TECHNISCHE UNIVERSITÄT DARMSTADT

Holistic Runtime Scheduling for the Distributed Computing Landscape

Vom Fachbereich Informatik
der Technischen Universität Darmstadt
genehmigte

Dissertation

zur Erlangung des akademischen Grades
Doktor-Ingenieur (Dr.-Ing.)

vorgelegt von
Marcel Blöcher
geboren in Gießen

Tag der Einreichung: 01. Februar 2021
Tag der Disputation: 17. März 2021

Referenten
Prof. Dr. Max Mühlhäuser, TU Darmstadt
Prof. Dr. Patrick Eugster, USI, Schweiz
Prof. Dr. Lin Wang, VU Amsterdam, Niederlande

Holistic Runtime Scheduling for the Distributed Computing Landscape

Genehmigte Dissertation von Marcel Blöcher aus Gießen
Tag der Einreichung: 01. Februar 2021
Tag der Disputation: 17. März 2021

Jahr der Veröffentlichung der Dissertation auf TUprints: 2021
Darmstadt - D 17

1. Gutachten: Prof. Dr. Max Mühlhäuser, TU Darmstadt
2. Gutachten: Prof. Dr. Patrick Eugster, USI, Schweiz
3. Gutachten: Prof. Dr. Lin Wang, VU Amsterdam, Niederlande

To my friends and family who made this work possible.

Abstract

Internet services have become an indispensable part of our lives, with billions of users on a daily basis. Example use cases include services for real-time communication and collaborative editing of documents. Furthermore, there are many hidden—nonetheless omnipresent—use cases like cashier systems and sensors of industry facilities. Users expect to use Internet services at any time at low cost with the desired service quality despite potential load spikes a service might face. A straightforward strategy to provide services with high availability is to allocate dedicated resources for each service. In turn, this strategy is likely to lead to over-provisioning and increased operating expenses, which contradicts offering services at a low price. A solution to this problem is to leverage resource scheduling to share the underlying resources among many different workloads and services. Sharing the underlying resources is a key enabler to offer highly scalable services while keeping operating expenses of each service low.

A wide range of resource scheduling systems for the distributed computing landscape has been proposed in the past, covering the application and infrastructure levels. Application-level scheduling focuses on problems such as given a set of resources, configuring an application to reach high throughput and good service quality. Many application-level resource scheduling systems lack support for *runtime* scheduling, often due to slow or unsuitable algorithms. Without *runtime* scheduling, resource scheduling must run in advance for many scenarios and, at best, repeats periodically to update scheduling decisions. This is likely to result in inefficient resource usage. In contrast to application-level scheduling, infrastructure-level scheduling is about orchestrating resources and serving resource requests of various applications, aiming at high resource utilization. Infrastructure-level scheduling leverages *generic resource* abstractions, *e.g.*, containers and virtual machines, to fulfill these properties. These abstractions make assumptions (*e.g.*, homogeneity, linear resource consumption) to simplify management, but ignore the fact that current distributed computing systems have been evolving in the post-Moore's law era and many of these assumptions need to be revised. In particular, the recent trend of new programmable networking devices, ushering in a new area of in-network computing (INC), overtaxes the generic abstractions of compute containers running on servers. The ever-growing demand for Internet services in general and increasingly heterogeneous resources combined with the highly varying demand in particular, require *runtime* solutions for *holistic* resource scheduling, covering both the application and infrastructure levels.

This dissertation presents four novel solutions to *holistic runtime* scheduling for the distributed computing landscape. Two solutions cover the application and two the infrastructure levels. We start with an analysis of the field of resource scheduling for the distributed computing landscape and classify involved systems, resources, and abstractions. Based on this, we present a classification of INC which helps to understand the design space of INC resource scheduling. Next, we discuss two scenarios at the application level and demonstrate how runtime scheduling improves resource efficiency. As a first scenario, we consider big data aggregation systems and present ROME, a middleware system to reduce the total aggregation time. ROME automatically analyzes at runtime the involved aggregation function's data stream and optimizes each node's responsibilities in the aggregation plan. ROME reduces total aggregation time even compared with manually fine-tuned systems. The second scenario discusses resource scheduling of distributed service function chains. We present STEAM, the first distributed runtime scheduler for this problem, that operates at packet-level granularity without requiring *a priori* information of traffic estimates and a global view of the systems. Compared with non-runtime solutions to this problem, STEAM achieves better service quality when using the same resources and reduces the amount of resources required to serve the same load.

For the data center infrastructure level, we present two mutually exclusive solutions. Our first solution is IncSched, a system that retrofits existing data center resource schedulers for INC. Based on the proposed classification of INC, IncSched presents a new resource model, translates resource requests to be compliant with the plugged retrofitted scheduler, and holds the logic for managing INC resources. IncSched makes existing resource schedulers compatible with INC for the first time, contributing to a broad acceptance of INC. For a holistic integration of INC in data center resource scheduling, we propose HIRE, a full-fledged resource scheduling solution for INC. HIRE extends the resource model of IncSched for automatic augmentation of resource alternatives and incorporates non-linearity property of INC resource usage. HIRE is the first scheduler that combines all server and INC resources in the same scheduling problem to attribute interdependencies on data center level. These novelties make HIRE more successful in satisfying resource requests with INC, finding better placements concerning locality, and reducing tail latencies. We evaluate all solutions using extensive simulations, and for some also using system prototypes and integrated benchmarks.

In summary, this dissertation proposes four novel solutions for holistic runtime resource scheduling. The contributions foster the importance of *runtime* resource scheduling for more efficient resource usage. Our contributions to *holistic* resource scheduling make shared INC available on a data center level for the first time.

Zusammenfassung

Internetdienste sind zu einem unverzichtbaren Bestandteil unseres Lebens geworden. Milliarden von Benutzern nutzen täglich Internetdienste, zum Beispiel zur Echtzeit-Kommunikation mit Video und kollaborativen Arbeit an Dokumenten. Darüber hinaus gibt es auch viele allgegenwärtige—teils versteckte—Internetdienste wie Kassensysteme für Restaurants und Sensoren in Industrieanlagen. Von Internetdiensten wird erwartet, dass diese stets in gewünschter Qualität verfügbar sind, auch bei hohem Nutzeraufkommen, und zu niedrigen Kosten genutzt werden können. Eine direkte Herangehensweise, um hohe Erreichbarkeit sicherzustellen, besteht in der Bereitstellung dedizierter Infrastruktur für jeden Dienst. Dieser Lösungsansatz führt wiederum sehr wahrscheinlich zu Überkapazität und ist mit dem Ziel niedriger Kosten praktisch schwer vereinbar. Eine alternative Lösung besteht darin, die zugrunde liegende Infrastruktur mittels Ressourcenplanung unter vielen verschiedenen Diensten zu teilen. Durch die gemeinsame Nutzung der Ressourcen wird es möglich, Dienste hochgradig skalierbar anzubieten, bei gleichzeitig niedrigen Betriebskosten für jeden einzelnen Dienst.

Eine breite Palette an Systemen zur Ressourcenplanung ist bereits in der verteilten Rechenlandschaft im Einsatz, sowohl auf Anwendungsebene als auch auf Infrastrukturebene. Die Systeme zur Ressourcenplanung haben das Ziel eine hohe Ressourcenauslastung bei gleichzeitig guter Servicequalität zu erreichen. Um diese Eigenschaften erreichen zu können und zur Gewährleistung einer breiten Einsetzbarkeit, basieren solche Systeme zu großen Teilen auf *generischen Ressourcenabstraktionen*, wie virtuelle Maschinen oder Container. Generische Ressourcenabstraktionen greifen auf Annahmen zurück, wie zum Beispiel Ressourcenhomogenität oder lineare Ressourcennutzung, die die Planung vereinfachen, jedoch für die jüngsten Entwicklungen verteilter Systeme in der nach-Mooreschen Ära überarbeitet werden müssen. Insbesondere der jüngste Trend zu programmierbaren Netzwerkgeräten, die eine neue Ära des “in-network computing” (INC) einläuten, passt nicht mehr zur generischen Abstraktion von Containern, die auf Servern ausgeführt werden. Darüber hinaus fehlt für viele solcher Ressourcenplaner die Möglichkeit zur *dynamischen Planung zur Laufzeit*, da die zugrunde liegenden Algorithmen zu hohe Laufzeiten aufweisen oder nicht für einen dynamischen Ansatz konzipiert wurden. Ohne dynamische Planung muss für viele Szenarien die Ressourcenplanung vorab ausgeführt werden und kann bestenfalls in regelmäßigen Zeitabständen Korrekturen vornehmen, was letztendlich zu ineffizienter Nutzung der Ressourcen führt.

Die ständig steigende Nachfrage nach Internetdiensten im Allgemeinen, und die zunehmende Hardware-Heterogenität kombiniert mit der Unbeständigkeit der Anwendungslast im Speziellen, erfordern *dynamische* Lösungen zur *ganzheitlichen* Ressourcenplanung, sowohl auf Anwendungsebene als auch auf Infrastrukturebene.

In dieser Dissertation werden vier neue Lösungen vorgestellt, um eine *ganzheitliche dynamische* Ressourcenplanung für die verteilte Rechenlandschaft zu realisieren. Zwei davon auf Anwendungsebene sowie zwei auf Infrastrukturebene. Zunächst wird das Forschungsfeld der Ressourcenplanung für die verteilte Rechenlandschaft eingegrenzt, indem die beteiligten Systeme, Ressourcen, Bedarfe und Abstraktionen analysiert und klassifiziert werden. Aufbauend darauf wird eine Klassifizierung für INC vorgestellt um den Entwurfsraum für die Ressourcenplanung zu erschließen. Die Lösungen auf Anwendungsebene zeigen für zwei Szenarien, wie dynamische Ressourcenplanung zu einer effizienteren Nutzung der Ressourcen beitragen kann. Als erstes Szenario betrachten wir Big Data Aggregationssysteme und stellen das System ROME vor. ROME analysiert automatisch den Datenstrom und berechnet zur aktuellen Umgebung die optimale Zusammenschaltung der verfügbaren Ressourcen, mit dem Ziel, die Aggregation frühestmöglich abzuschließen. ROME zeigt selbst gegen manuell eingestellte Systeme ein besseres Laufzeitverhalten. Das zweite Szenario diskutiert die Ressourcenplanung von verteilten Funktionsketten. Wir präsentieren STEAM, der erste verteilte Ressourcenplaner für dieses Problem, der die Planung auf Paketebene während der Laufzeit durchführen kann, ohne globales Wissen und Vorabinformationen zur aufkommen Last zu benötigen. Im Gegensatz zu bisherigen Arbeiten erzielt STEAM bessere Servicequalität bei gleichen Ressourcen, bzw. reduziert die benötigten Ressourcen bei gleicher Nutzerlast.

Auf Infrastrukturebene werden zwei alternative Lösungen präsentiert. Zunächst wird IncSched vorgestellt, ein System welches bestehende Rechenzentren-Ressourcenplaner für INC nachrüstet. Aufbauend auf der vorgestellten INC Klassifikation, führt IncSched ein neues Ressourcenmodell ein, übersetzt dieses für bestehende Ressourcenplaner und kapselt die Logik zur Verwaltung von INC. Dadurch werden existierende Ressourcenplaner erstmals kompatibel zu INC, was auf breiter Basis die Zugänglichkeit zu INC in Rechenzentren positiv beeinflussen wird. Um INC noch besser in die Ressourcenplanung integrieren zu können, wird als letztes HIRE vorgestellt, ein eigenständiger Rechenzentren-Ressourcenplaner. HIRE erweitert das Ressourcenmodell von IncSched um automatisch alternative Ressourcen spezifizieren zu können, bildet die Nichtlinearität von INC-Ressourcen ab und berücksichtigt die wechselseitigen Abhängigkeiten zwischen Servern und INC. Durch diese Neuerungen bedient HIRE im Vergleich zu IncSched-Lösungen mehr Anwendungen mit INC, reduziert die maximale Wartezeit, und trifft darüber hinaus noch qualitativ bessere Entscheidungen zur Auswahl der Ressourcen um Lokalität zu berücksichtigen. Alle vier Lösungen werden in umfangreichen Simulationen, zum Teil auch mittels Prototypen sowie integrierter Benchmarks, evaluiert.

Zusammenfassend präsentiert diese Dissertation vier neue Lösungen die zur ganzheitlichen dynamischen Ressourcenplanung der verteilten Rechenlandschaft beitragen. Damit zeigen wir die Relevanz *dynamischer* Ressourcenplanung und wie mittels *ganzheitlicher* Ressourcenplanung gemeinsam genutztes INC auf Rechenzentrumsebene zugänglich wird.

Acknowledgments

Completing this thesis would not have been possible without the continued support of my supervisors, colleagues, family, and friends. Looking back, I am amazed at how much I have learned thanks to the people who accompanied me. Here I want to express my gratitude to them.

I wish to express my gratitude to my supervisor, Patrick Eugster, for his continued support, optimism, and guidance, which were essential to this dissertation and the whole process of becoming a researcher. Thanks for creating a work environment that gave my colleagues and me the freedom to pursue long-standing research ideas and openly speak our minds. I do not take this for granted. This has contributed to how I scrutinize problems and conduct research. I wish to express my gratitude to my co-advisor Lin Wang for his support and guidance to strengthen my methodological foundation and his insightful comments and feedback and Max Mühlhäuser for his comments and willingness to act as my co-referee. Thanks to Ramin Kahlili for his collaborative support and for the many fruitful and welcome discussions we had during this journey. Max Mühlhäuser, Carsten Binnig, Amr Rizk, and Stefan Schmidt, whom I had the honor of collaborating with throughout my PhD journey. I would also like to thank Carsten Binnig, Iryna Gurevych, and Felix Wolf to act as my committee members.

A big thanks to Max Lehn, Christof Leng, and Michael Stein, who inspired me to do research, Alejandro Buchmann, who recommended joining Patrick Eugster's group, and Patrick Jahnke, who, finally, recruited me. Michael Stein, Andreas Rücklé, Malte Viering, and Ramin Kahlili for proof-reading and providing valuable feedback.

I would like to thank my collaborators and fellow PhD students Malte Viering, Michael Stein, Andreas Rücklé, Alexander Frömmgen, Patrick Jahnke, Seema Kumar, Matthias Eichholz, Pascal Weisenburger, Benjamin Hilprecht, Torsten Zimmerman, Bastian Alt, Tobias Ziegler, Sabrina Klos, Roland Speith, and the whole MAKI group for their support and many interesting discussions, work related and non-work related.

Last but not least, a big thanks goes to my parents, my sisters, and Heike Gute, supporting me through the years and being with me during highs and lows.

Contents

1 Introduction

Chapter Outline

Internet services have become an indispensable part of our lives, with billions of users using Internet services on a daily basis. This covers services of different complexities like distributed back-ends running in cloud data centers and service function chains (SFCs) spanning multiple (edge) data centers. Example use cases include services for real-time communication, collaborative editing of documents, and many hidden but omnipresent use cases like cashier systems and sensors of industry facilities. This vast number of pervasive Internet services would not have been possible without the availability of compute infrastructure on a global scale, constituting the distributed computing landscape. Users expect to use Internet services at any time at low cost, with the desired service quality, despite potential load spikes a service might face. A straightforward strategy to provision services with high availability is to allocate dedicated resources for each service. In turn, this strategy is likely to lead to over-provisioning and hence high operating expenses (OPEX), which contradicts offering services at a low price [TJP16; Arm+10]. A standard solution to this problem is to leverage resource scheduling—also called server consolidation—to share the underlying resources among many different workloads and services to achieve gains on OPEX [BCH13; JS14; TJP16].

Challenges. Resource scheduling takes a central role in the distributed computing landscape. On the application level, applications run their resource scheduling to plan the processing for available resource shares and actual workloads. Whereas on the infrastructure level, resource managers perform resource scheduling to give resource shares to applications. A wide range of resource scheduling systems for the distributed computing landscape has been proposed in the past, covering the application and infrastructure levels. A typical goal of these systems is to achieve at the same time high resource utilization and good service quality. Resource scheduling typically leverages generic resource abstractions, like virtual machines (VMs) and containers, to fulfill these properties and be compliant with a broad set of services. With the end of Dennard scaling and Moore's law, specialization is the way to further improve performance with domain specific architectures (DSAs) [HP19]. As a consequence, system heterogeneity becomes a norm and many assumptions in the distributed computing landscape have to change. Existing schedulers with generic resource abstractions fall short to accommodate these changes, *e.g.*, hardware heterogeneity and resource sharing of DSAs.

Data center resources are becoming increasingly heterogeneous in recent years, with servers offering accelerators like tensor processing units (TPUs) and graphics processing units (GPUs). As an example use case, consider an application that trains a machine learning (ML) model. Such an application benefits from servers with accelerators to finish the training faster than servers without specialized hardware. Not only servers have become heterogeneous, but network switches have become increasingly programmable over the past decades, capable of taking over an application's processing (and storage), the beginning of in-network computing (INC). Adding INC to the ML example improves the training latency by enabling INC aggregation when individual servers send their parameter updates after each iterative computation [Sap+17]. INC provides new opportunities for performance optimization of distributed applications if the (typically scarce) INC resources are used judiciously [Tok+19; Ben19; McC+19a; Kog+19; Jin+18]. While recent resource scheduling solutions consider server accelerators, INC is beyond the scope of existing resource scheduling solutions. In particular, the recent trend of INC overtaxes the widely used generic abstractions of compute containers running on servers.

Furthermore, many application-level resource scheduling systems lack support for *runtime* scheduling, often due to slow or unsuitable algorithms. Without runtime scheduling, many scenarios require resource scheduling to run in advance. However, properties required for scheduling, such as detailed information of the workload, are often available the earliest at runtime. Missing or inaccurate information required for scheduling is likely to result in inefficient resource usage [Rza+20; Jyo+16; Kul+17]. At best, without runtime scheduling, scheduling repeats periodically to update scheduling decisions.

The ever-growing demand for Internet services in general and increasingly heterogeneous resources like in INC combined with the highly varying demand in particular, require runtime solutions for holistic resource scheduling, covering the application and infrastructure levels.

This Dissertation. In this dissertation, we present four novel solutions to holistic runtime scheduling for the distributed computing landscape. Two solutions cover the application level and two the infrastructure level. Our contributions work towards the goal of a holistically organized distributed computing landscape. However, the proposed solutions stand for themselves as individual systems that contribute to this overarching goal. In more detail, in Section 1.2 we outline the problem statement, in Section 1.3 we present our hypothesis, and in Section 1.4 we summarize our contributions. At a high level, these solutions contribute to the following three building blocks:

1. *Fitting the application to the resources*, for application-specific resource usage that leads to more efficient and more flexible matching of a distributed application's execution plan to the available resources.

2. *Rethinking the scheduling problem as a runtime problem* to make scheduling independent of *a priori* information, *i.e.*, considering actual demands and available resources, and potentially allow for more fine-grained scheduling decisions.

3. *Proposing a holistic resource model and scheduling logic* that cover INC and server resources jointly and that let users express their applications' resource demands with more variety, while abstracting the complexity.

Our solutions at the application level focus on the first two building blocks mentioned above, whereas the solutions at the infrastructure level cover all building blocks. In Chapter 2 we provide background information and start with an analysis of the field of resource scheduling for the distributed computing landscape, and summarize involved systems, resources, and abstractions. Based on this, in Chapter 3 we present our **INC classification** which helps to understand the design space of INC resource scheduling, which we discuss in Chapter 6 and Chapter 7.

Solutions at the Application Level. First, we discuss the solutions at the application level in Chapter 4 and Chapter 5. Our first proposed solution covers the scenario of big data aggregation systems. In Chapter 4 we present our runtime systems for optimizing aggregation overlays for data analytics frameworks, realized in the ***Robust Aggregation Overlays Minimizing Execution Time*** **(ROME)** system. The dissertation by William Culhane [Cul15] (see Section 1.4) discusses the theoretical foundation of optimized aggregation overlays and the predecessor system of ROME (see Section 1.4). For the reader's convenience, we first give an overview of the existing model for a class of problems termed *compute-aggregate* in Chapter 4, and then introduce a framework for adapting aggregation overlays to near-optimal ones. Then, we introduce ROME, discuss its architecture and systems optimizations, and explain how ROME integrates into two well-known frameworks, Spark [Apa14] and Flink [Apa11]. Finally, we discuss the evaluation of ROME using microbenchmarks and integrated benchmarks with Spark and Flink. ROME reduces total aggregation time for various workloads, including single aggregation runs and iterative workloads. While supporting manual configurations, ROME is a fully automatic system for optimizing the aggregation overlay at runtime. In summary, our proposed system ROME demonstrates the importance of *adapting* an application's execution plan to the available resources at runtime.

Our second solution at the application level focuses on the scenario of SFCs. In Chapter 5 we introduce two systems, namely IA-MPP and STEAM. The chapter starts by formally introducing the *SFC runtime traffic scheduling problem*, *i.e.*, assigning SFC traffic to available servers at runtime. We then propose a throughput-optimal solution called **integer allocation maximum pressure policy (IA-MPP)**, which solves the scheduling problem using *global* knowledge of the network. Then we introduce **multi-site cooperative IA-MPP (STEAM)**, a distributed heuristic based on IA-MPP. We elaborate on the techniques that make STEAM a distributed *runtime* solution feasible to run in real-world deployments. The evaluation of IA-MPP and STEAM uses simulations and a server benchmark. We compare IA-MPP and STEAM with adapted state-of-the-art solutions for the SFC runtime traffic scheduling problem. Due to STEAM's runtime nature, we show that using STEAM helps to utilize the resources more efficiently. As a result, STEAM reduces the required resources to serve the SFC traffic compared with state-of-the-art schedulers, including SGH [Kuo+16] and SPH [ZLZ19]. IA-MPP and STEAM demonstrate the benefit of rethinking an application's resource scheduling problem as a *runtime* scheduling problem.

