with fine-grained messages. Offloading network processing to dedicated hardware addresses such challenges and has already seen wide industry adoption [56, 20, 22], but exposes a new bottleneck for microservices: the RPC layer itself.

The RPC layer running on the general-purpose CPUs is far behind the advancements seen in networking technologies. The overheads of the RPC software layer can be of a few tens of microseconds [7]. Putting this latency number into perspective, the service time for a simple microservice is in the same ballpark. It is shown that microservices spend up to 75% of their on-CPU time in the RPC and transport layers [1], which is a direct consequence of a deep software stack with multiple layers of functionality that is executed for every RPC message. The RPC software overhead is not only on the critical path of every RPC message, but also accounts for a significant fraction of a datacenter's load. Google reported that the RPC layer consumes about 12% of the total CPU cycles at their datacenters [24], which are not limited to running microservices. In our experiments, we have observed the RPC layer taking up to 90% of the microservices' on-CPU time and lagging behind the state-of-the-art and future NICs [8, 9] by up to three orderers of magnitude. Overall, the RPC software stack accounts for a significant fraction of both a single request's latency and the datacenter's total compute capacity; thus, optimizing the hardware-software stack for RPCs is of critical importance.

This chapter motivates the need for an RPC processor to address the growing RPC cost in datacenters. First, we describe how the move to microservices and their heavy reliance on communication along with technology trends have led to RPCs becoming a performance bottleneck in datacenters. We then quantify the cost associated with the RPC layer, and after dissecting the RPC layer, we present insights on (i) why the RPC layer is costly and CPUs are ill-suited to perform the RPC layer functionalities, and (ii) why multi-threading fails to improve the RPC layer's performance. Finally, we conclude with why RPC processing is a good fit for hardware acceleration.

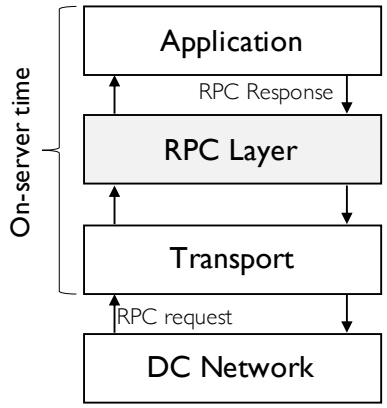


Figure 3.1: System stack exercised in a microservice's invocation.

3.1 The Need for Faster RPC Processing

Online services have been transforming from single-binary monoliths to a plethora of fine-grained modules known as microservices. A microservices architecture simplifies development and deployment by creating independent modules responsible for subsets of the application's functionality, and enforcing modularity between each module [2, 1, 6]. Microservices are deployed across many servers and thus require inter-server communication using an API such as Remote Procedure Calls (RPCs). Figure 3.1 breaks down the layers of the system stack exercised by a microservice. Upon the arrival of a new request, the server first performs transport protocol processing and hands the request to the RPC layer to begin operation. After the RPC layer completes, the actual application-layer code in the microservice runs and then sends its response to the original requester. The response goes through the same layers in reverse before the message leaves the server.

Decomposing a monolithic service into a concert of microservices means that each microservice is leaner than the original monolith. However, each microservice must now traverse the lower levels of the communication stack at least twice (i.e., to receive an incoming RPC and send the matching response). Simply put, decomposing a service into a chain of ten microservices will result in each component doing 10% of the application-level work on average while increasing the aggregate communication time by 10×. As microservices continue to proliferate, the amount of application-level work performed per RPC (i.e., the computation to

communication ratio) will decrease, thus creating a challenge to minimize the "tax" associated with each RPC.

The above walkthrough presents a simplified case where the application layer is self-contained and completes its processing in isolation (i.e., there is only one incoming and one outgoing message for each request). However, microservices commonly perform several nested RPCs while processing a request, for reasons such as retrieving data from other microservices or sending information as inputs to other microservices. The inclusion of nested RPCs means the execution of the business logic is interrupted multiple times by repeatedly traversing the RPC and transport layers. Such behavior exacerbates the microservice's computation to communication ratio and further adds to the microservice's time spent in the RPC and transport layers. Prior work reports that microservices spend up to 75% of their on-CPU time in the RPC and transport layers [1]. Oscillating between the microservice and the RPC layer also negatively impacts the CPU's instruction supply, leading to a higher number of instruction misses than would be experienced by an RPC-free application.

While the tax on inter-microservice communication includes both the RPC layer and the underlying network stack, ongoing research has drastically reduced the latency of both transport protocols and datacenter network traversals. Modern datacenter network topologies [17, 18] and protocols for optimized congestion control [10, 12, 14] achieve network traversals of a few microseconds (μs) with high predictability. Furthermore, optimized endpoints running protocols in userspace (e.g., DPDK [19]) or hardware (e.g., RDMA [20], Scale-Out NUMA [21], or LTL [22]) have drastically shrunk the cost of the transport layer from 10s of $\mu s$for kernel-based TCP [23] to as low as sub-μsvalues [13]. Hence, the time spent in the RPC layer is becoming a significant fraction of the end-to-end cost of invoking a microservice. We claim that the RPC layer in its current form will inevitably dominate the future latency of microservices, particularly as the processing times of the microservices themselves shrink to the microsecond scale [7]. We now proceed to quantify the costs of the RPC layer.

3.1.1 The Cost of RPCs

To quantify the costs of the RPC layer, we study microservices from DeathStarBench [1] and measure the fraction of CPU cycles spent in the RPC layer versus the function code itself, where the actual business logic of the application is performed. We also measure how much of the microservices' instruction working sets belong to the RPC layer. We first go over our experimental setup and then present our findings.

Evaluated Microservices. We select five microservices from DeathStarBench [1] that differ in the following primary parameters that dictate the cost of the RPC layer: number and complexity of functions, frequency of nested RPCs, and message size/format complexity. We evaluate UniqueId (UID), User (USR), UrlShorten (URL), SocialGraph (SG), and ComposePost (CP), comprising one, six, one, seven, and six underlying functions, respectively. The selected microservices represent DeathStarBench's various microservice classes. Other microservices in this benchmark suite behave identically or similarly to those we evaluated. In particular, most of the microservices are similar to SG and CP, which contain little business logic and spend most of their execution time just passing data along to other microservices or data stores via nested RPCs.

All microservices use Apache Thrift [32] as their RPC layer, to which we have added a new hardware-terminated transport protocol based on NEBULA [25]. We study each microservice in isolation and create mock components for the other microservices surrounding the isolated one. Due to our use of isolated microservices, we report the CPU cycles expended in only the RPC and application layers. Therefore, our results are independent from the underlying transport and network protocols.

Microservice Characterization. To accurately measure the breakdown between the functions and RPC layer, we instrument the microservices' code to record cycles expended in the following three steps: (i) the RPC processing that occurs upon new requests arriving, (ii) nested RPCs that occur during the function's execution, and (iii) the function code itself. Therefore, the cycles we attribute to the function quantify only the time spent executing the application's

Cores	ARM v8; 64-bit, 2GHz, 4-way OoO TSO, 128-entry ROB Next-line instruction prefetcher
L1 Caches	64KB 4-way L1d, 64KB 4-way L1i, 64B blocks 2 ports, 32 MSHRs, 4-cycle latency (tag+data)
LLC	Shared block-interleaved NUCA, 8MB total 16-way, 1 bank/tile, 8-cycle latency
Coherence	Directory-based Non-Inclusive MESI
Memory	45ns latency, 2×25.6GBps DDR4-3200
Interconnect	2D mesh, 16B links, 3 cycles/hop

Table 3.1: Parameters used for cycle-accurate simulation.

business logic. Reported cycle counts are the average number of cycles expended per request across all functions for each microservice. To estimate instruction working set sizes, we apply the methodology used for profiling workloads in Google datacenters [24]: we collect the trace of executed instructions and measure how many unique cache lines cover 99.9% of the trace when ranked by popularity.

Simulation Setup. We use the QFlex cycle-accurate full-system simulator [57] to simulate a 16-core ARMv8 CPU running Ubuntu Linux 18.04. Table 3.1 summarizes our system's configuration parameters. 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, and delivers notifications to the CPU through the NEBULA transport stack. The load generator also emulates all the mock microservices, mimicking their behavior and instantly responding to RPCs with pre-constructed messages.

Figure 3.2 breaks down each microservice's mean on-CPU time when processing various request types into the following three categories: (i) the RPC layer for the initial request message and its final corresponding response, (ii) the RPC layer for any nested RPCs that the microservice generates, (iii) the application-layer functions. In all cases, the RPC layer takes a

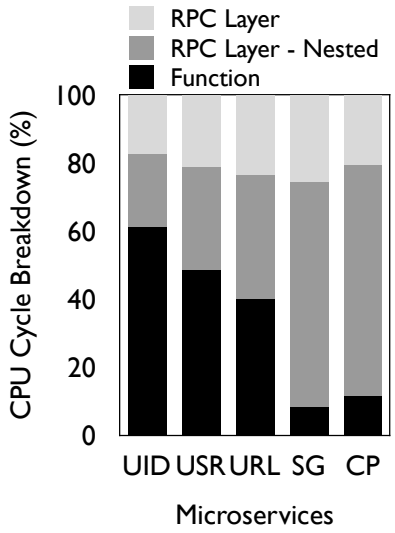


Figure 3.2: Breakdown of CPU cycles expended in microservices - Function vs. RPC Layer.

significant fraction of the microservice's runtime, accounting for 40–90% of the on-CPU time. The fraction of time spent in the RPC layer varies widely because the functions comprising these five microservices have different complexities, input/output message types, and number of nested RPCs. For example, UID has a single function that generates a globally unique integer, and one nested RPC to upload that ID to another microservice. In contrast, CP has six simpler functions, but each can have up to 13 nested RPC calls to other microservices; hence, ~70% of CP's expended cycles are attributed to nested RPCs. We also observed throughputs for the RPC layer in the range of $100-200MB/s$, which are an order of magnitude lower than a commodity NIC's throughput (i.e., $40Gbps$), and two to three orders of magnitude behind the more advanced NICs that are on the roadmap for both Ethernet and Infiniband with processing rates of up to $1Tbps$ [8, 9].

Additionally, the RPC layer hurts the microservice's performance indirectly by bloating the microservice's instruction working set. The decomposition of a monolith into microservices drastically shrinks each individual component's working set. For the microservices we evaluated, we observed working set sizes of 70-190KB, which even though are one to two orders of magnitude smaller than that of monolithic services, they still exceed the typical L1 instruction cache capacity (e.g., AMD's Zen v3 [58], Intel's Ice Lake [59]) by 4–6×. Essentially, although the

microservice itself could indeed be L1-resident, encapsulating it in a bloated RPC layer results in the working set outgrowing the L1 cache's capacity. Our study on the five microservices shows that 30-70% of the working set is attributed to the RPC layer. Many individual functions are small enough to fit inside a 32KB instruction cache, whereas when the RPC layer's instructions are included, the total working set exceeds the instruction cache size. As a result, CPU frontend stalls remain an important bottleneck to address for microservices, as they still account for 20–60% of total CPU slots [60]. In our experiments, ~20% of CPU cycles are wasted on frontend stalls.

This study shows that once microservices are deployed using optimized transport and network layers, the RPC layer is a prime contributor to a server's expended CPU cycles. Of equal importance is that nested RPCs cannot be overlooked: as microservices become more specialized and modular, the greater the cumulative RPC overheads. Given the significant overhead of the RPC layer in terms of latency and instruction footprint and its universal use by all microservices, it is the next logical step to revisit to improve performance. In order to resolve this emerging RPC bottleneck, it is first necessary to decipher its underlying operations. Hence, in the following section, we dissect the RPC layer, examine its internal operations and identify their associated costs.

3.2 Dissecting the RPC Layer

RPC frameworks such as Apache Thrift [32] or Google's gRPC [34] are themselves multi-layered architectures. Figure 3.3 expands the RPC layer to display the common functionalities comprising these frameworks in a simple request-response case (i.e., without nested RPCs). An RPC message consists of two parts: a header and a payload. Upon receiving a new request, the RPC layer first has to parse the header, which has fields indicating the message type and the requested function. The header may also include a sequence number to be used for demultiplexing outstanding RPCs completing out-of-order. The dispatch module then runs, which looks up the function ID in a table to retrieve the handler associated with this function and passes control to it. Finally, the handler prepares the input arguments by deserializing

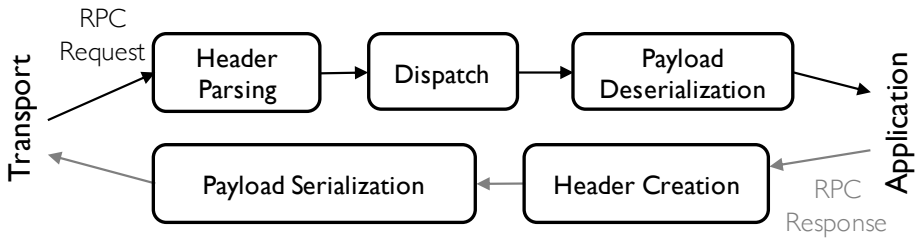


Figure 3.3: Operations within the RPC layer.

the message's payload and calls the requested function, terminating the RPC layer. Response messages are handled by the RPC layer in a similar manner, applying the same operations in inverse order.

We categorize the aforementioned operations into three modules: (i) header manipulation, which contains header parsing and creation, (ii) payload manipulation, which contains payload serialization and deserialization, and (iii) dispatch. We refer to the two manipulation modules together as *data transformation*, as they essentially boil down to the same type of operations, which we see later in Section 3.2.1. While these modules are mandatory for every RPC, there exist additional modules such as compression, encryption and authentication, that the RPC layer may optionally employ. However, for many of the microservices that only exchange small amounts of data and are not user-facing (i.e., are internal to a datacenter), these modules are often omitted.

We classify RPC layer time into the cycles expended in the three aforementioned modules, using the same experimental setup as in Section 3.1.1 and further instrumentation of the code. Figure 3.4 depicts our results. All cycle counts are cumulative over the request RPC, the final response RPC, and the nested RPCs that occur within the microservice's functions. Payload manipulation stands out as the largest component, accounting for ~60% of the RPC layer's total expended cycles. The absolute cost of payload manipulation is a function of each message's size and layout and adds up with each nested RPC. CP and SG's aggregate payload manipulation cycles in Figure 3.4 are greater than UID, USR, and URL because they create more nested RPCs, and each individual payload manipulation task is costlier due to larger and more complex messages. In contrast, header manipulation uses an identical format (i.e.,

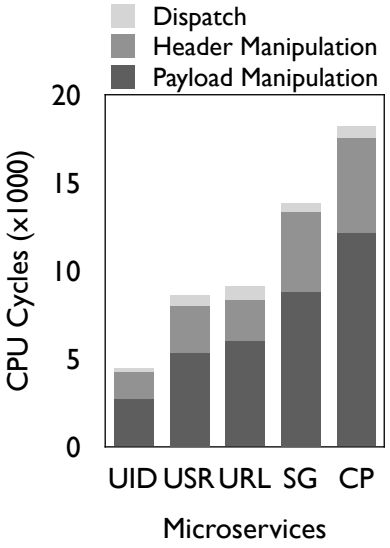


Figure 3.4: Breakdown of CPU cycles expended in the RPC Layer.

data types and values) across all of the microservices, and therefore the total cost of header manipulation only depends on the number of nested RPCs. The same is true for the dispatch module. Therefore, microservices like SG and CP have a far greater aggregate cost for header manipulation and dispatch than those similar to UID.

This study shows that the RPC overheads are concentrated in the RPC layer's payload and header manipulation modules (also known as data transformation), which together make up ~95% of RPC cycles. Hence, we first look further down into data transformation tasks in order to have a better understanding of what they do and why the are necessary. We then look into why these operations are so costly and how we can potentially make them to run faster in Section 3.3.

3.2.1 Data Transformation

Because the microservices architecture consists of fine-grained software components with enforced modularity that are interconnected through RPCs, each microservice is often developed by separate teams using different programming languages and tools that are best suited for that microservice [2, 1, 6]. Hence, RPCs that cross format boundaries must perform data transformation (DT) to and from the desired format. Such DT tasks are also found in

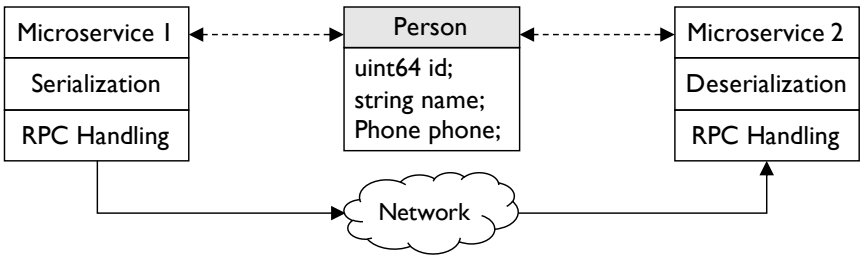


Figure 3.5: Two microservices communicating using RPC.

workloads other than microservices, such as databases, data analytics and generally wherever there are multiple software components that need to work together in order to achieve an end goal. Moreover, it is common for storage systems to store objects in a serialized format such as Protobuf's binary format. RPC frameworks such as Apache Thrift [32] or Google's gRPC [34, 24] provide a structured way for application developers to define the format of each message and generate the code required to transform that message to and from the wire representation.

To illustrate the essential nature of data transformation, we present an example software stack in Figure 3.5, where microservice 1 performs an RPC to microservice 2 with a `Person` object as its argument (i.e., message's payload). The `Person` object must first be serialized to its wire representation on the sender side, and then deserialized later on the receiver side before being ready in microservice 2's acceptable format. Both the serialization and deserialization processes are performed using the code generated by the RPC framework. Critically, this step takes place for every network message between microservices, even those using the same data format, as the data must be flattened into a byte-stream at the sender and unflattened on the receiver side.

Listing 3.1 shows pseudo-code for the serialization process that is done by software frameworks such as Thrift [32] or Protobuf [61], which is part of Google's gRPC [34]. The serialization process converts objects to a series of keys and values, which are then sent over the network. We use Figure 3.6 to aid our explanation — it shows the fields of a basic `Person` object and its final binary wire representation in the Protobuf format. As transformation operations are similar across frameworks, we use Protobuf as a reference framework throughout this section without loss of generality.

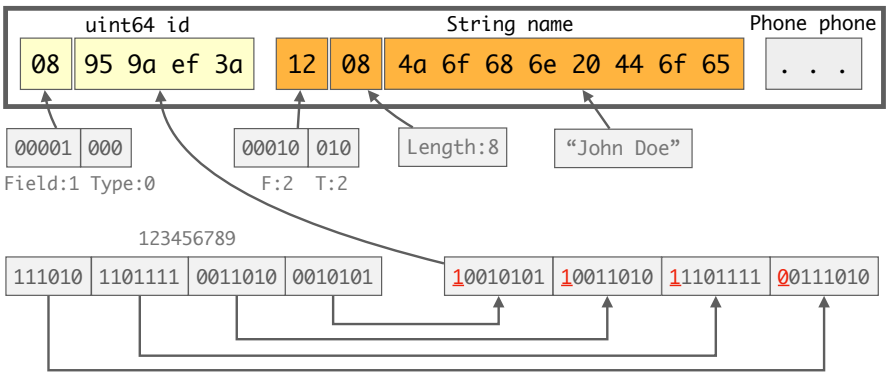


Figure 3.6: Sample Person object in Protobuf binary format.

Listing 3.1: Serialization Pseudo-code.

```
1 serialize (byteStream target):
2     for (field f from 1 to N):
3         serializeField(f, target)
```

To serialize a `Person`, each field is individually transformed into its binary representation using the `serializeField` function, where the operation that is performed depends on the field type. For some fields, such as `float`, the source data is directly copied, but for others, the source is completely transformed before being written. The output of each field contains a key (aka. tag), which acts as an identifier for the field, and the serialized bytes of data. For example, the second field in Figure 3.6, `name`, has its tag set to `0x12`, which comes from its type, 2 representing a `string`, and its field ID, which is 2. Because the third field is an embedded message called `Phone`, `serializeField` recursively calls `serialize`, and all the output bytes corresponding to this message will be placed into the output stream following the tag. The `Phone` message is shown in Figure 3.6 as an ellipsis.

The wire representation for the `id` field is a common data type in DT frameworks such as Protobuf or Thrift and is called a `varint`, an encoding that depends on the data value. Only the number of bits required to encode the value (e.g., 32 bits to represent the value `123456789`, even though the language specifies 64 bits) will be sent on the wire. In order to signal to the receiver that there are more bytes to be processed that represent this integer value, the upper

bit of each byte is reserved for the *continuation bit*, and is set to 1 if there are more bytes to come. These bits are shown as red and underlined in Figure 3.6. Reading from left to right, each continuation bit would signal the receiver to "keep reading" as there are more bytes to come. The receiver stops processing the `varint` when it reaches a continuation bit with the value of 0. These complete transformations are the most challenging and compute-intensive operations. All DT frameworks contain these types of transformations, such as the `itoa` operation in JSON.

3.3 Toward Faster RPC Processing

Because the majority of the RPC cost comes from the data transformation tasks, we first look into why these tasks are so costly, and CPUs are ill-suited to execute them. We then explore the use of threads to accelerate DT tasks individually and the RPC layer as a whole. We show that neither multi-threading nor other parallelization techniques such as staging can help to accelerate the RPC layer. Finally, we conclude this section and this chapter by making a case for an RPC processor.