Solutions at the Infrastructure Level. Next, we discuss our solutions at the infrastructure level in Chapter 6 and Chapter 7. Chapter 6 starts with a discussion of the challenges of INC resource scheduling and presents key insights. Then, we present INCSCHED, a resource management framework for **INC-aware Scheduling (INCSCHED)**. We discuss its architecture, including the resource model for INC resources and the mechanism of selection strategies for selecting resource alternatives. To demonstrate universal compatibility with existing schedulers, we conduct case studies with three schedulers combined with INCSCHED. These case studies cover centralized and fully distributed scheduler architectures and three commonly used scheduler designs, namely queue-based, dominant resource fairness delay scheduling, and power of two choices. IncSched makes existing resource schedulers compatible with INC for the first time, contributing to the overall goal of *holistic* resource scheduling.

The last main chapter, Chapter 7, introduces **Holistic INC-aware Resource managEr (HIRE)**, which extends the solution of IncSched as follows. First, we elaborate on the challenges of INC resource scheduling with a more in-depth analysis of what IncSched considers. This includes non-linear resource usage and locality interdependencies of servers and switches. Then, we present the extended resource model and HIRE's design, including the design of a flow network to model the scheduling problem of joint INC and server resource scheduling. We evaluate HIRE using trace-driven simulations with a 4000 machines data center and compare it with retrofitted variants of four existing schedulers using IncSched. HIRE supersedes IncSched solutions concerning scheduling performance of satisfied INC resource requests, locality of placement decisions, and tail latencies.

In summary, this dissertation presents **(1)** a novel classification of INC, **(2)** describes two application-level *runtime* scheduling solutions, namely ROME and STEAM, **(3)** and two *holistic runtime* scheduling solutions at the infrastructure level, namely IncSched and HIRE, which make shared data center INC accessible. These solutions contribute to a holistically organized distributed computing landscape but also stand for themselves.

In the rest of this chapter, we discuss the problem statement (1.2), thesis statement (1.3), and contributions in detail (1.4). Before this, in the following section we elaborate on the demand on resource scheduling in the distributed computing landscape and present general related work, primarily addressing the unfamiliar reader.

1.1 The Demand on Resource Scheduling

This section presents general related work in resource scheduling for the distributed computing landscape. It discusses why resource scheduling is becoming increasingly dynamic and complex. Related work specific to each of the contributions is reviewed in Section 4.1.2, Section 5.1.2, Section 6.1.2, and Section 7.1.2, respectively.

Resource scheduling in the scope of the distributed computing landscape has been a very active research topic in the recent decade, from both the application- and infrastructure-level perspectives. We identify three main properties that come with the distributed computing landscape's dynamic nature, making efficient resource scheduling challenging.

1.1.1 Highly Heterogeneous Compute Resources

Data center resources are becoming increasingly heterogeneous in recent years. The broad adoption of ML components in data center applications increased the demand for accelerators attached to servers, including TPUs and GPUs, to boost ML models' training and inference times. Recent domain-specific scheduling solutions show the demand for scheduling heterogeneous server resources, particularly for shared accelerator clusters for ML workloads [Gu+19; Mah+20; Pen+18; Xia+18; Nar+20].

Not only servers have become heterogeneous, but network components like switches and network interface cards have also become increasingly programmable over the past decades. More recently, programmable switches can take over application-specific processing [PN19] "in the network" on the path between communicating parties. This trend has ushered in a new era of in-network computing (INC). Many application scenarios, including distributed systems concerns like agreement [Jin+18; Dan+15] and caching [Liu+17; Jin+17], and even high-level application functionality like machine learning [Sap+19; XZ19] benefit from offloading application-specific processing to switches. Existing work on INC focuses mostly on isolated scenarios, where a single INC service uses all switches. So far, multiplexing of INC services is limited to a single network device [ZBH18; HM16; Zha+19; Wan+20]. However, supporting INC in a data center context requires resource scheduling support of sharing switches between users and INC services.

INC overtaxes the "compute on servers" approach simply because INC resources are not directly attached to servers like other accelerators including GPUs and field-programmable gate arrays (FPGAs). INC adds compute (and storage) capacity to switches sitting in the network topology. State-of-the-art data center resource models consider all application components to run on servers, not partially/jointly on switches. A resource scheduler is likely to perform suboptimal decisions if the resource model cannot express applications in a holistic view, *i.e.*, including components running on both servers and switches.

1.1.2 Unpredictable Workloads and Fluctuations

The flexible nature of data center applications introduces challenges to run applications with an efficient resource configuration. Many properties are hidden for the application [Bur+16] or available the earliest at runtime: like exact information on how many worker nodes are available [Rza+20], their interconnection properties [Bal+11], availability of accelerators (and their features) [Fir+18], data locality [Zah+10], and properties of the workload [Pen+18]. Unpredictable workloads and fluctuations pose challenges, as we shall detail in the following.

If not exploited for resource planning by the data center scheduler and by the application itself, each of these runtime properties is a potential source for resource inefficiency. In case such information is missing, a resource scheduler might struggle to find the best match of candidate nodes and pending jobs. The application might miss opportunities for adapting its execution plan to the given resources. For example, when the actual workload differs from the expected workload, an application might be under-provisioned or over-provisioned.

To further clarify the impact of unpredictable workloads on application-level scheduling decisions, we discuss **big data aggregation systems**. Aggregation is common in data analytics and crucial to distilling information from large datasets. However, current data analytics frameworks do not fully exploit the potential for optimization in such phases. Many aggregation functions are associative [YGI09], so a natural choice is to aggregate results along an overlay (network) such as a tree connecting leaf nodes to a root. It becomes clear that there are simple customizations to such aggregation trees created for a broad range of aggregation functions. When deploying a data aggregation system, not all workload dynamics may be known. For example, a parameterized aggregation function or unknown data characteristics may lead to inefficient resource configurations.

Next, we discuss the example of **service function chain (SFC) deployments**. Most existing works for SFC focus on the placement problem, *i.e.*, which server should run which function and how many resources each function gets [Coh+15; Add+15; Mar+15; Mij+15; Wan+16; Kuo+16]. These solutions perform chaining of functions in a mostly static manner, with pre-runtime configured load-balancing performed among them. Few non-static solutions exist [QAS16; Era+17; Sat+18; ZLZ19; Anw+15; Pal+15], which periodically adapt deployment of these functions and their resource assignments to changes in the network traffic and topology. However, these solutions are still coarse-grained [Era+17], where the adaptation takes seconds to take effect [Pal+15] or the scheduling logic is too slow due to its high complexity [ZLZ19] and the involvement of disruptive service function instances migration [Era+17]. Missing workload information at the time of deployment (submitting a resource request) and unpredictable workload dynamics caused by fluctuations make good resource planning challenging.

1.1.3 Applications with Resource Alternatives

There is an ever-growing heterogeneity in data center resources, as discussed in Section 1.1.1. Servers provide accelerators like GPUs, FPGAs, and TPUs and INC switches take over application-specific processing. One side effect of resources of different performance, capabilities, and availability is resource alternatives. Hence, applications end up having multiple resource requests and application-level execution plans to perform processing (of the same work).

INC resources are relatively scarce compared to server resources. For example, on-chip stateful memory is tens of MB on a typical Tofino switch [Wan+20]. Similarly, server accelerators tend to be scarce or at least not available to all servers in a data center [Fir+18; Xia+18; Mah+20]. Given this scarcity, requests for INC or accelerator resources may be unsatisfiable within a non-trivial timeframe.

Fortunately, accelerator-enabled applications can also run without “accelerator resources.” For example, a partition/aggregate job can run without INC, though probably taking longer or requiring more (not-accelerated) servers to execute in the same timeframe. Similarly, an application performing training on a deep neural network (DNN) runs faster if it uses an accelerator like a GPU but could also run on server CPUs only. Nevertheless, some applications might require accelerators, *e.g.*, because of strict requirements on processing latency. Considering the heterogeneity factors mentioned earlier (see Section 1.1.1), an accelerator-enabled job can be specified by a set of substantially different, interchangeable resource demands. Typically with varying performance properties and multiplexing dependencies. Such flexibility (having alternative job demands) adds an extra dimension to the scheduling problem: which resource demands to accept for an accelerator-enabled job.

Domain-specific solutions, that can handle jobs with alternative resource demands, exist for example for GPU clusters and other accelerators [Gu+19; Mah+20; Pen+18; Xia+18; Cha+20; Nar+20; Zha+17b; Jeo+19; Pen+19; Le+20]. However, these solutions often depend on jobs implemented using specific frameworks and require performance estimates for each job's resource alternative. Furthermore, none of the existing solutions offers data center scheduling support for INC resources.

1.2 Problem Statement

Resource scheduling for the distributed computing landscape is becoming increasingly dynamic and complex. This is likely to disconnect resource scheduling decisions, the workload, and the underlying resources. In the following we discuss two major problems in more detail.

1. *Application-level planning and scheduling ignores runtime properties*
   Existing applications and schedulers do not leverage available runtime properties to the full extent, leading to reduced resource efficiency and reduced delivered service quality. We identify the following main reasons that cause these effects:

   (a) *A priori workload information differs from the actual workload*
   Typically, an application utilizes ahead-of-runtime workload estimates for performing application-level planning and scheduling of resources. However, *a priori* workload information is often coarse-grained and may miss important details, leading to a mismatch between expected and actual workloads. Hence, an application may plan insufficient resources for a specific task or may intentionally over-provision tasks for budgeting a resource backup. As a result, resource schedulers allocate available resources inefficiently for serving the actual demand. Especially for applications with latency-sensitive quality of service (QoS) requirements, *e.g.*, in 5G scenarios, the delivered service quality depends on the quality of application-level scheduling decisions [Sat+18].

   (b) *Scheduling algorithms designed for coarse-grained or offline decisions*
   Application-level scheduling often shows characteristics that do not match the requirements of a *runtime* scheduler [ZLZ19]. Reasons for this are the

computational complexity or algorithmic dependencies on information that is not available (*e.g.*, all buffer states in a large distributed setup). Another problem is the level of granularity these scheduling algorithms offer, often being coarse-grained due to their offline design. As a result, applications run at configurations that do not match perfectly to the actual demand.

(c) *Planning for the generic case to cope with unpredictable workloads*
Like the example discussed with data aggregation (see Section 1.1.3), many applications do application-level planning for the generic case without considering the actual workload characteristics. Applications perform planning, *e.g.*, of the aggregation overlay, ahead of runtime, considering a generic setup and workload—like a *one-size-fits-all* approach—to deal with the unpredictable workload at the time of scheduling. Naturally, the more cases application-level decisions consider, the better an application performs for a specific situation. In this example, slower aggregation latency will degrade resource efficiency and lead to lower service quality delivered.

2. *Applications cannot leverage INC resources in a shared data center*
Existing data center INC services are not ready for being used in a shared data center. Data center resource schedulers cannot account for shared INC resources, leading to unused or statically assigned INC resources. We identify the following main reasons that cause these effects:

(a) *Data center INC is not democratized*
Existing data center INC services focus on isolated scenarios, where switches benefit a single application. So far, multiplexing of different INC services is limited to a single network device [HM16; Wan+20]. Runtime sharing of INC is required to treat INC as a first-class member of data centers. INC runtime sharing would enable applications to access INC switches dynamically and even maybe share an INC instance.

(b) *Resource managers do not account for compute capacity on switches*
Existing data center resource schedulers use resource models that consider computing capacity on servers. Typically, they support additional feature flags for servers with accelerators (*e.g.*, GPU, FPGA) and failure domains. So far, the network is beyond the scope of most existing resource managers, except for virtual network embedding algorithms specifically used for network bandwidth reservations [Bal+11; Guo+10; Pop+12; Xie+12]. In particular, popular cluster resource managers [Vav+13; Bur+16; Cur+19] are completely agnostic to the status of the network managed by a separate entity, the network controller, thus being unlikely to support INC resources directly. Simply treating INC switches like servers leaves many questions unresolved, like how to (i) manage tear-up/-down phases of INC services (which often requires the

network controller to be involved), (ii) resolve multi-tenancy on switches which differs from multi-tenancy on servers, (iii) manage an INC service shared by several applications, (iv) incorporate network locality of server-switch communication when INC is used, and many other problems, as we will show.
This raises the question of how to manage INC resources (*i.e.*, INC switches) next to servers to make INC available via common front ends of data center resource managers.

1.3 Thesis Statement

In this dissertation, we introduce solutions for *holistic runtime* scheduling for the distributed computing landscape to improve distributed systems' performance and resource efficiency.

The hypothesis is that by modeling application-level resource planning problems as runtime scheduling problems, applications require fewer resources to serve the same workload and achieve higher service quality using the same resources. Furthermore, with a holistic INC-server resource model and scheduling logic, data center resource managers can better account for resource interdependencies, thus serving more INC resource requests at the infrastructure level.

To support the hypothesis, we discuss two scenarios at the application level, *i.e.*, we design the resource planning problem as a runtime scheduling problem and evaluate the performance and resource efficiency. First, we propose ROME, an integrated middleware system for big data aggregation to automatically reduce aggregation latency by applying runtime scheduling. Second, we model a runtime solution for the service function chain scheduling problem and discuss two system implementations. The system IA-MPP for performing global optimization and the distributed system STEAM with scalability and real-time constraints in mind.

On the infrastructure level, we validate our hypothesis as follows. We characterize the INC resource challenges, introduce a holistic resource model and scheduling logic, and evaluate these compared with retrofitted schedulers. We propose two systems. First, we propose the framework INCSCHED, which hides INC complexity and is sufficiently flexible to extend existing server-only data center schedulers. Second, we extend the INCSCHED resource model and propose HIRE, a holistic scheduling approach which considers more INC specific side-effects and constraints to serve more INC resource requests.

1.4 Contributions of This Dissertation

In this dissertation, we introduce mechanisms and solutions for holistic runtime scheduling at the application and infrastructure levels to mitigate the above discussed problems. More specifically, this dissertation describes five principal contributions:

1. **Classification of in-network computing (INC).** We characterize in-network computing (INC) by elaborating on the three dimensions we identify an offloaded processing function must fulfill to belong to the realm of INC. Namely, physical, semantic, and logical characteristics. Our **classification of INC** helps to distinguish INC from classic network functions. We use the classification as a foundation in this dissertation when discussing resource scheduling of INC.

2. **Runtime planning of application's aggregation plan.** Based on previous work by Culhane et al. [Cul+14; Cul15; Cul+15] which adapts aggregation overlays of big data aggregation workloads for reduced processing latency, we make the following contribution: The **ROME system** for big data aggregation workloads. This middleware system acts either as a standalone aggregation system or as an aggregation orchestrator for other aggregation systems (*e.g.*, Spark and Flink). When using the default autodetection mode, ROME automatically chooses the optimal aggregation plan at runtime to fully leverage all available compute resources while taking the properties of the actual aggregation function into account.

3. **Runtime scheduling of distributed SFCs.** We propose to treat the SFC scheduling problem as a runtime scheduling problem. The contributions of this is a **theoretically optimal solution IA-MPP** and a **distributed systems solution STEAM** for the runtime traffic scheduling problem of SFCs. IA-MPP transforms the scheduling problem into a stochastic processing network (SPN) and follows the maximum pressure policy (MPP) by Dai and Lin [DL05]. IA-MPP and STEAM rethink the SFC scheduling problem as a runtime scheduling problem. Our solutions are independent of *a priori* information like traffic estimates and more efficient in utilizing all available processing capacities. Compared with state-of-the-art solutions, our proposed systems reduce the required processing capacities to serve the same workload, and at the same time, achieve better latencies.

4. **Democratizing data center INC.** We introduce a new resource model for data center scheduling and a corresponding scheduling framework to democratize INC, called **IncSched**. IncSched adds INC resource compatibility to existing data center schedulers and provides a flexible API to implement strategies for dealing with resource demands under various load situations. We describe how IncSched integrates with three existing data center schedulers, each combined with different strategies for coping with INC.

5. **Holistic INC and server scheduling.** We propose a holistic data center scheduler that jointly considers INC and server resources. The **HIRE data center scheduler** handles the INC data center scheduling problem without depending on IncSched. HIRE follows the path of flow-based schedulers [Isa+09; Gog+16] but introduces a flow network that supports making scheduling decisions for (1) servers and (2) INC, and (3) fine-granular flavor selection, all within the same flow network and scheduling round. Furthermore, we propose a cost model that optimizes placement latencies while trying to reach the highest possible success rates in serving INC resource requests. This holistic INC scheduling enables HIRE to encode global scheduling goals using the proposed cost model. We also demonstrate that these extensions to flow-based schedulers have negligible effects on scheduling runtime performance.

Declaration of Originality. All models, algorithms, and implementation details described are the results of my work. I built the IA-MPP, STEAM, IncSched, and HIRE implementations from scratch. However, colleagues and students have at times assisted me in extending and evaluating specific prototype components. Furthermore, co-authors of the involved publications contributed to the models and algorithms during intense discussions while working on the publications.

In particular, ROME integrates and extends previous work, notably an initial description of the idea of leveraging aggregation function characteristics for optimizing corresponding phases [Cul+14] and an exploration of theoretical foundations for such optimized aggregation [Cul+15]. William Culhane discussed in his PhD thesis [Cul15] the mathematical model and the system NOAH, the predecessor of ROME. This dissertation presents ROME's architecture, APIs, and the evaluation of ROME's performance while integrated into well-known frameworks Apache Spark [Apa14] and Apache Flink [Apa11]. Pascal Kleber contributed a fault tolerance implementation of ROME under my supervision during his Master's thesis [Kle17]. Max Herbst contributed to the Spark implementation of ROME under my supervision during his programming lab in the DSP group.

The initial ideas for IA-MPP to build on the MPP [DL05] for SPNs came from Ramin Khalili. Daniel Failing implemented and evaluated STEAM's DPDK prototype under my supervision during his Bachelor's thesis [Fai19].

For IncSched, Marco Micera explored the weaknesses of existing data center schedulers when adding INC resources and discussed early ideas of IncSched under my supervision during his Master's thesis [Mic20]. Furthermore, Marco Micera and Max Schmidt contributed to the implementation of IncSched, HIRE (min-cost max-flow solver and cost model), Yarn, and CoCo in the simulator under my supervision.

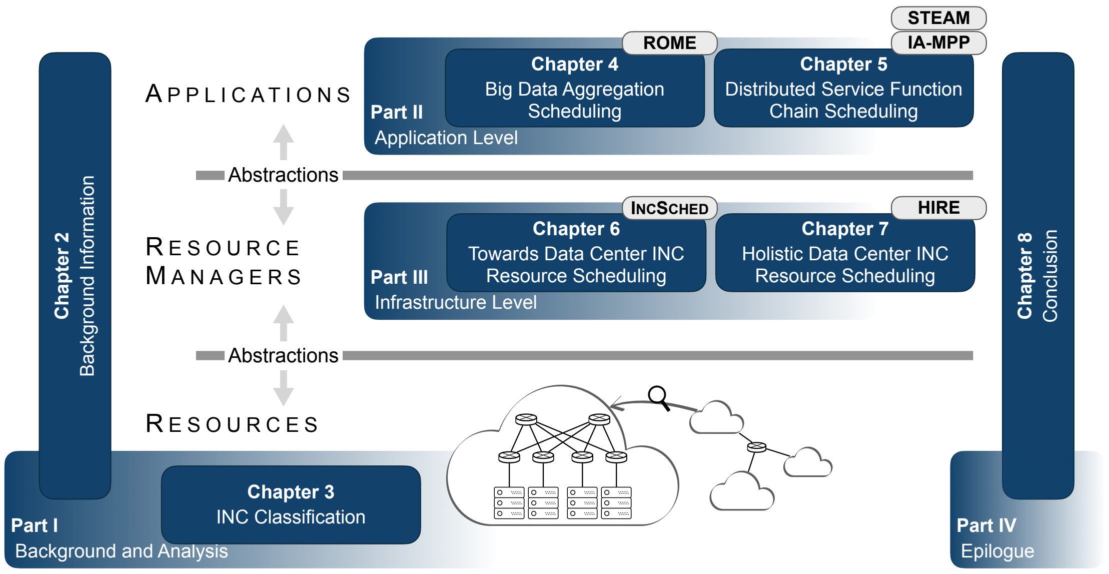


Figure 1.1: This dissertation's organization: Part I provides an overview of relevant background information and introduces our classification of INC. Next, Part II and Part III present our holistic runtime scheduling solutions. Finally, Part IV concludes this dissertation.

1.5 Overview of This Dissertation

This dissertation is structured as outlined in Figure 1.1:

Part I The first part lays the foundation of all other parts with an overview of resource scheduling in the distributed computing landscape. Furthermore, we introduce a classification of INC that contributes to the general discussion of offloaded application logic.

Part II The second part discusses two application scenarios and highlights the importance of runtime resource scheduling for efficient usage of the distributed computing landscape. Chapter 4 discusses the application scenario of data aggregation of big data applications and introduces ROME. Then, Chapter 5 presents two runtime scheduling solutions for the traffic scheduling problem of distributed service function chains. One variant which uses a global centralized policy (IA-MPP) and a fully distributed variant (STEAM). This part presents standalone resource scheduling solutions on the application level. Both scenarios could run with an existing or one of our proposed (in Part III) data center resource scheduling solutions.

Part III The third part focuses on the infrastructure level and presents two data center resource scheduling solutions for managing server and INC resources. Chapter 6 presents the resource management framework IncSched with a new resource model and an INC scheduling logic, which can be used jointly with existing data center resource schedulers to make these for the first time compatible with INC resources. Lastly, Chapter 7 presents the new resource manager HIRE which supersedes IncSched. This part presents the first resource scheduling solution for shared INC on a data center wide level.

Part IV The last part finally concludes the dissertation by summarizing the contributions and elaborating on directions for future work.

1.6 Related Publications

Parts of this dissertation have been covered in international peer-reviewed publications, listed below. All publications have joint authorship. Unless otherwise stated at the beginning of a chapter and in Section 1.4, the material presented in this dissertation is the author's contribution.

[Blö+21] Marcel Blöcher, Lin Wang, Patrick Eugster, and Max Schmidt. "Switches for HIRE: Resource Scheduling for Data Center In-Network Computing". In: *Proceedings of the 26th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).* ACM, 2021, pp. 268–285. DOI: `10.1145/3445814.3446760`

[Blö+20b] Marcel Blöcher, Ramin Khalili, Lin Wang, and Patrick Eugster. "Letting off STEAM: Distributed Runtime Traffic Scheduling for Service Function Chaining". In: *Proceedings of the 39th Conference on Computer Communications (INFOCOM).* IEEE, Aug. 2020, pp. 824–833. DOI: `10.1109/INFOCOM41043.2020.9155404`

Parts of this dissertation are based on manuscripts that are currently under submission, listed below. All publications have joint authorship. Unless otherwise stated at the beginning of a chapter and in Section 1.4, the material presented in this dissertation is the author's contribution.

[Blö+20a] Marcel Blöcher, Emilio Coppa, Pascal Kleber, Patrick Eugster, William Culhane, and Masoud Ardekani Saeida. "ROME: All Overlays Lead to Aggregation, but Some Are Faster than Others". Submitted for publication. Mar. 2020

I have also authored or co-authored the following publications during my PhD time. Some of these have impacted the work presented in this dissertation but did not directly contribute to its contents.

[Blö+19] Marcel Blöcher, Matthias Eichholz, Pascal Weisenburger, Patrick Eugster, Mira Mezini, and Guido Salvaneschi. "GRASS: Generic Reactive Application-Specific Scheduling". In: *Proceedings of the 6th SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS).* ACM, Oct. 2019, pp. 21–30. DOI: `10.1145/3358503.3361274`

[Heu+18b] Jens Heuschkel, Lin Wang, Erik Fleckstein, Michael Ofenloch, **Marcel Blöcher**, Jon Crowcroft, and Max Mühlhäuser. “VirtualStack: Flexible Cross-layer Optimization via Network Protocol Virtualization”. In: *Proceedings of the 43rd Conference on Local Computer Networks (LCN)*. IEEE, 2018, pp. 519–526. DOI: 10.1109/LCN.2018.8638106

[Heu+18a] Jens Heuschkel, Rick Vogel, **Marcel Blöcher**, and Max Mühlhäuser. “Blow up the CPU Chains! OpenCL-assisted Network Protocols”. In: *Proceedings of the 43rd Conference on Local Computer Networks (LCN)*. IEEE, 2018, pp. 657–665. DOI: 10.1109/LCN.2018.8638096

[Blö+18] **Marcel Blöcher**, Tobias Ziegler, Carsten Binnig, and Patrick Eugster. “Boosting Scalable Data Analytics with Modern Programmable Networks”. In: *Proceedings of the 14th International Workshop on Data Management on New Hardware (DAMON)*. ACM, 2018, pp. 1–3. DOI: 10.1145/3211922.3211923

[Blö+17] **Marcel Blöcher**, Malte Viering, Stefan Schmid, and Patrick Eugster. “The Grand CRU Challenge”. In: *Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems (HotConNet)*. ACM, 2017, pp. 7–11. DOI: 10.1145/3094405.3094407

Part I

Background and Analysis

The first part of the dissertation introduces relevant technical background this dissertation is based on and provides a detailed introduction and analysis of the field of resource scheduling in the distributed computing landscape, including a classification of INC.

Chapter 2 starts with a discussion of the distributed computing landscape, the involved server and networking resources, resources abstractions for these, and an overview of data center resource scheduling. This lays the foundation for all topics following in Part II and Part III. Finally, this part introduces a classification of in-network computing in Chapter 3, and gives an overview of recent in-network computing solutions. Thereby contributing to the general understanding of the requirements for INC resource scheduling for Part III.

2 Distributed Computing Landscape

Chapter Outline

This chapter provides technical background that this dissertation is based on. We start with an overview of the distributed computing landscape. Next, Section 2.1 gives an overview of server resources, Section 2.2 follows with the network fabric, and Section 2.3 summarizes resource abstractions for server and networking resources. Finally, Section 2.4 gives an overview of data center resource scheduling.

Figure 2.1 shows the distributed computing landscape and depicts locations where processing (⚙) of an Internet service can take place. As we shall detail in the following, processing can take place—on servers (▤) and switches (⊜)—at various locations in the

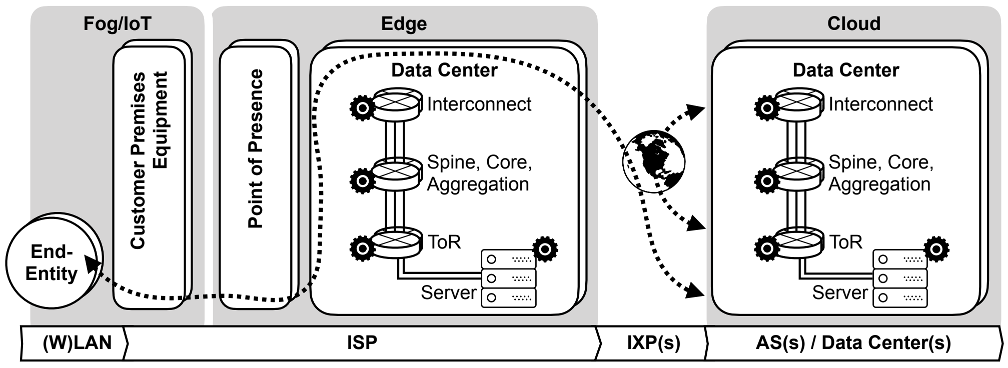


Figure 2.1: The distributed computing landscape. Arrows exemplify communication patterns of applications, and ⚙ possible locations where processing of an application can take place: on servers (▤) and switches (⊜)

distributed computing landscape, *e.g.*, on a server at an edge data center close to an involved end entity. In this dissertation, we consider edge data centers and cloud data centers, and discuss resource scheduling solutions for distributed applications (Part II) and data center resource managers (Part III). This covers all locations highlighted in Figure 2.1. In the following sections we discuss involved resources and abstraction.

2.1 Servers and Accelerators

In a shared data center, applications use server resources either by "bare-metal" access (*i.e.*, dedicated root access), or by using server virtualization techniques like VMs and containers (*e.g.*, containerd [CNC19], Docker [Ber14]). If a server virtualization technique is used, applications typically get (multiplexed) resources with the abstraction of (shared) CPU cores, dedicated and elastic memory portions, and network access using bridged mode or Single Root I/O Virtualization (SR-IOV), among other techniques [BCH13]. More recently, shared data centers offer server accelerators including TPUs, GPUs, SmartNICs, and FPGAs. Various techniques are used for sharing access to accelerators, depending on how these accelerators are integrated in the data center infrastructure. For example, an accelerated host networking platform [Fir+18; Fir17], that uses FPGAs for providing a high performance virtual switch platform, does not expose the accelerator directly to the application.

Resource Sharing. Dynamically sharing the underlying resource among many different workloads and frameworks is crucial to achieving high resource utilization for data center computing [Hin+11; Sch+13]. Resource sharing can be implemented either by

(a) elastic compute approach, *i.e.*, re-allocate resources when latency critical services face low load or idle times, and delegate spare compute resources to offline workloads

(b) resource overcommitment, *i.e.*, co-locate workloads on the very same machine and set priorities to handle overload scenarios

In both resource sharing options, containerization is one of the key enablers for the powerful and efficient operation of the distributed computing landscape. Containerization is a technique that encapsulates all software dependencies of an application (component) into an image, which then could run on a generic container infrastructure running on commodity servers. These (application) container images are also called "hermetic container image" [Bur+16] due to their full encapsulation of all required pieces to run the software. Containerization brings the benefits of less maintenance workloads introduced by many different hardware and software versions that need to be supported. "*Containerization transforms the data center from being machine-oriented to being application oriented* [Bur+16]."

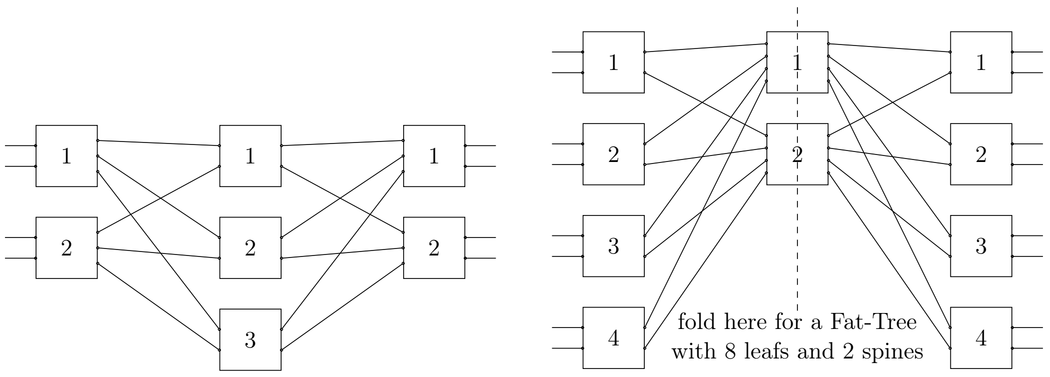
There is no universal answer to choosing always elastic compute approach over resource overcommitment, or vice versa. However, containerization gives both options to the resource manager. In particular, for small, energy efficient servers (with small resource capacity) and resource hungry services, the elastic compute approach seem to match better [Fac18].

2.2 Network Fabric

The data center network fabric is a critical part of the data center infrastructure, and sets the limitations and capabilities for server communication with respect to latency and available bandwidth. Beside resources for packet-forwarding, programmable switches offer resources for in-network computing, which we will discuss in Chapter 3. In the following, we discuss packet-forwarding resources.

A server rack houses a group of servers and interconnects them with one or multiple so called top-of-rack (ToR) switches. The ToR switch is also the interconnection of a server rack to the network fabric using the ToR's uplink port(s). Figure 2.1 shows at a high level the typical layers of the network fabric, from servers bottom up, ToR switches connect to the multi-tier network topology comprising aggregation, core, and spine switches, and finally the interconnect which connects a data center with the *outside world* (*e.g.*, other data centers).

Typically, the network fabric uses a multi-path topology, to provide better fault tolerance, higher total throughput, and lower latency and jitter [BCH13]. Many data center network topologies are based on the Clos [Clo53] topology [Sin+15], like Fat-Tree [Lei85], Facebook's 4-post [FA13], F16 and HGRID [AWE19]. Clos networks have three stages, each with a number of switches (crossbars) with inputs and outputs. Figure 2.2a shows an example Clos network. A Clos network is defined by 3 parameters (n, m, r) for setting the number (n) of sources for each ingress stage crossbar (and outputs of each egress stage crossbar), the number (r) of ingress (and egress) stage crossbars, and the number (m) of middle stage crossbars.



(a) Strictly nonblocking Clos network $(n, m, r) = (2, 3, 2)$.

(b) Rearrangeable nonblocking Clos network $(n, m, r) = (2, 2, 4)$.

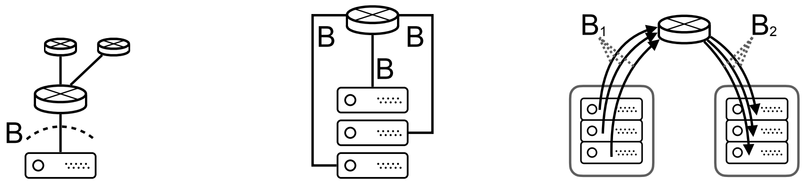
Figure 2.2: Clos network as a building block for many data center topologies.

Clos networks can easily be used to build networks with more than three stages, by replacing the middle stage crossbars with Clos networks (*i.e.*, recursively stacking Clos networks). Figure 2.2 shows two Clos networks, and how to build a Fat-Tree topology based on a Clos network. When folding the Clos network in Figure 2.2b at the dashed line, the ingress and egress stage of the Clos network become the leaf layer in a Fat-Tree, the middle stage becomes the spine layer.

Multi-stage topologies can be classified according to the blocking characteristics [New89, Chapter 4]. A topology satisfies the *non-blocking* property, if it can connect any free input with any free output, regardless of other connections. If existing connections must be rearranged to be non-blocking, a topology satisfies *rearrangeable non-blocking*. Otherwise, a topology belongs to the *blocking* group. The blocking/non-blocking concept is applicable for networks exhibiting circuit switching. Data center switches typically[1] are packet switched networks, however, blocking/non-blocking becomes important when considering available link bandwidth. With $m \geq 2n - 1$, a Clos network satisfies the strictly nonblocking property, with $m \geq n$ it is rearrangeably nonblocking.

Network Resource Sharing. In data centers, servers share the network fabric according to a policy, *e.g.*, best effort routing, strict packet priorities, static reservations, among others [TJP16; BCH13]. There are two reasons for the demand for resource sharing policies. First, typically the topology is *blocking*, *i.e.*, the total link network capacity of all servers (of a subtree of the topology) exceeds the total aggregated link capacity of the corresponding subtree switches. This effect is described with network oversubscription. Oversubscription reduces the size and cost of the network fabric. Second, Incast is another reason for the demand for resource sharing policies. Incast describes a throughput

[1] Recent works [Bal+20] on optically-switched architectures emulate packet-by-packet switching by reconfiguration at nanosecond-timescales.



(a) Hose: Aggregated bandwidth B at a server.

(b) Virtual cluster: Bandwidth B for each server towards a switch.

(c) Tenant application graph: Outgoing (B_1) and incoming (B_2) bandwidth for each server of a group towards another group.

Figure 2.3: Common data center resource abstractions for network bandwidth guarantees.

collapse effect (which happens even without oversubscription). A typical cause of incast is when many network streams arrive at a switch and are all routed to the same output port, thereby exceeding the capacity of the outgoing port.

2.3 Resource Abstractions

Most data center resource managers use resource abstractions that focus solely on server resources. A broad set of data center resource managers provide scheduling abstractions dedicated for server accelerators, often targeting a domain-specific solution like those for ML workloads [Gu+19; Mah+20; Pen+18; Xia+18; Nar+20]. Some data center resource managers also consider networking resources, but mostly with respect to the network link capacities of the involved servers, not the network fabric. Only few data center resource managers handle networking resources with respect to reserved bandwidth shares [Bal+11; Guo+10; Pop+12; Xie+12]. Typical abstractions for describing the demand on networking resources include hose [Duf+99], virtual cluster (VC), virtual oversubscribed cluster (VOC) [Bal+11], and tenant application graph (TAG) [Lee+13].

Figure 2.3 shows three common resource abstractions. Hose (see Figure 2.3a) describes the reserved aggregated uplink and download capacity of a server, without considering possible communication paths. A VC (see Figure 2.3b) is similar, but describes a group of servers. Each server of a group has a reserved bandwidth share towards a virtual switch. A VOC consists of a hierarchical group of VCs, optionally with oversubscription of some of the connections. A TAG (see Figure 2.3c) is a directed graph of groups (servers) and edges (network links). Each group represents a set of servers each with the same network bandwidth and server resource demand. Directed edges describe network bandwidth reservations from any server of a group towards another server group (by using a virtual switch for specifying the bandwidth reservations). In Figure 2.3c, each server in the left group has bandwidth B_1 (outgoing) and B_2 (incoming) for communicating with the right group.

All described resource models for networking resources focus on bandwidth reservations, and differ mostly in the level of detail for supporting nested bandwidth constraints, and how pair-wise bandwidth reservations are considered.

2.4 Resource Scheduling

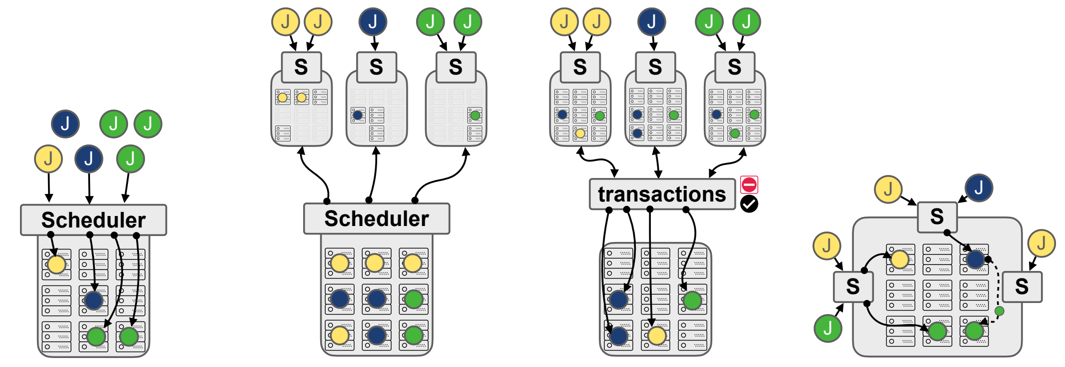
Data center resource scheduling is about assigning compute resources (*e.g.*, CPU, memory) to jobs in a way reaching some set requirements on resource efficiency, task placement latency, and scalability, among others [SB18]. Data center resource scheduling falls into three layers [Zha+15]:

1. **Application (Software) Layer** providing the Software as a Service stack. Scheduling of application components onto virtual resources (*i.e.*, VMs).
2. **Virtualization (Platform) Layer** providing the Platform as a Service stack. Scheduling of virtual resource (*i.e.*, VMs) onto physical resources (*i.e.*, servers).
3. **Deployment (Infrastructure) Layer** providing the Infrastructure as a Service stack. Scheduling focuses on topics like data routing and multi data center management.

In this dissertation we focus on the distributed computing landscape, which concerns all layers mentioned above, but most dominantly the Software as a Service (in Part II) and Platform as a Service (in Part III) layers. We discuss resource scheduling for the distributed computing landscape from the perspective of the resource abstractions of the involved parties, rather than a strict separation by using the classification of the three layers.

Both single-resource [Ous+13; Gog+16] and multi-resource [Gho+11; Gra+14; Cho+16] scheduling have been well studied. Most related work in this area focuses on containers (and the corresponding jobs/tasks) as the unit of scheduling and serves resources among multiple applications/frameworks (*e.g.*, Spark, Flink, and TensorFlow) [Hin+11; Ver+15]. To cope with this sharing, resource managers exist with a variety of architectures (*e.g.*, centralized and distributed) and scheduler designs (*e.g.*, queue-based and graph-based).

Architectures. Figure 2.4 shows four common generic architectures for data center resource managers, as outlined by Schwarzkopf [Sch15]. The *centralized* monolithic scheduler manages all data center resources from a centralized perspective (Figure 2.4a), *i.e.*, users send their resource requests to the centralized manager, which in turn runs a global policy for scheduling. A variant of the centralized architecture is the statically partitioned architecture, which partitions the data center resources and assign each partition to a



(a) Centralized (b) Two-Level (c) Shared-State (d) Fully Distributed

Figure 2.4: Data center resource management architectures.

single sub-scheduler. Hadoop [Zah+10], Quincy [Isa+09], and Firmament [Gog+16] are examples of centralized or statically-partitioned centralized schedulers

The *two-level* approach (Figure 2.4b), also called pessimistic concurrency [Sch+13], splits the resource allocation problem into two parts: the data center resource manager acts as a central coordinator which decides how many resources each sub-cluster gets. Each sub-cluster is controlled by another sub-scheduler, often an application framework specific scheduler. Mesos [Hin+11] is a well-known example of this architecture using resource offers.

Next, the partially distributed *shared-state* architecture (Figure 2.4c), also called optimistic concurrency [Sch+13], uses multiple scheduler instances each with full access to all data center resources. This architecture requires concurrency control to resolve potential resource allocation clashes (conflict resolution). Omega [Sch+13] is a shared-state scheduler which uses a transaction like interface for applying resource allocations to the global cluster state. Apollo [Bou+14] is another shared-state scheduler performing conflict resolution at the level of server queues.

Finally, the *fully distributed* architecture (Figure 2.4d) uses a scheduler design which allows to run the scheduling logic in a fully distributed manner. Sparrow [Ous+13] is a well-known example, which applies a variant of power of two choices [Mit01] to sample candidate machines for each resource request. Each machine in turn manages a queue of pending resource requests, and eventually probes the requesting job in the event of local resource availability.

None of these generic scheduling architectures strictly surpasses every other architecture, each shows advantages for certain domain-specific problems. Furthermore, many resource schedulers use an architecture that combines multiple of these generic architectural design principles.

2.5 Summary

In this chapter, we discussed basic concepts and related work that is required for this dissertation. We started with an overview on data center resources and resource abstractions. Server resources are typically shared among different workloads using elastic compute approach or resource overcommitment. Existing server resource abstractions often use the abstraction of containers with optional feature flags and a multi dimensional resource demand vector. Existing abstractions for networking resources focus on network bandwidth, *e.g.*, to set strict routing priorities and to statically reserve bandwidth. Next, we introduced data center resource scheduling, and discussed typical architectural design options for resource managers, namely centralized, two-level, shared-state, and fully distributed.

In Part II, we present application-level schedulers, using a centralized and a distributed architecture with statically partitioned sub-clusters. Followed by Part III, which presents two data center resource scheduling solutions: A resource management framework (Chapter 6) with a centralized architecture that works in tandem with other scheduler architectures outlined in this chapter and a standalone centralized scheduler (Chapter 7). Before we present our scheduling solutions in the following parts, the next chapter introduces our classification of in-network computing.