3.3.1 Limitations of Data Transformation on CPU

The process of transforming data on CPUs has two critical limitations, both inherently connected to the use of the ISA as an abstraction to represent the underlying operations. Performing DT on CPUs entails a high instruction count per serialized field and relies on implicit instruction-level rather than explicit field-level parallelism. The object layout expresses the field-level parallelism, but the task granularity is too big for CPU's ILP window. At the same time, the tasks are variable-sized and too fine-grained for thread-level parallelism to amortize synchronization costs.

The cause for instruction bloat is the fact that many format encodings (e.g., `varint`) require performing an operation on each byte of the source data and the extra operations needed to access the complex data structures generated by the DT framework. In Listing 3.2, we show

Listing 3.2: Serialization Pseudo-code for Variable Intger 64.

```
writeVarInt64 (uint64 value, byte* target):
    while(value >= 0x80):
        *target = value | 0x80
        value = value >> 7
        ++target

    *target = value | 0x80
```

pseudo-code for transforming a 64-bit integer to a `varint`. For each byte of the input value, the code performs a branch to check whether or not the value is large enough to require the use of this byte (e.g., when checking the third byte, the source value must be greater than 2^{14}). Then, bitwise operations are performed to isolate the correct byte and write it to the output buffer. Moreover, because of the complex data structures generated by the DT framework, and the fact that this code is performed by a general-purpose DT framework with a deep software stack, a series of instructions must be executed before and after the actual transformation function gets executed. For example, the code path goes through 10-15 functions before reaching the leaf `writeVarInt64` function.

We quantify the cost of such DT operations using a microbenchmark running on a commodity ARM X-Gene server. We use an X-Gene system-on-chip with eight ARM Atlas A57 4-wide OoO cores running at 2.4GHz available on CloudLab [62]. Our microbenchmark runs (de)serialization tasks using Google Protobuf v3.7 and the message types in Table 3.2. The sizes of these messages are chosen to represent the fact that the majority of network packets sent by latency-critical applications are sub-1KB [41, 14]. The R/W ratio is the number of bytes that must be read for each byte written in the serialized output and depends on each field's depth and type. For example, `varints` are converted to different byte-streams based on their values, where `strings` have an R/W ratio of one (not considering the tag and size). Moreover, the depth of each field (i.e., the number of sub-messages that must be parsed before returning to the top level) increases the read/write ratio because an additional tag and size values must be written in the serialized output. We use the C API for Unix time and `perf` to measure transformation throughput and dynamic instruction count, respectively.

Message Type	R/W Ratio	Max Depth	Size (B)
Type 1	2.6	1	485
Type 2	2.75	2	297
Type 3	4.25	2	232

Table 3.2: Message types and their characteristics.

We measured that 25 dynamic instructions are executed on average per output byte, adding up to hundreds of instructions per field and thousands of instructions per message. Executing these many dynamic instructions per field means CPU's ILP window is not able to extract the available field-level parallelism, thus capping the achievable transformation throughput at roughly 1Gbps. Improving the performance of serial instruction streams requires boosting the CPU's IPC. Unfortunately, transforming fields such as `varints` results in a data-dependent branch per byte; these branches are known to be difficult to predict and limit the attainable instruction-level parallelism. The limited success of control speculation, when applied to data-dependent branches, has also been observed by prior work studying ETL workloads for data cleaning and ingestion [63]. Although classical microarchitecture techniques such as predication [64] would help the performance of `varint` encoding, today's CPUs only support partial predication and lack the ability to perform conditional stores [65].

We measured an IPC of 1.5 on this 4-wide OoO core when running the transformation microbenchmark, indicating the application itself does not have enough ILP. In this case, the transformation throughput could get up to 2.8Gbps in the best case, assuming the core was able to achieve an IPC of four (i.e., the full width of this OoO core) when doing these transformations. However, even with utilizing the full width of the core, the throughput would still be far away from the network's line rate (40Gbps). In fact, at this rate of 25 instructions per byte, in order to match the network's line rate, the CPU core would have to achieve an IPC of 56. Even if the application exhibited that much instruction parallelism, it is unlikely that any micro-architectural optimization would improve IPC by that much. The raw number of instructions required will inevitably exceed the limits of hardware techniques for instruction-level parallelism.

Unfortunately, parallelizing the data transformation tasks with software threads will not help

either due to synchronization costs and the granularity of tasks. Assuming all the fields' types and addresses are known in advance, the most efficient way to parallelize serialization of fields with threads requires a scatter phase in the beginning, which distributes the fields to threads, and a gather phase at the end that brings all the transformed fields back into a single serialized buffer. Hence, there are at least two synchronization points. Considering using a simple locking mechanism to signal other threads, the synchronization cost will be at least $200ns$ [66]. To put this number into perspective, serializing a header or the payload of small messages like the ones used by the UID microservice of Section 3.1.1 takes roughly the same amount of time. In addition to this synchronization cost, there is also the cost of extra copies needed in the gather phase to bring the serialized data from each thread's local buffer into the final serialized buffer. Hence, the parallel version of this task is, in fact, going to perform worse than its sequential version.

With bigger messages, the synchronization cost gets amortized over the whole execution, making it less of a bottleneck. To illustrate, consider the serialization of a message of Type 3 from Table 3.2, for which CPU takes $\sim 1\mu s$ to complete. Parallelizing this task with four threads only gives a 2.2× speedup in the best case, assuming perfect parallelism and no additional cost other than the $200ns$ synchronization cost. However, the cost of gathering the data is non-negligible, and it scales with the message size and number of threads.

Moreover, as messages get bigger, they tend to get deeper as well, meaning they have more sub-messages. Such nested messages make both scatter and gather phases more difficult to balance because a sub-message is treated as one single field and all of its data has to be written together in the output buffer. The existence of such fields, along with the fact that even other primitive data types have different transformation latencies lead to significant load imbalance for each thread, penalizing the overall benefit. Additionally, in the case of online services where cores are busy processing multiple requests in parallel, multi-threading is only beneficial if it gives super-linear speedups (e.g., more than 4× with four threads); otherwise, the overall throughput of the system gets hurt. For instance, in the above example, assuming running four threads gives 2.2× speedup for the DT tasks, the overall system throughput is

55% of the case where four threads run separately to process parallel requests.

Additionally, while parallelizing serialization is theoretically possible, deserialization is not easily parallelizable in software because it is not known in advance where the data of each field lies in the serialized buffer. One thread has to run ahead to read the data blocks from the serialized buffer, detect the serialized fields and then pass them to other threads for parsing, which significantly limits the attainable parallelism. Hence, we do not expect that parallelizing DT with software threads will yield performance improvements. In the following section, we look into the possibility of improving the RPC layer's performance using another software parallelization technique to run the whole RPC layer in parallel with stages.

3.3.2 Limitations of Staging the RPC Layer

We do not expect to get any benefit from other parallelization techniques like software pipelining or staged execution either. Staged execution is a well-known technique that improves the performance by improving locality and increasing parallelism when the program execution can be broken into several segments or stages that are chained together, forming a producer-consumer pipeline [67, 68, 69, 70]. As shown in Figure 3.3, processing an RPC request is, in fact, comprised of a chain of tasks (i.e., parsing the header, dispatch, parsing the payload, function, header creation, and payload serialization). While staged execution may seem a good fit for RPC processing, it is not beneficial for similar reasons as to why multi-threading fails to improve the performance of DT tasks—i.e., the tasks are variable-sized and too fine-grained to amortize synchronization costs.

Staging or software pipelining works best when (1) the processing rate of all stages is the same, and (2) the synchronization overhead (i.e., the data transfer cost between two stages and queue contention among threads of the same stage) is negligible compared to the latency of each stage. When the stages do not operate at the same rate, the stages with faster processing rates will be stalled, hence, wasting resources and limiting the overall throughput of the system. Unfortunately, the aforementioned chain of tasks comprising the processing stack exhibits significant latency variability, both across and within tasks. For example, while the

dispatch and header manipulation modules take roughly the same amount of time across messages (i.e., around $100ns$ and $200ns$, respectively), payload manipulation can take from just a few nanoseconds (in the case of an empty message) all the way up to a few microseconds, depending on the structure of the payload.

Moreover, one stage of this pipeline is the function itself that runs the business logic of the program, which not only exhibits significant variability within itself, but also can have latencies that are orders of magnitude higher than the other stages. Such high variability leads to poor resource utilization. Additionally, as shown in Section 3.1.1, microservices have several nested calls to other microservices within the function (e.g., up to 13 times in the case of the ComposePost microservice). These nested calls mean the function stage of the pipeline has to be stalled while the nested RPC is completed due to the synchronous nature of the program. Coroutines can help in such cases to break the function into smaller segments and make it reentrant. However, the fine-grained granularity of the segments, along with the extra complexity of such design that makes it error-prone and reduces development velocity, prevent it from being a viable solution; thus, we do not consider such design. As such, nested calls exacerbate the load imbalance and utilization issue.

Even with perfect load balancing and utilization, the extra synchronization overhead precludes any potential performance gain. To illustrate, we assume the latching overhead between two stages is at least 50 CPU cycles [71], which is just the cost for signaling the other thread that the previous stage is completed. Considering each of the aforementioned six tasks of the chain to be a stage in the request processing pipeline, the aggregate latching overhead is at least 250 cycles. Additionally, because of the nested calls, the execution flow goes through the RPC stages multiple times over the course of processing a request. These additional latching overheads bring up the aggregate latching overhead to even a couple of microseconds, which is non-negligible knowing that the whole microservice is about a few microseconds.

While merging the stages into fewer stages lowers the aggregate latching overhead at the cost of less scalability for each stage, there are still other sources of synchronization overhead that limit the staging gains. Each stage consists of multiple threads contending over a single pair

of input and output queues, which introduces extra synchronization overhead. This extra overhead is at least $100ns$ for each stage, assuming using optimized queue-based locks with hardware support [72]. By merging all stages into three stages, one for all the pre-processing tasks that occur before the function, one for the function itself, and one of all the post-processing tasks that occur after the function, the total synchronization cost will be at least $350ns$ for the case where there are no nested calls. The additional synchronization overheads incurred by such nested calls lead to non-negligible microsecond-scale overheads. To compare, the service time of our five microservices from Section 3.1.1 is $6-10\mu s$.

Contrary to prior work in which staging is used to gain better instruction and data locality [69, 68, 70], the only advantage of staging in the case of an RPC processing pipeline is better instruction locality because there is no data locality within each stage. In our experiments on the five microservices, we observed on average ~ 20% of the cycles being wasted due to instruction misses. Even assuming staging can reclaim all those cycles, staging will still hurt the overall system performance as the microsecond-scale synchronization overheads and resource under-utilization offset the reclaimed cycles.

3.3.3 The Case for an RPC Processor

Despite the critical nature of the RPC layer for microservice performance, CPUs are ill-suited to execute the RPC layer tasks. We showed that the underlying data transformation inherent to header and payload manipulation modules is a parallel task, but software implementations are unable to extract said parallelism. The object layout expresses the field-level parallelism, but the task granularity is too big for the CPU's ILP window. At the same time, the tasks are variable-sized and too fine-grained for thread-level parallelism to amortize synchronization costs. In principle, each field in `Person` could be independently transformed if the hardware is made aware of each field's type and memory location. In that case, while the `varint` encoding is being performed, the `name` can be copied, and the `Phone`'s data can be fetched. Serial instructions are the wrong abstraction to expose these types of independent operations because the problem is inherently parallel. Unfortunately, neither software threads nor CPU

ISAs is the right form to represent this parallelism between fields.

The dispatch module is also unlikely to be efficiently executed on CPUs because it contains multiple data-dependent and indirect branch instructions, which are dependent on the incoming message. Furthermore, staging the RPC processing does not help either for similar reasons as to why multi-threading fails to improve the performance of DT tasks. In addition to these issues, it has already been reported that CPUs are plagued by instruction supply problems when executing microservices [1]. This problem will worsen with the number of functions, message types, and nested RPCs that make up a microservice. When the inefficiencies of CPU-centric data transformation are combined with the instruction supply issues in microservices, using dedicated hardware for RPC tasks becomes an attractive solution.

To justify the investment in dedicated hardware, it must be widely applicable and also configurable for the sake of future software deployments. Our breakdown of the RPC layer shows these exact characteristics are true for its three modules; despite the diversity and rapid evolution of microservices, they all depend on these three ubiquitous modules. Furthermore, the maturity of the RPC layer [24, §4] indicates that dedicated hardware for its modules will not be immediately obsolete. Via the use of a dedicated abstraction to represent both payload and header manipulation, such hardware can be made applicable to any RPC message and framework. Hence, we argue it is feasible to design hardware that is drastically more effective in executing the RPC layer than CPUs.

Although the use of FPGA-equipped NICs has been proposed to accelerate RPC layer operations [73, 74, 75, 76], no existing design has managed to target all of the RPC layer modules we describe in section 3.2. The most difficult challenge is to support the header and payload manipulation tasks because the message objects in production RPC layers are complex pointer-based data structures. Processing such software-readable objects requires judiciously co-designing RPC hardware and software around the constraints of the server's DMA engine, which only transfers opaque chunks of bytes or scatter-gather arrays [76]. Therefore, we argue that it is logical to handle such tasks with hardware integrated on chip, removing the DMA engine's constraints as well as the extra latency incurred whenever data must be moved across

the I/O interconnect. Details of existing proposals are further discussed in chapter 8.

Although it may appear logical to limit the scope of integrated hardware accelerators to the two manipulation modules because they make up ~95% of the execution time, the seemingly small dispatch module creates a critical bottleneck that must be addressed. Because the dispatch module is logically wedged between the two manipulation tasks and remains on the CPU, the CPU serves as the coordinator that creates offload tasks after a new network message arrives. The resulting split in the RPC layer between software and hardware and using the accelerator as a co-processor introduces excessive fine-grained CPU-accelerator offload overheads. These offload overheads are a critical obstacle limiting the performance gains, particularly because their cost accumulates when a microservice uses many nested RPCs. We conclude that although it is logical to invest in hardware for the two common manipulation tasks, keeping the dispatch module on the CPU cripples end-to-end performance due to cumulative offload overheads, and therefore it must also be done in hardware as well.

The inclusion of dedicated hardware for all three of the RPC layer's modules has the side benefit of improving the CPU's instruction supply. Many individual functions are small enough to nearly fit inside a 32KB instruction cache, whereas when the RPC layer's instructions are included, it bloats the total working set to a few times larger than the instruction cache. In a server with hardware support for the RPC layer, these instructions vanish, and any remaining L1 instruction cache contention occurs when the execution of multiple functions is interleaved on the same core. Deploying the RPC processor as a NIC integration rather than CPU extension enables a unique performance optimization opportunity by doing network-centric RPC request steering. The RPC processor can have the ability to assign requests to cores in a manner that is aware of the function being requested to increase instruction cache locality and lower the inter-function contention.

Summary: The rise of RPC-coupled microservices, combined with developments in networking, is leading to a need to address the overheads in the RPC layer. Unfortunately, despite the critical nature of the RPC layer for microservice performance, CPUs are ill-suited to execute the RPC layer tasks. We cannot rely on software optimizations like multi-threading or staging

to shrink the gap between the CPU and network processing rates either. Despite extreme diversity among microservices and their rapid evolution, each microservice depends on the ubiquitous RPC layer. The commonality of RPC tasks justifies the investment in dedicated hardware for processing the RPC layer. This RPC processor must incorporate all the three common modules of the RPC layer in order to be most effective. Performing the RPC layer in a dedicated hardware unit integrated with the NIC opens up further opportunities such as network-centric RPC request steering based on function affinity. In the next chapter, we present our hardware/software co-design for rapid and flexible RPC processing.

4 Designing an RPC Processor

In Chapter 3, we motivated the need for an RPC processor that addresses the growing overheads of the RPC layer in datacenters. In this chapter, we describe the design space for an RPC processor, hereafter referred to as an RPCProc. Our design is guided by the following six design goals: (G1) the CPU should only need to run the business logic of the microservice rather than the RPC layer, (G2) the RPCProc should be autonomous and not CPU-controlled, (G3) the RPCProc should be synergistic with state-of-the-art NIC architectures, (G4) the RPCProc should be able to scale with the NIC's line rate and allow the cores to run services at NIC's rate, (G5) the RPCProc should have minimal silicon requirements, and finally (G6) the RPCProc should be compatible with existing frameworks and programmable to allow compatibility with future frameworks. In this chapter, we present the design of a specialized RPC processor that achieves all the aforementioned goals and is able to completely remove the RPC layer's burdensome tasks identified in Chapter 3.

4.1 High-Level Architecture

4.1.1 Logical Workflow

In current systems, the NIC directly interacts with the CPU cores to signal the arrival of incoming work, as shown in Figure 4.1. After terminating the network and transport protocols,

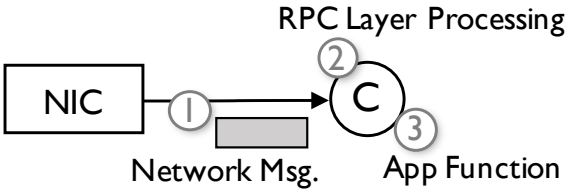


Figure 4.1: Current system design, where both the RPC layer and the application function are executed by CPU cores.

the NIC sends a notification to a CPU core through its interface (①), and then the core begins processing the RPC layer (②), before executing the requested application-level function (③). In this case, we assumed the NIC is capable of terminating the transport protocol, which is becoming commonplace with integrated NICs [21, 25]; otherwise, the core needs to also process the transport layer in addition to the RPC layer.

As seen in Section 3.2, the two header and payload manipulation tasks constitute ~ 95% of the cycles spent in the RPC layer. Because both tasks boil down to common data transformation tasks and together they make up most of the RPC cost, it may seem logical to limit specialized hardware design efforts to these two modules. Figure 4.2 depicts such a design where the CPU core offloads the header parsing and payload parsing tasks to the RPCProc (②, ④), but it has to perform the dispatch functionality, which is wedged between the two tasks (③).

Because the dispatch module is logically wedged between the two manipulation tasks and is processed on the CPU, the CPU serves as the coordinator that creates offload tasks after a new network message arrives. The resulting split in the RPC layer between software and hardware and using the accelerator as a co-processor introduces excessive fine-grained CPU-accelerator offload overheads. While in this design the RPCProc is able to accelerate the dominating fraction of the RPC layer, the unnecessary offload overheads limit the performance gains, particularly for small messages where the offload overhead is comparable to the time RPCProc takes to process the message. Hence, the seemingly small dispatch module creates a critical bottleneck that must be addressed.

Offloading the dispatch functionality to the RPCProc is not enough to achieve our second

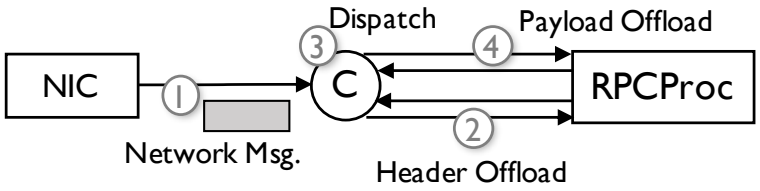


Figure 4.2: Design with explicit CPU-controlled offloads.

design goal, mandating the RPCProc to work independent of the CPU (G2). In this case, even if the RPCProc was able to process the dispatch module as well the two manipulation modules, because the CPU is the unit that takes the message from the NIC, there needs to be at least one explicit task offload from the CPU to the RPCProc. A critical requirement for the RPCProc is to receive incoming requests directly from the NIC and process the full RPC layer to completion before involving the CPU. The same is true for outgoing requests, except the RPC layer must entirely complete with a single RPCProc call by the CPU to start the sending process.

Offload overheads stemming from partial acceleration of the RPC layer or CPU's involvement for explicit task offloads form a critical obstacle limiting the performance gains, particularly because their cost accumulates when a microservice uses many nested RPCs. Hence, we conclude that although the dispatch module takes less than 5% of the RPC layer runtime, it must happen in the same location as the payload and header manipulation tasks. While it is logical to invest in hardware for the two common manipulation tasks, keeping the dispatch module on the CPU cripples end-to-end performance due to cumulative offload overheads, and therefore it must also be done in hardware as well. Moreover, the RPCProc must have a direct interface with the NIC to be able to operate without the CPU's involvement. Without either of these requirements, the system reverts to the behavior in Figure 4.2, where the dispatch and manipulations are logically split, necessitating the RPCProc to be under CPU control and resulting in unnecessary offload overheads.

Realizing our first two design goals (G1 and G2) requires the RPCProc to be a transport protocol endpoint. The use of lean hardware-terminated protocols actually enables this design change because there is no extra processing to be done once the incoming message exits the NIC.

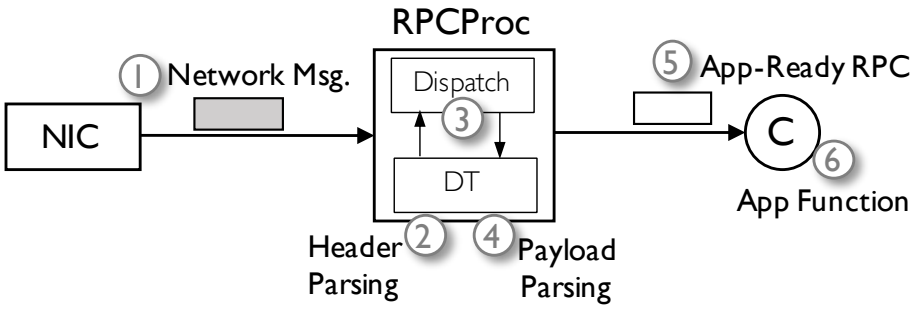


Figure 4.3: Design with a NIC-interfaced RPCProc.

Therefore, we target such a design for our RPCProc, because it not only allows us to meet our first two goal, but also is sufficient to meet our third design goal (i.e., RPCProc being synergistic with the NIC). Message delivery to the RPCProc could be accomplished by any well-established signaling method, such as in-memory queues [77] or MSI-X interrupts [78].

Figure 4.3 displays the architecture of an RPCProc that achieves our first two design goals: it receives incoming RPC requests directly from the NIC (①) and immediately begins processing the RPC layer without the CPU's involvement, starting with header parsing (②). It then performs the dispatch module using hardware logic, which reads the function ID from the parsed header, looks it up in a dispatch table, and finds the metadata describing how to parse the corresponding type of payload (③). Using this information, the accelerator parses the payload (④) and passes an application-readable RPC to the CPU (⑤) that begins executing the requested function (⑥). Placing the RPCProc's dispatch module near the header manipulation module is natural because these two stages form a logical pipeline, where the dispatch logic depends on the value produced in the RPC's header indicating which function is being requested.

An RPCProc with this design is not only essential for solving the RPC layer bottleneck, but it also can provide additional benefit to the CPU when it executes the microservice's business logic. As our design performs all three RPC modules together in the RPCProc, all of the instructions previously executed in the RPC layer are completely bypassed, reducing instruction cache pressure on the cores. Additionally, the decision of selecting the core to run the incoming

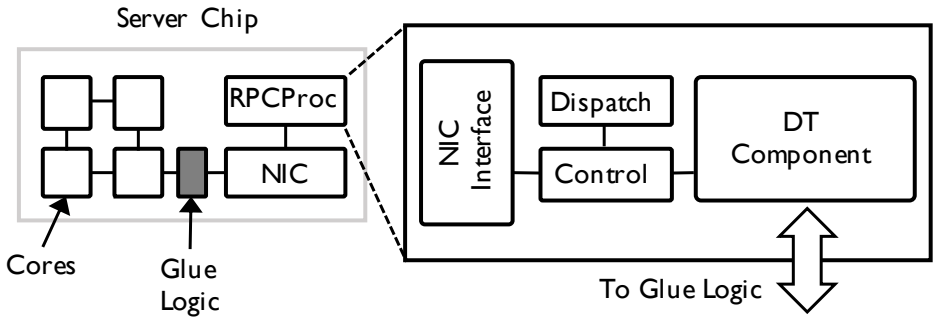


Figure 4.4: Architecture of an on-chip RPC processor.

RPC takes place after the function type is already known from the completed header parsing step (②). Based on this information, the RPCProc can choose the core to process this function based on any policy—in particular, we identify temporal locality as a beneficial one. Assigning RPCs to cores that have just executed the same function virtually guarantees that the core's instruction cache is warm and that function will execute with fewer stalls. We call this approach *affinity-based request steering*. Next, we present our RPCProc's system integration, which is critical to meet two of our remaining design goals (G3 and G5).

4.1.2 Server System Integration

Figure 4.4 shows the architecture of an on-chip RPCProc and the components that execute the header manipulation, dispatch, and payload manipulation modules of the RPC layer. Keeping with G3, we assume the existence of an on-chip integrated NIC with a hardware-terminated transport protocol, because such designs are a natural substrate for a server that is intended to handle communication-intensive microservices. Architectures featuring integrated NICs are already becoming commonplace, following seminal work showing they reduce the total cost of ownership [79]. Academic examples include Scale-Out NUMA [21], the FAME-1 RISC-V RocketChip SoC [47], and the NanoPU [48]. Commercial examples include Oracle's Sonoma [49], Calxeda's ARM SoC [50], and integrated Ethernet MACs in Intel's Xeon-D line [51].

We integrate the RPCProc with the on-chip NIC to reduce silicon costs and deployment

complexity (G3 and G5) because the RPCProc and NIC share glue logic that connects them to the CPU's memory hierarchy. In particular, both components need a small cache and its matching MMU, which the NIC uses to read/write data coherently and the RPCProc will use to operate on that data when it performs the RPC layer. By moving the endpoint of the transport protocol to the RPCProc, it now must be the agent which communicates with the CPU cores to inform them of incoming RPCs (c.f., Step ⑤ of Figure 4.3). As CPU-NIC communication has been shown to be problematic for small data transfers, it is logical for an RPCProc meeting G3 to leverage highly optimized architectural support in state-of-the-art NIC designs [80, 25].

An alternate design point is to provision an RPCProc per CPU core, sharing the CPU's glue logic rather than the NIC's. However, we choose to use a single shared RPCProc for two reasons. First, the silicon overheads of a design with replicated RPCProcs are considerable, thus contradicting G5. Second, a per-core design precludes the RPCProcs from employing affinity-based request steering because requests are first sent to the per-core RPCProcs by the NIC before the function IDs are known.

Integrating the RPCProc with the server's NIC enables function-to-core affinity as one of the potentially many policies in the stages of the NIC that assign incoming requests to cores. State-of-the-art NICs already contain support for assigning work to cores based on metrics such as load balancing [52] or TCP connection locality [81], and therefore, it is logical to provide affinity-based request steering in the same location. The RPCProc component for the dispatch module can be easily adapted to extract the function ID from the header-parsing stage, and provide it to the NIC's core-assignment stages once the entire RPC stack processing is concluded. We now present our RPCProc's interfaces.

4.1.3 Interfaces

Based on our analysis in Chapter 3, we claim that accelerating the RPC layer requires an abstraction that expresses the parallelism inherent in the underlying data transformation tasks (i.e., transforming independent fields). Such an abstraction solves the bottlenecks of expressing transformations in traditional ISAs. When a transformation is compiled into a

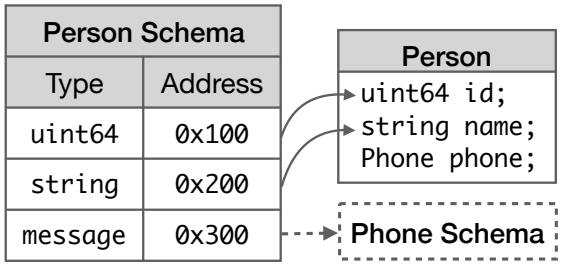


Figure 4.5: Sample Person object and its schema.

CPU's ISA, field-level parallelism is only unlocked if the core is able to speculate far enough ahead to issue instructions that actually operate on two fields simultaneously. Our analysis in Section 3.3 shows that doing so requires many hundreds or thousands of instructions, greatly exceeding the practical limits of a CPU's speculative state. An efficient abstraction, therefore, requires representing transformations in an explicitly parallel fashion. The hardware can then unpack the field-level parallelism and enjoy the performance benefits.

The critical observation that leads to our novel transformation abstraction is that the transformation on each field is completely described by its type (thus identifying the operation the hardware must perform) and the address of the input data. Therefore, a data structure containing these two pieces of information for each field is the leanest abstraction required to express all of an object's transformations. Such an abstraction also enables the RPCProc to work independent of the high-level software RPC framework that is employed as long as the framework sets up the schemata used by the application (`G6`).

We call our abstraction the *schema*, which resides in memory and holds the type and address of each field. The schemata are generated by the application, and passed to the RPCProc to instruct it on how to process RPC messages. The software framework (e.g., Protobuf or Thrift) needs to be modified to create the schemata during the process of creating the message, which can be done by updating the *setter* methods generated by the framework compiler to populate the schema's `address` field as well as the message's value. Figure 4.5 shows an example of the schema for a `Person` after each field has been initialized.

Our schema design achieves two goals: first, it enables the hardware to operate on each field

in parallel by scanning the schema, accessing the data to be transformed, and performing the requested operations. Second, it enables any framework to use the accelerator (G6); the only requirement is to update the schema while creating the message. To provide compatibility with various frameworks, our RPCProc also has an interface for applications to program custom transformations. Upon requesting a new transformation context, the application also has the option to issue system calls to program custom operations into the RPCProc itself. We now present the internal building blocks of our RPCProc and how they implement the above interfaces.

4.2 Components for RPC Tasks

An RPCProc's most important component is the one that handles payload and header manipulations, because those two tasks constitute the vast majority of RPC latency (c.f., Figure 3.4). Both manipulation tasks essentially reduce to the same low-level operation: given an in-memory object, create its wire format, or vice versa. Due to the prevalence of these manipulation operations and the associated CPU limitations, it is logical to have a bespoke component for object (de)serialization. Although such a component can provide impressive speedups for the manipulation tasks, having this component by itself is not enough as it cannot operate without CPU involvement, missing G1 and G2. In this section, we describe the set of components required to address both the manipulation and dispatch tasks of the RPC layer.

Enabling the RPCProc's manipulation hardware to directly handle incoming requests from the NIC requires additional control logic that initiates accelerator processing in response to incoming requests, replacing the CPU as the controller. Due to the complexity and recursive nature of the underlying objects inside the requests, the control logic should make no attempt to represent that structure, and instead be limited to creating tasks for the accelerators. This control logic simply indicates the correct schema to the data transformation (DT) component.

Figure 4.6 shows the specialized hardware components comprising our RPCProc. We start

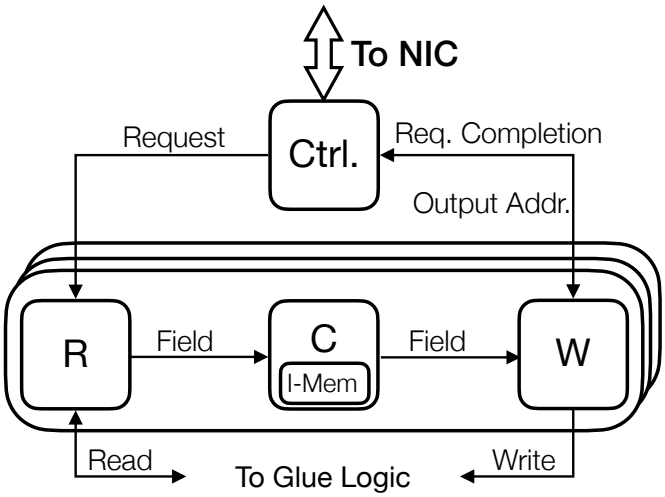


Figure 4.6: The Building Blocks of an RPC Processor.

with RPCProc's control logic (labeled Ctrl.), which is responsible for interacting with the rest of the system (i.e., the NIC in our case). The Controller contains internal registers which are read/written by the NIC when invoking new tasks. Upon receiving a new request, the Controller unpacks the request information and schedules a new task for the DT component which includes an array of transformation pipelines.

4.2.1 Handling Data Transformations

Conceptually, the DT component is organized as an array of independent transformation pipelines, each featuring a set of hardware units operating in a decoupled access-execute mode [82]. During serialization, its input units read data from the memory hierarchy based on the *Address* fields and feed transformation units, which feature simple ALUs that transform data according to the installed schema's rules. The output units write the transformed data to the designated memory buffer. An architecture like the DT component we have presented is sufficient to handle both RPC manipulation modules (header and payload).

Each transformation pipeline consists of three components dedicated to parsing the schema and fetching data blocks from the memory hierarchy, performing transformations, and writing back results. We first describe the specialized Converters (denoted C in Figure 4.6) and how they achieve complex transformation operations (e.g., `varint` encoding) in a few cycles.

Recall from Section 3.2.1 that serializing a `varint` requires at least one branch, two arithmetic operations, and one memory access per byte. Specialized hardware can achieve this complex operation in a single cycle by extracting each byte from the source data independently, performing the range checks, and inserting the correct continuation bits. Each such operation is read from a small instruction memory (I-Mem) which resides in the Converter. By designing the DT component's Converter around such specialized operations, the accelerator can attain transformation throughput at higher rates than traditional cores.

In keeping with our goal that the RPCProc architecture should be applicable to various frameworks (`G6`), we also make the Converter's I-Mem programmable by system software. The DT component is still usable for rare transformation operations and is forward-compatible with new software. Such custom operations will inevitably have reduced transformation performance due to the return of ISA limitations, but they will still reap the field-level parallelism enabled by using our schema. In such cases, a design with more Converters per transformation pipeline can overlap costly transformations to keep the overall transformation performance high. The software can customize the I-Mem's contents for new transformations at initialization time when the RPCProc is set up and the application context is created.

Each transformation pipeline also includes two decoupled components responsible for accessing the memory hierarchy, which overlap the memory accesses with actual transformations. These two components are the Reader and Writer (denoted R and W in Figure 4.6 respectively), which access data and stream it to/from the programmable Converter. Our schema also enables the Reader to perform multiple parallel memory accesses without requiring speculation, as each field's address is explicitly written in the schema by software (in the case of serialization). All memory accesses are performed through the glue logic shared with the NIC.

Once the transformation task is complete, the Controller will notify the NIC through its interface. The usage of parallel transformation pipelines with specialized Converters and decoupled components for accessing the memory hierarchy (i.e., the Reader and Writer) enables the DT component to process DT tasks at NIC's line rate, hence, achieving `G4`. We analyze this component thoroughly in Section 7.4.

4.2.2 Handling Dispatch

In order to eliminate excessive offload overheads and meet G2, the RPCProc must also contain dedicated logic to perform the RPC layer's dispatch module. The dispatch module must retrieve the target function ID from the parsed header, call the subroutine that deserializes the message's payload, and then transfer control to the appropriate function when payload deserialization finishes.

To realize this in hardware, the in-memory schema used by the accelerator must include information to specify which header field acts as the function identifier. We argue that as the Reader component is already able to parse the schema, it can be trivially enhanced to use this information to extract the function ID from the deserialized header and perform a table lookup to find the corresponding schema describing this function's payload manipulation task. Hence, there is no need for an additional component to perform the dispatch functionality. This logic can be trivially performed with a table lookup that returns the expected payload format and address of the matching function. The Reader will then fetch the payload's schema and starts feeding the Converter with the payload's fields.

In our design, we assume the function ID is an integer value or an index that can be directly used to perform the table lookup by the Reader. Because the RPC layer code is generated by a compiler based on the described services and message types, the RPC layer compiler (e.g., Thrift) can generate these function IDs. In a case where the function IDs are not treaded as a simple index value, the RPCProc needs to hash the function ID fetched from the message's header to get an index and perform the table lookup. Prior work has proposed simple hardware units for hashing and table lookups [83]; hence, similar hardware can be employed for RPCProc's dispatch module in such cases.

5 Cerebros: an RPC Processor

Simultaneously meeting all of the design goals of Chapter 4 for an RPCProc requires the following architectural characteristics: First, it needs to support the execution of all three modules comprising the RPC layer. Using dedicated hardware is feasible because the vast majority of the required functionality can be accomplished by an accelerator configured to perform any microservice's RPC tasks. Second, it must reside between the server's NIC and its CPU cores to eliminate excessive offload overheads and allow CPU cores to execute only the application business logic. Third, integrating the RPCProc with the server's NIC minimizes silicon deployment costs and enables affinity-based request steering. Fourth, the use of a powerful schema that uses simple type identifiers and memory addresses, enabling field-level parallelism and making the accelerator compatible with various frameworks. Finally, specialized hardware converters, which can perform data transformations in a handful of cycles and support a variety of operations defined by the software.

In this chapter, we present Cerebros, our implementation of a full RPC processor following the design principles presented in Chapter 4. We first briefly introduce the critical features of our assumed network hardware and discuss Cerebros' integration with NEBULA's network stack (Section 5.1). Next, we present Cerebros' software interface (Section 5.2), followed by the description of Cerebros' components that replace the RPC layer's modules (Section 5.3 and Section 5.4). We then conclude this chapter with the extensions for affinity-based request

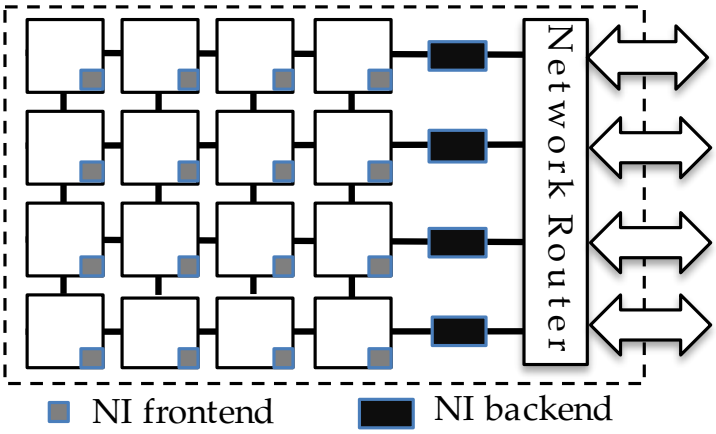


Figure 5.1: A server equipped with the NEBULA architecture following the Split-NI design.

steering (Section 5.5). Figure 5.3 presents the Cerebros architecture, with indicators showing the process of receiving and processing an RPC. Alphabetic indicators show events associated with the NIC, where numeric indicators show Cerebros' operations.

5.1 Integration with NEBULA

As motivated in Section 4.1.2, it is logical for Cerebros to be constructed over a baseline system featuring an on-chip integrated NIC and hardware-terminated protocol. We, therefore, select the NEBULA [25] architecture as our baseline. The NEBULA architecture features an RPC-oriented hardware-terminated transport and an integrated NIC attached to the server's on-chip network. NEBULA's NIC also supports load-balancing of incoming messages across threads of the same application.

5.1.1 NEBULA's Baseline Architecture

The NEBULA architecture is based on the Scale-Out NUMA (soNUMA) architecture [21] and follows the "Manycore Network Interface" architecture (aka. Split-NI) [80] for CPU-NIC interactions as shown in Figure 5.1. In this architecture, the network interface (NI) is split into two parts, a frontend and a backend, which together implement the soNUMA communication protocol [21] and NEBULA's extensions.

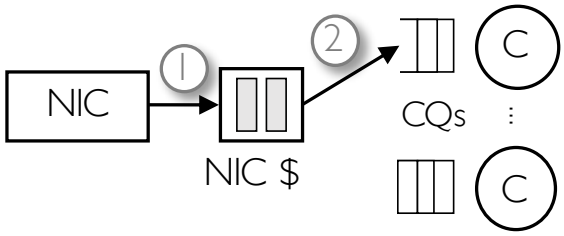


Figure 5.2: High-level overview of the baseline NEBULA architecture.

Software endpoints communicate with the NEBULA stack by using an RDMA-like memory-mapped Queue-Pair (QP) interface [77]. The frontend part of the NI handles all the control path interactions with QPs, and thus it is colocated with each core to accelerate the on-chip interactions. The backend part of the NI is replicated across the chip's edge and is in charge of all the data path interactions. It handles the arrival of new network packets and reading data from the memory hierarchy of the server. The communication between the two parts occurs via special packets sent over the on-chip network of the server.

Following the soNUMA architecture, the NIC in the NEBULA architecture is comprised of three independent pipelines: the Request Completion Pipeline (RCP), the Request Generation Pipeline (RGP), and the Remote Request Processing Pipeline (RRPP). The RGP is responsible for sending new RPC messages. The RRPP handles incoming requests, and the RCP handles incoming message replies. In the Split-NI design, the RGP and RCP are split between the backend and frontend components of the NIC, while the RRPP is only included in the backend. For further details of the Split-NI and its underlying pipelines, see [35].

Figure 5.2 shows the high-level key events that take place when the baseline NEBULA architecture receives a new RPC request. When packets arrive at the server, NEBULA's NIC pipelines extract the RPC message from the network packet by terminating the transport and reassembling the possibly fragmented network packets into a full message and place it into the NIC's dedicated cache, which is coherent with the server's memory hierarchy (①). For load-balancing reasons, NEBULA keeps the arrived messages in a NIC-private memory-mapped queue until a CPU core becomes available to process a new RPC. When a core indicates its availability, NEBULA creates a new entry in that core's Completion Queue (CQ), pointing

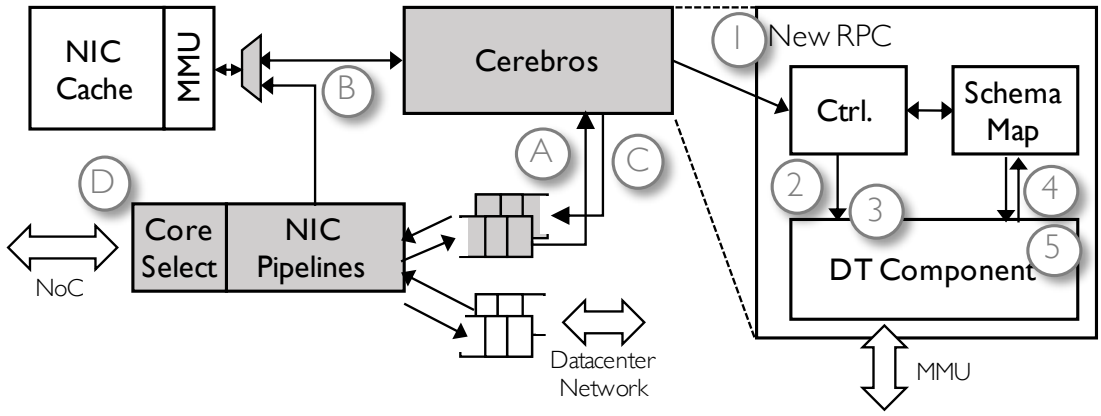


Figure 5.3: Architecture of Cerebros. Shaded components are modified or newly added.

to the received message's buffer location in memory (②). The core receives the RPC arrival notification by polling its private CQ.