3 In-Network Computing

Chapter Outline

This chapter presents a classification of in-network computing (INC). Even though many INC services have been published recently, there is not yet a classification of INC which distinguishes INC from related concepts like network function virtualization (NFV) and classic network functions in general. This chapter closes this gap and introduces a classification of INC, which is used to distill the requirements for a resource management solution for INC in Chapter 6 and Chapter 7.

3.1 Deployment Targets

Before going into the detail of our classification, this section gives an overview on the enabling technologies to push application processing from servers to locations “in the network.” We consider not all of these deployment targets as valid instances of what we consider to belong to the realm of INC.

Software Network Functions. Network function virtualization (NFV) is a technique that uses software implementations on top of commodity hardware instead of using expensive, specialized hardware middleboxes. This decouples the network function from the underlying hardware, *i.e.*, NFV uses virtualization techniques to install software network functions on *any* matching hardware. Example use cases include network address translation (NAT), deep packet inspection (DPI), and firewalling. A popular NFV platform is ClickOS [Mar+14]. Flick [Ali+16] and In-Net [Sto+15] are two example solutions based on NFV which term their solution as in-network processing.

Programmable NICs. SmartNICs are programmable network interface cards (NICs) that can be used to offload application processing, *e.g.*, for reducing CPU load on networking-intensive tasks. Typical architectures of SmartNICs include pipelined designs [Fir+18], many-core designs[1], and reconfigurable match tables (RMT) designs [Kau+16; Pon+19]. The latter is also the typical design of a programmable switch. A popular use case of SmartNICs is AccelNet [Fir+18], an accelerated host networking platform which is deployed on a data center level in Azure cloud. IncBricks [Liu+17] and NICached [SB17] are two examples that use SmartNICs and call their approaches INC. IncBricks uses a co-design of SmartNICs and switches for the prototypical implementation for doing INC, whereas NICached proposes to use solely SmartNICs for doing INC.

Programmable Switches. Programmable switches offer programming abstractions originally intended to serve as prototyping or deployment methods for more flexible and novel network(-wide) services and protocols, *e.g.*, HULA [Kat+16b]. These solutions used the programming abstractions mainly for accessing packet headers, but not for performing application-level processing. With the recent advances on programmable switches, the programming abstractions became more expressive and the hardware more powerful, enabling new use cases. Data plane programming languages such as P4 [Bos+14] and frameworks like μP4 [Son+20] and Lyra [Gao+20a] provide programming abstractions for the network data plane, allowing network devices to be customized for application-specific computation. Various hardware architectures are used for programmable switches, but most dominantly application-specific integrated circuits (ASICs) with an RMT design [Bos+13]. Recent work on program synthesis for programmable switches provides an extensive overview of hardware architectures [Gao+19; Gao+20b; Jos+15].

[1] `https://www.netronome.com/products/smartnic/overview/`

3.2 Classification of INC

No exiting work provides a classification of INC that is sufficient to differentiate INC from related techniques such as NFV, NIC offloading, and classic network functions (*e.g.*, NAT).

Inconsistent Naming. Existing definitions of INC are indefinite, *e.g.*, "In-network processing, where data is processed by special-purpose devices as it passes over the network, ..." [Mus+19]. Furthermore, no consistent naming is used by recent work in this area. We observe three naming conventions used dominantly, namely: *in-network processing* [Mai+14; Anw+15; Sto+15; Ist+16; Ali+16; Jin+17; Blö+18; Mus+19; PN19; Alo+19; Zen+20; Blö+17], *in-network computing/compute* [Sap+17; Liu+17; Liu+19; Ben19; PN19; TMZ18; Zhu+19; Kog+19; Blö+21], and *in-network X* or *in-switch X*, where *X* can be caching [Jin+17; Ren+14; Liu+19; Li+16b], aggregation [KB20], database acceleration [Ler+19; Blö+18], replication protocol [Zen+20], network sequencing [Li+16a; LMP17], among others.

Previous work use the same term for different types of offloading techniques and for different target devices. Flick [Ali+16] and In-Net [Sto+15] are two example solutions based on NFV which term their solution as in-network processing. IncBricks [Liu+17] uses a co-design of SmartNICs and switches for the prototypical implementation, however, conceptually IncBricks considers programmable switches for their offloading solutions. NICached [SB17] proposes to use SmartNICs for doing in-network processing.

Ports and Nelson [PN19] introduce a taxonomy which helps to identify which data center application can benefit from offloading computation to programmable networking devices. This taxonomy considers three dimensions, namely (a) number of operations per packet, (b) state per packet, and (c) packet gain, *i.e.*, how many packets a networking device sends when receiving a packet. For each of these dimensions, the authors consider the asymptotic behavior, *e.g.*, for packet gain the cases $O(1)$ (same number of input and output packets), $O(r)$ (output to r replicas), and $O(1/r)$ (output a single aggregate result from r replicas). Based on this taxonomy, the authors discuss best deployment targets for an in-network solution. Possible deployment targets include middleboxes, FPGAs, network processing units (NPUs), SmartNICs, and switches. Furthermore, for each of the three dimensions, they identify examples where the dimension is the dominant resource bottleneck of a deployment. Their classification of INC includes deployment targets that are located at the servers, *e.g.*, FPGAs with network ports directly connected to the fabric. However, we do not consider accelerated host networking as belonging to INC.

Benson [Ben19] introduces another taxonomy of INC, focusing on the functionality of deployed INC services. Benson defines INC as the "shift of network functions that act at layer-7 for the purpose of pure networking, to include computing functionality." The introduced taxonomy with two dimensions classifies the *type* of the offloaded application

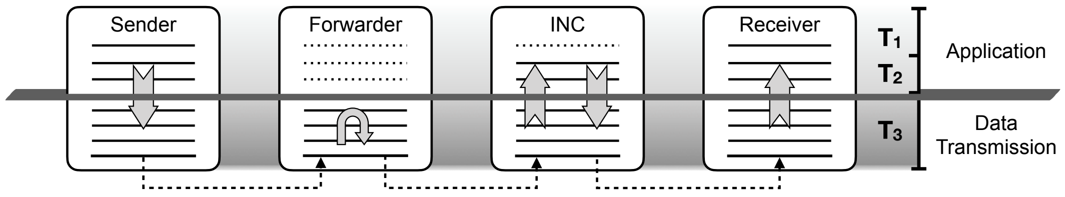


Figure 3.1: Logical representation of communicating parties and the abstract tier they work at.

functionality to be INC or *pure networking functions*, and the *mode* dimension identifies if the mode of operation is *offloaded* or *transparent*. The offloaded mode requires to rewrite server applications, while the transparent mode is defined as when the functionality is unattached to the application. This classification depends on the blurring of the definition of computing. For example, a load balancer could be a classic network infrastructure function, *e.g.*, ECMP [Hop00], but also a solution with computing functionality, *e.g.*, R2P2 [Kog+19], that we consider to belong to INC.

In the following, we present our INC classification based on three characteristics that must be fulfilled by an offloaded processing function for being classified as INC.

Physical Characterization. The first characteristic we set for INC refers to one of its main intuitions, *i.e.*, INC processing takes places somehow on a device in the network. An INC device could be either *en route* between communicating parties (*e.g.*, two servers), or the INC device could be itself one of the end-to-end communicating parties in the network (*e.g.*, when a single server communicates with an INC device).

Definition 3.1 characterizes offloading approaches as not belonging to INC, if the approach uses a device whose primary intent is not to forward network packets. For example, a middlebox is an "intermediary box performing functions apart from normal, standard functions of an IP router on the data path between a source host and destination host" [CB02], which classifies all NFV based approaches as not belonging to the realm of INC. Furthermore, this definition excludes approaches that offload processing on a SmartNIC attached to a server (*e.g.*, [SB17])—we consider this to belong to accelerated host networking.

Definition 3.1. *INC processing takes place on a device in the network, whose primarily intent is to forward network packets, ideally at line rate.*

Semantic Characterization. The second characteristic of an offloading approach potentially qualifying as INC is on the kind of processing that the offloaded component is performing. The main purpose of the offloaded function must be connected to an application running on communication parties connected to the network. A function that is deployed on a network wide level, which is required to operate the network, does not belong to the realm of INC.

Definition 3.2. *The duty of INC processing exceeds sole data transmission purpose to operate the network, and is strictly intertwined with specific applications running on connected servers.*

Logical Characterization. The third characteristic of INC covers the logical aspect of what the offloaded processing function is acting with. Figure 3.1 depicts communicating parties and abstract tiers (T_1 - T_3) the processing of each device is accessing. Data transmission belongs to tier T_3, INC operates at tier T_2, and the application's sender and receiver at tier T_1. An INC node may be itself the sender or receiver. Tier T_1 and tier T_2 might refer to the same logical abstraction. A classic networking device involved in forwarding network packets along the path works solely on logical abstractions intended for data transmission (T_3).

A counter example to INC is DPI, if it is deployed on a network wide level. Such a DPI solution might access information from abstract tier T_2, which fulfills the logical characteristic (Definition 3.3), but its pure purpose is to keep the network operating, *i.e.*, it contradicts with the semantic characteristic (Definition 3.2).

Definition 3.3. *INC denotes processing which operates on logical layer* not *part of data transmission abstractions.*

Summary. In this dissertation, we use the term INC as described below in Definition 3.4, which combines the three characteristics we set in Definition 3.1, Definition 3.2, and Definition 3.3.

Definition 3.4. *In-network computing (INC) denotes the general concept of offloading processing (and/or storage) to programmable networking devices, which fulfills the following three criteria:*

1. *INC processing takes place on a device in the network, whose primarily intent is to forward network packets, ideally at line rate.*
2. *The duty of INC processing exceeds sole data transmission purpose, and is strictly intertwined with specific applications running on servers connected to the network.*
3. *INC denotes processing which operates on logical layer* not *part of data transmission abstractions.*

3.3 Overview of INC Services

INC has been explored for various scenarios including data aggregation, caching, and coordination, among others. All these approaches share a common goal of achieving gains on performance [PN19] and/or energy efficiency [Tok+19]. In the following, we explain some INC services in more detail.

Data Aggregation. A straightforward candidate for INC is data aggregation since many data center jobs (*e.g.*, SQL joins, MapReduce, graph processing, and machine learning) involve partition/aggregate patterns, and network traffic can be significantly reduced by in-network aggregation [Sap+17; Blö+18; Gra+16a; Jep+18; Ler+19]. Even though they share a similar goal, they differ in the hardware that is used, the integration into the software stack running at the servers, and the way an application deploys its configuration to use the INC devices.

A typical approach for in-network aggregation is to base the data plane on P4 and have the software defined network (SDN) controller build aggregation trees [Sap+17; Sap+19]. The controller sets up an aggregation overlay consisting of a list of INC switches that need to be configured. Each of the involved switches maintains a list of identifiers of active aggregation overlays, hence an INC switch performs aggregation when receiving data from a predecessor node, either an INC switch or a server. Switches can also be configured through direct interactions with the involved servers, in place of the SDN controller in some designs [Gra+16a].

SHArP [Gra+16a] is another solution to in-network aggregation and requires switches with a custom designed ASIC. SHArP proposes a *Scalable Hierarchical Aggregation Protocol* which allows to offload a subset of MPI [WD96] and OpenSHMEM [Cha+10] to SHArP enabled switches. The SHArP *Aggregation Manager* hands over the list of involved INC switches to the involved servers. After receiving the INC resource descriptors, the servers interact with the INC switches on their own. SHArP supports simple single switch aggregation overlays and also large tree setups.

Caching. Another representative application scenario for INC is caching. Essentially, it offloads key-value pairs with high access counts to INC switches, which reduces the load of the servers and improves latency at the same time [Jin+17; Liu+17; Liu+19]. The main idea is as follows: When a server wants to access a key either with write or read access, it sends a query which passes involved INC switches. In case of a write access, each INC switch checks if it has to prune a cached entry, and forwards the query towards the server holding the specific key. When a server answers with a specific key-value pair, the INC switches check whether they should cache the key-value pair for the next query while forwarding the reply. The SDN controller retrieves access statistics and decides on updating the list of hot items for each of the INC switches.

In addition to rack-scale implementations such as NetCache [Jin+17], global data center solutions have also been proposed [Liu+17; Liu+19], where an SDN controller is used to set up INC switches for a specific key-value store and to steer corresponding queries through all involved INC switches.

Coordination. Another application for INC commonly described in literature [Jin+18; Dan+15; LMP17; Dan+16; Yu+20] is coordination. While serving a different purpose, in-network coordination can also be based on a distributed fault-tolerant key-value store (or log) at its core, making the deployment of in-network coordination solutions similar to that of in-network caching, despite maintaining consensus on the involved switches. An SDN controller usually serves to configure the INC switches and handle requests [Jin+18].

In NetChain [Jin+18], when a client wants to use the INC service, a server's NetChain client translates the API requests to NetChain's own protocol, addressing the chain's tail or head switch, for read or write requests, respectively. The head switch processes write requests and sends them to the next INC switch of the chain. Finally, the tail switch replies to the NetChain client. The NetChain client is aware of all involved INC switches of a chain, which was set up by the SDN controller beforehand. The underlying key-value store uses consistent hashing and maps each key to a key ring using a replication of $f + 1$ for each ring segment, to allow up to f switch failures.

Typically, using the example of NetChain [Jin+18], an INC service provides high-level configuration parameters, that directly affect the deployment and configuration properties of the involved INC switches. NetChain provides configuration parameters like key length (bytes), value length (bytes), and replication factor (chain length). When the high-level configuration parameters and the target switches are known, the exact resource requirements (*e.g.*, allocated stages of the RMT on a switch) can be determined, sometimes with the involvement of SDN or INC specific deployment tools [Jos+15].

3.4 Summary

This chapter discussed possible deployment targets for offloading application processing and introduced a classification of in-network computing (INC). This classification defines three characteristics that offloaded processing functions must fulfill to belong to the realm of INC. Definition 3.4 summarizes the three characteristics, namely physical, semantic, and logical.

In Part III we will present two scheduling solutions for data center resource scheduling, that offer resource scheduling of server and INC resources simultaneously.

Part II

Application-Level Resource Scheduling

The second part of this dissertation presents two solutions to runtime resource scheduling at the application level, covering two scenarios: big data aggregation and distributed service function chains. These solutions demonstrate the advantage of using runtime resource scheduling solutions at the application level, by the example of two application scenarios.

Chapter 4 presents ROME, a system for resource planning of big data aggregation systems. ROME automatically adapts the aggregation overlay to the aggregation function to reduce total aggregation time. Then, Chapter 5 introduces our solutions IA-MPP and STEAM for runtime traffic scheduling of distributed service function chains.

4

ROME: A Middleware System for Optimized Aggregation Overlays

Chapter Outline

This chapter introduces ROME, an application-level aggregation system for use within data analytics frameworks like Spark and Flink or standalone. ROME uses a set of novel heuristics based primarily on basic knowledge of aggregation functions combined with deployment constraints to efficiently aggregate results from computations performed on individual data subsets across nodes (*e.g.*, merging sorted lists resulting from top-k). The user can either provide minimal information which allows our heuristics to be applied directly, or ROME can autodetect the relevant information at little cost. We integrated ROME as a subsystem into the Spark and Flink data analytics frameworks. We use real world data to experimentally demonstrate speedups up to 3× over single level aggregation overlays, up to 21% over other multi-level overlays, and 50% for iterative algorithms like gradient descent at 100 iterations.

With the exception of portions of Section 4.7, the contents of this chapter are reprinted, with permission, from **Marcel Blöcher**, Emilio Coppa, Pascal Kleber, Patrick Eugster, William Culhane, and Masoud Ardekani Saeida. "ROME: All Overlays Lead to Aggregation, but Some Are Faster than Others". Submitted for publication. Mar. 2020 [Blö+20a].

This chapter extends previous work by Culhane et al. [Cul+14; Cul+15; Cul15] as described in detail in Section 1.4. Section 4.1, Section 4.2, Section 4.3, and Section 4.5 are predominantly based on previous work.

4.1 Overview

Data analytics systems typically split data across nodes and compute information from subsets of the data, then aggregate the partial results.

Many aggregation functions are associative [YGI09], so a natural choice is to aggregate results along an *overlay (network)* such as a tree connecting leaf nodes (where original computations occur) to a root (where the final result will be available). It becomes clear that there are simple customizations to such aggregation trees created for a broad range of aggregation functions. Exactly which customizations are applicable—most prominently affecting *fan-in* of the tree—depends on the characteristics of the aggregation function which affect the size of the data (more details in Section 4.2).

We propose to automatically tailor aggregation overlays to *specific problems*, using the ratio R of the output size of the aggregation function at hand to its input size, which is easy to find or estimate for many aggregation functions. For example, a *top-k aggregate* (simply top-k for short) overlay reports the k groups according to a selection criteria on the group's score [IBS08]. If all data belonging to a group resides on a single partition, so that each group's score is computed based on a single partition, the aggregation function produces output of size k—a ratio of $R = 1$ (*i.e.*, k/k). If some group's data is spread across all partitions, on the other hand, the output is larger than any individual input, similar to merging sets. Table 4.1 shows common aggregation problems grouped by relevant ratios.

Common problems	$R = \frac{\lvert\textbf{Output}\rvert}{\lvert\textbf{Input}\rvert}$
Production jobs at Facebook, Yahoo! [Che+11], Google [DG08]; sieve and deduplication algorithms	< 1
Top-k on pre-partitioned data, k-means clustering, square matrix multiplication, word count with fixed dictionary	$= 1$
Top-k or word count on arbitrary data, sort	> 1

Table 4.1: Some common aggregation functions and their size ratios of output to *one* input ($|a|$ means the size of a).

In this chapter, we present a novel holistic approach for optimizing aggregation implemented in a system called ROME (*Robust Aggregation Overlays Minimizing Execution Time*). In short, our approach is threefold:

1. an *Analysis* stage obtains relevant constraints from applications, based on which,
2. an *Overlay* stage determines theoretically (near-)optimal overlay trees for idealized settings; finally,
3. a *Mapping* stage applies several heuristics to tailor these overlays to real-world (non-ideal) deployment constraints at hand.

Like most currently popular systems for data analytics [Zah+12; Ale+14; Ven+13], ROME works in-memory to significantly reduce latency and unpredictable timings. It is designed for easy integration into existing data analytics systems.

4.1.1 Design Challenges

ROME faces three design challenges, which we discuss in this chapter:

Compatibility and Application Transparency. A deployment of ROME should be transparent to the application and support a broad range of existing applications and workloads for performing data aggregation. Not only the functional compatibility of the provided API is important, but also the operational behavior of the system, *e.g.*, fault tolerance, plays an important role. This ensures easy integration into existing applications and frameworks with minimal efforts, which fosters an easy integration into many existing systems.
To achieve this, ROME provides an API for total aggregation problems which fits to many typical problems in the compute-aggregate area. We provide a standalone implementation of ROME, but also an integration into Apache Spark [Apa14] and Apache Flink [Apa11]. For all implementations, ROME provides a set of features to integrate transparently into many application deployments, including fault tolerance features and operation modes for incremental computations (*i.e.*, append-only updates and infrequent updates).

Automatic Configuration. In order to use an optimal aggregation overlay with ROME, the system needs to know the optimal configuration for setting up the aggregation overlay. A system that requires a user to configure all parameters manually, is likely to face low adoption/acceptance rate and the system may not run at the best performance for several reasons. The user may choose wrong parameters, or some of the required information may not be available at the time when the user configured the system.
The intended way of using ROME is to max out its capabilities of its autodetection system. When running in autodetection mode, ROME runs a partial aggregation to infer the required information, and continues the remaining aggregation using the detected system behavior. However, ROME also provides APIs for fine-tuned manual configuration, which allows to invoke ROME's aggregation overlay with a self-configured setup, but still using ROME's configuration system to prevent mis-configured setups (*e.g.*, prevent cases where nodes do not get any predecessor data).

Fast Processing and Minimal Overhead. ROME's purpose is to provide better latency for aggregation phases, compared with default or manually configured aggregation overlays. Consequently, ROME, when running combined with Spark and Flink, should not induce noticeable performance overhead when running in the automatic configuration mode, compared with a perfectly manually configured counterpart of Spark or Flink running without ROME.

To achieve this, ROME implements performance optimization techniques including co-location of aggregation nodes, root-node bypass, and tree balancing mechanisms. These techniques make ROME run fast by minimizing latency of the slowest aggregation branch and reducing redundant "write-out read-in" network transfers.

4.1.2 Related Work

Research which underscores the importance of minimal aggregation latency largely ignores the impact of overlays [Ke+15; Kum+16]; existing big data analytics frameworks mostly use one-size-fits-all overlays, leading to unnecessarily high latency for distributed data aggregation. When customizable overlays are available in a framework, users must *manually* configure them properly for a particular problem. Such static optimization might become simply impossible with non-trivial aggregation functions (*e.g.*, composed functions) and jobs, or when the same code/code portion is executed on several distinct datasets with different data skewness.

The data analytics systems discussed in Section 4.2.3 are popular in part because of their power to adapt to current processing needs. Data analytics has quickly outgrown many early attempts at aggregation which required special architecture, such as processors in tree networks and fully parallelizable functions [CR90; KJL96]. Aggregation is so fundamental to data analytics that the MapReduce framework has been modified to include aggregation more than once. Incoop [Bha+11] exploits `Combiners` to aggregate between the map and reduce phases, and Map-Reduce-Merge [Yan+07] implements aggregation after the reduce phase. Yu et al. [YGI09] also attempt to optimize aggregation between phases. All these approaches attempt to add the necessary aggregation functionality to MapReduce, but do not consider the effect of topology.