5.1.2 NIC Interface and Execution Flow

To meet the goal of performing the RPC layer without CPU involvement (Chapter 4, G2), Cerebros needs to be inserted into the flow of incoming RPCs as a logical step between NEBULA's transport protocol termination and core notification. As our design goals are best fulfilled by integrating the RPC processor with the NIC, we choose to add a simple interface comprising two hardware queues between NEBULA's NIC pipelines (i.e., the RGP backend and the RRPP) and Cerebros' control logic. Cerebros only begins RPC processing *after* network protocol handling completes; the inverse is true for outgoing RPCs.

As in the NEBULA baseline, the NIC pipelines place incoming RPC messages into the NIC cache. The NIC invokes Cerebros' Controller (labeled as Ctrl.) through a hardware queue (Figure 5.3, Ⓐ), passing the address of the newly arrived message. Cerebros' data accesses all go through NEBULA's existing MMU (Ⓑ) and find the target data already resident in the NIC cache. Once Cerebros completes its processing tasks, its Controller returns a message to the NIC pipelines indicating that RPC processing is complete, which contains a metadata structure with all of the RPC's corresponding data (Ⓒ). The NIC pipelines' final stages then execute the core selection logic and use NEBULA's default mechanism to notify the selected

core through its QP (Ⓓ). In Section 5.5, we present how we use NEBULA's core selection logic for affinity-based request steering.

5.1.3 Memory Management

During its payload manipulation stage, Cerebros unpacks the incoming message's arguments and prepares them for the software to read. However, this implies that a buffer must be provisioned for the deserialized payload, in addition to the transport buffers reserved for and managed by NEBULA's protocol stack. Instead of adding another disparate memory reservation stage in Cerebros' hardware, a more efficient alternative is to unify this buffer management with NEBULA's transport buffer management and make them "all-or-nothing" atomic. Allocating both the transport and application buffers together avoids the need for additional logic in the NIC to handle cases where transport allocation succeeds but RPC layer allocation fails, which is likely to be rare and complex to handle. To unify the two buffering stages, we extend NEBULA's buffer manager (which originally only manages transport buffers) to also reserve memory for the deserialized payload. If either memory reservation fails, NEBULA returns a NACK to the sender according to its existing protocol; the sender reacts to the NACK according to a policy of its choice.

To ensure the allocated size for the application buffer is sufficient to contain the deserialized payload, we use the insight that in production RPC stacks, the maximum possible field-level compression is 4×. This compression occurs only in variable-length integers, which can shrink from eight bytes in their application format to two bytes in the network format. All other primitive types have lower compression factors due to additional metadata, and the same is true for composite types such as `Maps`. Therefore, Cerebros allocates 4× the network message's size for the deserialized payload, which is guaranteed to be sufficient memory even if the entire incoming message consists of variable-length integers. All RPC layer memory comes from the arenas pre-allocated and installed by the microservice through Cerebros' control interface.

5.2 Software Interface

Cerebros' control path is used at initialization time by microservices that wish to offload their RPC layer. Software must provide Cerebros with the following information in order for the full RPC stack to execute in hardware: i) its function IDs and their respective payload types, ii) the metadata (schema) describing each function's payload layout, ii) the globally shared format for header manipulation, and iv) a set of memory arenas used by the manipulation accelerator to place its output into. Each of these parameters is created once on application start, and programmed into Cerebros' memory-mapped control registers via `ioctl` system calls.

Each of the microservice's threads creates and registers a dedicated QP that is used for sending and receiving network messages. Incoming messages placed in the thread's QP have been completely processed by Cerebros and can be directly processed by the function whose ID is indicated in the new QP entry. Outgoing nested RPCs and responses are placed by the microservice directly in the QP without invoking software RPC processing, which is completely performed by Cerebros before the message is delivered to the NIC for transport encapsulation. In case Cerebros cannot process a message (e.g., due to an unrecognized function), a fallback mechanism sends the unprocessed message to a thread, indicating with a null function ID that the RPC layer must be executed in software. Next, we discuss the architecture of the components comprising Cerebros.

5.3 Data Transformation Component

Due to the commonality between the operations, both header and payload manipulation can be handled by a single hardware component performing data transformations. Cereal [84] is an example of an accelerator that targets data transformations with bespoke hardware components. However, it only works with a dedicated serialization format, limiting its generality. Following our `G6` from Chapter 4, we believe a data transformation component (DTC) that does not require changing each microservice's data format to match the specific DTC implementation is more applicable to datacenter microservices. Therefore, Cerebros adopts

such a DTC design for header and payload manipulation.

The DTC's key enabling feature is the use of a transformation schema, an in-memory data structure that represents the parallel sub-tasks comprising each manipulation request. Cerebros uses this transformation schema as a flexible accelerator interface that allows defining all types of parallel data manipulation tasks, facilitating compatibility with any RPC framework after the schema's format is established.

The DTC is internally organized as an array of independent transformation pipelines. A Transformation Pipeline is architected as a decoupled access-execute pipeline [82] and includes a Reader, Converter, and Writer. We now present the implementation details of our DTC. Figure 5.4 displays the microarchitecture of our DTC, comprising its transformation pipelines and their three internal components. As we describe each component, we walk through the process of *serializing* a message. A similar process applies to *deserialization*.

5.3.1 Reader

The Reader parses the schema that comes from the Controller, fetches all the fields from the memory hierarchy, and sends them to the Converter. The Reader receives a request's schema pointer from the Controller through a hardware queue and issues a memory request for that address to NEBULA's MMU. The Reader gets a cache line containing schema fields, which it stores in a dedicated Field Buffer. The Reader then fetches a field from the Field Buffer, extracts the data pointer, and issues a read request to the MMU. If a field is a sub-message, such as the `Phone` field of `Person` in Figure 4.5, the Reader recursively fetches the schema of that sub-message in a depth-first manner.

The Reader gets a cache line containing the field's raw data, which it then stores in its Data Buffer. The Reader then extracts the required data (in *Chunks*) from the Data Buffer based on the field's type, and forwards it to the Converter to carry out the transformation. The Reader also calculates the offset where the Writer should place the transformed data, again depending on the schema. To illustrate this process, consider the `string` field in Figure 4.5. Once the

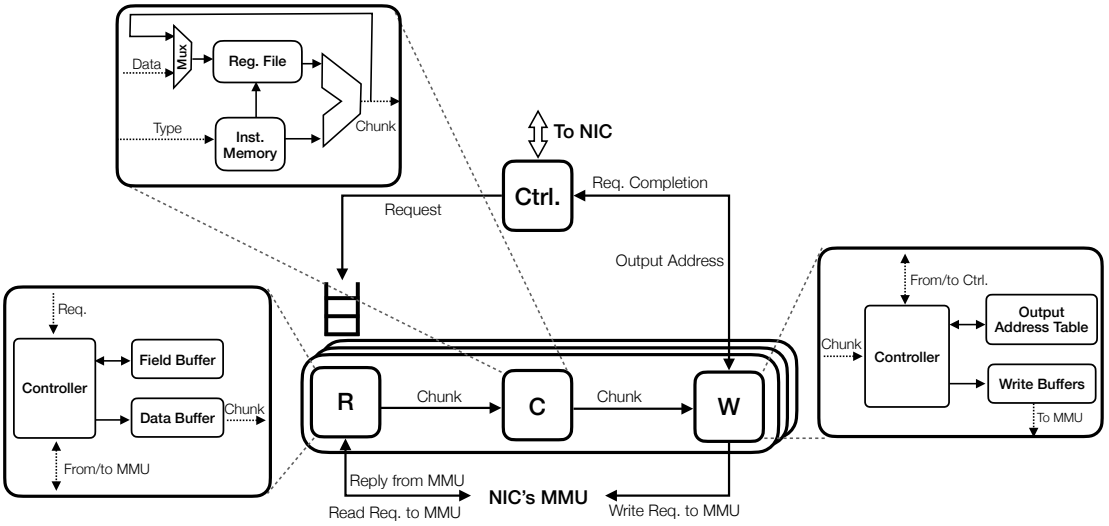


Figure 5.4: Overview of the microrchitecture of Cerebros' data transformation component.

second field of the schema is present in the Field Buffer, the Reader determines that it is a `string` of length eight. The Reader picks the correct eight bytes from the cache line, forms a Chunk with the correct output buffer offset and transformation type, and sends it to the Converter. During deserialization, the Reader fetches data from the serialized buffer, finds the corresponding field in the schema, and passes the information to the Converter.

5.3.2 Converter

The Converter takes in the Chunks sent by the Reader and performs the required data transformation. Once completed, the Converter passes the transformed field to the Writer. The Chunks contain information that identifies the field's type and therefore what operation to execute. A small (128-entry) instruction memory stores a sequence of instructions for each application-defined type to perform the conversion. This memory is initialized when the application requests to use the accelerator and is indexed by the type field in the Chunk. After data is transformed, the Converter passes the converted bytes to the Writer to be written to the output buffer.

The Converter is implemented as a simple pipeline with the following four stages: instruction fetch, decode and register file read, execute, and register file write-back. The field type

included in the Chunk indicates the entry in the instruction memory that the Converter executes. For common data types (e.g., `varint`), a single instruction performs the conversion. Other transformations that do not have specialized instructions can execute a sequence of instructions at the cost of reduced throughput. While in our case the Converter takes four cycles for each Chunk (a single instruction), the number of cycles the Converter requires to transform data varies depending on the complexity of the transformation. As such complex transformations that are not supported with a single instruction can benefit from having more Converters (in the same transformation pipeline) that work in parallel on various fields of the same object.

The optimal number of Converters per transformation pipeline is easily determined with Little's Law. If the expected latency (given by W) of a non-dedicated transformation is ten cycles and the arrival rate of Chunks (given by λ) from the Reader(s) is one Chunk per two cycles, the correct number of Converters to provision is given by: $L = \lambda\ W = \frac{1}{2} \times 10 = 5$. We provision a single Converter per pipeline because when operating at peak throughput, a Reader can produce one Chunk per cycle if it is picking bytes from a contiguous array. A single Converter and Writer can keep up with this peak throughput. Once the Reader has finished queuing all of the Chunks for a message, it can continue to the next message while the Converter and Writer complete the transformations and write-backs.

5.3.3 Writer

The Writer receives transformed data from the Converter and writes it at the appropriate location in the output buffer, which is identified by a `(base,offset)` pair. The base address is supplied by the NIC, and is passed to the Writer by the Controller, while the offset is calculated by the Reader and passed to the Writer. The Writer contains internal write buffers that assemble a cache line of transformed data from the Converter and writes it through NIC's MMU. During deserialization, the Writer also writes the data pointers in the schema. Finally, once the Writer issues all the writes for a request, it notifies the Controller of completion.

An architecture like the DTC we have presented is sufficient to handle both RPC manipulation

modules (header and payload); the next component that needs to be addressed is the one handling dispatch.

5.4 RPC Dispatch

Moving the dispatch module into hardware is mandatory for complete RPC layer processing on Cerebros without CPU involvement. We now walk through the tasks performed by Cerebros when the dispatch stage executes, using Figure 5.3 as a guideline. When a new RPC task arrives at Cerebros from the NIC (①), Cerebros' Controller assigns the RPC to an available transformation pipeline and passes the request's metadata to it (②). After the DT component first parses the header (③), Cerebros must (i) determine the function ID being requested, and (ii) prepare the payload manipulation task corresponding to that function's message type.

To meet these two requirements, we extend Figure 4.5's schema format to include a special marker indicating which field of the header contains the function ID. The Reader in the DTC's pipeline uses the schema to extract the function ID from the deserialized header. After the function ID is known, Cerebros uses a small table, called the Schema Map, that maps this ID to the correct schema corresponding to the incoming request's payload format (④). The Schema Map is exposed via an in-memory configuration space and programmed by the microservice at start-up through Cerebros' control path. The Schema Map's storage requirements are limited because we expect the number of concurrently active functions to be a few tens.

We also introduce the idea of a *split schema*, which decomposes each schema into two parts. The first part is the `Type` column of Figure 4.5, which only represents the data types pertaining to a particular message class. As all of the messages reaching a particular function are of the same type, the `Type` schema remains immutable and is shared among all messages of the same type, including headers. The second part is unique for each individual message and contains the data pointers (the `Address` column). Dividing the schemata in this fashion roughly halves their storage requirements, as Cerebros will access the same read-only `Type` schema for all incoming messages to the same function. Additionally, such division eliminates the need for

Cerebros' DT component to create a new `Type` schema in memory for every request.

Returning to Figure 5.3, after header parsing completes, Cerebros uses the special marker in the header schema to extract the function ID. It then looks up the Schema Map (④) which returns the `Type` schema for the corresponding function's payload type. To prepare the payload manipulation task for the incoming RPC, Cerebros creates a blank `Address` schema in the memory previously reserved by the NIC's pipelines for the DT component to fill out with each `Type`'s address. Cerebros' DTC then begins parsing the payload by reading the raw payload and filling out the `Address` schema with the addresses where the application-readable fields were placed (⑤). When payload manipulation completes, Cerebros sends two pieces of information to the NIC's core selection stage (Ⓒ): a pointer to the buffer with the application-readable request, and the function ID.

5.5 Affinity-Based Request Steering

The final task remaining is to select a core to send this RPC to—the result of this process is what allows us to realize affinity-based request steering. NICs already implement logic to perform core selection based on a variety of metrics (e.g., load balancing [52] or TCP 5-tuple [81]). Cerebros contains a core selection stage that obtains a set of desirable cores for handling this function from a table called the *function map* (Ⓓ). The function map is a direct-mapped table storing a FIFO list of recently executed function IDs for each CPU core. When a new RPC is assigned to a core, the function ID is added to the head of the core's list, and the tail of the list is dropped. 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.

Selecting a core for a new RPC involves comparing the function map's entries against the incoming function's ID, and considering that a core has affinity to this function if the incoming ID matches. To preserve load balancing, Cerebros' core selection stage then chooses the core with the fewest number of outstanding RPCs from the set of all cores having affinity to this function. Such policies that assign requests based on the number of outstanding requests

per core have been implemented in hardware by prior work [52, 85]. Further core assignment policy optimizations (e.g., increasing the depth of the list in the function map in the case where multiple functions have constructive code sharing) are interesting extensions to our proof-of-concept implementation.

While the core is being selected, NEBULA's NIC pipelines create a metadata structure containing a pointer to the incoming message's *Address* schema, the corresponding request buffer, and a function pointer that indicates the address where the core must begin executing. Cerebros 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.

6 Evaluation Methodology

In this chapter, we detail our evaluation methodology. We first describe the microservices and simulation setup used for evaluating Cerebros (Section 6.1). Next, we present the methodology used for a deeper evaluation of the data transformation component along with the implementation details of a stand-alone version of the data transformation component (Section 6.2).

6.1 Full RPC Layer Acceleration

6.1.1 Evaluated Microservices

We choose microservices from DeathStarBench [1] that differ in the following primary parameters that dictate the RPC layer's cost breakdown: number and complexity of functions, frequency of nested RPCs, and message size/format complexity. Our microservices are UniqueId (UID), User (USR), UrlShorten (URL), SocialGraph (SG), and ComposePost (CP), which comprise one, six, one, seven, and six underlying functions, respectively. The selected microservices represent DeathStarBench's various microservice classes. Other microservices in this benchmark suite behave identically or similarly to those we evaluated. In particular, most of the microservices are similar to SG and CP, which contain little business logic and spend most of their execution time just passing data along to other microservices or data stores via nested RPCs. Facebook has also recently revealed their web services (the closest

workload to DeathStarBench's microservices) spend as little as 18% of their execution time in the application logic [86].

All microservices use Apache Thrift [32] as their RPC layer, to which we have added a new hardware-terminated transport protocol based on NEBULA [25]. We study each microservice in isolation and create mock components for the other microservices surrounding the isolated one. Due to our use of isolated microservices, we report the CPU cycles expended in only the RPC and application layers. Therefore, our results are independent from the underlying transport and network protocols.

6.1.2 Request Processing Model

Our evaluated RPC layer implements a synchronous request processing model, where each microservice polls for incoming requests and executes them to completion. Threads also synchronously poll for the results of their nested RPCs, which Cerebros guarantees will be returned to the same thread. An asynchronous processing model (where threads begin processing new requests instead of polling for responses to nested RPCs) would provide higher throughput at the cost of extra CPU cycles spent for context switching and higher programming complexity [7]. A user-level threading library such as Arachne [87] would be mandatory for handling the μs-scale execution times of our evaluated microservices. We emphasize that because Cerebros' primary target is the reduction of CPU cycles expended per request, it benefits either processing model, and the saved cycles can be re-purposed to increase concurrency if asynchronous RPCs are used.

6.1.3 Microservice Characterization

To accurately measure the breakdown between the functions and RPC layer, we instrument the microservices' code to record cycles expended in the following three steps: (i) the RPC processing that occurs upon new requests arriving, (ii) nested RPCs that occur during the function's execution, and (iii) the function code itself. Therefore, the cycles we attribute to the function quantify only the time spent executing the application's business logic. Reported

Cores	ARM v8; 64-bit, 2GHz, 4-way OoO TSO, 128-entry ROB Next-line instruction prefetcher
L1 Caches	64KB 4-way L1d, 64KB 4-way L1i, 64B blocks 2 ports, 32 MSHRs, 4-cycle latency (tag+data)
LLC	Shared block-interleaved NUCA, 8MB total 16-way, 1 bank/tile, 8-cycle latency
Coherence	Directory-based Non-Inclusive MESI
Memory	45ns latency, 2×25.6GBps DDR4-3200
Interconnect	2D mesh, 16B links, 3 cycles/hop

Table 6.1: Architectural simulation parameters for evaluating Cerebros.

cycle counts are the average number of cycles expended per request across all functions for each microservice. To estimate instruction working set sizes, we apply the methodology used for profiling workloads in Google datacenters [24]: we collect the trace of executed instructions and measure how many unique cache lines cover 99.9% of the trace when ranked by popularity.

6.1.4 Simulation Setup

We evaluate Cerebros using cycle-accurate full-system simulation. We use the QFlex simulator [57] to simulate a 16-core ARMv8 CPU running Ubuntu Linux 18.04. Table 6.1 summarizes our system's configuration parameters. 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, and delivers notifications to the CPU through the NEBULA transport stack. For the User microservice, in particular, we use the following popularity distribution: 5% of the requests are for the `RegisterUser` function, 5% for the `RegisterUserWithId` function, 10% for the `Login` function, 30% for the `UploadCreatorWithUserId` function, 30% for the `UploadCreatorWithUsername` function,

and the remaining 20% requests are for the `GetUserId` function. The load generator also emulates all the mock microservices, mimicking their behavior and instantly responding to RPCs with pre-constructed messages.

6.2 Study of the Data Transformation Component

The data transformation (DT) component is the most critical component of Cerebros as the majority of the RPC layer's cost is associated with the data transformations inherent in both the header and payload manipulation tasks. Cerebros's performance and its ability to perform RPC processing at NIC's line rate is dictated by how fast its DT component can process DT tasks. Hence, we see the need to perform a stand-alone study of the DT component.

Such DT tasks are also found in workloads other than microservices, such as databases, data analytics, and generally wherever multiple software components need to work together to achieve an end goal. Moreover, it is common for storage systems (e.g., key-value stores) to store objects in a serialized format such as Protobuf's binary format. Because our DT component has use cases in other application domains, we propose and implement a stand-alone version of our DT component or a DT accelerator (DTA) that can process DT tasks without being integrated with the NIC. On the contrary to what we proposed in Chapter 4, this design presents a more relaxed and less intrusive point in the design space, where the server system does not require integrated NICs, and the overall throughput of the system is more important than its service latency. We still maintain the goal of achieving the NIC's line-rate performance.

We now describe our design for a DTA, followed by our implementation of such a stand-alone DTA, called Optimus Prime. In Chapter 7, we evaluate the performance of the stand-alone DTA using micro-benchmarks (Section 7.4) and also compare this implementation to our full RPC processor in the context of microservices to highlight the impacts of offload overheads (Section 7.5).

6.2.1 Designing a Stand-Alone DTA

In this section, we lay out the design of a DTA, prioritizing the following design goals. First, the DTA must be able to directly communicate with the cores, to get new tasks and notify the cores on completion of the tasks. Second, the DTA should perform DT tasks at NIC line rate, as this DTA will still be used by software components that commonly communicate with each other. Third, similarly to the RPCProc, the DTA should be compatible with existing DT frameworks and programmable to allow compatibility with future data formats. Third, a DTA should have minimal impact on existing server architecture, limiting deployment cost. To achieve these goals, we seek to answer the following questions: i) what interfaces should the accelerator have with the software framework and the server system, ii) what additional components are needed as building blocks compared to the DT component we designed for our RPCProc, and iii) where should the accelerator reside in the server?

Interfaces

We argue that the combination of our proposed transformation schema (Section 4.1.3) with programmable Converters (Section 4.2.1) is enough to allow the DTA to be framework-agnostic. The only requirement is that the software framework needs to update the schema while creating the message. Moreover, the schema enables the hardware to operate on each field in parallel by scanning the schema, accessing the data to be transformed, and performing the requested operations. We now present the interface design between the DTA and the rest of the system.