Morozov and Weber [MW13] integrate efficient aggregation into their design of merge trees, an abstraction for aggregating large structured datasets. Their system monitors data attributes in different branches and recomputes more efficient trees for the prevailing network status. This system allows to deal well with changing network conditions rather than optimizing an overlay upfront as with our heuristics.

Kumar et al. [Kum+16] consider the need for an aggregation overlay to respond with the most accurate result available within a deadline. The proposed solution determines the probability of a child node providing additional results within a given timeframe, then decides to wait or forward the current known results in order for the parent node to meet its deadline. This work is complimentary to our focus on overlays. Astrolabe [Van+02] and STAR [Jai+07] use a holistic data management approach, and optimize aggregation within their given hierarchies. Astrolabe uses gossip protocols for large-scale data propagation. STAR, extending the work of SDIMS [YD04] to build on top of distributed hash tables, has more flexible and configurable aggregation options, but tries to use its topology rather than a prescribed fan-in. SDIMS allows the user to specify an aggregation precision with the understanding that very low latency is sometimes a priority over perfect answers. Kumar et al. [Kum+16] also consider the ability to selectively drop data during processing to minimize latency for applications which need very fast aggregation. PIER [Hue+03] is another system built on DHTs to distribute workload, this time for database use. The overlays are once again restricted by the underlying framework, but the system itself efficiently aggregates results to respond to queries.

While not specific to aggregation, Kim et al. [KJL96] extend the work by Cheng and Robertazzi [CR90] to optimize load distribution on processors connected by a tree network. The newer work maximizes parallelization for fastest completion because there is no computation to aggregate results from each processor. Work with sensors optimizes overlays for power consumption while conforming to the routing restrictions imposed by the location and communication capabilities of sensors [CT00; TK03]. TinyDB [Mad+05] furthers this in determining when to sample, which is equivalent to local computation. Valerio et al. [Val+12] consider aggregation when servers span multiple administration domains that do not trust each other. Their auditing system detects if a server may bias the aggregation result when manipulating the aggregation overlay. We assume all nodes are within the same administration domain. Morozov and Weber [MW13] consider *merge trees* for distributed computations, an abstraction for combining subsets of large structured datasets. Their system monitors data traits in different branches and recomputes a better tree. It does not optimize for aggregation functions or extrapolate to find optima. Naiad [Mur+13] considers aggregation as part of iterative or cyclic computations. It allows distributed data access and updates to be interleaved, and is not aggregation-specific. Like resilient distributed datasets put forth by Zaharia et al. for

Spark [Zah+12], Naiad relies on data retained in memory. This offers latencies which can be orders of magnitude better than with disk accesses. This addresses the problem raised by Venkataraman et al. [Ven+13] that data access is a significant bottleneck for iterative calculations on many distributed frameworks.

While users may be able to use our heuristics to manually configure aggregation in in-memory distributed data analytics systems like Spark [Zah+12] and Presto [Ven+13], as mentioned, this approach may fail when datasets are yet unknown, which is commonly the case with non-initial computation phases. One work which does explicitly consider topology is CamCube [Abu+10], which allows users with full environmental control to define neighbor nodes which bypass traditional network routing to minimize latency. Each machine is limited to 6 such neighbors, but the configuration has been shown to decrease latency in some jobs using a MapReduce style framework with built-in aggregation called Camdoop [Cos+12].

Chuprikov et al. [Chu+17; Chu+18] investigate compute-aggregate problems in setups where network links have different capacities/costs and "hard" topological constraints (*e.g.*, compute nodes may not aggregate) focusing on hardness of optimal distribution, and lower bounds.

The initial intuition of using the R ratio for optimizing the execution of basic compute-aggregate tasks was presented at a workshop by Culhane et al. [Cul+14], however without thorough consideration of dealing with real-world (discrete) settings, or implementation experience. The formal aspects of the problem were studied subsequently [Cul+15], again in an idealized setting without considering implementation, further adding focus on the case of data streaming.

4.1.3 Contributions

In this chapter, we make the following contributions:

- We introduce ROME, a full-featured system implementing our heuristics. We describe its architecture and API. We then integrate ROME as a subsystem into two common data analytics frameworks, Apache Flink [Apa11] and Apache Spark [Apa14], to increase the efficiency of end-to-end data analytics (Section 4.4).

- We evaluate ROME through Flink and Spark. Within real world systems, job execution time shrink by a factor of up to 3 over systems using single level aggregation overlays like `reduce` in Flink [Apa11] and Spark [Apa14], by 21% over the `treeReduce` in Spark, and 16% if using Spark's feature to *manually* configure an overlay. When running an iterative algorithm like gradient descent, total runtime improves by %50 at 100 iterations over Spark's Mllib implementation, which also uses a multi-level aggregation overlay (Section 4.6).

4.2 Model

This section outlines the compute-aggregate family of problems considered, and how it fits into the landscape of data analytics.

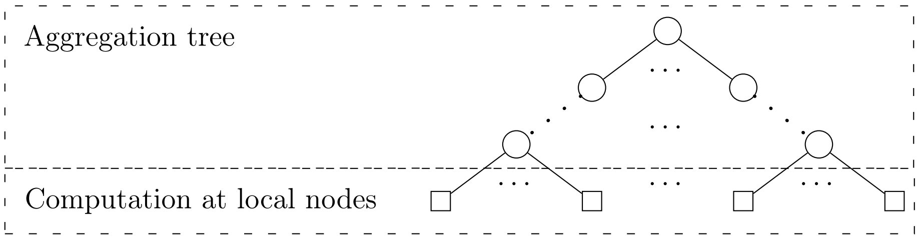


Figure 4.1: Visual representation of the computation and aggregation phases.

4.2.1 Problem Definition

Intuitively, compute-aggregate problems consist of two phases (see Figure 4.1): a *(i)* compute phase processes distributed subsets of input in parallel; a subsequent *(ii)* aggregation phase combines the results of the first phase to obtain a final output.

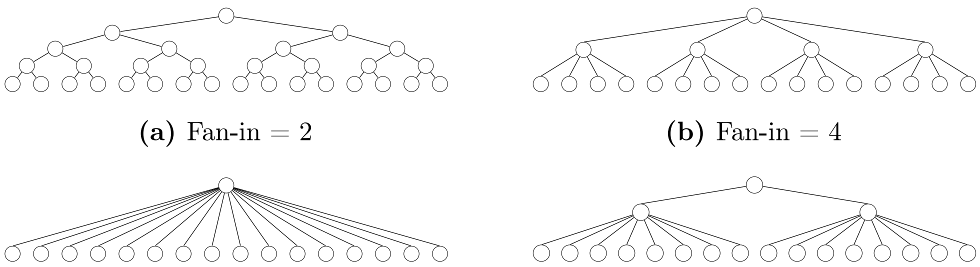
More formally, using the notation summarized in Table 4.2:

Definition 4.1. *A compute-aggregate task produces output $h(\overline{z})$ from input $\overline{z} = z_1, \ldots, z_n$ where $h(\overline{z})$ is decomposable into computations on partial inputs, $f(z_1), \ldots, f(z_n)$, and an aggregation function $g()$ such that $h(\overline{z}) \equiv g(x_1, \ldots, x_n)$ with $\forall i \in [1..n] \quad x_i = f(z_i)$.*

Intuitively, each computation node contains some subset of the intial data. After computation (i), a system aggregates (ii) the results along an *aggregation tree* (henceforth simply tree) communication structure to create the final output. With the exception of passing the results of the computation to the aggregation tree the two phases are independent from each other. The two phases are visually represented in Figure 4.1.

We consider optimizing the aggregation phase. Optimizing computation requires knowledge about the data, data structures, and computation for each specific problem. We show optimizing the tree often only requires knowing very basic information about the aggregation function.

Aggregation can be triggered by the completion of the computation phase or run periodically on the current state of the data, as long as the data is formatted for aggregation. Aggregation applies some function g to all of the outputs of the computation nodes, $g(f(z_1), \ldots, f(z_n))$. This does not have to be done in a single step. Aggregation can be applied to the results of previous aggregation. When aggregation begins, each output from a leaf is sent to a single aggregation node. The aggregation is applied to all inputs received at the node, and the node outputs the aggregated result. The outputs from those nodes, if there are indeed multiple such nodes, are in turn aggregated. The final aggregate result contains exactly one path to each leaf, so each computation output is included exactly once, resulting in an explicit tree structure. Figure 4.2 shows how 16 leaf nodes can be placed in four different trees that only differ in their fan-ins.



(a) Fan-in = 2

(b) Fan-in = 4

(c) Fan-in = 16

(d) Fan-in = 2/8

Figure 4.2: Four aggregation trees with 16 leaves.

4.2.2 Function Requirements

We consider aggregation functions which take $\overline{x} = x_1 \ldots x_m$ and output an aggregate $x^{1..m}$, *i.e.* $x^{1..m} = g\left(\overline{x}\right)$. Functions must be able to handle any number of inputs in order for the increased fan-in to be effective, as there is no advantage to having multiple inputs available it they cannot be used.

Functions are cumulative, commutative, and associative. This essentially means inputs may be aggregated in any order with any group of inputs, including those which are outputs of non-root nodes of the tree. Definition 4.2, Definition 4.3, and Definition 4.4 capture the properties more precisely. Definitions require equivalency ($\equiv$), not necessarily identical output. For example, if a system is supposed to output the single word with the

Token	Meaning
n	Number of computation/leaf nodes.
$f\left(\right)$	Initial computation function.
$g\left(\right)$	Aggregation function.
$h\left(\right)$	(Composed) function to compute and aggregate.
z	Data formatted for input/computation.
x	Data formatted for aggregation.
F	Fan-in of the tree, making the height $O(\log_F n)$.
$g\left(\overline{x}\right)$	The aggregation function for a set of inputs $\overline{x}$.
$g^t\left(\overline{x}\right)$	Returns the time taken for $g\left(\overline{x}\right)$ (with communication).
t	Time per unit of data for linear $g^t\left(\overline{x}\right)$; $g^t\left(\emptyset\right) = 0$.
x_0	Output from one computation node.
R	Ratio of the final aggregate output size to $\lvert x_0 \rvert$.
R_1	Ratio of output sizes of individual levels.

Table 4.2: List of notation used.

System	Fan-in	In-mem	Aggregation types
MapReduce [DG08]	Manually configurable		By partition
Flink [Apa11]	Manually configurable	✓	By partition
Spark [Apa14]	Manually configurable *or* tree template by height	✓	By partition *or* total aggregation *or* add-only semantics
ROME	Adaptive *or* self-adaptive	✓	Total aggregation

Table 4.3: System comparison.

maximum number of occurrences (word count) and two words are tied for that distinction, either word may be returned.

Definition 4.2 (Cumulative Aggregation). $g\left(g\left(\overline{x}\right), g\left(\overline{x}'\right)\right) \equiv g\left(\overline{x}, \overline{x}'\right)$

Definition 4.3 (Commutative Aggregation). $g\left(\overline{x}', \overline{x}\right) \equiv g\left(\overline{x}, \overline{x}'\right)$

Definition 4.4 (Associative Aggregation). $g\left(g\left(\overline{x}, \overline{x}'\right), \overline{x}''\right) \equiv g\left(\overline{x}, g\left(\overline{x}', \overline{x}''\right)\right) \equiv g\left(\overline{x}, \overline{x}', \overline{x}''\right)$

4.2.3 In Perspective

Before detailing how we optimize aggregation, we put it in the perspective of state-of-the-art data analytics systems which can natively aggregate data (see Table 4.3).

Aggregation in many big data analytics frameworks follows the MapReduce [DG08] approach, and map data to disjoint processing partitions. This *aggregation by partition* model still inspires a significant portion of data analytics. It works well for problems like word count where aggregation can be partitioned. We show this is inefficient for problems such as top-k in Section 4.4.

For *total aggregation* jobs where all data must be compared (*e.g.*, *top-k*, *sort*), at least transitively, to each other to find a global result, aggregation by partition must use only a single partition. That is, users must either create a single reducer (fan-in of n) or run iterations of the problem to prune data at the cost of remapping at each iteration. While Flink [Apa11] addresses some issues with MapReduce (*e.g.*, processing in-memory for lower latency), it still suffers from similar aggregation limitations. Aggregation by partition and total aggregation are disjoint in the big data problem space, as shown in Figure 4.3, even if the tools for one may be applied to the other.

Spark [Apa14] also uses an in-memory model. Additionally, it adds an extra *aggregator* functionality to the MapReduce model, allowing aggregation to be distributed across multiple reducers. These aggregators require "add-only" semantics, which require monotonic and non-transitive operators. This is insufficient for a wide range of problems like top-k sorting, which rely on score comparisons *across* nodes to be transitive. Consequently, for

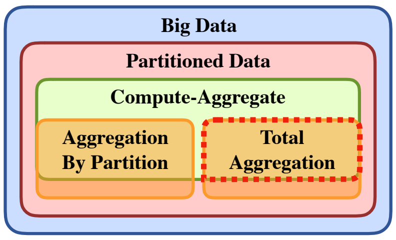


Figure 4.3: Compute-aggregate and total aggregation in the data analytics problem space.

aggregation methods more complex than a monotonically increasing counter, Spark must use its `reduce` operator, effectively limiting the deployment to a fan-in of n.

The `treeReduce` functionality (added 2015) in Spark enables user-specified aggregation topologies, however with several drawbacks. Firstly, the topology in `treeReduce` is defined by *height* rather than *fan-in* using a *scale* factor internally

$$scale := \mathsf{max}(2, \lceil partitions^{1/height} \rceil).$$

When a user calls `treeReduce`, Spark runs partial aggregation rounds a long as

$$remaining\ partitions > scale + \lceil remaining\ partitions / scale \rceil$$

evaluates positively, and updates

$$remaining\ partitions := remaining\ partitions / scale$$

accordingly. As a consequence, the user sets only an upper bound of the total height of the aggregation tree. Hence, Spark is forced to fit the aggregation overlay to the number of partitions, workers, and given (max) height, which ignores the actual aggregation function.

Secondly, Spark does not provide guidance on how to pick an appropriate value. Simply setting height, *i.e.* the upper bound, to a very large value will cause Spark to build up aggregation trees of maximum height, always enforcing a fan-in of 2. Informed users may use our heuristics presented shortly based on the ratio R of final output to one input, and the number n of nodes across which data is originally distributed, to manually determine a height. However, one must recalculate the height when the application is deployed with a different number of worker nodes or workload. Even then resulting overlays may be skewed by the greedy hash partitioner Spark uses to form an aggregation tree based on a rough fan-in derived from the provided height.

Clearly, manual calculation of an aggregation overlay requires that R and n are known before the application is deployed which can be hard. Thus, the ability of ROME to

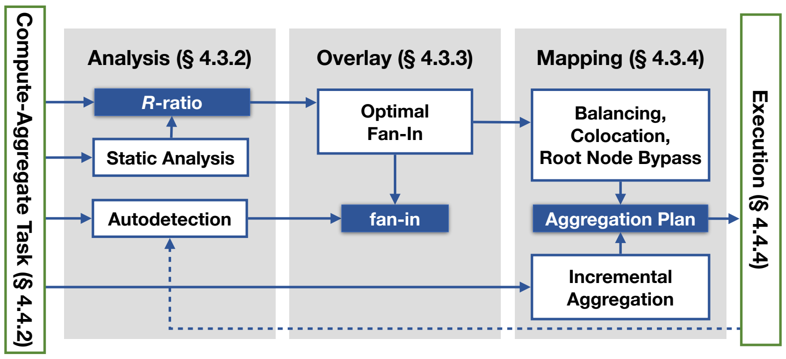


Figure 4.4: ROME's three-staged approach.

autodetect R, as we shall detail in Section 4.3.2, and use a properly determined topology based on fan-in at execution time is a more applicable way and leads to lower job latency.

4.3 Optimizing Compute-Aggregate

While the aggregation function $g()$ is fixed for a given problem, the overlay over which to apply it is configurable. Our goal is thus to find an aggregation overlay yielding minimal latency, using the set of available resource to its best. This section discusses our threefold approach towards that goal.

4.3.1 Optimizing Overlays

To optimize aggregation overlays ROME applies a three-stage approach as shown in Figure 4.4 to an aggregation job submitted by an application. In short:

1. The *Analysis* stage obtains relevant constraints for the job. This consists first and foremost in the mentioned characteristic R-ratio *between the aggregation function output and a single input* (see Table 4.2).

2. An *Overlay* stage uses R to determine the *fan-in* of a (nearly) optimal overlay tree for an idealized setting.

3. A *Mapping* stage applies several heuristics to tailor such an overlay to the real-world deployment constraints at hand; this includes catering for workload rebalancing, trees with non-fractional height, limited resources of on-path aggregation nodes, and strategic reuse of resources.

The job is then executed by ROME. In case the job was submitted through a data analytics framework that ROME is integrated into, this execution happens in a concerted manner with the framework as we shall elaborate on shortly.

Next we discuss the three phases in more detail.

4.3.2 Analysis Stage

The first stage is concerned with obtaining characteristic information from jobs that allow the necessary aggregation to be optimized. Chiefly this consists in the R-ratio. Other parameters used in later stages such as for incremental aggregation (see Figure 4.4) are (currently) obtained explicitly as parameters of the job.

Static analysis. R-ratios are well-known for a number of aggregation scenarios such as the ones presented in Table 4.1. ROME can thus perform a simple static analysis of aggregation jobs passed to it to determine if they consist in/use any of these functions directly. Depending on the integration of ROME into a larger framework, that framework can share a map with ROME that outlines any of the framework's own known/pre-defined aggregation functions and their corresponding R-values (or correspondences to ROME's pre-defined ones). Simple embeddings of these functions inside composite functions, *e.g.*, inside loops, allow for automatic inference of R in a good number of cases.

Autodetecting R. The static inference of R from a complex aggregation function $g()$ is not always possible. For the very same reason, one cannot automatically extract or synthesize an aggregation function $g()$ from a function $f()$ naïvely applied to an entire dataset, in order to allow for automatic breakdown of $h()$ into compute and aggregation functions $f()$ and $g()$ and corresponding distributed multi-phase execution (see [Mor+09; LHM11]).

For these cases, ROME supports automatic runtime detection of R. ROME then builds the first level of aggregation with a fan-in of 2. Once nodes at this level complete their local aggregation, R is locally computed and sent to the controller, which builds the heuristic tree for the average computed value of R and the $\frac{n}{2}$ results already computed. We use a fan-in of 2 for initiation as it allows estimating R very quickly while also preserving as much aggregation as possible for the optimal overlay calculated.

User-provided. To avoid autodetection of R and its overheads (or to bypass static analysis), a user can always submit a job with an explicit value for R. As a matter of fact, internally in ROME, a negative R value represents the *absence* of such a pre-defined value. We will elaborate on this later in the context of the ROME system and its API (see Section 4.4.2).

4.3.3 Overlay Stage

Our goal is thus to find an aggregation overlay with minimal latency for a given aggregation function. Figure 4.2 shows four overlays created with different fan-ins, yielding equivalent results for compute-aggregate tasks. Smaller fan-ins like that in Figure 4.2a yield higher parallelism at the lowest levels. Figure 4.2c instead obtains all input at

R	**Optimal fan-in**	**ROME fan-in**
$R < 1$	2	2
$R = 1$	e	3
$1 < R < n$	$\min\left(n, (1 - \log_n R)^{-\log_R n}\right)$	$\min\left(n, \lceil(1 - \log_n R)^{-\log_R n}\rceil\right)$, see Section 4.3.4
$R \geq n$	n	n

Table 4.4: Optimal value for F to minimize the latency of a single input block, and values ROME chooses.

the first level of aggregation. That level will thus take longer than a single level in Figure 4.2a, but there are fewer levels to run. Consider the aggregation of occurrences of words in a word count job. Each word is considered at each level. More branching increases parallelism, but at the cost of redundancy at multiple levels. To determine the best trade-off between parallelism and redundancy, we need to reason on the factors that impact the latency when using an aggregation overlay.

Optimal Fan-In. The aggregation time at a level—composed of the time to receive input from the level just beneath it and the time to create the output for the level—depends on the size of the input, some set of partial results $\overline{x}$. We use $g^t(\overline{x})$ to denote the time required by $g(\overline{x})$ to aggregate input of size $|\overline{x}|$, including communication time. Aggregation on the same level of the overlay happens in parallel, so only the time of a single branch is modeled.

The optimal fan-in for an aggregation tree can intuitively be derived from the given aggregation function $g()$ deployed by considering *two* measures of function complexity [Cul15]:

(a) a measure of space complexity in the form of the R-ratio

(b) the time complexity based on $g^t()$

The next section focuses on proving optimal values for R for minimizing aggregation time in several scenarios of (b). However, somewhat skipping forward and maintaining the bigger picture, these proofs will be based on an idealized setting in order to stay tractable. We thus have to adapt these heuristics in several ways for application in ROME (see Table 4.4).

Fractions. The first consideration removes fractional fan-ins. The equations in the theoretical model assumes all variables are continuous; however aggregation must use discrete inputs in real-world. Hence, in practice, a system as to round up (or down) the fan-in F derived from the model, *e.g.*, considering the ceiling $\lceil F \rceil$. For instance, applications such as top-k sorting on pre-partitioned data which output the same size as one input, *i.e.* $R = 1$, use a fan-in of 3, whilst the theory would insist on a fan-in of e.