In the left half of Figure 6.1, we see a traditional multi-core server system, with a number of cores connected by an on-chip network (NoC). Because the DTA must be able to interact with the cores directly, it must use its own “glue logic” rather than relying on the NIC’s. In particular, the DTA exposes a set of internal registers to the system software, which we map to I/O virtual addresses (IOVAs) in each process’ address space to enable kernel-bypass and minimize invocation latency [44]. Applications use memory-mapped I/O (MMIO) writes to

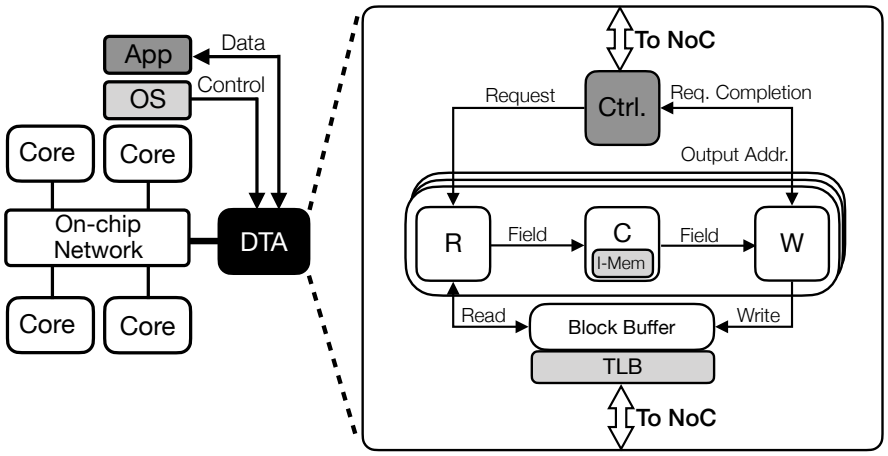


Figure 6.1: Architectural overview of a DTA. Light grey structures are configured by the control path, and dark grey structures directly communicate with the application.

these IOVAs to request new transformations, and repeated MMIO reads to poll for completions. Additionally, as the DTA needs access to the schemata and objects, it requires access to the server's virtual memory system. Given such access to the virtual memory system, the DTA can use regular pointers for schemata and messages, avoiding wasteful copies. Each request contains pointers to the schema and the output buffer for this transformation.

For an application to begin using the DTA, it performs a system call that returns a private context containing: (i) a set of per-core memory arenas where all messages from the application must be constructed, and (ii) the I/O virtual addresses where new requests are to be submitted. It is common for DT frameworks to use arena-based memory management [88], which follow the principles of user-level allocators like `jemalloc`. Because software already builds its messages in these arenas, the system call provides the arenas' virtual addresses to the DTA so it can access the messages to be transformed. While there are rare cases where the application has pre-populated responses in its internal data structures, we focus on the general case where the application creates a new response message for every request.

To provide compatibility with a variety of DT frameworks, our DTA also has an interface for applications to program custom transformations. Upon requesting a new transformation context, the application also has the option to issue system calls to program custom operations into the DTA itself. We now present the internal building blocks of our DTA and how they

implement the above interfaces.

Building Blocks

The right half of Figure 6.1 shows the specialized hardware components comprising the DTA. The main two changes compared to the RPCProc's DT component are in the Controller and the Block Buffer. We start with the Controller (labeled Ctrl.) component as it is responsible for interacting with the server's cores (as opposed to interacting with the NIC pipelines in the case of RPCProc). The Controller contains the DTA's internal registers, which are read/written by the cores when invoking new transformations. Upon receiving a new request, the Controller unpacks the schema and output buffer pointers and sends them to a transformation pipeline. Once the transformation is complete, the Controller's registers are updated and the core will see the completion with its next MMIO read.

The design of the transformation pipeline is the same as in the RPCProc—i.e., architected as a decoupled access-execute pipeline [82] of three components dedicated to parsing the schema and accessing data from the server's memory (Reader), performing transformations (Converter), and writing back results (Writer). However, all memory accesses are now performed by means of a non-coherent Block Buffer, which acts as a scratchpad for the Reader and Writer, translates the virtual addresses of each field, and issues the corresponding reads and writes to the server's NoC. We next discuss the physical placement of DTA in the server.

Physical Location

Placing the DTA off-chip and connected over PCIe offers the lowest cost and least intrusive design point. However, the $\sim 1\mu s$ latency of the PCIe interconnect [89] quickly becomes an obstacle for common nested messages (e.g., `Person`), due to the multiple PCIe roundtrips to fetch the nested message's pointers. Therefore, we focus on the tradeoffs inherent to an on-chip DTA, and discuss the following options: a private DTA co-located with each core, or a shared DTA that is placed on one of the chip's tiles.

These two choices expose a critical tradeoff between transformation latency and silicon provisioning. In the case where a DTA is co-located with each CPU core, it shares the core's L1 cache and TLB, eliminating the need for the Block Buffer component. However, private DTAs will require the Reader, Writer, Converter and Controller to be replicated. Effectively, choosing private DTAs costs more silicon but lowers data access latency and eliminates variability in the access latency, which is attributable to the NUCA architecture of the server's LLC. In contrast, attaching the DTA to the NoC as a shared component accepts the variability but attains more efficient silicon provisioning. We claim that despite the higher memory access latency and variability, the DTA should be shared due to the fact that the silicon costs of private DTAs quickly add up with increasing core counts. Our DTA is therefore shared and sits at the chip's edge, as shown in Figure 6.1.

6.2.2 Optimus Prime

In this section, we present Optimus Prime (OP), our implementation of a stand-alone DTA, which follows the principles in Section 6.2.1. OP includes two main changes compared to the Cerebros's DT component presented in Figure 5.4: (i) the Controller component is modified to enable direct interaction with CPU cores, and (ii) OP includes a new component called Block Buffer that facilitates the Reader's and Writer's data accesses.

Controller

The Controller receives transformation requests from the cores and notifies the corresponding core upon completion. It contains a set of dedicated control registers which CPU cores access through MMIO to request new transformations. Each request includes a pointer to the schema to be transformed, a pointer to the output buffer where data has to be written, a pointer to the serialized buffer (for deserialization only), and a valid bit. When a new request arrives in the control registers, the Controller's internal control logic passes the schema pointer to the Reader and output buffer pointer to the Writer.

Upon request completion, the valid bit is cleared and the CPU core will determine the trans-

formation is completed with its next MMIO read. In our implementation, transformations are synchronous in nature; therefore, a core waits for a request completion before it issues another. Asynchronous transformations can also be implemented by writing each request to a different control register and polling each one.

Block Buffer

OP has a virtually-indexed, virtually-tagged Block Buffer, which is not coherent with the rest of the on-chip hierarchy. Synonyms are resolved by tagging each entry with the core ID associated with the transformation. If a data request to the Block Buffer results in a miss, the Block Buffer issues an explicit read request using the cache coherence protocol. The Block Buffer has a TLB, which contains the virtual-to-physical translations for per-core memory arenas where applications construct their objects. The OS allocates and pins a per-core arena at initialization time for each application and fills the TLB with the translations.

We assume services that wish to use OP run on dedicated cores, which is a common practice in datacenter workloads. As such, the TLB has as many entries as cores, is directly indexed by core ID and maintains the translation for as long as the microservice is active. Such a direct-mapped table has a small silicon footprint even with hundreds of cores. As the cores create every object in their private arenas whose translations are pre-installed, the TLB never misses. The total amount of pinned memory for the arenas is also relatively small, given that modern servers integrate hundreds of GB of DRAM [90]. Filling translations into the TLB at initialization time is a low-cost operation required only in the case of a context switch.

Optimizations

Each transformation pipeline's throughput heavily depends on memory access latency. Before any Converter can begin transforming data, a Reader must perform at least two memory accesses, one for schema and another for the corresponding data. More accesses are required for fields that are objects themselves (i.e., sub-objects). However, as the Reader has access to the object's schema in its Field Buffer, it can issue prefetches for each upcoming field and

overlap the access latency. These prefetches attain 100% accuracy because the object's schema explicitly contains the address of each field.

Even with prefetching, we find that the pipeline still spends the majority of its cycles waiting for memory accesses. To increase utilization further, the pipeline can be time-shared among multiple requests. This technique is similar to coarse-grained multi-threading in CPUs [91] and requires keeping multiple request contexts per Reader, which can be rotated in one cycle. The Converter and the Writer do not require contexts as they do not retain the message state.

Time-sharing provides almost the same performance as physically replicating the entire transformation pipeline. The optimal degree of time-sharing is limited by the pipeline's idle fraction. For example, a pipeline that is stalled 75% of the time will have a utilization rate of 100% with four contexts. Adding contexts beyond this point will only increase request latency. To aid the explanation of OP's possible configurations, we introduce the following notation: $OP_{\{i,j\}}$ refers to OP with i physical transformation pipelines, with each being time-shared among j DT requests.

6.2.3 Methodology

Microbenchmark

To study the performance of our stand-alone DTA, Optimus Prime (OP), we use a multi-threaded micro-benchmark that generates (de)serialization tasks based on Google Protobuf. In order to directly evaluate the maximum throughput of OP, the micro-benchmark sends (de)serialization requests to OP in a tight loop. This scenario represents the upper bound of the load offered to OP, as in a real-world deployment, the application will also consume CPU time. As specified in Section 6.2.1, each core sends a transformation request to OP with MMIO writes, and repeatedly polls the address to check if the request is complete. Once the request completes, the core generates a new request and sends it to OP.

To choose representative objects to be transformed, we create three object classes shown in Table 6.2. The sizes of these objects are chosen to represent the fact that the majority of

Object Type	R/W Ratio	Max Depth	Size (B)
Flat	2.6	1	485
Mixed	2.75	2	297
Nested	4.25	2	232

Table 6.2: Object types and their characteristics.

network packets sent by modern online applications are sub-1KB [41, 14]. The R/W ratio is the number of bytes that must be read for each byte written in the serialized output, and depends on each field's depth and type. For example, `varints` are converted to different byte-streams based on their values, where `strings` have an R/W ratio of one. Moreover, the depth of each field (i.e., the number of sub-objects that must be parsed before returning to the top level) increases the read/write ratio.

Area and Power Analysis

To estimate OP's area and power, we implemented OP in VHDL and synthesized it with the Synopsys Design Compiler [92] using TSMC 28nm technology (Core library: TCBN28HPMBWP35, Vdd: 0.9V). We use a 2GHz clock rate and set the compiler to the high area optimization target. The synthesized RTL only takes into account the Controller, Reader, Converter, and Writer. We add the power and area of the Block Buffer and TLB using CACTI 6.5 [93]. Finally, we compare our area and power overheads with Cortex-A57 numbers from prior work [94] in Table 7.1.

System Organization and Simulation Parameters

We simulate a 16-core ARMv8 server running Ubuntu Linux 18.04 in full-system cycle-level detail using the QFlex simulator [57], which combines the QEMU emulator with the timing models from the Flexus simulator [95]. Table 6.3 summarizes the simulation parameters. All workloads are pinned on 15 cores, leaving one core for OS threads and interrupts. OP is attached to a corner tile of the NoC mesh, which has access to two NoC links. Therefore, OP has a total read/write bandwidth of 32 bytes/cycle (i.e., 512Gbps).

To quantify the implications of offload overheads, we estimate the best-case performance

Cores	ARM v8; 64-bit, 2GHz, OoO 3-wide dispatch/retirement, 128-entry ROB, TSO
L1 Caches	32KB 2-way L1d, 48KB 3-way L1i, 64-byte blocks 2 ports, 32 MSHRs, 2-cycle latency (tag+data)
LLC	Shared block-interleaved NUCA, 8MB total 16-way, 1 bank/tile 6-cycle access
Coherence	Directory-based Non-Inclusive MESI
Interconnect	2D mesh, 16B links, 3 cycles/hop
Memory	45ns access latency
OP	Block Buffer: 8KB, 2-way, 64-byte blocks, LRU 64MSHRs, 1 cycle hit, 2 read/write ports TLB: 2MB pages, 64 entries, direct mapped

Table 6.3: Architectural simulation parameters for the stand-alone DTA study.

for Optimus Prime—i.e., we assume no queuing delays on the accelerator and that all data it requires is delivered with a single access to the cache hierarchy. We model the accelerator's processing time as the cycles required by OP's transformation pipeline to process all the fields of the message. To calculate the cost of a single synchronous offload, we model five sequential traversals of the server's on-chip network: (i) the CPU invokes the accelerator through MMIO writes; (ii) the accelerator reads the metadata describing the task, which is delivered separately from the invocation; (iii) the accelerator reads the data block(s) corresponding to the task; (iv) the data to be returned is written back to the cache hierarchy; and (v) the accelerator notifies the CPU. Each of these traversals incurs a latency of 40 cycles, measured using our cycle-accurate simulator.

7 Evaluation

Chapter 4 presented a set of design guidelines for building an RPC processor. In Chapter 5, we introduced Cerebros, a full RPC processor integrated with the NEBULA architecture and implemented following the guidelines of Chapter 4. In this chapter, we evaluate the performance impact of Cerebros on microservices and justify the design choices we made in Chapter 4.

We begin our evaluation by showing Cerebros' ability to virtually eliminate the RPC processing tax and to achieve our first design goal: that CPU cores only execute the microservices' business logic and not the RPC layer (Section 7.1). Next, we demonstrate how fully offloading the RPC layer actually improves the performance of the microservices themselves (Section 7.2), and show the benefits of affinity-based request steering (Section 7.3). We then focus on data transformation (DT) component and evaluate our stand-alone DT component, Optimus Prime, and perform a scalability study to show how it achieves line-rate DT processing (Section 7.4). We also show the implications of only accelerating the data transformations within the RPC layer and quantify the performance implications of offload overheads, demonstrating the need for Cerebros to directly interact with the NIC and perform the entire RPC layer (Section 7.5).

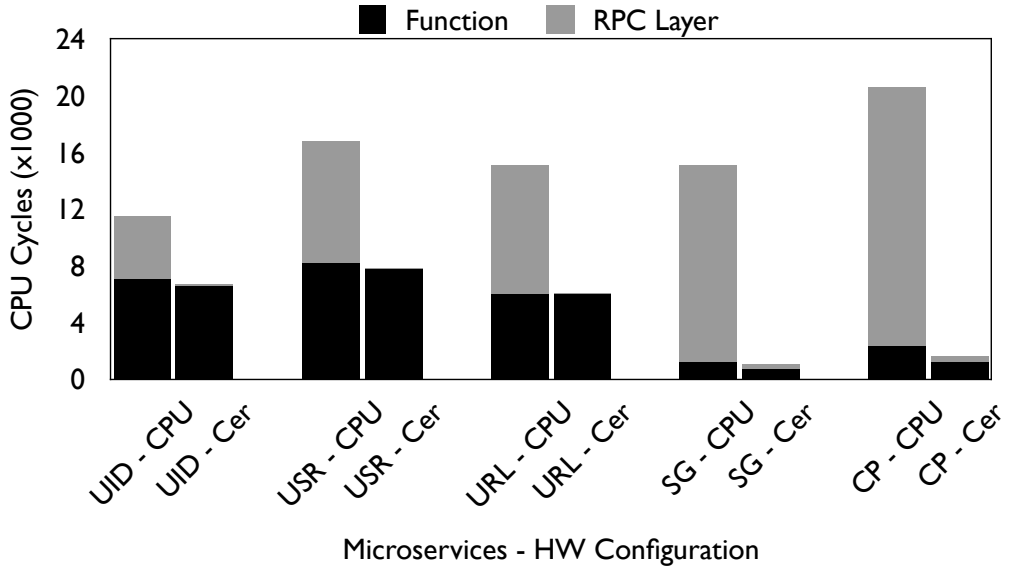


Figure 7.1: Average on-server cycles per request.

7.1 RPC Layer Acceleration

Figure 7.1 shows the mean on-server expended CPU cycles per request, broken down into the RPC layer and the application-level function. Cerebros virtually eliminates the cycles spent in the RPC layer and thus reduces the expended CPU cycles by 1.8–14.2×, depending on the fraction of cycles attributed to the RPC layer in the baseline.

The effect of RPC layer offload to Cerebros is most pronounced for SG and CP, as they spend ~90% of their cycles in the RPC layer due to heavy use of nested RPCs (Section 3.1.1). The functions of these two microservices primarily pass along the information contained in their input messages to other microservices, performing tiny amounts of business logic. A function in CP can include up to 13 nested RPCs, accounting for up to ~70% of the microservice's total expended cycles on average (as shown in Figure 3.2). In contrast, SG has a maximum of five nested RPCs, but the messages it exchanges with other microservices include complex nested objects and are larger than CP's. Message size and complexity make SG's breakdown of RPC versus function time similar to CP's, despite fewer nested RPCs. Cerebros is able to effectively eliminate the RPC layer's overheads, whether the underlying root cause is deep RPC nesting or transformation complexity.

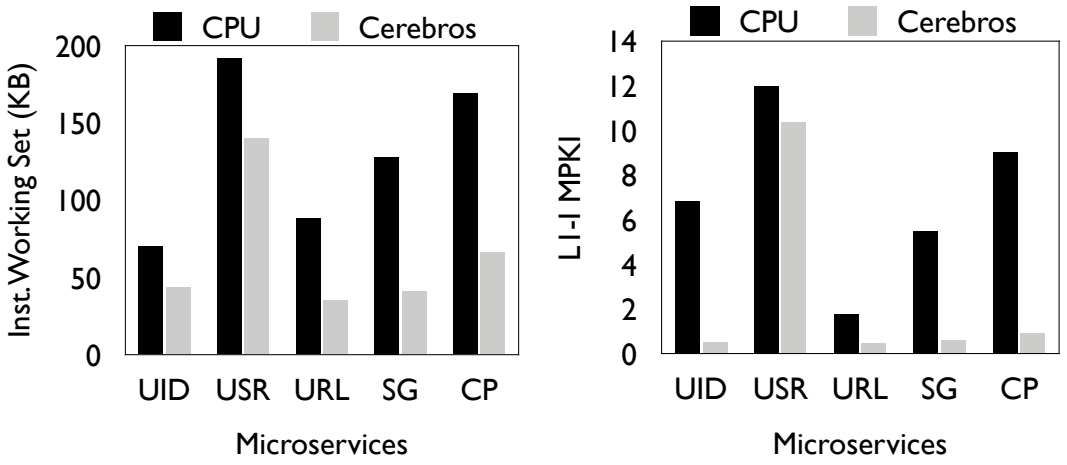
Business logic in UID, USR, and URL is more complex, hence forming a more notable fraction of the expended cycles. Additionally, UID's messages do not contain complex objects and are smaller than 50B in size. Despite the relative simplicity of UID's RPC tasks, Cerebros still attains a 1.8× reduction in CPU cycles.

7.2 Improved Function Performance

As a side-effect of full RPC layer offload, Cerebros reduces the on-CPU service time of each microservice's business logic as well. Figure 7.1 shows 2–49% fewer expended cycles in the functions as a result of improved CPU frontend performance due to reduced instruction working set. To clearly show the effect on the CPU frontend, we measure the working set sizes and and the number of misses per thousand instructions (MPKI) of our five microservices in two configurations: when the RPC layer is performed by the CPU and when it is offloaded to Cerebros.

Figure 7.2a shows the instruction working sets of our evaluated microservices. In the baseline CPU system, the bloated RPC layer results in total working sets that exceed the L1-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 L1-I hit rate. The working sets are most visibly reduced for SG and CP, as they have little business logic in their functions and their instruction footprints correspond more directly 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 a reduction of more than 60% in their instruction working sets. 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 7.2b depicts the L1-I MPKI before and after the 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



(a) Instruction working set.

(b) MPKI.

Figure 7.2: Frontend behavior of microservices.

on instruction misses by 5–93%. Instruction miss reduction also yields a 2–49% reduction in function cycles, as shown in Figure 7.1, highlighting that RPC layer offload has a significant positive side-effect on the CPU performance.

USR benefits the least among all microservices because it includes two functions with working sets larger than 90KB in size. It also experiences the smallest reduction in the instruction working set, as shown in Figure 7.2a. In such cases where the aggregate working set of all the functions still outstrips the L1-I cache, even fully offloading the RPC layer to Cerebros provides limited benefits to the CPU's frontend. We now evaluate the performance of affinity-based request steering that ameliorates CPU frontend inefficiencies in these exact cases.

7.3 Affinity-Based Request Steering

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 core selection policy does not take into account function locality, all six functions will compete for L1-I cache capacity, resulting in the high number of instruction misses visible in Figure 7.2b even after Cerebros' RPC layer offload. This phenomenon particularly hurts the performance of functions for which L1-I misses account for a large fraction of total execution time.

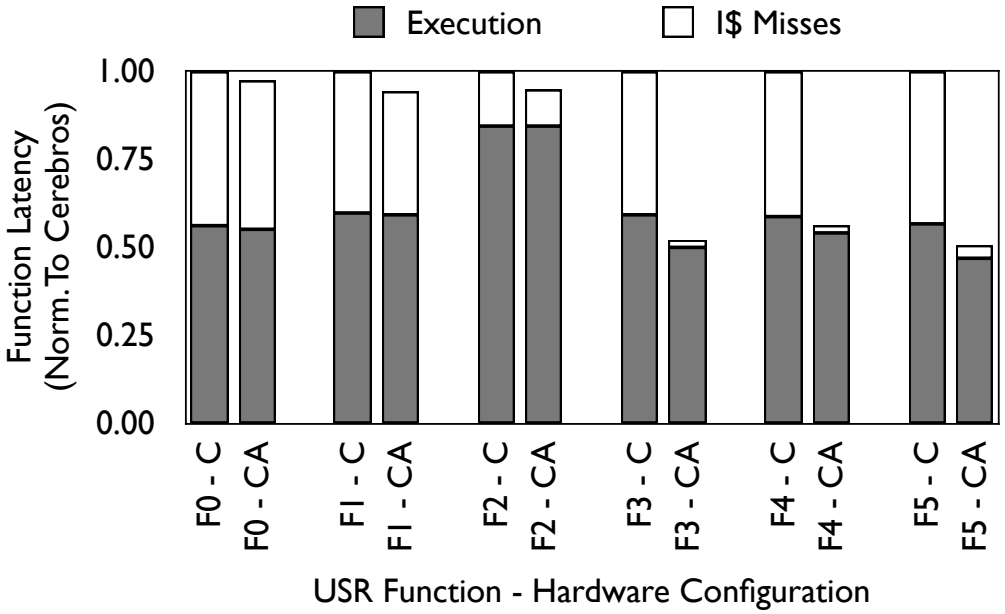


Figure 7.3: Breakdown of USR' functions into execution time and instruction cache misses.