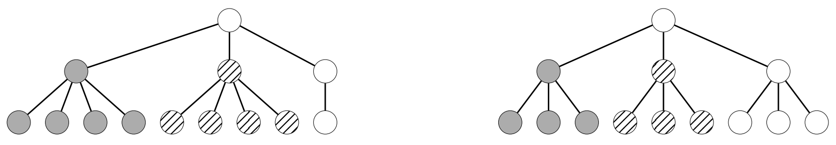
Values in the range $1 < R < n$ are likewise rounded to the smallest fan-in with the same height as the suggested ideal. We round up because rounding down might increase the height of the overlay.

Time Complexity. We also omit the time complexities of aggregation functions (sub-linear, linear, or super-linear) given that these variables, while necessary for mathematical rigor, do not sensibly impact the calculated fan-in for any of the scenarios with *proven* optimal values; for unproven cases we observe that similarly the time complexity of practical aggregation functions have no sensible effect in practice.

4.3.4 Mapping Stage

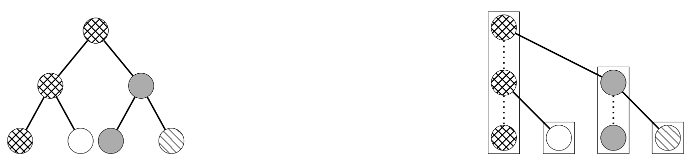
Further heuristics are required in practice to deal with a non-ideal setting, *e.g.*, to deal with non-full trees, and to effectively map a conceptual overlay to an actual application topology.

Balancing Mechanism. A theoretical model can easily assume that overlay trees are perfectly balanced using continuous variables. This often requires fractional tree heights. Applying fan-ins obtained after rounding blindly creates trees where some nodes have more children than others; thus the coarse-grained heuristics usually result in unbalanced trees which can be tuned further.



(a) Unbalanced overlay **(b)** Balanced overlay

Figure 4.5: The effect of the balancing mechanism.



(a) Logical overlay **(b)** Physical overlay

Figure 4.6: Reusing workers to reduce resources and latency.

Figure 4.5 shows a simple example of imbalance skewing performance. In Figure 4.5a, the model chooses a fan-in of 4 and expects the height of the overlay to be $\log_4 9 \approx 1.58$. The actual height of the associated overlay is 2. There are also nodes with fewer than 4 children, creating a performance skew between branches. Figure 4.5b also has a height of

2. However, the lowest level of the longest branch has a fan-in of 3 instead of 4. Because the input size is reduced the corresponding node will run faster, and there is no increase in height to offset this. Thus the latency of the slowest branch is decreased.

To analyze this more formally, let us consider an aggregation function $g\left(x_1 \dots x_\alpha\right)$. Let c be the amount of time to process one input to the aggregation. If $g\left(x_1 \dots x_\alpha\right)$ is linear on the size of its combined inputs, the latency of a single run of the function is simply $c\,\alpha$. For an unbalanced tree using a heuristic fan-in α, we model the branch with the highest latency, *i.e.* with most input. This branch has ϵ levels using α, and $\left(\lceil \log_\alpha n \rceil\right) - \epsilon$ levels with a fan-in of no more than $\alpha - 1$. Consequently, a total time for the branch can be estimated as follows:

$$\sum_{k=0}^{\epsilon} R^k c\alpha + \sum_{k=\epsilon+1}^{\lceil \log_\alpha n \rceil - 1} R^k c\left(\alpha - 1\right).$$

Note that aggregation only makes sense for $\alpha \geq 2$, which implies this formulation holds for $R \geq 1$. When $R > 1$, the formulation simplifies to

$$c\left(\alpha \frac{R^{\epsilon+1} - 1}{R - 1} + \left(\alpha - 1\right)\left(\frac{R^{\lceil \log_\alpha n \rceil} - R^{\epsilon+1}}{R - 1}\right)\right).$$

Also, when $R = 1$, it simplifies to

$$c\left(\epsilon + 1 + \lceil \log_\alpha\left(n\right) \rceil \left(\alpha - 1\right)\right).$$

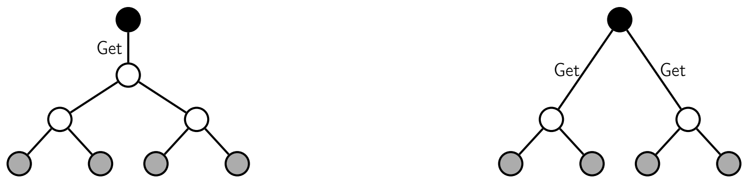
These equations are minimal when ϵ is minimized, *i.e.* when the entire path uses the smaller fan-in.

As long as imbalance remains, we can apply this logic inductively to decrease the fan-in of a node on whichever is the highest latency branch until there are no nodes which have a different number of children than other nodes at the same height. This inductive process creates as balanced an aggregation overlay as possible for a given height and number of leaf nodes.

ROME thus implements an explicit balancing mechanism. The system first finds the non-fractional height of a tree using the heuristic. For a heuristically determined fan-in α, this is simply $\lceil \log_\alpha n \rceil$. By our inductive reasoning, ROME finds the smallest fan-in which creates an overlay with the same height as the heuristic. Thus the balancing mechanism uses the equation $\lceil \log n / \lceil \log_\alpha n \rceil \rceil$ as the practical fan-in where α is the value returned by the original heuristic. Observe that this only affects $1 \leq R < n$; there is no fan-in less than 2, and any fan-in less than n results in greater tree height.

Colocation. Data-intensive aggregations can spend a significant amount of time just sending serialized objects over the network. A natural heuristic for ROME is to colocate distinct aggregation nodes which communicate with each other at the same worker as long as this does not create resource contention. For example, consider the aggregation tree with 4 leaves and fan-in of 2 in Figure 4.6a. Since aggregation at different levels does not run concurrently, workers can host nodes along a single branch without resource contention. Nodes of the same color can be harmlessly colocated on the same worker machine. In addition to reducing the number of workers required for an overlay, using RAM for intra-worker communication (dashed lines) instead of the network as for inter-worker communications (solid lines) reduces communication latency. Figure 4.6b shows an optimized physical overlay where boxes represent physical machines. Such optimization reduces the communication time of each parent by a factor of $\frac{F-1}{F}$.

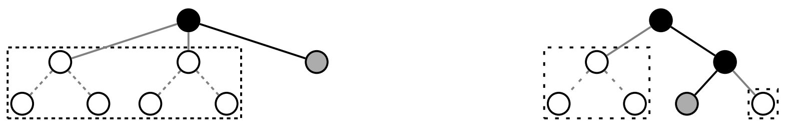
Root Node Bypass. When ROME is integrated into a third-party system we can improve latency by running the final aggregation in that system instead of in ROME. The aggregation is the same, in fan-in and result, yet we save one overlay level and thus networking.



(a) Traditional overlay **(b)** Optimized overlay

Figure 4.7: Final aggregation in ROME vs a third-party system. Black nodes produce data. Uncolored nodes are ROME workers. Gray nodes are third-party system workers storing results.

Figure 4.7 shows an aggregation performed in this manner. The ROME nodes run most of the aggregation. The results from the level beneath the root are sent to a component of the third-party system which acts as the root of the overlay and completes the aggregation. Note that in the case of $R \geq n$, which means a fan-in of n, the ROME workers are not used at all. In this case the overlay contains no ROME nodes, and the entire aggregation is completed in the third-party system.



(a) Append-only update **(b)** Infrequent updates

Figure 4.8: Minimizing reaggregation. Gray nodes have new or updated data. Black nodes perform (re-) aggregations. Results from boxed nodes are obtained from caches.

Incremental Computations. Many big data applications deal with dynamic datasets, meaning that new partitions may be added, or existing partitions may have their data changed. Aggregating from scratch upon changes in these cases is not necessary, and highly inefficient. ROME thus allows users to specify two ways for efficiently handling incremental data:

Append-only updates: Append-only data is the norm for some applications and filesystems [GGL03; Shv+10]. In this case, new data is aggregated directly with the results cached at the root of the original overlay. Figure 4.8a illustrates it. The new data in the gray node is aggregated with the result from the unshaded nodes, stored in the black node. This is no more work than the black node would have done during reaggregation anyway, and the rest of the overlay is not needed, freeing up resources and reducing latency. This refined approach can be naturally extended to adding multiple new nodes at once, creating a new overlay where one leaf is the old root node and other leaves compute the new incoming data.

Infrequent updates: In this scenario, some existing partitions of the dataset change over time. In this case, the overlay will be kept alive by ROME after the final aggregation, and partial results remain in an in-memory cache managed by each worker. When a new version of a partition reaches a worker, only the aggregation along the path to the root is rerun, since other results are unaffected. Figure 4.8b shows an example where only a single partition (gray node) is changed: only black nodes perform reaggregation.

Table 4.4 summarizes the parameters of the aggregation overlay ROME uses when running with autodetection. Next we discuss optimality of these parameters.

4.4 ROME System

We present our ROME system for optimized compute-aggregate task processing leveraging the heuristics presented in the previous sections. We focus on its architecture, API, fault tolerance support, and integration into general-purpose data analytics frameworks.

4.4.1 System Architecture

Figure 4.9 shows the architecture of ROME, which is implemented 6K lines of Java code. There are two core components inside ROME: *(i) workers* that are deployed on all nodes, and *(ii)* a *controller* that coordinates workers. A full processing environment also requires an *invoker*, a set of *producers*, and a *consumer*. These can be implemented inside ROME, but will typically reside in a third-party framework. More precisely the complete set of components/component types in ROME with their respective duties is as follows:

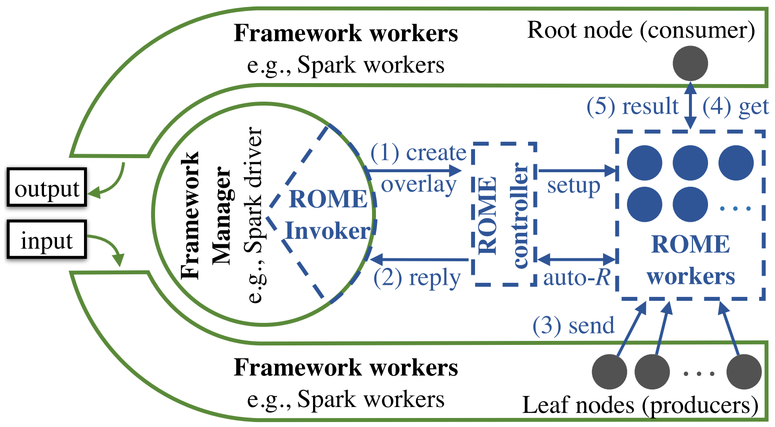


Figure 4.9: Architecture of ROME.

Workers: aggregate data received from other nodes and send the results to their parent in the aggregation tree. If requested, results are stored inside an in-memory cache to avoid repetitive aggregations.

Controller: directs the workers in creating and maintaining the overlay, tracks the status of each worker, and repairs any active aggregation overlay and restarts aggregation as needed if a worker leaves the system (or fails, see Section 4.4.3).

Invoker: a client which interacts with the ROME controller to initialize the overlay, and relays relevant data between a third-party analytics system and ROME. This role is typically played by the "job manager" of the former system.

Producers: feed the data into the leaf nodes of the aggregation overlay. The data may be read directly from a local or distributed file system, but typically producers are worker components of a third-party system.

Consumer: receives the final aggregation result. This role is typically played by a worker of the third-party system.

```
interface Accumulator<T> extends Externalizable {
    T get();
    void add(List<Accumulator<T>> list);
}
```

Listing 4.1: `Accumulator` interface (`public` visibility).

4.4.2 API

Because we assume aggregation is associative we need a standard interface to link up the aggregating nodes. The `Accumulator` interface is shown in Listing 4.1. There are only two necessary functions.

1. `add()` simply allows an input to be added or replaced. When a producer or worker lower in the overlay calls `add()` the data is placed in the worker. If data already exists for that child the existing data is replaced. If after the call there is data from every child available aggregation at that worker begins.
2. `get()` retrieves the result. A communication manager on each worker thus fetches ("`get()`s") results from its node and sends them to its parent.

Each aggregation requires a separate `Accumulator` implementation. We implemented several types of `Accumulator`s during our experimentation. Table 4.5 shows how little implementation complexity they require. The entire implementation of an `Accumulator` is often less than 100 lines of code.

Accumulator	**Description**	**Lines of code**	R
`MergeLists`	Merge sorted lists into a list of all elements in sorted order.	79	n
`TopKSort`	Output a sorted list of the k highest scoring elements from sorted input.	71	1 on pre-partitioned, ≥ 1 on arbitrary data
`LCS`	Output longest substring, which is contained in all sequences.	62 (without generalized suffix tree)	< 1
`WordCount`	Take mappings of `key`$\mapsto$`int` and combine the counts of same keys.	114	≥ 1
`SVM+SGD`	Iterative gradient descent on mini-batches.	118	1

Table 4.5: Examples of `Accumulator` implementations.

Table 4.6 outlines our simple API, and Figure 4.9 shows where in the workflow the API calls occur. An invoker calls `initialize ()` with a list of nodes ROME can use for the workers and controller. To start a new aggregation, the invoker calls the `createOverlay ()` procedure (step 1 in Figure 4.9) with an ID to uniquely identify the overlay. This function sends R and a number of leaf nodes to the controller. A flag specifies whether to enable incremental computations (see Section 4.3.4). If there are not enough available nodes, the controller returns an error; otherwise it sets up an overlay and replies (step 2) to the invoker with a list of ROME workers where producers should send their data and the worker which is the overlay root. The overlay is maintained until the invoker calls `releaseOverlay ()`. At that point workers drop their connections and release any cached partial results.

A leaf node forwards its local compute results to the assigned ROME worker by calling the `send()` procedure (step 3). Each ROME worker independently merges the received `Accumulators` from its children and then sends the result to its parent in the aggregation tree.

The node which receives the final result of the aggregation (the consumer) invokes `get()` (step 4). When the aggregation is completed by the root, the result is returned (step 5).

Note that to request autodetection of R, when calling `createOverlay ()`, an invoker simply specifies a negative R. Since ROME will not build the full aggregation tree, the reply received by the invoker from the controller will contain an invalid root node. When the consumer invokes `get()`, our framework transparently retrieves the actual root from the controller to obtain the aggregation result.

Signature	**Action**	**Return value**
`initialize (nodes)`	Setup	–
`createOverlay (id, R, n, flags)`	Overlay configured	List of nodes to send data *and* root node
`send(id, Accumulator , node)`	`Accumulator` sent to `node`	–
`get(id, root)`	Get result from `root`	Aggregation result
`releaseOverlay (id)`	Overlay dissolved	–

Table 4.6: Core ROME API.

4.4.3 Fault Tolerance

Our fault recovery strategy [Kle17] is similar to that of Spark [Zah+12]. We maintain data from unaffected portions of the overlay and only recompute what was lost. For some component recoveries, we rely on the fault tolerance mechanism of the third-party system without hampering safety or liveness. Below we discuss the process for each component:

Controller: Zookeeper [Hun+10] can maintain the controller's state and make it fault tolerant. Heartbeats from workers can be rerouted with techniques similar as in Heron [Kul+15] to further reduce load if scalability is an issue. In our deployments, the controller load was low enough that this was not necessary.

Invoker: If a failure happens during `createOverlay ()`, the controller aborts the overlay. Otherwise, the new instance of the invoker can recover details about a previously created overlay using the unique overlay identifier used in creation.

Producer: The responsibility for producers remains with the invoking system. If a producer fails after calling `send()`, it is presumably restarted, and calls `send()` again.

The second call is ignored unless the overlay is configured to accept incremental changes. If so, the new `send()` is treated like new data and data is reaggregated along that branch.

Consumer: Likewise the third-party system is responsible for recovering consumers. Any consumer wishing to receive the aggregation result can invoke `get()`. This can be done multiple times (it is idempotent) by different consumers as long as `releaseOverlay ()` has not been invoked.

Worker: Workers send heartbeat messages to the controller. They also notify the controller if another worker is not responding to attempts to send data along the overlay. Upon suspecting a worker failure, the controller creates a new worker at an unused node. It also notifies the failed worker's parent and children, and sends them the address of the new worker. If a worker fails after it sends all its results to its parent, the new worker (and its children) do not need to perform any additional task for recovery. Otherwise, the children need to (re)send their results to the new worker. Observe that if a child of a failed worker misses a portion of the result in its memory, it needs to recursively ask for the corresponding portions from its own children. ROME has no direct hooks into third-party systems. Thus in the case of producers part of such a failure, ROME kills the processes at the end of the associated branches. This forces them to restart and re-`send()` their data. If no unused nodes are available, ROME returns an error to the invoker.

4.4.4 Integrating ROME

We designed ROME to be easy to integrate with popular more generic data analytics systems. Developers should be able to accelerate their system when performing aggregations with minimal implementation effort. In this section, we describe how we have integrated ROME in Apache Spark [Apa14] 2.4 and Apache Flink [Apa11] 1.1. We chose a more stable version of Flink in favor of some benchmarks, however, we adapted a recent version of Spark to benefit from recent updates on `treeReduce`. For both systems, 5k lines of Java code were needed for successful integration.

Spark. We extend Spark's API with a `reduceWithROME ()` operator. Besides the aggregation function, the user can also provide the output to input ratio R. If R is unknown to the user, a negative value can be passed in its stead.

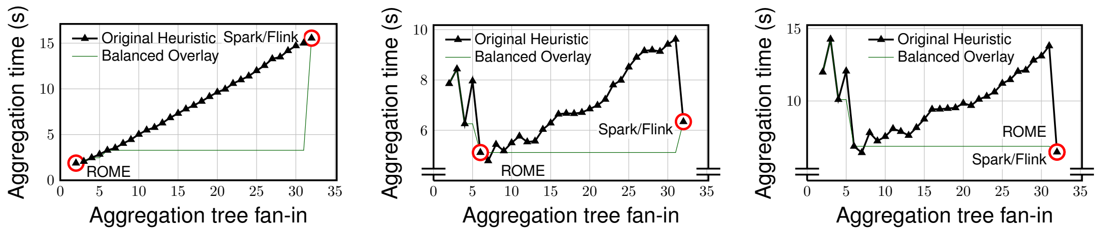
As discussed in Section 4.4.2, aggregators in ROME implement the `Accumulator` interface. Our class `SparkAccumulator` wraps the aggregation function provided by a user and transparently reuses several utilities supplied by Spark (*e.g.*, serializers). To execute `reduceWithROME ()`, the following changes to Spark were also required: *(i)* during the submission of a job using `reduceWithROME ()`, the Spark driver invokes `createOverlay ()`; *(ii)* upon executing the job, each Spark worker wraps the local data and the aggregation function into a `SparkAccumulator` and sends it to ROME; *(iii)* the Spark driver invokes `get()` as many times as needed to get the partial results from ROME and then does the final aggregation.

Flink. We implement a `reduceWithROME()` method to extend the `ReduceFunction` interface. The method is parameterized by the ratio R. Again, a negative value can be passed to indicate an unknown R. We also support a variant of `ReduceFunction` called `GroupReduceFunction`, that runs an aggregation over an object *list* instead of a single pair.

As with Spark, integrating ROME with Flink requires little effort. We provide a `FlinkAccumulator` to wrap the `ReduceFunction` provided by the user and some Flink-specific utilities. The following changes are also made to Flink: *(i)* the Flink job manager calls the `createOverlay()` procedure if `reduceWithROME()` is used; *(ii)* each Flink worker forwards a `FlinkAccumulator` object with the results of local computation on its data partition to ROME using the `send()` procedure; *(iii)* the Flink worker elected by the job manager for performing the final aggregation retrieves the partial results from ROME via `get()`.

4.5 Overlay Evaluation

First we evaluated the accuracy of our heuristics against one-size-fits-all overlays with simulated workloads on m3.medium nodes started from a single image. All nodes were provided on demand, so no network location or locality information was available. For the overlay evaluation we measured aggregation latency in isolation from computation. To that end aggregation was delayed until all leaves completed computation. The controller then initiated aggregation and timed from that point until the root node reported completion.



(a) Varying fan-ins, $R = \frac{1}{n}$ **(b)** Varying fan-ins, $R = \sqrt{n}$ **(c)** Varying fan-ins, $R = n$

Figure 4.10: Overlay comparison for various values of R.

4.5.1 Varying Fan-in

Our first experiments verified our heuristics and balancing mechanism using 32 leaf nodes. The compute phase generated a set number of random integers. Aggregators generated a list of random numbers proportional to the size of their inputs and pruned the list as necessary for multiple values of R. We show both balanced and naïve versions of the heuristic. The overlays using the original heuristics assigned children to an aggregator until the prescribed fan-in was met, then continued with the next node in a left-to-right fashion. Our balanced overlays apply the balancing formula to the given fan-in to get a fan-in which balances the overlay, then construct the appropriate overlay. This means there were often large jumps in performance when the chosen fan-in crosses a threshold changing the height of the overlay.

Figure 4.10 shows the average results from this experiment over 25 runs. We circle the performance of two overlays in each graph to highlight the comparison of our performance to the overlay of fan-in n commonly seen in practice. The second overlay is unavoidable for total aggregation problems when aggregating by partition, such as with Spark without `treeReduce` or Flink as described in Section 4.2.3.

For both $R = \frac{1}{n}$ (Figure 4.10a) and $R = n$ (Figure 4.10c) ROME's heuristics correctly chose the fastest aggregation overlay. This is especially visible with $R = \frac{1}{n}$ as the fastest overlay outperformed the slowest overlay of a single aggregation, used in frameworks aggregating by partition, by 86%.

When $R = \sqrt{n}$ (Figure 4.10b) there was an overlay outperforming our chosen one by 6%. This very small difference happened in one overlay adjacent to the chosen one, suggesting the heuristics are successful at finding a nearly optimal overlay.

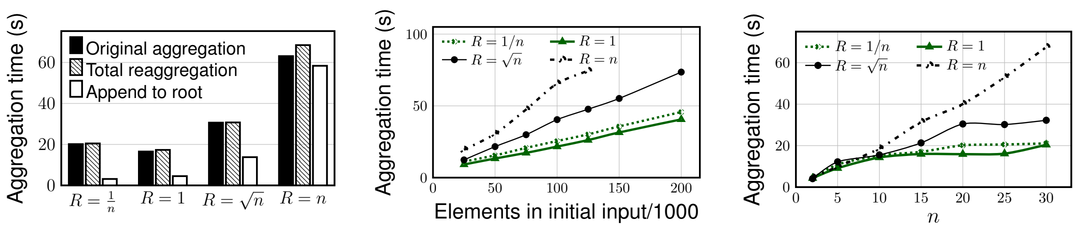
The difference between the balanced and unbalanced overlays is most obvious in the range of $[7, 31]$, where the unbalanced overlays become significantly more skewed. In this range all the unbalanced overlays diverged from the balanced ones. This is most notable in the case of $R = n$, when the idealized model [Cul+15] monotonically decreases because it allows fractional overlay height. Given that the trend increases in that range in practice and that the balanced overlays thus outperformed the unbalanced ones we can assert that our balancing mechanism effectively improves performance.

4.5.2 Append-only Updates

Next, we tested our append-only update technique by appending a single node to an existing 25-node overlay. Figure 4.11a shows the difference between creating a new overlay with 26 leaves vs aggregating the prior output with the new data.

Complete reaggregation took slightly longer than the original aggregation. Aggregating the previous results with the new data was faster in all cases, although the amount of savings depended on R, *i.e.* on the size of the output of the previous data. With $R = \frac{1}{n}$, aggregating the prior output with the new data directly was 85% faster than using a new overlay. With $R = n$ the benefit was a 15% speed-up.

If the overlay was reused each time, the second approach would require time equal to the original aggregation time plus the time to aggregate those new results with the data from the new node. In all cases that combined time was greater than the cost of using a new overlay. Thus it makes sense to use this approach only for append-only data.



(a) Strategies for new data. **(b)** Varying input size. **(c)** Varying leaf node #.

Figure 4.11: Aggregation latency when changing input data or number of leaf nodes.

4.5.3 Input Size and Distribution

In order to fully understand the performance of ROME, we next considered the effects of the number of leaf nodes and the input data size. These are environmental parameters imposed by the user and problem, and are thus not configurable by ROME. Nonetheless, both of course significantly affect performance. For each of the chosen values of R, we varied the number of leaves n from 3 to 30 and, independently, the input size from 25000 elements per leaf to 200000.

Figure 4.11b shows the effect of changing the input size across 25 nodes using the same R values as the earlier experiments. The final two points for $R = n$ are omitted as memory requirements at the root exceeded the available RAM, and latency increases dramatically with disk accesses. With the exception of the omitted points, latency is very predictably correlated with the input size.

Figure 4.11c shows how the number of leaves affects the latency, with the input size per leaf remaining the same. Thus a cluster with 20 leaves is processing twice the total amount of input data as a cluster of 10 leaves. Observe that smaller values of R, which have smaller fan-ins and thus higher parallelism, are less sensitive to changes in the number of nodes. The latency increased in a sublinear fashion relative to the number of leaves, suggesting that the job should be parallelized to a larger degree when possible.

For $R = n$, the relationship is predictably linear. This is not surprising given that all aggregation takes place at a single node, which means that we were increasing the workload of a single machine by a factor equal to the change in fan-in.

4.6 Integrated Evaluation

In this section we evaluate compare performance of unmodified versions of Spark and Flink to versions integrated with ROME. In the unmodified systems, each worker node was placed on a separate AWS instance and used the whole available main memory.

In the case of Spark+ROME (respectively Flink+ROME), a ROME node is colocated with a Spark (or Flink) worker node, and uses half of the instance's memory. We differ-

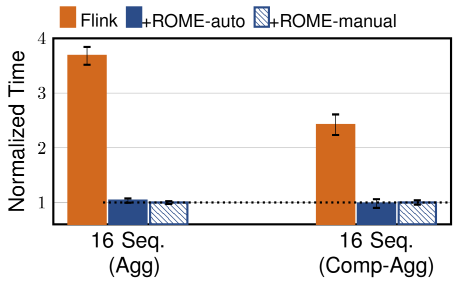


Figure 4.12: LCS with Flink.

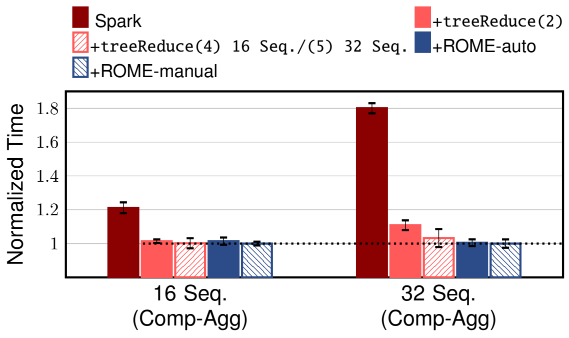


Figure 4.13: LCS with Spark.

entiate between ROME-manual and ROME-auto. The first represents a user providing the correct value of R, while the second represents a user providing a negative value, meaning our system autodetects R with a single layer of aggregation (see Section 4.3.2).

Flink and Spark use a fixed fan-in of the number of worker nodes. We also compared with Spark using `treeReduce` to specify tree depth. Since `treeReduce` uses a depth of 2 as default, but takes a user-provided depth as an upper bound, we show results for both `treeReduce` with default depth and `treeReduce` (d) with d equivalent to the optimal (but unbalanced) overlay of ROME.

We chose three experiments exploring behavior in different ranges of R. The first experiment considers a longest common substring problem where $R < 1$. The second experiment considers a top-k sort on not pre-partitioned data comprising two aggregation phases, with $n \geq R \geq 1$ and $R = 1$ respectively. The third experiment runs an iterative algorithm—a gradient descent with mini-batches with the goal of learning a classification problem. Each iteration runs an aggregation phase with $R = 1$.

We run each experiment 10 times and report the average and standard deviations as error bars.

4.6.1 Longest Common Substring

We ran the longest common substring (LCS) problem on a DNA dataset [The]. Worker nodes were assigned unique DNA sequences from a section of the genome. Each worker built an suffix tree containing all the substrings contained within its sequence. This data structure was then compared with those from other workers to remove substrings which are not contained in all sequences. Since the output size was smaller than the input, the fan-in for ROME is 2.

Because of the amount of computing power and the memory requirement to aggregate at a single node with Flink and Spark, we chose m3.2xlarge nodes on AWS, which have 8 virtual CPUs, 30 GB of RAM, and 2 SSDs for permanent storage. Since the suffix trees necessary for aggregation are about 700 MB per sequence, Flink is unable to process the results from 32 sequences, even with 30 GB of RAM. We thus run Flink with 16 sequences, but Spark with 16 and 32 sequences.

We run three variants of this test. The first, labeled “16 Seq. (Agg)”, reads the precomputed data structures from disk. The time to read 700 MB from an SSD is minimal, so this essentially isolates the aggregation phase for sequences without needing to modify Flink or Spark with a stopping mechanism between compute and aggregate to synchronize a timer. The second test, “16 Seq. (Comp-Agg)”, represents the entire compute-aggregate workload for 16 sequences including building the LCS structures online from the DNA sequence. Similarly we have a “32 Seq. (Comp-Agg)” test which runs the entire compute-aggregate job with 32 sequences on 32 nodes. The Comp-Agg variants show how much effect the improvement on aggregation time has on the total job latency. The times in Figure 4.12 and Figure 4.13 are all normalized to the time that ROME-manual (fastest system) takes.

Figure 4.12 shows the results when running Flink. Flink+ROME improves latency by a factor of 3.68 over Flink when reading the precomputed data structures (“16 Seq. Agg”). Computing the 16 LCS structures online increases job latency, which lowers the benefit of running Flink+ROME—a speedup of 2.42 over Flink when considering the entire compute-aggregate workload for 16 sequences (“16 Seq. Comp-Agg”). We see that if a user provides R to Flink+ROME brings almost no benefit over ROME to autodetect R, since autodetection uses the optimal R anyway during the probing phase.

Figure 4.13 shows the LCS problem using Spark with 16 and 32 DNA sequences. The 32 Seq. (Comp-Agg) test taxes the standard Spark system. In part because the memory requirement at the aggregating node is very close to the total allotment, the execution takes 260% of the time of the comp-agg job on 16 sequences. The fastest configuration is Spark+ROME when R is provided, reducing total time for the entire compute-workload by a speedup of 21% and 80% over vanilla Spark when working on 16 and 32 sequences, respectively. The overhead of automatically detecting R is less than 1%, simply because autodetection runs the initial aggregation phase with the optimal R.

Using `treeReduce` for the 16 sequences with the manually calculated ideal height (4) shows same performance as ROME when running with autodetection. However, `treeReduce` becomes less efficient when running the problem of 32 sequences (using optimal height 5), adding an overhead of almost 5% compared with ROME with autodetection. Spark `treeReduce` with the default configuration (height 2) shows similar performance for the small problem of 16 sequences, but adds an penalty of 11% over ROME with autodetection when running the problem of 32 sequences.

The sublinear relationship between latency and the number of machines matches the earlier results and shows ideal overlays are more important as data sizes grow.

4.6.2 Top-k Sort

Next we analyzed Wikipedia page accesses [Wik] to find the top-k most visited pages. We distributed 35.6 GB of relevant data, part of a much larger dataset available for more

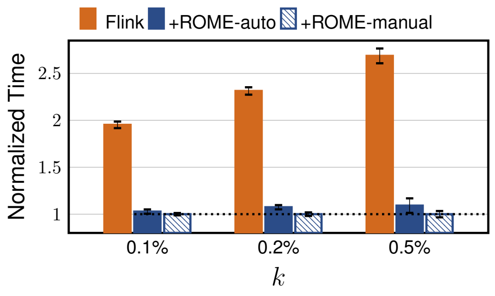


Figure 4.14: Top-k with Flink.

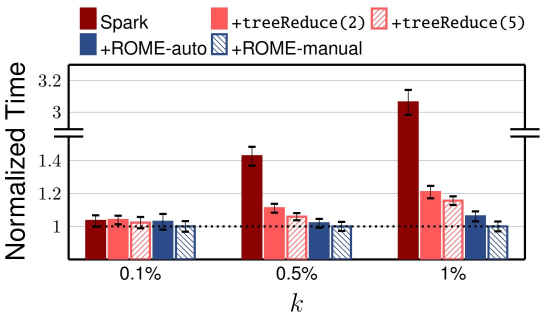


Figure 4.15: Top-k with Spark.

expressive querying, across 32 i3.large workers. The benchmark calculated each page's score in a first aggregation phase (R-ratio s.t. $n \geq R \geq 1$). Then, in a second aggregation phase each compute task found the k most visited pages in its partition. Aggregation involved finding the k highest scoring pages from all those, so the R-ratio of the second phase is 1. This benchmark stresses the R autodetection of ROME since ROME uses a non-optimal temporary fan-in while running the first intermediate aggregation.

Figure 4.14 shows the completion time for 3 different values of k. We normalize the values in this graph to the time Flink+ROME takes when R is provided to determine the pages with the top k% most accessed pages. When k is 0.1%, autodetecting R adds less than 3% overhead. Running vanilla Flink requires almost 2 times as long as running Flink+ROME.

Increasing k to 0.2% minimally impacts the compute phase, but doubles the aggregation load. As a result, the Flink vanilla deployment adds a slowdown of factor 2.3 compared with Flink+ROME with manual configured R. Flink+ROME autodetection overhead grows to 7%.

When we increase k to 0.5%, garbage collection comprises a third of the resulting runtime for vanilla Flink. In contrast, garbage collection does not affect either Flink+ROME setup despite the reduced RAM allocation to each workers. As a result ROME shows a speedup of almost 2.7 over vanilla Flink, with only 9% overhead to autodetect R.

Figure 4.15 shows the Top-K sort problem when using Spark with different configurations for k. Spark shows more efficient resource usage compared with Flink, so we run the problem with larger values of k. We normalized the values in this graph to the time ROME takes with manual configured R.

For the smallest k (0.1) all systems perform similarly (Spark vanilla adds a penalty of 4% over ROME), since the aggregation load is very small. Increasing k to 0.5% leads to five times higher aggregation load, which unveils differences in performance. ROME with autodetection adds 1.9% overhead compared with manual configuration of ROME, but Spark vanilla needs 43% more time. With Spark's default depth of 2 `treeReduce` is 11% slower than ROME; using `treeReduce` with the same depth as ROME is 6% slower.

When k was increased to 1% the aggregation overlay became even more important. Autodetection of ROME adds 6% overhead compared with a manually configured ROME. `treeReduce` with manually set optimal tree depth (same as ROME) adds 16%, whilst default `treeReduce` adds 21%. Vanilla Spark took 3× more than ROME.

This experiment also showed a worst case scenario for ROME-auto during the first experiment phase with $n \geq R \geq 1$. When R is autodetected there must be a level of aggregation to learn R, adding overhead in this case.

4.6.3 Gradient Descent

Gradient descent (GD) is an iterative method for optimizing a differentiable objective function, by updating the parameters of the function in the opposite direction of the gradient of the function. GDs are widely used by many data-intensive machine learning tasks including training of neural networks [Rec+11]. Mini-batch GD is a variant which uses a small (randomly sampled) subset of the data to perform the GD update (instead of the complete dataset in each iteration). Spark MLlib[1] provides a GD implementation supporting mini-batching. In each iteration, a mini-batch uses an aggregation task with a `treeAggregate` overlay (used internally by `treeReduce`) for computing and summing up the subgradients, hence an aggregation with R=1.

In this benchmark we trained a Support Vector Machine (SVM) using Spark's MLlib (`SVMWithSGD`) to perform binary classification. We used a 21.4 GB large data set from KDD Cup 2012 [SIG], which holds feature vectors and corresponding classification; in total 55 million binary features [Jua+16].

We compare Spark using the default implementation of MLlib's `SVMWithSGD` (using `treeAggregate`) and a modified version using ROME as an aggregation overlay using 32 i3.xlarge workers. The ROME aggregation function uses the same logic as the default `SVMWithSGD` function, hence only 118 lines of code were changed for integrating the ROME overlay (mostly accounting for wrapping/boilerplate code). We set the number of iterations (from 1 to 100) used by SVM to learn and ignore earlier convergence, so that all iterations run. Using ROME for aggregation does not change PR/ROC (quality) so we focus again on total running time.

Figure 4.16 shows the time for varying number of iterations and Figure 4.17 the savings of ROME over Spark. We normalized the values in this graph to the time ROME needs for 1 iteration. When running a single iteration, Spark is 11% slower than ROME, even though ROME needs to set up a larger aggregation overlay. With increasing number of iterations, the advantage of ROME over Spark increases. At 10 iterations ROME runs 28% faster, at 20 iterations 40% faster, and at 100 iterations 50% faster than Spark. The trend indicates higher savings with more iterations.

[1] `https://spark.apache.org/mllib/`

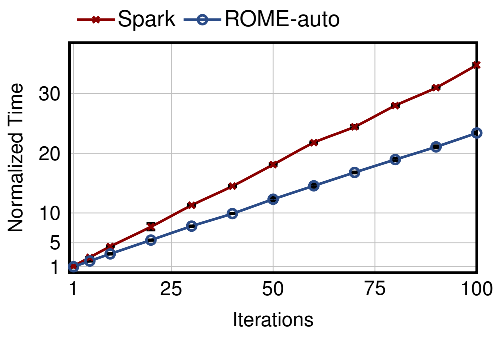


Figure 4.16: Gradient descent with Spark.

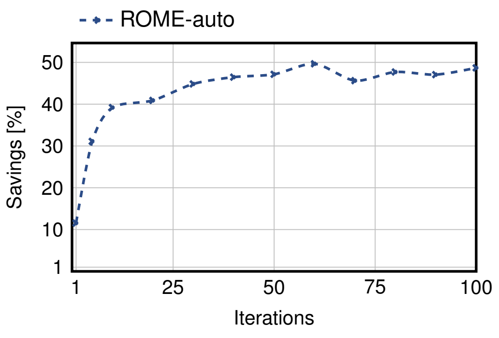


Figure 4.17: Gradient descent time savings.

Table 4.7: Network optimizations. Average and standard deviation in seconds.

	No colocation	**Colocation**
No root bypass	166.4 ± 5.1	158.3 ± 3.5
Root bypass	160.5 ± 4.1	154.2 ± 2.6

4.6.4 Parent-child Colocation and Root Node Bypass

Our next tests were for the two location-based optimizations. Since both the root node bypass and node colocation optimizations were targeted at reducing network latency by decreasing the amount of traffic, we reran the top-k experiment with $k = 1\%$. With its slightly higher network component, this test was best suited to see small but significant network latency savings. We reran the test using four configurations of Spark+ROME – with and without root node bypass, with and without parent-child colocation (see Table 4.7). The time taken with both mechanisms applied was 7.3% faster than when neither was applied. Applying only colocation or root node bypass only saved 4.9% or 3.5% respectively. The combined savings was less than the sum of the two because we could not colocate the final parent in the case of root bypass.

4.6.5 Fault Tolerance Overhead

This experiment used the top-k most visited pages of Wikipedia, and was run 31 times [Kle17]. ROME used a fan-in of 3 (see Figure 4.18). We ran the experiment on i3.large spot instances. Each of the 32 nodes contained a Spark worker and a ROME worker. The controller node ran Spark's cluster manager and ROME controller. We also deployed a Spark driver on a separate instance.

We first evaluated the overhead of providing fault tolerance in ROME as shown in Figure 4.19. This was the only experiment which ran ROME without fault tolerance. Because this experiment was solely testing ROME internal functions we show only the results with Spark. The mean execution time was around 44 seconds with and without fault tolerance. The standard deviation was around 2 seconds for both cases. This shows that ROME's fault tolerance has negligible impact.

To evaluate the recovery time under failure, we put a tier two ROME worker node along with its parent and one of its children on the same worker node, and crashed that worker node at different stages of computation, and before the second tier node finished its computation (black node in Figure 4.18). We observed that upon failure the latency increased by 50%.

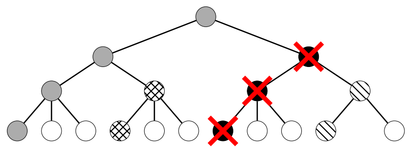


Figure 4.18: Failure affecting multiple tree levels.

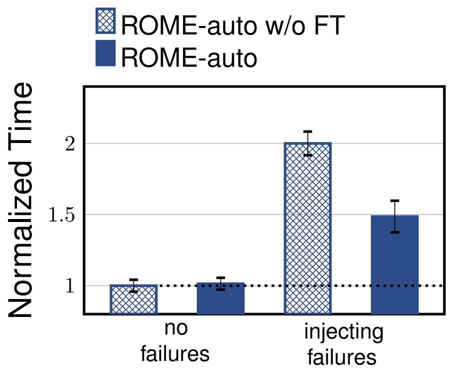


Figure 4.19: ROME fault tolerance.

4.7 Conclusions

This chapter introduced an aggregation system with runtime resource scheduling for use within data analytics frameworks or in isolation. We present ROME to construct and maintain low latency aggregation overlays. ROME chooses an overlay based on the ratio of aggregation output to one input—a ratio highly characteristic of performance of distributed overlay-based aggregation—which we call R. Our targeted reuse of nodes further decreases latency and resource requirements. We empirically show our overlays are nearly, if not actually, optimal.

We propose that R is easy to find in most practical scenarios, at least to the granularity of the ranges $R < 1$, $R = 1$, and $R \geq n$ identified by our heuristics. If such estimation is not possible the user can inform our system to compute R on-the-fly via partial aggregation with low overhead: only by 9% in the worst case in our experiments.
We integrate ROME into Flink and Spark and validate the effects of the overlay in end-to-end distributed data analytics with real world data and problems. ROME decreases latency by a factor of up to 3 over unmodified systems that do not use tree overlays for aggregation, and up to 21% and 16% over a Spark deployment using `treeReduce` with default and *manually tuned* configurations, respectively. When running iterative algorithms with many aggregation phases, ROME decreases total runtime, *e.g.*, by 50% over the fastest Spark configuration after 100 iterations of gradient descent. These im-

provements are despite sharing the RAM allocation between ROME and Spark/Flink. We demonstrated in this chapter that fitting the application to the resources at runtime helps making distributed applications more efficient and flexible to fit to available deployment targets. ROME performs runtime adaption of an application's aggregation plan to fit available resources and actual workload characteristics, with the overall goal of lower aggregation latency and higher resource efficiency. This enables aggregation systems to leverage most of the set of available resources, instead being pre-configured for a specific presumed resource setup. Using ROME makes big data aggregation system ready for a dynamic data center environment, especially for cases where spare resources are not always available.

In Chapter 6 and Chapter 7, we will discuss why runtime adaption of application to available resources is essential, especially in the presence of INC resources. With the example of INC data aggregation [Sap+17; Gra+16a; Mai+14], INC switches (required for performing INC aggregation) may not be available for all jobs, hence an application must react at runtime to the set of available resources. The next chapter discusses the second scenario of application-level scheduling, the problem of traffic scheduling of distributed service function chains.

5 STEAM: Distributed Runtime Scheduling of Service Function Chains

Chapter Outline

This chapter discusses the second scenario of application-level scheduling, the problem of traffic scheduling of distributed service function chains, and introduces two solutions called IA-MPP and STEAM, a throughput-optimal solution (IA-MPP) and a distributed heuristic (STEAM), respectively. STEAM closely matches IA-MPP in terms of throughput, and significantly outperforms (possible adaptations of) existing static or coarse-grained dynamic solutions, requiring 30%-60% less server capacity for similar or better service quality. This reduces the amount of resources in the network that need to be allocated to provide a target quality of service guarantee for serving SFCs.