Figure 7.3 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 steering (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 L1-I. All other functions have their CPU times divided roughly equally between execution and instruction stalls.

With affinity-based request steering enabled, the fraction of time stalled on L1-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 able to be steered to a core that had just executed the same function type, highlighting the fact that function affinity is plentiful for our deployment. For F3–F5, affinity-based steering virtually eliminates L1-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 L1-I cache. Affinity-based steering allows F3–F5 to execute with zero L1-I misses for 94% of requests.

Despite their high number of L1-I misses in the baseline, F0–F1 benefit only marginally from affinity-based steering because their L1-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 of 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.

7.4 Line-Rate DT Acceleration

The DT component is the most critical component of Cerebros as the majority of the RPC layer's cost is associated with the data transformations inherent in both the header and payload manipulation tasks. DT tasks are also found in workloads other than microservices, such as databases, object stores, and data analytics (Section 3.2.1). Hence, in this section, we focus on evaluating a stand-alone version of our RPC processor's DT component, Optimus Prime (OP), primarily on its ability to transform data at the bandwidths of modern NICs.

We first evaluate a single transformation pipeline (Section 7.4.1), and then move to designs that exploit parallel pipelines to match the NIC bandwidth (Section 7.4.2). Next, we evaluate the impact of time-sharing and its effectiveness in improving pipeline utilization, in cases where the data access latency is the main bottleneck, thus reducing the number of required physical pipelines (Section 7.4.3). Finally, we analyze the power and area cost of various OP configurations using our synthesized RTL (Section 7.4.4).

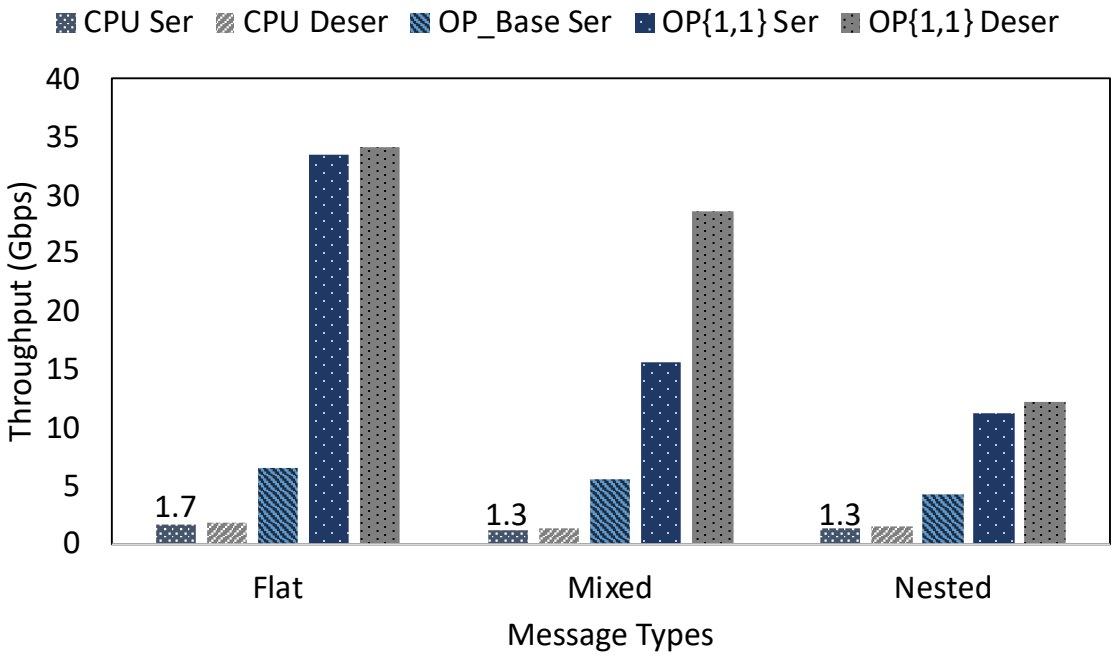


Figure 7.4: Data transformation throughput comparison of a single core with $OP_{\{1,1\}}$.

7.4.1 Single Transformation Pipeline

We first measure the performance of a configuration of OP with a single transformation pipeline ($OP_{\{1,1\}}$) against CPU-centric DT and plot the results in Figure 7.4 for all three object classes. To isolate the improvement from Converter specialization, we also measure a configuration labeled *OP_Base Ser* that disables pipelining and prefetching. Figure 7.4 shows that a CPU core can at best achieve a throughput of ~1.7Gbps for serialization, while OP is ~5× faster. *OP_Base Ser*'s throughput is limited because it spends most of its time waiting for data from the memory hierarchy.

Next we enable pipelining and prefetching; the throughput of this configuration is shown by the $OP_{\{1,1\}}$ Ser and $OP_{\{1,1\}}$ Deser bars. Prefetching and pipelining overlap the latency of transformations and memory accesses, improving throughput by another $2-4\times$. The key enabler for this overlap is our schema, which represents each field as a {type, address} pair, allowing OP to extract field-level parallelism. Such parallelism allows $OP_{\{1,1\}}$ to reduce the average field read latency from 27 cycles to 9 with prefetching. The CPU baseline does not attain this parallelism because the transformations are compiled into serial instruction slices

that are lengthy, and highly control- and data-dependent.

The Nested object class represents the worst case performance for OP with a throughput of ~11Gbps, because every field in this object class is a sub-object. Therefore, each schema field must be read before the sub-object's data can be forwarded to the Converters. Additionally, the prefetcher only operates at the top object level, and therefore it does not overlap accesses further than the schemata of the first level of sub-objects. The Flat object class exhibits ~33Gbps of throughput, because all the schemata and data can be prefetched in parallel. Mixed objects have characteristics of both flat and nested, with OP reaching ~15Gbps.

Deserialization exhibits higher throughput because OP reads already-serialized items from a contiguous buffer, rather than making dependent accesses to the data elements, thus enjoying high spatial locality. However for objects which exhibit more nested fields, the degree of dependent accesses to the schema grows and limits throughput. The bottleneck in $OP_{\{1,1\}}$ is the serial processing of objects by a single transformation pipeline. Given that objects are naturally independent of each other, we now evaluate configurations with multiple pipelines. For brevity's sake, all further experiments only display results for serialization as deserialization has similar performance.

7.4.2 Parallel Transformation Pipelines

Although $OP_{\{1,1\}}$ attains 9-20× higher serialization throughput than a core, there is significant headroom left to attain the 40Gbps sustainable by modern NICs. Next, we measure a scale-up OP by adding transformation pipelines which operate in parallel. Figure 7.5 depicts the serialization throughput for $OP_{\{n,1\}}$ as we vary the number of transformation pipelines.

Flat objects achieve 40Gbps with only two pipelines, benefiting the most because they have the fewest dependent memory accesses. In contrast, Nested objects require four pipelines to achieve 40Gbps, and Mixed objects require three. Throughput increases linearly with up to three pipelines in all cases, because each extra pipeline adds additional independent memory accesses and transformations. Overall, our OP design can easily meet the target NIC

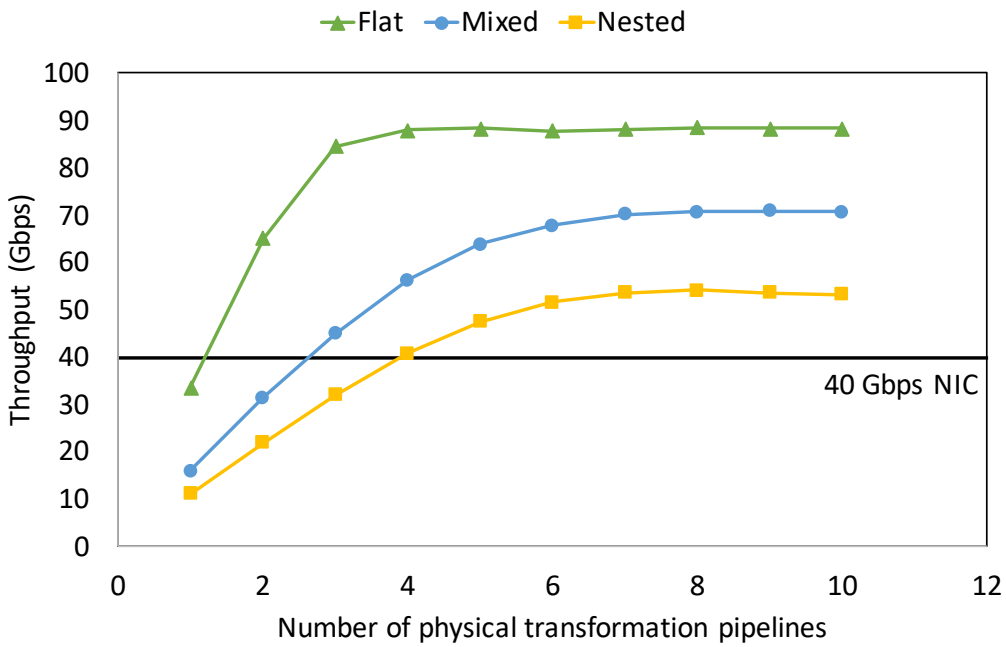


Figure 7.5: Serialization throughput with $OP_{\{n,1\}}$.

bandwidth of 40Gbps for all the three object classes.

The throughput plateaus beyond a certain number of pipelines because they, in aggregate, exhaust the available NoC bandwidth. Flat objects are the most read-efficient (lowest R/W ratio) class of objects, and therefore generate less NoC traffic and higher throughput. In contrast, Nested objects require more reads per write (i.e., they have a greater R/W ratio), thus limiting the OP's serialization throughput to ~50Gbps. We confirm that each configuration has reached the maximum link bandwidth of 512Gbps by summing the bandwidth needed for the schema and object data, the additional NoC header overhead, and other on-chip coherence protocol requests.

When OP saturates the NoC links of the tile it is attached to, the whole NoC has an average link utilization of 16%. Preserving the NoC bandwidth that is available to the core on the contended link would require slightly over-provisioning the NoC's link width. The silicon costs of doing so are negligible, as the per-tile cost of the NoC components has been shown to be less than 1.5% [96].

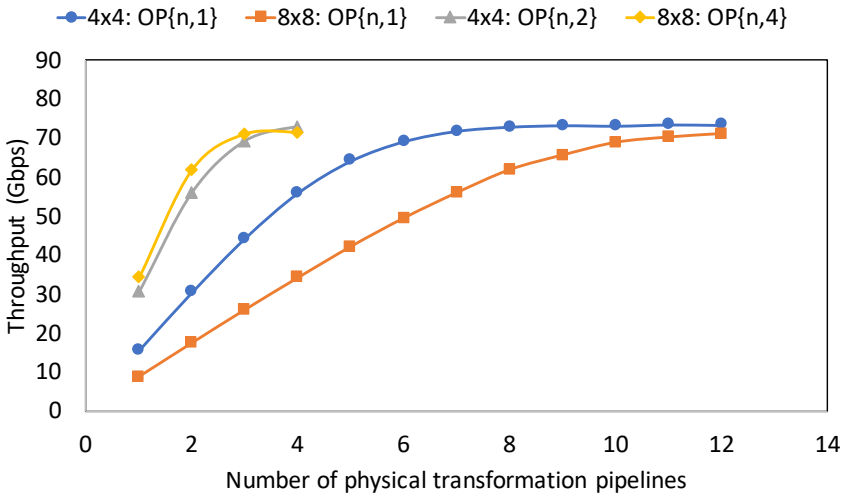
7.4.3 Time-Shared Transformation Pipelines

This section quantifies the benefits of time-sharing transformation pipelines and studies the effects of a larger diameter NoC on OP's performance. Figure 7.6 illustrates the impact of longer average memory access latency (AMAT) on the number of pipelines required to attain peak throughput, by plotting $OP_{\{n,1\}}$'s throughput and latency per 100B for Mixed objects for a 4×4 and an 8×8 mesh. The 8×8 mesh has twice the AMAT of the 4×4 mesh, resulting in half the throughput for an equivalent OP configuration. Doubling the number of pipelines for the case of 8×8 mesh allows OP to recover the original throughput of the 4×4. Following this trend, while the NoC link attached to OP saturates with six pipelines in the case of the 4×4 mesh, we need 12 pipelines to saturate the same link in an 8×8. Beyond this point, increasing the number of pipelines results in elevated latency due to contention for NoC bandwidth.

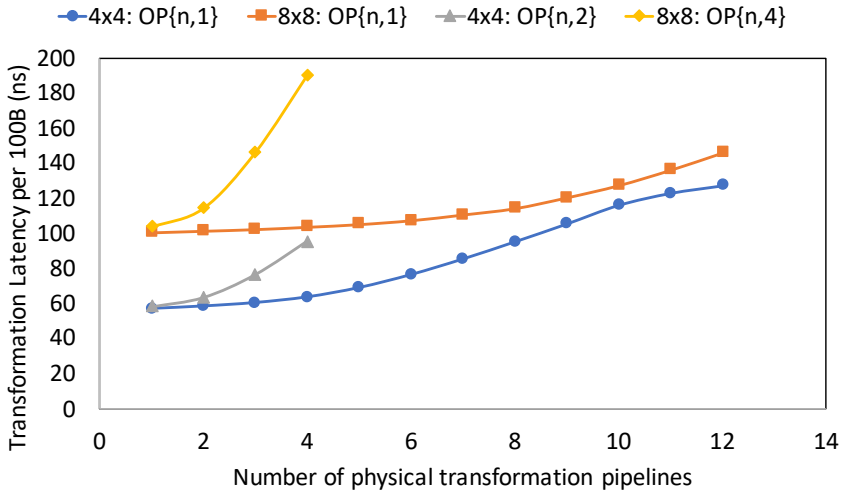
When time-sharing is enabled, OP can continue to issue memory accesses to hide cycles where the pipeline is idle. For instance, with a time-sharing degree of two ($OP_{\{n,2\}}$) we are able to saturate OP's NoC link in the 4×4 mesh with three pipelines as opposed to six in Figure 7.5. The 8×8 mesh benefits more from time sharing due to its larger average latency, and requires a time-sharing degree of four to saturate OP's NoC link with three physical pipelines. Figure 7.6a shows that a time-sharing degree of four, $OP_{\{n,4\}}$, has a nearly identical throughput curve with increasing n as the time-sharing degree of two for a 4×4 mesh. Time-sharing enables OP to achieve roughly the same throughput per pipeline even with different NoC sizes. As long as there is available NoC bandwidth and request-level parallelism, adding more pipelines or time-sharing pipelines increases attainable throughput.

7.4.4 Area and Power Analysis

To model the area and power consumption of OP, we synthesized our RTL design using TSMC 28nm technology, and display the results in Table 7.1. Configured with 12 pipelines, $OP_{\{12,1\}}$ requires $0.45mm^2$ of area, and consumes $532mW$ when operating at $2GHz$. Additionally, each time-shared pipeline requires an enhanced 4-way Reader to switch between different trans-



(a) Transformation bandwidth.



(b) Average Transformation latency per 100B.

Figure 7.6: OP throughput and latency for serialization over Mixed objects comparing different NoC sizes.

	Power [mW]	Throughput [Gbps]	Area [mm^2]	Performance per Watt
CPU	5400	1.3	2.57	1
$OP_{\{1,1\}}$	58	8.8	0.12	655
$OP_{\{12,1\}}$	532	73	0.45	593
$OP_{\{3,4\}}$	152	73	0.19	2075

Table 7.1: Synthesis results for different configurations of OP, compared to the CPU baseline. All throughput numbers are for serializing Mixed objects on the 64-core setup, and all performance per watt numbers are normalized to the CPU.

formation contexts. Such time-shared Readers have ~20% greater area and power overhead compared to normal Readers. Fortunately, time-sharing reduces the number of transformation pipelines required at saturation from 12 to 3. Therefore the $OP_{\{3,4\}}$ configuration achieves the same performance as $OP_{\{12,1\}}$ with a silicon area reduction of 60% and a performance/watt improvement of 3.5×. Compared to a CPU core, $OP_{\{3,4\}}$ achieves 2075× higher performance/watt.

Finally, we compare the area of a shared version of OP to a core-private version (as discussed in Section 6.2.1) and Cerebros. Private OPs do not require a Block Buffer and share the CPU core's TLB, and therefore require $0.03mm^2$ of area. The silicon area of $OP_{\{3,4\}}$ is only ~7% of a single CPU core, whereas having 64 private OPs, one for each core, costs ~75% of a core. This overhead from replication justifies our choice of having a shared accelerator. We estimate the area of Cerebros' additional storage using CACTI, and compare it to those reported for OP. Cerebros' NIC integration eschews the need for a dedicated cache (i.e., OP's Block Buffer) or dedicated control registers for each core, as it uses the NIC's cache and the queue pairs already provided by the transport interface, thus requiring only half of the area that OP requires.

7.5 Impacts of Offload Overhead

We now quantify the performance implications of offload overheads, demonstrating the need for Cerebros to directly interact with the NIC and run the entire RPC layer. The performance

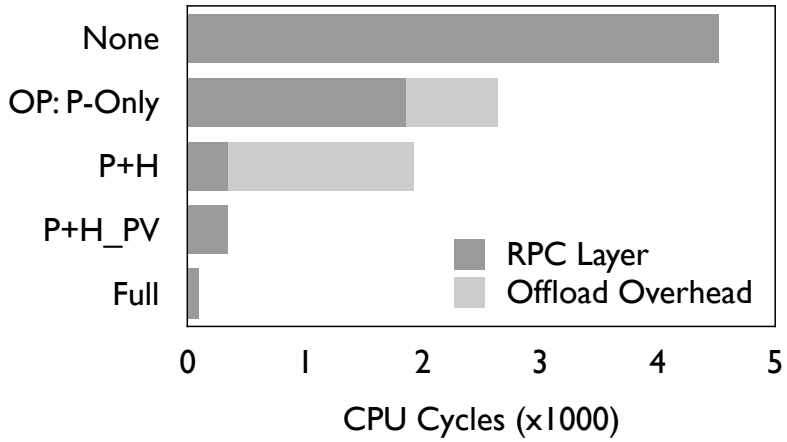


Figure 7.7: RPC layer cycles for various offload options.

improvements from a design that uses the CPU as a coordinator for an RPC accelerator depend on the accelerator's per-module speedup and the overhead of each module offload. In Figure 7.7 we instantiate our analytical model for the following five designs executing the UID microservice: the CPU baseline (None), offloading the RPC's payload to the OP accelerator (OP: P-Only), offloading both the payload and header (P+H), using a private accelerator per core (P+H_PV), and Cerebros that performs the entire RPC layer (Full).

Accelerating payload manipulation (P-Only) leaves the rest of the RPC layer processing to the CPU and only reduces RPC layer cycles by 1.7×. Additionally offloading header manipulation (P+H) frees up 40% of the remaining cycles, bringing the total speedup to 2.3×. In this case, the only remaining part that is executed on the CPU is the dispatch module, which takes only 5% of the RPC layer's cycles. However, the offload overhead grows because the CPU must explicitly request the processing of both manipulation modules independently, forming a lower bound on the performance of the RPC layer. Each request to the UID microservice generates four header manipulation and four payload manipulation tasks, resulting in a total offload overhead of ~1600 cycles, compared to an accelerator processing time of only 90 cycles. For microservices with more nested RPCs (e.g., SG or CP), offload overheads dominate the cost even more overwhelmingly.

A brute-force solution to mitigate offload overheads is to integrate a private accelerator with each CPU core. Figure 7.7's P+H_PV bar shows the performance of this solution, where the

only module remaining that contributes to RPC layer cycles is the dispatch layer. However, such accelerator replication requires 5× more area and 6× more power than a single shared accelerator with a 16-core chip, which is a steep cost for hardware operating at only 1% utilization in the case of UID. These costs grow with higher core counts, making replication an inefficient design choice.

We expect that the best solution to reduce offload overhead without a direct NIC-accelerator interface requires optimizing the accelerator's interface to coalesce its input data with the CPU's requests (similar to RDMA NICs) [97], and using a low-diameter on-chip interconnect such as Multi-drop Express Channels [98] or NOC-Out [99]. The combination of these two techniques can reduce offload overheads by 3.36×, reducing the exposed offload overhead to 60 cycles per module. Therefore, the performance of such an optimized system would fall between P+H and P+H_PV.

Only the solution with a fully capable RPC processor (Full) can simultaneously offload all elements of the RPC layer and avoid silicon over-provisioning. A NIC-interfaced RPC processor cuts expended cycles in the RPC layer by 50× when compared to the CPU baseline, and 21.5× compared to the case where only dispatch is performed on the CPU (P+H). Integrating the RPC processor with the NIC itself allows a 50% reduction in the area of the DT component, as it now shares the NIC's cache and MMU (§4.2).

8 Related Work

In this chapter, we discuss prior work related to the topics of this thesis. We first discuss the software and hardware proposals targeting RPC processing acceleration across the communication stack: transport (Section 8.1.1), the RPC layer itself (Section 8.1.2), and data transformation in particular (Section 8.1.3). Next, we review proposals to reduce CPU-accelerator offload overheads (Section 8.2) and address the instruction supply bottleneck (Section 8.3).

8.1 RPC Processing Acceleration

8.1.1 Accelerating the Transport Layer

The use of RPCs as a datacenter communication API has resulted in a plethora of research to optimize RPC performance. FaRM [100], FASST [101], HERD [102], and NEBULA [25] all study the specifics of how to best implement RPCs over hardware-terminated transports to optimize scalability. In contrast, R2P2 [85] and RPCValet [52] both target improved load balancing. eRPC proposes an RPC API, wire protocol, and threading model which can receive RPCs at the rate of a 40Gb NIC [13]. None of these customized systems use a production RPC layer providing communication among microservices that cross language and data format barriers; therefore, our work is largely orthogonal to all of these systems. Although we chose to implement Cerebros on top of NEBULA due to its integrated NIC, it is possible to integrate

Cerebros with any solution using hardware transport termination. In that case, Cerebros would need to re-implement an interface to the server's memory hierarchy and to the CPU cores, increasing its hardware cost.

8.1.2 Accelerating the RPC Layer

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 [73] 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 [32] software stack. Similarly, NICA proposes a programming model and runtime to accelerate application-level tasks on FPGA-enhanced NICs [74], citing message de-serialization as a potential application. Neither Dagger nor NICA currently supports the underlying modules of production RPC layers we describe in Section 3.2 because they lack support for the header and payload manipulation modules in production Thrift [73, §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 of the object's pointers. Although it may be possible to realize such a design in the future, as argued by Wolnikowski et al. [75] and Raghavan et al. [76], 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 commodity 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 between the FPGA accelerator and the host's CPU cores, and reduce 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.

iPipe proposes a scheduling algorithm to minimize tail latency for microservices offloaded to SmartNICs [103]. Combining affinity-based request steering with a scheduler such as iPipe's would result in a system that can simultaneously improve instruction locality and maintain tail latency.

8.1.3 Accelerating Data Transformation

We focus on Protobuf [61] and Thrift [32] as they are the two state-of-the-art and commonly-used frameworks for generalized cross-language data transformation (DT). Cerebros and Optimus Prime are designed to be a *general-purpose* RPC processor and DT accelerator, respectively. Hence, they are compatible with other frameworks provided they simply create a compatible schema for each object (e.g., through the setter methods). Customized RPC frameworks such as Cap'n Proto [104] or FlatBuffers [105] use a pre-flattened RPC message format, which inherently removes the aforementioned challenges with pointer-based objects. However, this comes at the price of more costly object creation [76], object immutability, less flexibility, and larger wire-format objects. We believe these trade-offs are particularly burdensome for microservices with many nested RPCs because message objects are commonly modified on every nested call. Cerebros can even benefit such frameworks by accelerating header parsing and with affinity-based request steering.

CPU vendors have also realized the difficulties in expressing transformations using existing ISAs. In fact, Intel has already been granted a patent for ISA extensions to x86-64 which provide dedicated support for specific DT operations [106]. The performance impact of ISA extensions would be more or less similar to our specialized Converters, which are specialized pipelines tailored for transformation. However, the serialization of an entire object is still expressed as a serial sequence of many transformation instructions with implicit parallelism. Our work goes further by proposing an entire new abstraction for explicit parallelism between the many fields of a message.

Intel has recently released the specifications for an integrated Data Streaming Accelerator (DSA), supporting operations like data copying, virtual switching, and integrity check-

ing [107]. Although DSA does not currently target general-purpose DT, our key insights would equally apply to DSA, as it follows many of our design choices (e.g., integration into to the virtual memory system). Applying our transformation schema would enable DSA's internal hardware to unlock the field-level parallelism inherent in transformation tasks. SoC designers wishing to perform general-purpose DT with DSA could construct specialized DSA Engines (which are in principle similar to our Converters) for common DT tasks.

Finite automata (FA) processing is an emerging computational model that promises orders of magnitude better performance than CPUs in executing Finite State Machines (FSMs) [108, 109, 110]. FSMs traditionally suffer from complex control flow, limiting the benefits of branch prediction, and irregular memory access patterns, reducing the effectiveness of caching. We observe similar behavior for code generated from DT frameworks. Therefore, FA accelerators for tasks such as pattern matching or replacement could easily be deployed as a type of Converter in our proposed data transformation pipelines. UDP [63] applies the FA model to a coarse-grained class of workloads such as data mining and CSV file parsing. Their architecture is targeted towards bulk loading and cleaning of batches of data, and motivate UDP by comparing CPU processing time to disk I/O. In contrast, our work targets eliminating DT as a bottleneck for latency-critical inter-microservice RPCs.

8.2 Reducing CPU-Accelerator Offload Overhead

Shao et al. have observed that data movement between CPUs and accelerators limits performance, and propose optimizations to pipeline data transfers with computation [111]. Although such techniques could reduce some of the offload overhead we identify in Section 7.5, 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 [112]. Systems such as Morpheus-SSD [113], GPUfs [114], and NVIDIA GPUDirect [115] 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.

Daglis et al. observe that for one-sided RDMA operations in rack-scale computing fabrics, cache coherence interactions can become a bottleneck for end-to-end latency; these interactions are a form of invocation cost. They propose Manycore Network Interfaces [80] to optimize CPU-NIC data transfers. In fact, our NEBULA [25] protocol baseline uses a Manycore Network Interface architecture for its on-chip communication. Our work studies different invocation costs associated with CPU-accelerator offloads for a single RPC request.

Many prior works have developed analytical modeling techniques for studying heterogeneous architectures [86, 116]. We instantiate a similar model to show the offload overheads associated with CPU involvement in the flow of RPCs (Section 7.5).

8.3 Instruction Supply in Servers

A plethora of micro-architectural solutions exist to address instruction supply bottlenecks [117, 118, 119, 120, 121, 122, 123], all of which depend on 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. Offloading the RPC layer to Cerebros benefits these frontend designs, as the RPC layer's code footprint vanishes and fewer capacity misses occur in the prefetcher's storage. Cerebros also goes further by proposing affinity-based request steering, which provides speedups in the case where a microservice's functions are too large to be contained by the CPU's frontend resources.

Profile-guided prefetching proposals, such as AsmDB [124] and I-SPY [125], perform offline analysis on datacenter-wide miss traces, and re-compile the profiled applications with software prefetches. Affinity-based request steering does not require recompilation or datacenter-wide profiling.

9 Concluding Remarks

Deploying and maintaining online services in warehouse-scale computers [2] has become a task so complex that it has changed the best practices for software development and deployment [1, 126, 127, 128, 129]. Instead of single-binary monoliths, datacenter-scale applications are now best constructed as *microservices*, consisting of numerous self-contained modules communicating through Remote Procedure Calls (RPCs) or RESTful APIs [130, 131, 60, 1, 53, 132]. The microservices architecture provides composable software design, with each microservice being responsible for a small subset of the application functionality. Hence, microservices not only simplify and accelerate software development, but also facilitate deployment, scaling and updating individual microservices independently [2, 1, 6]. Moreover, architecting the software in this way enables independent development of each microservice using the programming language and tools best suited to its purpose, and simplifies correctness and performance debugging, as each microservice can be isolated easily [2, 1, 6].

Although decomposing a monolith into microservices implies that each microservice does only a small fraction of the application-level work, the total time spent on inter-microservice RPCs increases in proportion to the number of microservices. The increase in communication to computation ratio creates a challenge to minimize the "tax" associated with each RPC. The importance of communication has resulted in a recent wave of fast evolution in datacenter network infrastructure, optimizing both hardware and software [10, 11, 12, 13, 14, 15, 16, 17,

18, 19, 20, 21, 22, 23]. Although the communication tax includes both the RPC layer and the underlying network stack, recent research has mostly targeted the network stack, leading to a drastic reduction of the networking latency. Hence, the time spent in the RPC layer itself is becoming a significant fraction of the end-to-end cost of invoking a microservice. As the microservices software architecture continues to proliferate, the common RPC layer gluing the microservices together is becoming a bottleneck—the RPC layer itself consumes 40 – 90% of the execution cycles of the microservices we study.

In this thesis, we made the observation that RPC processing is a common yet costly task in modern datacenters and we made a case for hardware-software co-design for rapid and flexible RPC processing. While modern fabrics continue improving network bandwidth, silicon's efficiency and density scaling met an abrupt slowdown with the end of Dennard scaling and the slowdown of Moore's law, putting more pressure on the RPC layer running on the general-purpose CPUs. We analyzed microservices from the DeathStarBench and quantified the RPC layer's cost. We then presented insights on why CPUs are ill-suited to perform the RPC layer functionalities, and why we need an RPC processor to shrink the gap between the CPU and network processing rates and to address the growing cost of RPCs in datacenters. The combination of increasing demands for fast inter-server communication, continuous network bandwidth scaling, and the slowdown of silicon scaling, necessitate hardware-software co-design to transfer functionality from the CPU to an accelerator designed to accelerate the communication software stack.

Motivated by the growing cost of RPCs in datacenters, we presented design principles and constraints guiding the architecture of RPC processing hardware that enables evading the RPC tax in datacenters. Specifically, we showed that an RPC processor must directly receive tasks from the NIC and execute the full RPC layer to completion before the CPU is involved. It should also be integrated with the server's NIC to minimize silicon deployment costs and enable affinity-based request steering, which improves instruction locality. Furthermore, the RPC processor requires a new abstraction called transformation schema, which is a collection of type identifiers and memory addresses, that not only enables parallelism but also

makes the accelerator framework-agnostic. Finally, the RPC processor comprises specialized components handling costly data transformations in a handful of cycles and support a variety of operations defined by the software.

Following our design principles, we implemented Cerebros, a proof-of-concept instance of such an RPC processor, which is NIC-integrated and executes the Apache Thrift RPC layer. We showed that Cerebros can process the RPC layer $37-64\times$ faster than a CPU. Additionally, offloading the RPC layer to Cerebros shrinks the microservice's instruction working set by $27-68\%$, and our novel affinity-based function steering policy provides a further $1.05-2\times$ reduction in execution time for microservices whose functions contend for cache space. Our evaluation using the DeathStarBench microservice suite [1] showed Cerebros can reduce the CPU cycles spent per microservice request by $1.8-14\times$. We believe Cerebros is an ideal candidate for inclusion in future server chips, to support microservices as they decompose into even finer granularity.

9.1 Future Directions

While our RPC processor design assumes the most common and mandatory functionalities of an RPC layer (i.e., header and payload manipulation, and dispatch), there exist additional modules such as compression, encryption, and authentication, that RPC layers may optionally employ. These layers can also be translated into simple data transformation tasks. In our future work, we look for bringing these other layers into our RPC processor design and add support for these optional layers.

Cerebros opens up a new world of possibilities for intelligent request steering. Because Cerebros can peek into the request's header and payload, it can provide more information to the request steering mechanism to decide the optimal request-to-core assignment. Our dispatch policy can now use parameters such as request type and size to decide the optimal request-to-core assignment taking into account data and instruction locality, as well as the current load on each core. The request type can also indicate potential synchronization points

in the application logic; hence, the request-to-core assignment can be adjusted accordingly to reduce the need for costly application-level synchronization. As part of our future work, we plan to explore the various parameters that can be used in a dispatch policy and evaluate different policies and the effects of each parameter for request steering.

RPCs are typically used for one-to-one, synchronous request/response interactions, where the client expects the response to arrive in a timely fashion. However, there are cases where asynchronous communication is preferred, or a request has to be processed by multiple services [28]. In such cases, microservices communicate through message queues or Publish/Subscribe (Pub/Sub) systems such as Amazon SQS [29] and SNS [30], and RabbitMQ [31]. These systems provide a form of asynchronous service-to-service communication and are particularly useful where services follow the publish-subscribe pattern.

At the high level, Pub/Sub systems include intermediary channels, knows as topics, and for each topic, they maintain a list of subscribers to relay messages to. To broadcast a message, the publisher simply pushes a message to a topic. Pub/Sub systems can be used to enable event-driven architectures, or to decouple applications in order to increase performance, reliability and scalability. Many service providers rely on such systems for their event-based computation, real-time analytics, data pipelining, stream processing, and IoT applications.

Pub/Sub systems are also used in the context of the serverless architecture, the native architecture of the cloud. Serverless computing simplifies cloud programming by enabling service providers to offload their operational responsibilities to cloud providers. It eliminates infrastructure management tasks such as deployment, scaling, fault tolerance, monitoring, and system maintenance [1, 133]. The stateless and ephemeral nature of serverless functions and lack of direct addressability of microservices necessitates data to be stored in persistent storage for subsequent functions to operate on it [1, 133, 134].

However, object storage services such as AWS S3 and Google Cloud Storage exhibit high access costs and high access latencies [1, 133]. Pub/Sub systems such as Amazon SQS [29] and SNS [30] can be used as a middle layer (aka rendezvous servers) to relay messages and allow

microservices to communicate. Even though they are faster than object storage services, they still add significant latency, sometimes hundreds of milliseconds [133], limiting the performance of the end-to-end service. Having a message relaying system that operates at the microsecond scale and provides high throughput can significantly improve the performance of the serverless applications [133].

In the same spirit of reducing the communication overhead of microservices, our future work will transcend from RPC layer and direct microservice communication to indirect communication through message queues or Pub/Sub systems. The underlying operations in such systems are fairly similar to the RPC layer, and mostly include data transformation, hashing, table lookup and pointer chasing. By extending our designed RPC processor to fully support the common protocols used in the Pub/Sub systems, we look for proposing a server design that is optimized for both throughput and latency.

Bibliography

[1] 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-XXIV)*, pp. 3–18, 2019. (Cited on pages xiii, 1, 2, 4, 6, 9, 10, 11, 19, 20, 21, 22, 23, 28, 38, 65, 97, 99, and 100)

[2] L. A. Barroso, J. Clidaras, and U. Hölzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*. Synthesis Lectures on Computer Architecture, Morgan & Claypool Publishers, 2013. (Cited on pages 1, 2, 4, 9, 10, 14, 15, 19, 21, 28, and 97)

[3] Internet live stats, "Google search statistics," 2021. (Date last accessed 30-May-2021). (Cited on page 1)

[4] Facebook, "Facebook company info," 2019. (Date last accessed 1-December-2019). (Cited on page 1)

[5] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, no. 2, pp. 74–80, 2013. (Cited on page 1)

[6] 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*, pp. 33:1–33:15, 2018. (Cited on pages 2, 9, 10, 19, 21, 28, and 97)

[7] 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 3, 14, 19, 20, 22, and 66)

[8] Infiniband Trade Association, "Infiniband Roadmap," 2018. (Cited on pages 3, 20, and 25)

[9] The Ethernet Alliance, "The 2018 Ethernet Alliance Roadmap," 2018. (Cited on pages 3, 20, and 25)

[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*, pp. 63–74, 2010. (Cited on pages 3, 22, and 98)

[11] S. Choi, B. Burkov, A. Eckert, T. Fang, S. Kazemkhani, R. Sherwood, Y. Zhang, and H. Zeng, "FBOSS: building switch software at scale," in *Proceedings of the ACM SIGCOMM 2018 Conference*, pp. 342–356, 2018. (Cited on pages 3 and 98)

[12] 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*, pp. 29–42, 2017. (Cited on pages 3, 16, 22, and 98)

[13] A. Kalia, M. Kaminsky, and D. G. Andersen, "Datacenter RPCs can be General and Fast," in *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 1–16, 2019. (Cited on pages 3, 22, 91, and 98)

[14] 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*, pp. 221–235, 2018. (Cited on pages 3, 16, 22, 32, 75, and 98)

[15] F. Ohler, M. C. Beutel, S. Gökay, C. Samsel, and K. Krempels, "A structured approach to support collaborative design, specification and documentation of communication protocols," in *Proceedings of the 13th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE 2018, Funchal, Madeira, Portugal, March 23-24, 2018.*, pp. 367–375, 2018. (Cited on pages 3 and 98)

[16] 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)*, pp. 325–341, 2017. (Cited on pages 3 and 98)

[17] 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*, pp. 51–62, 2009. (Cited on pages 3, 16, 22, and 98)

[18] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, 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," in *Proceedings of the ACM SIGCOMM 2015 Conference*, pp. 183–197, 2015. (Cited on pages 3, 16, 22, and 98)

[19] "Data Plane Development Kit." (Cited on pages 3, 16, 22, and 98)

[20] 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*, pp. 202–215, 2016. (Cited on pages 3, 16, 20, 22, and 98)

[21] 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-XIX)*, pp. 3–18, 2014. (Cited on pages 3, 17, 22, 42, 45, 54, and 98)

[22] 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)*, pp. 7:1–7:13, 2016. (Cited on pages 3, 14, 16, 20, 22, and 98)

[23] S. M. Rumble, D. Ongaro, R. Stutsman, M. Rosenblum, and J. K. Ousterhout, "It's Time for Low Latency," in *Proceedings of The 13th Workshop on Hot Topics in Operating Systems (HotOS-XIII)*, 2011. (Cited on pages 3, 22, and 98)

[24] S. Kanev, J. P. Darago, K. M. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. M. Brooks, "Profiling a Warehouse-Scale Computer," *IEEE Micro*, vol. 36, no. 3, pp. 54–59, 2016. (Cited on pages 3, 4, 14, 19, 20, 24, 29, 38, and 67)

[25] 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)*, pp. 199–212, 2020. (Cited on pages 6, 17, 23, 42, 46, 54, 66, 91, and 95)

[26] A. Pourhabibi, M. Sutherland, A. Daglis, and B. Falsafi, "Cerebros: Evading the RPC tax in datacenters," in *54th IEEE/ACM International Symposium on Microarchitecture (MICRO 2021), 16-20 October 2021, Athens, Greece*, 2021. (Cited on page 7)

[27] A. Pourhabibi, 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-XXV)*, pp. 1203–1216, 2020. (Cited on page 7)

[28] Chris Richardson, "Building Microservices: Inter-Process Communication in a Microservices Architecture," 2015. (Date last accessed 1-Sept-2019). (Cited on pages 12 and 100)

[29] "Amazon Simple Queue Service." (Date last accessed 1-Sept-2019). (Cited on pages 13 and 100)

[30] "Amazon Simple Notification Service." (Date last accessed 1-Sept-2019). (Cited on pages 13 and 100)

[31] "RabbitMQ." (Date last accessed 1-Sept-2019). (Cited on pages 13 and 100)

[32] Apache Software Foundation, "Thrift." (Cited on pages 13, 23, 26, 29, 66, 92, and 93)

[33] Facebook Inc., "Facebook Thrift." (Cited on page 13)

[34] P. Grosu, M. Rehman, E. Anderson, V. Pai, and H. Miller, "gRPC." (Cited on pages 13, 26, and 29)

[35] A. Daglis, "Network-compute co-design for distributed in-memory computing," p. 230, 2018. (Cited on pages 14, 20, and 55)

[36] L. A. Barroso, J. Dean, and U. Hölzle, "Web Search for a Planet: The Google Cluster Architecture," *IEEE Micro*, vol. 23, no. 2, pp. 22–28, 2003. (Cited on page 14)

[37] 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)*, pp. 1–12, 2017. (Cited on page 14)

[38] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. R. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and

D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proceedings of the 41st International Symposium on Computer Architecture (ISCA)*, pp. 13–24, 2014. (Cited on page 14)

[39] Intel Corp., "Intel® Optane™ Technology for Data Centers." (Cited on page 15)

[40] Samsung Electronics Co., Ltd., "Ultra-Low Latency with Samsung Z-NAND SSD," 2017. (Cited on page 15)

[41] 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*, pp. 123–137, 2015. (Cited on pages 16, 32, and 75)

[42] Alexey Andreyev, "Introducing data center fabric, the next-generation Facebook data center network," 2014. (Cited on page 16)

[43] M. Technologies, "Mellanox 200Gb/s ConnectX-6 Ethernet Single/Dual Port Adapter IC." https://www.mellanox.com/products/ethernet-adapter-ic/connectx-6-en-ic, 2020. (Cited on page 16)

[44] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, "IX: A Protected Dataplane Operating System for High Throughput and Low Latency," in *Proceedings of the 11th Symposium on Operating System Design and Implementation (OSDI)*, pp. 49–65, 2014. (Cited on pages 16 and 69)

[45] 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 page 16)

[46] "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. (Cited on page 16)

[47] 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)*, pp. 29–42, 2018. (Cited on pages 17 and 45)

[48] 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 System Design and Implementation (OSDI)*, pp. 239–256, 2021. (Cited on pages 17 and 45)

[49] T. Halfhill, "Oracle shrinks sparc m7," *Linley Group Microprocessor Report*, September 2015. (Cited on pages 17 and 45)

[50] B. Wheeler, "Calxeda spins 4w server-on-a-chip," *Linley Group Microprocessor Report*, November 2011. (Cited on pages 17 and 45)

[51] "Intel xeon processor d-1500 product family." https://cdrdv2.intel.com/v1/dl/getcontent/333423, 2016. (Date retrieved: 6 March 2020). (Cited on pages 17 and 45)

[52] 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-XXIV)*, pp. 35–48, 2019. (Cited on pages 17, 46, 63, 64, and 91)

[53] A. Sriraman and T. F. Wenisch, "μtune: Auto-tuned threading for OLDI microservices," in *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018.*, pp. 177–194, 2018. (Cited on pages 19 and 97)

[54] 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)*, pp. 365–376, 2011. (Cited on page 19)

[55] David Brooks, "What's the future of technology scaling?," 2018. (Cited on page 19)

[56] 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)*, pp. 51–66, 2018. (Cited on pages 19 and 20)

[57] E. Parallel Systems Architecture Lab (PARSA), "Qflex," Mar. 2020. (Cited on pages 24, 67, and 75)

[58] B. Wheeler, "Ryzen 5000 Rides Zen 3 to the Top," *Linley Group Microprocessor Report*, November 2020. (Cited on page 25)

[59] T. Halfhill, "Ice Lake Debuts 10nm, New Cores," *Linley Group Microprocessor Report*, August 2019. (Cited on page 25)

[60] Y. Gan and C. Delimitrou, "The architectural implications of cloud microservices," *Computer Architecture Letters*, vol. 17, no. 2, pp. 155–158, 2018. (Cited on pages 26 and 97)

[61] Google, "Protocol Buffers." (Cited on pages 29 and 93)

[62] The University of Utah, "CloudLab Hardware." Retrieved 15-Jan-2020. (Cited on page 32)

[63] Y. Fang, C. Zou, A. J. Elmore, and A. A. Chien, "UDP: a programmable accelerator for extract-transform-load workloads and more," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 55–68, 2017. (Cited on pages 33 and 94)

[64] D. N. Pnevmatikatos and G. S. Sohi, "Guarded Executing and Branch Prediction in Dynamic ILP Processors," in *Proceedings of the 21st International Symposium on Computer Architecture (ISCA)*, pp. 120–129, 1994. (Cited on page 33)

[65] S. A. Mahlke, R. E. Hank, J. E. McCormick, D. I. August, and W. mei W. Hwu, "A Comparison of Full and Partial Predicated Execution Support for ILP Processors," in *Proceedings of the 22nd International Symposium on Computer Architecture (ISCA)*, pp. 138–150, 1995. (Cited on page 33)

[66] T. David, R. Guerraoui, and V. Trigonakis, "Everything you always wanted to know about synchronization but were afraid to ask," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 33–48, 2013. (Cited on page 34)

[67] T. Blackwell, "Speeding up Protocols for Small Messages," in *Proceedings of the ACM SIGCOMM 1996 Conference*, pp. 85–95, 1996. (Cited on page 35)

[68] M. Welsh, D. E. Culler, and E. A. Brewer, "SEDA: An Architecture for Well-Conditioned, Scalable Internet Services," in *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 230–243, 2001. (Cited on pages 35 and 37)

[69] J. R. Larus and M. Parkes, "Using Cohort-Scheduling to Enhance Server Performance," in *USENIX Annual Technical Conference*, pp. 103–114, 2002. (Cited on pages 35 and 37)

[70] S. Harizopoulos and A. Ailamaki, "A Case for Staged Database Systems," in *Proceedings of the 1st Biennial Conference on Innovative Data Systems Research (CIDR)*, 2003. (Cited on pages 35 and 37)

[71] 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)*, pp. 342–358, 2017. (Cited on page 36)

[72] 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)*, pp. 170–180, 1997. (Cited on page 37)

[73] 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 *ASPLOS 2021, To appear.*, pp. 36–51, 2021. (Cited on pages 38 and 92)

[74] 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)*, pp. 345–362, 2019. (Cited on pages 38 and 92)

[75] 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)*, pp. 206–212, 2021. (Cited on pages 38 and 92)

[76] 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)*, pp. 199–205, 2021. (Cited on pages 38, 92, and 93)

[77] 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 44 and 55)

[78] J. Coleman, "Reducing Interrupt Latency Through the Use of Message Signaled Interrupts," 2009. (Cited on page 44)

[79] S. Li, K. T. Lim, P. Faraboschi, J. Chang, P. Ranganathan, and N. P. Jouppi, "System-level integrated server architectures for scale-out datacenters," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 260–271, 2011. (Cited on page 45)

[80] 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)*, pp. 567–579, 2015. (Cited on pages 46, 54, and 95)

[81] Intel, "Introduction to intel ethernet flow director and memcached performance," 2014. (Cited on pages 46 and 63)

[82] J. E. Smith, "Decoupled Access/Execute Computer Architectures," *ACM Trans. Comput. Syst.*, vol. 2, no. 4, pp. 289–308, 1984. (Cited on pages 49, 59, and 71)

[83] Y. O. Koçberber, B. Grot, J. Picorel, B. Falsafi, K. T. Lim, and P. Ranganathan, "Meet the walkers: accelerating index traversals for in-memory databases," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 468–479, 2013. (Cited on page 51)

[84] 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)*, pp. 322–334, 2020. (Cited on page 58)

[85] 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)*, pp. 863–880, 2019. (Cited on pages 64 and 91)

[86] 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-XXV)*, pp. 733–750, 2020. (Cited on pages 66 and 95)

[87] 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 System Design and Implementation (OSDI)*, pp. 145–160, 2018. (Cited on page 66)

[88] Google, "C++ Arena Allocation Guide ." (Cited on page 70)

[89] 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*, pp. 327–341, 2018. (Cited on page 71)

[90] Amazon Web Services, Inc., "Amazon EC2 Instance Types." (Cited on page 73)

[91] M. Nemirovsky and D. M. Tullsen, *Multithreading Architecture*. Synthesis Lectures on Computer Architecture, Morgan & Claypool Publishers, 2013. (Cited on page 74)

[92] Synopsys, "Synopsys Design Compiler." (Cited on page 75)

[93] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 3–14, 2007. (Cited on page 75)

[94] A. Pahlevan, J. Picorel, A. Pourhabibi, D. Rossi, M. Zapater, A. Bartolini, P. G. D. Valle, D. Atienza, L. Benini, and B. Falsafi, "Towards near-threshold server processors," in *Proceedings of the 2016 Design, Automation, and Test in Europe Conference and Exhibition (DATE)*, pp. 7–12, 2016. (Cited on page 75)

[95] 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 page 75)

[96] M. McKeown, A. Lavrov, M. Shahrad, P. J. Jackson, Y. Fu, J. Balkind, T. M. Nguyen, K. Lim, Y. Zhou, and D. Wentzlaff, "Power and Energy Characterization of an Open Source 25-Core Manycore Processor," in *Proceedings of the 24th IEEE Symposium on High-Performance Computer Architecture (HPCA)*, pp. 762–775, 2018. (Cited on page 85)

[97] A. Kalia, M. Kaminsky, and D. G. Andersen, "Design Guidelines for High Performance RDMA Systems," in *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*, pp. 437–450, 2016. (Cited on page 90)

[98] B. Grot, J. Hestness, S. W. Keckler, and O. Mutlu, "Express Cube Topologies for on-Chip Interconnects," in *Proceedings of the 15th IEEE Symposium on High-Performance Computer Architecture (HPCA)*, pp. 163–174, 2009. (Cited on page 90)

[99] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, Y. O. Koçberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Özer, and B. Falsafi, "Scale-out processors," in *Proceedings of*

the 39th International Symposium on Computer Architecture (ISCA), pp. 500–511, 2012. (Cited on page 90)

[100] 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)*, pp. 401–414, 2014. (Cited on page 91)

[101] A. Kalia, M. Kaminsky, and D. G. Andersen, "FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs," in *Proceedings of the 12th Symposium on Operating System Design and Implementation (OSDI)*, pp. 185–201, 2016. (Cited on page 91)

[102] A. Kalia, M. Kaminsky, and D. G. Andersen, "Using RDMA efficiently for key-value services," in *Proceedings of the ACM SIGCOMM 2014 Conference*, pp. 295–306, 2014. (Cited on page 91)

[103] 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*, pp. 318–333, 2019. (Cited on page 93)

[104] Kenton Varda, Sandstorm.io, "Cap'n Proto." (Cited on page 93)

[105] Google, "FlatBuffers." (Cited on page 93)

[106] J. D. Guilford and V. Gopal, "Instruction set for variable length integer coding," 2016. (Cited on page 93)

[107] Intel Corp., "Intel Data Streaming Accelerator Preliminary Architecture Specification," 2019. (Cited on page 94)

[108] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes, "An Efficient and Scalable Semiconductor Architecture for Parallel Automata Processing," *IEEE Trans. Parallel Distributed Syst.*, vol. 25, no. 12, pp. 3088–3098, 2014. (Cited on page 94)

[109] V. Gogte, A. Kolli, M. J. Cafarella, L. D'Antoni, and T. F. Wenisch, "HARE: Hardware accelerator for regular expressions," in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 44:1–44:12, 2016. (Cited on page 94)

[110] A. Subramaniyan and R. Das, "Parallel Automata Processor," in *Proceedings of the 44th International Symposium on Computer Architecture (ISCA)*, pp. 600–612, 2017. (Cited on page 94)

[111] 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)*, pp. 48:1–48:12, 2016. (Cited on page 94)

[112] 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)*, pp. 617–632, 2019. (Cited on page 94)

[113] 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)*, pp. 53–65, 2016. (Cited on page 94)

[114] 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-XVIII)*, pp. 485–498, 2013. (Cited on page 94)

[115] N. Corp., "Developing a Linux Kernel Module using GPUDirect RDMA," 2020. (Cited on page 94)

[116] M. S. B. Altaf and D. A. Wood, "LogCA: A High-Level Performance Model for Hardware Accelerators," in *Proceedings of the 44th International Symposium on Computer Architecture (ISCA)*, pp. 375–388, 2017. (Cited on page 95)

[117] 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)*, pp. 1–10, 2008. (Cited on page 95)

[118] 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)*, pp. 272–283, 2013. (Cited on page 95)

[119] C. Kaynak, B. Grot, and B. Falsafi, "Confluence: unified instruction supply for scale-out servers," in *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 166–177, 2015. (Cited on page 95)

[120] 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)*, pp. 16–27, 1999. (Cited on page 95)

[121] 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)*, pp. 493–504, 2017. (Cited on page 95)

[122] 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-XXIII)*, pp. 30–42, 2018. (Cited on page 95)

[123] 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)*, pp. 65–78, 2020. (Cited on page 95)

[124] 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)*, pp. 462–473, 2019. (Cited on page 95)

[125] 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)*, pp. 146–159, 2020. (Cited on page 95)

[126] T. Hoff, "Lessons learned from scaling Uber to 2000 engineers, 1000 services, and 8000 Git repositories," 2016. (Cited on page 97)

[127] S. Kramer, "The Biggest Thing Amazon Got Right: The Platform," 2011. (Cited on page 97)

[128] T. Mauro, "Adopting microservices at Netflix: Lessons for architectural design," 2015. (Cited on page 97)

[129] A. Schaffer, "Testing of microservices," 2018. (Cited on page 97)

[130] 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)*, pp. 49–60, 2013. (Cited on page 97)

[131] A. Cockcroft, "Microservices the good bad and the ugly," 2015. (Cited on page 97)

[132] H. Zhou, M. Chen, Q. Lin, Y. Wang, X. She, S. Liu, R. Gu, B. C. Ooi, and J. Yang, "Overload control for scaling wechat microservices," in *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2018,Carlsbad, CA, USA, October 11-13, 2018*, pp. 149–161, 2018. (Cited on page 97)

[133] E. Jonas, J. Schleier-Smith, V. Sreekanti, C. che Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. J. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson, "Cloud Programming Simplified: A Berkeley View on Serverless Computing," *CoRR*, vol. abs/1902.03383, 2019. (Cited on pages 100 and 101)

[134] S. Fouladi, R. S. Wahby, B. Shacklett, K. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein, "Encoding, Fast and Slow: Low-Latency Video Pro-

cessing Using Thousands of Tiny Threads," in *Proceedings of the 14th Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 363–376, 2017. (Cited on page 100)

Arash Pourhabibi-Zarandi

Computer Science PhD Candidate
Swiss Federal Institute of Technology in Lausanne (EPFL)

Contact Info

Address: EPFL IC IINFCOM PARSA, INJ 238, Station 14, 1015 Lausanne, Switzerland
Phone: +41 21 693 13 79
Email: arash.pourhabibi@epfl.ch
Website: http://arash.pourhabibi.info

Interests

I am broadly interested in the field of computer systems and interdisciplinary systems-level problems found in modern, large-scale datacenters, from cloud services to data stores and all the way down to server systems. By characterizing modern datacenter applications, I look for maximizing the compute density of server systems and minimizing their energy footprint through specialization of various system components and better system integration. My current focus is on the evasion of the RPC tax in datacenters through hardware-software co-design.

Education

2015-2021 | Ph.D. in Computer & Communication Sciences
Swiss Federal Institute of Technology in Lausanne (EPFL), Lausanne, Switzerland
Doctoral Research Assistant at PARSA under supervision of Prof. Babak Falsafi
Thesis: Hardware-software Co-design to Evade the RPC Tax in Datacenters
Related Courses: Advanced Multiprocessor Architecture, Topics on Datacenter Design, Understanding Datacenter Software Dynamics

2013-2015 | M.Sc. in Computer Engineering (Software Engineering)
Shiraz University, Shiraz, Iran
Thesis: Design & Implementation of a Scheme for Big Data Processing on GPU
Advisor: Dr. Farshad Khunjush
Ranked First: Achieving the highest course GPA (19.84/20) among all M.Sc. students
Related Courses: Advanced OS, Advanced Computer Architecture, Multicore Programming, Parallel Algorithms, Grid Computing, Software Architecture, Text Mining

2009-2013 | B.Sc. in Computer Engineering (Software Engineering)
Shiraz University, Shiraz, Iran
Ranked First: Achieved the highest GPA in CS courses (18.88/20) among all B.Sc. students

Work Experience

Current, Sep. 2015 | Doctoral Research Assistant
PARSA Lab, EPFL, Lausanne, Switzerland
Contributed to several research projects focused on hardware and software co-design for future generations of datacenter servers. Supervised junior students and summer interns. Member of the CloudSuite team, and part of the core team responsible for its 3rd release. Member of the Flexus simulator maintenance team, and part of the core team working on its new incarnation branded as QFlex.

2010-2013 | Member of the IT Task Force
CS Department, Shiraz University, Iran
In charge of the maintenance of department's network infrastructure and IT services. Proposed and implemented new services for the department such as a CMS.

2011-2013 | Freelance Java and iOS Developer
Involved in development of a payment switch system (in Java). Co-founded the Mobile Programming group at Shiraz University and developed a bill payment app for iPhone.

2010-2011 | Intern at Shiraz University's CERT Center *(ShirazAPA)*
Involved in research and development of several security-related projects such a secure update manager.

Technical Skills

Programming: Python, C/C++, Java, PThreads, OpenMP, CUDA, MPI
Operating Systems: macOS, Linux, Windows
Miscellaneous: Git, Docker, LaTeX, Shell Scripting, Agile Development
Basic Familiarity: Objective-C, Ruby, PHP, HTML, JavaScript

Publications & Patents

1. **Cerebros: Evading the RPC Tax in Datacenters.** A. Pourhabibi, M. Sutherland, A. Daglis, B. Falsafi. *In Proceedings of the 54th IEEE/ACM International Symposium on Microarchitecture, MICRO'21*, Athens, Greece, October 2021.
2. **Equinox: Training (for Free) on a Custom Inference Accelerator.** M. Drumond, L. Coulon, A. Pourhabibi, A. Yüzügüler, B. Falsafi, M. Jaggi. *In Proceedings of the 54th IEEE/ACM International Symposium on Microarchitecture, MICRO'21*, Athens, Greece, October 2021.
3. **Data Transformer Apparatus.** A. Pourhabibi, S. Gupta, H. Kassir, M. Sutherland, Z. Tian, M. Drumond, B. Falsafi, C. Koch. *Patent (Pending)*, March 2021.
4. **Optimus Prime: Accelerating Data Transformation in Servers.** A. Pourhabibi, S. Gupta, H. Kassir, M. Sutherland, Z. Tian, M. Drumond, B. Falsafi, C. Koch. *In Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'20*, Lausanne, Switzerland, March 2020.
5. **Towards near-threshold server processors.** A. Pahlevan, J. Picorel, A. P. Zarandi, D. Rossi, M. Zapater, A. Bartolini, P. G. Del Valle, D. Atienza,L. Benini, and B. Falsafi. *In Proceedings of the 2016 Conference on Design, Automation & Test in Europe (DATE)*, Dresden, Germany, March 2016.
6. The official Persian translation of "Engineering SaaS: An Agile Approach Using Cloud Computing" written by Armando Fox and David Patterson. *Currently under preparation.*

Awards & Honors

Dec. 2018	Awarded the **Teaching Assistant Award** for teaching excellence School of Computer & Communication Sciences (IC), EPFL, Lausanne, Switzerland
Summer 2014	Ranked 2nd at the first Iran Programming Skill Challenge (Java section) Held by Sharif University of Technology and Tehran University's Faculty of Entrepreneurship, Iran
Sep. 2013	Honorary admission to the M.Sc. program without university entrance exam Shiraz University, Shiraz, Iran
June 2013	Awarded as the **Best Undergraduate Student** in Computer Engineering Shiraz University, Shiraz, Iran
Feb. 2012	Nominated for the **Best Mobile Application** for SAHA (payment app for iPhone) The First Iran Mobile Innovation Awards, held by Sharif University of Technology, Tehran, Iran

Professional & Extracurricular Services

ASPLOS 2020	Artifact Evaluation Committee Member
Dec. 2014	Co-organizer of an Hour of Code Event Organized a one-day workshop, participating in the Hour of Code program, for tens of high-school and middle-school students and taught them the basics of computer programming and algorithmic thinking.
2010-2014	BreakTime In University (BTiU) BTiU is a three-day annual conference consists of tens of parallel workshops held by a group of university students during the summer at Shiraz University. Hundreds of talented high-schoolers attend this event to learn more about various study majors, practice teamwork, life and social skills, and learn how to be creative and innovative. I had the chance to be a part of the organizing team for five years.
May 2012	Member of Conference Organizing Committee Internet and technical services assistant at the 16^{th} CSI International Symposiums on Computer Architecture & Digital Systems (CADS 2012) and Artificial Intelligence & Signal Processing (AISP 2012) held at Shiraz University, Shiraz, Iran.

Languages

Persian:	Native Proficiency
English:	Full Professional Proficiency
French:	Elementary Proficiency

Open Source Contributions

2015-2021 | **CloudSuite**

CloudSuite is a benchmark suite of cloud services. The benchmarks are based on real-world software stacks and represent real-world setups. It is one of the early benchmark suites that is representative of modern datacenter services and is included in Google's PerfKit Benchmarker. It has become an industry standard and been used to drive the design of modern datacenter-oriented CPUs, such as Cavium ThunderX. I have been a core member of the team responsible for the maintenance and the third release of CloudSuite, which is a major enhancement over prior releases both in benchmarks and infrastructure.

2016-2021 | **QFlex**

The QFlex project targets quick, accurate, and flexible simulation of multi-node computer systems proceeding along four fronts: QEMU, a popular open-source full-system machine emulator, Flexus, a powerful and flexible simulation framework that enables detailed cycle-accurate simulation, SMARTS, which applies rigorous statistical sampling theory to reduce the simulation time while achieving high accuracy, and NS-3, a popular and flexible network simulation stack. I have been a member of the Flexus maintenance team and the team responsible for its new incarnation branded as QFlex.

Teaching Experience

Fall 2019, Fall 2018, Fall 2017, Fall 2016 | **Introduction to Multiprocessor Architecture**

Assisted in redesigning the course: constructed a new syllabus and prepared course material including lecture slides, exercises, programming assignments, and exams. Graded assignments and exams. Led weekly lab sessions and guided students. | EPFL

Spring 2020, Spring 2019 | **System Oriented Programming**

Assisted in redesigning the course: constructed a new course project and weekly tasks for students. Created a grading and feedback infrastructure for the weekly tasks, graded the final project and led weekly lab sessions. | EPFL

Spring 2018 | **Systems for Data Science**

Graded students' assignments, projects and exams, held lab sessions and gave guidance to students for their projects. | EPFL

Spring 2017 | **Programming II (Using C++)**

Graded students' programming assignments, projects and exams. Led weekly lab sessions and gave guidance to students. | EPFL

Spring 2014 | **Grid Computing**

Graded students' programming assignments and projects. | Shiraz University

Spring 2014 | **Software Architecture**

Graded students' programming assignments and projects. | Shiraz University

Spring 2014, Spring 2013 | **Database Laboratory**

Completely redesigned the course from scratch. Constructed the syllabus and prepared the course material. Led weekly lab sessions, gave guidance to students and graded their assignments and projects. | Shiraz University

Fall 2013 | **GPU Programming**

Prepared and graded students' programming assignments and projects, led weekly lab sessions and gave guidance to students. | Shiraz University

Spring 2013 | **Design & Implementation of Programming Languages**

Prepared students' programming assignments and projects. | Shiraz University

Fall 2012, Fall 2010 | **Fundamentals of Computer and Programming Using Python**

Constructed the syllabus and prepared the course material (programming assignments, labs, and projects). Led weekly lab sessions, gave guidance to students and graded their assignments and projects. | Shiraz University

Spring 2012, Spring 2011 | **Principles of Programming Using C**

Prepared and graded students' programming assignments and projects, led weekly lab sessions and gave guidance to students. | Shiraz University

Fall 2011 | **Advanced Programming Using Java**

Prepared and graded students' programming assignments and projects, led weekly lab sessions and gave guidance to students. | Shiraz University