The solutions in this chapter are examples for rethinking a scheduling problem as a runtime problem, to make the scheduling solution become independent of *a priori* information. As we will show with this scenario, runtime scheduling solutions are likely to perform more fine-grained scheduling decisions and are more easily capable to incorporate actual demands and available resources.

5.1 Overview

The dynamic multi-service network architecture for 5G and beyond bears a demand for SFCs provisioning mechanisms [Nat18; ZLZ19; Sat+18] that are capable of creating SFC instances and scheduling traffic through them *on the fly* [Nat18]. This requires making decisions on both **(1) placement** of service function instances (SFIs) and **(2) scheduling of traffic** through them [HP15].

Most existing works focus on the placement problem (1), deciding *where* SFIs should be deployed (*e.g.*, on which server) and *how many* resources (*e.g.*, CPU shares) should be assigned to each of them [Coh+15; Add+15; Mar+15; Mij+15; Wan+16; Kuo+16]. Figure 5.1 shows how state-of-the-art solutions tackle the SFC resource provisioning problem. These solutions perform chaining of service functions (SFs) in a mostly *static* manner, where traffic is steered through deployed SFIs in the network with load-balancing performed among them. Few *dynamic* solutions are discussed [QAS16; Era+17; Sat+18; ZLZ19; Anw+15; Pal+15], in which the deployment of these SFIs and their resource assignments are periodically adapted to changes in network traffic and topology, as required for future carrier networks. However, these solutions are still coarse-grained [Era+17] (*e.g.*, based on peak hour interval traffic demand), where the adaptation takes seconds to take effect [Pal+15], or cannot be applied in real-time due to its high complexity [ZLZ19] and the involvement of disruptive SFIs migration [Era+17]. Therefore, these proposals are not able to explore and exploit the resources that become available on the fly as a result of real-time, sudden, changes in network traffic. Yet, as we shall show, such

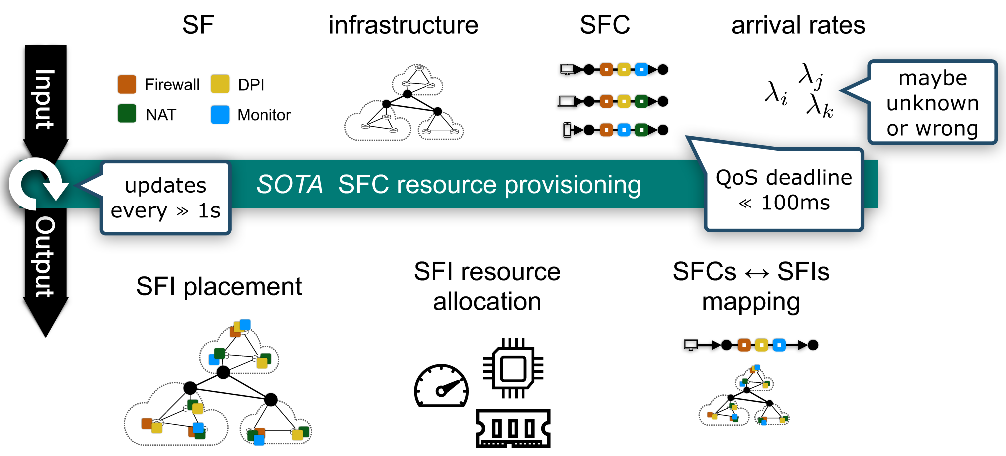


Figure 5.1: SFC resource provisioning of state-of-the-art solutions.

a fine-grained approach based on per-packet scheduling is required for achieving high resource utilization under high traffic dynamics.

Differently from previous work, IA-MPP and STEAM treat the SFC traffic scheduling problem as a *runtime* scheduling problem, as shown in Figure 5.2. The goal is to dynamically assign packets with specific processing requests, to active SFIs in the network. We assume that SFIs are already deployed on servers using any of the existing algorithms (see Section 5.1.2). However, these SFIs are not pre-assigned any resources or traffic. Our goal is to select an appropriate SFI *for each packet* and to decide on the amount of resources that should be assigned to each SFI at runtime.

5.1.1 Design Challenges

IA-MPP and STEAM face three design challenges, which we discuss in this chapter:

No *a Priori* Knowledge of Traffic Distribution. The major challenge is to quickly react to dynamic traffic conditions, without any *a priori* knowledge of traffic distribution. We characterize the SFC traffic scheduling problem with a stochastic model and show that this problem can be reduced to the scheduling problem in an SPN [Wil16]. We propose the IA-MPP for SFC scheduling, a derivation of MPP, which we show is throughput-optimal. It is also asymptotically optimal for minimizing a cost function of buffer occupancy levels in the network, providing approximate guarantees on latency. Importantly, IA-MPP and STEAM require no *a priori* information about network traffic patterns.

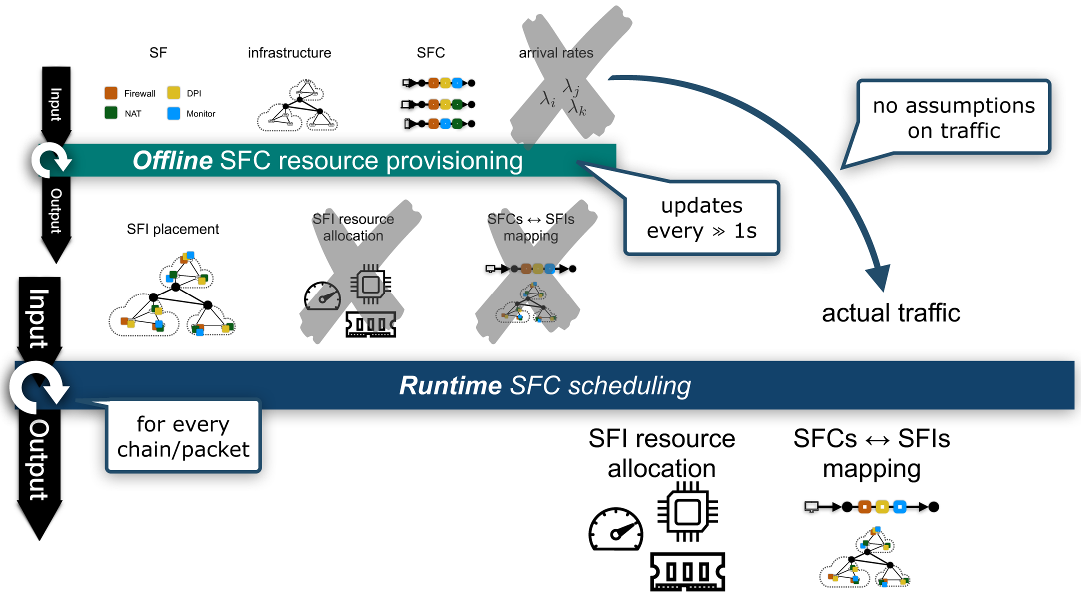


Figure 5.2: Tackling the SFC resource provisioning problem as a runtime scheduling problem.

Scheduling Without Global View of the System. In large SFC deployments, a scheduler does not have access to a global view of the system (with up to date information of all queues, buffers, and link and server load). Based on practical constraints in large deployments, we present a novel distributed variant of IA-MPP called STEAM. With STEAM, a scheduler instance is running at each SFC deployment site *using only site-local state*, performing site local optimal scheduling. In order to consider the global view when performing distributed scheduling in STEAM, STEAM has a component called SALVE which helps to perform cooperative distributed scheduling. SALVE forwards traffic to other sites before its local site gets overloaded.

Low Runtime Overhead for Packet-level Scheduling Decisions. The goal of IA-MPP and STEAM is to perform packet-level scheduling decisions without adding significant runtime latency overhead. A fine-grained approach based on per-packet scheduling is required for achieving high resource utilization under high traffic dynamics. IA-MPP's scheduling logic acts on packet level granularity. Furthermore, we show that the time complexity of IA-MPP is bounded by a linear term on the number of sites in the network. Our practical system's solution STEAM applies further optimizations, and provides mechanisms to adapt scheduling granularity from single packet-level to batches of packets. Beside our theoretical discussion of the runtime complexity, we present measurements of our STEAM prototype which achieves 10^6 to $4 * 10^6$ scheduling decisions per second (using 1 CPU core) when running in per-packet scheduling mode.

5.1.2 Related Work

In short, our work differs from all previous works related to virtualized network function (VNF) placement/scheduling in one or more of following aspects: (1) We consider *runtime* traffic scheduling *without a priori knowledge* of traffic distribution. (2) We target global optimization as a distributed scheduling problem, assuming *no complete view of the system.* (3) Scheduling decisions are made at *packet-level vs. flow-level*, making our solution more adaptive to traffic dynamics.

Some recent works [Kul+17; Kat+18; Men+18; Kat+16a] consider optimizations and scheduling at the level of a *single server or CPU core.* In particular, NFVnice is a VNF framework for CPUs that aims for fair and efficient resource allocation of chains, considering the impact of different VNFs on resource usage. Katsikas et al. [Kat+18] propose an intertwined setup of network devices and servers, allowing to reduce inter-core transfers of packets on the server and by this improving single-server VNFs throughput. Meng et al. [Men+18] split an SFC into smaller semantically equivalent VNFs, enabling reuse of parts of an SFC across others.

Many research efforts have recently targeted network-wide VNF scheduling, inside a single data center or across multiple data centers. Nevertheless, most of them consider centralized SFC/VNF scheduling, assuming a scheduler with *global knowledge of the network* and often statistical information on traffic distribution. Mechtri et al. [MGZ16] consider joint placement and scheduling of SFCs for infrastructures mapping to undirected graphs, using *a priori* knowledge of the required static bandwidth of each network flow. Similar problems are investigated by others [Coh+15; Add+15; Mar+15; Mij+15; Wan+16; Kuo+16; Anw+15]. Assuming perfect knowledge about traffic volumes, these placement/scheduling solutions can be applied only offline, or take decisions ahead of flow arrival. Qu et al. [QAS16] consider dynamic flow demands, but allow a server to run only one SF instance and a link to forward only traffic from one flow at a time. Eramo et al. [Era+17] allow traffic to change while being processed, but still require knowledge of flows' nominal and maximum traffic volumes. Anwer et al. [Anw+15] use the input and output traffic volume of all SFIs in the system to dynamically update the routing of SFC.

Different optimization objectives have been considered for VNF placement and scheduling [MK16; LRS18; Fei+18; Sat+18; Xia+19]. Marotta et al. [MK16] tackle energy-efficiency in SFCs placement and scheduling, minimizing the number of involved switches and servers. Caggiani et al. [LRS18] focus on internal switching of VNFs on a server to reduce the total switching overhead of an SFC. However, networking cost (*e.g.*, latency) is neglected. Fei et al. [Fei+18] consider demand prediction for VNFs, based on which VNFs are scaled up by adding new instances and traffic is split among instances, with the objective of minimizing prediction error and system configuration cost. Yikai et al. [Xia+19] uses deep learning to reduce the cost of involved servers, but employs coarse-grained scheduling per SF.

Some earlier work related to fair queuing is also relevant here. Bennet and Zhang [BZ97] propose hierarchical fair queuing to provide network load balancing by scheduling packet flows over available paths. The proposed solution requires *a priori* knowledge of each flow type's share of assigned resources and arrival rates. Stoica et al. [SSZ98] use predictions of arrival rates of flow types to decide on the share of resources which should be assigned to each flow type and the corresponding link that a packet should be scheduled over. These solutions therefore fall into the same category as the other proposals mentioned above. Besides, they do not consider any chaining of SFs.

The works most closely related to ours are those of (1) Bhamara et al. [Bha+17] and (2) Satyam et al. [Sat+18]. (1) applies queuing models for servers and links in a multi-cloud environment to minimize inter-cloud traffic and response time. (2) studies VNF placement and CPU allocation for co-located VNFs in 5G networks to minimize end-to-end traffic latency. Both (1) and (2) assume *a priori* knowledge of packet arrival rates.

5.1.3 Contributions

In this chapter, we make the following contributions:

1. We introduce a model of SFC provisioning infrastructure based on the SFC RFC 7665 [HP15], and formulate the SFC runtime scheduling problem (Section 5.2).

2. We present a throughput-optimal solution called integer allocation maximum pressure policy (IA-MPP). IA-MPP transforms the SFC runtime scheduling problem to the scheduling problem in an SPN. The proposed solutions has linear time complexity (in the number of sites), where schedulers have access to each other's state (Section 5.3).

3. We introduce a distributed heuristic called multi-site cooperative IA-MPP (STEAM), where schedulers have access only to their local state and scheduling costs are amortized over batches. For cooperative scheduling among sites, STEAM provides an admission control policy called STEAM T-valve (SALVE), which helps to forward traffic to other sites before a local site gets overloaded (Section 5.4).

We evaluate our solutions (Section 5.5) based on a discrete-event packet-level simulator, showing that our solutions significantly outperform dynamic variants of existing solutions: (i) STEAM reduces the required amount of resources by 30%-60% compared with the baselines, while providing similar or even better service quality; (ii) STEAM's scheduling quality does not suffer from small batch sizes (≤ 64), making runtime scheduling feasible in practice. Furthermore, we describe a prototype implementation of STEAM and show the feasibility of running STEAM in real-time, achieving 10^6 to $4*10^6$ scheduling decisions per second (1 CPU core).

5.2 Model and Problem

In this section we introduce a comprehensive model for the runtime traffic scheduling problem for SFC. We use calligraphic fonts for sets (*e.g.*, $\mathcal{S}$), capital letters to refer to members of a set (*e.g.*, $S \in \mathcal{S}$), lower-case letters to refer to variables (*e.g.*, v), and letters with arrows, such as $\vec{z}$, to refer to vectors. For two vectors $\vec{x}$, $\vec{y}$ of the same size, $\vec{x} \times \vec{y}$ denotes the cross product of the vectors and $\vec{x} \cdot \vec{y}$ denotes their dot product. Table 5.1 summarizes major notation used.

5.2.1 System Model

Infrastructure. We consider an architecture similar to the one proposed in RFC 7665 [HP15]. Our network consists of geographically distributed sites, each of which holds servers for running SFIs as depicted in Figure 5.3. Attached to each site is a set of service function forwarders (SFFs), which are responsible for forwarding traffic within their site and among sites. We model the network of SFFs across sites with a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, with $\mathcal{V}$ the set of SFFs and $\mathcal{E}$ the set of links interconnecting SFFs. For any link $E \in \mathcal{E}$, $d_E(l)$ denotes the l-th packet propagation latency, where $\{d_E(l), l \geq 1\}$ is assumed to be a sequence of i.i.d. random variables, with an average noted $\bar{d}_E$ and a finite variance. We thus consider that this propagation latency is time-varying. Moreover, we assume that network planning, as discussed in [Cis; NN10] for different use cases, is performed beforehand, as a result of which enough capacity is assigned to transmission links among SFFs.

Symbol	**Description**
$\mathcal{V}$	Set of SFFs in the network
$\bar{d}_E$	Average network delay of link $E \in \mathcal{E}$
$\mathcal{S}$	Set of servers
c_S	Processing capacity of server $S \in \mathcal{S}$
$\mathcal{F}$	Set of SFs
μ_F	Processing rate of SF F using one resource unit
$\mathcal{I}$, $\mathcal{I}_S$	Set of all SFIs and of those running on server S
w_I	Resource share of SFI I at a server
$\mathcal{B}$	Set of all buffers at all SFFs
$\vec{z}$	Vector of buffer utilization levels
$\mathcal{A}$	Set of all activities for the corresponding SPN
$\mathcal{H}(t)$	Set of all feasible allocations at time t
$\mathbf{R}$	Input-output matrix of the network
θ_l, θ_h	Thresholds used by SALVE
ϕ_b, $\phi_{w,S}$	Batch size and threshold of $s \in \mathcal{S}$ used by STEAM

Table 5.1: List of notation used for IA-MPP and STEAM.

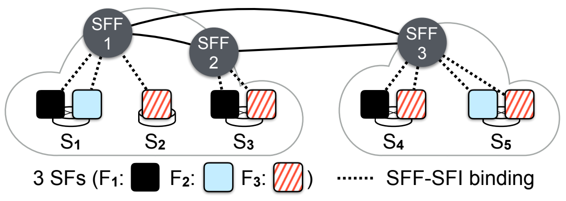


Figure 5.3: Small scenario with two sites, three SFFs, five servers, and three SFs with multiple SFIs of each.

Attached to each SFF, we have a set of servers each running SFIs. $\mathcal{S}$ denotes the set of all servers in the system. Each server $S \in \mathcal{S}$ has a total processing capacity of c_S ($c_S > 0$), and hosts at least one SFI. In addition, each SFI belongs to one of the SFFs at the same site, which means that this respective SFF considers the SFI for scheduling purposes. Servers and SFFs in a site are interconnected by a high-throughput low-latency network. For the sake of tractability, our optimal solution (Section 5.3) assumes no latency and bandwidth constraints for communication *within* a site. This assumption is relatively realistic as modern data center networks can provide ultra-low latency and full bisection bandwidth between any pair of servers [Gho+17; Han+17]. We relax the assumptions in Section 5.4.

Service Function Instances (SFIs). An SF is a piece (type) of processing logic applied to network packets, while an SFI is an instantiation of an SF deployed on a server. For simplicity, we focus on stateless SFIs, where packets from the same flow can be scheduled separately. (Stateful SFIs can be built on top of stateless SFIs using a distributed data layer [Kab+17; ARI+18].) We denote by $\mathcal{F}$ the set of all SFs in the system. Multiple SFIs of the same SF $F \in \mathcal{F}$ might be deployed in the network (see Figure 5.3 as an example). We denote by $\mathcal{I}$ the set of all running SFIs of all SFs in the network and by $\mathcal{I}_S$ the set of SFIs running on server S.

We consider that SFIs are already deployed in the network, by using any of the solutions proposed in the literature (*e.g.*, [Coh+15; Mar+15]; see also Section 5.1.2). Differently from these studies, however, the SFIs are not pre-assigned any resources or traffic. The scheduler dynamically decides where to send a packet and how many resources to assign to each SFI. Without loss of generality [Har00; DL05; DL08; Kat+18], we assume that all SFIs of the same type in the system have the same processing rate when given equal resources. We denote by μ_F the processing rate of SF $F \in \mathcal{F}$ when provided one resource unit. Thus, $k \cdot \mu_F$ is the processing rate of an SFI of type F using k units of resources. Moreover, we assume that the processing capacity of a server is shared among all co-located SFIs according to some given policy. Under such a policy, an SFI $I \in \mathcal{I}_S$ receives a share of w_I of the total capacity of server $S \in \mathcal{S}$ and the total capacity of the server is constrained by enforcing $\sum_{I \in \mathcal{I}_S} w_I \leq 1$. Furthermore, we assume that each SFI holds a local buffer to store incoming packets and applies non-preemptive execution of the corresponding SF. The assumptions indicated above are not restrictive, representing many real-world use cases [LeB10].

Network Traffic. Each SFF $V \in \mathcal{V}$ runs a classifier and a scheduler and also behaves as both ingress and egress for the network traffic. We assume that the network traffic is composed of many flows originating from different users connected to the network. The set of ingress and egress nodes of a flow, which are SFFs in $\mathcal{V}$, is determined using the source and destination addresses of the flow. We assume that there are buffers at each SFF, which store incoming packets (new packets arriving to the site via this SFF, or packets forwarded by other SFFs) before scheduling over the available resources.

Like in RFC 7665 [HP15], we assume that the *classification* of packets in the network is known. This classification is performed at the ingress node of the flow and the classification information can be embedded in the header of each packet of the flow, *e.g.*, by using network service headers (NSHs) [QEP18]. Each packet, after classification, will be assigned an SFC that the packet has to go through. The packet header maintains the processing stage of the packet, specifying by which SF in its SFC it is to be processed next. An SFC in the system is specified by an ordered set of SFs that a flow packet should be processed through, *i.e.* $C = (F_1, ..., F_k)$, $F_1, ..., F_k \in \mathcal{F}$, where k is the number of SFs on the SFC. In addition, each SFC C is given a set of QoS metrics that the handling of packets undergoing C has to conform to, which in our considered scenarios contains the end-to-end delay. $(F_1, F_2) \in C$ denotes that both F_1 and F_2 are part of C and that F_1 precedes F_2 in C. An SF in C can be handled by any of its corresponding SFIs deployed in the network.

5.2.2 Problem Description

We now describe the **SFC runtime traffic scheduling problem** (in short, SFC scheduling problem). A *packet class* defines a set of packets in the network (1) to which the same SFC needs to be applied and (2) which are at the same processing stage within that SFC. At each SFF, we maintain a set of buffers, each holding packets falling into a same packet class. The SFC scheduling problem consists in deciding at runtime how to assign packets from buffers to the corresponding SFIs and how to allocate the resources to SFIs. For each packet, the end-to-end delay is the sum of the delays at SFFs, SFIs, and propagation delays between SFFs. Our objective is to maximize the system's processing throughput, while constraining the average delay experienced by packets.

5.3 Optimal Scheduling Policy

We show that the SFC scheduling problem is reducible to the scheduling problem in stochastic processing networks (SPNs) [Har00], and propose a scheduling policy achieving the above objectives (Section 5.2.2) with SFFs accessing each other's state.

5.3.1 Background on SPNs

SPNs are a general class of network models that have been used to characterize a wide range of application fields [Wil16], including manufacturing systems and cross-training of workers at a call center. The key elements of an SPN include a set of *buffers*, a set of *processors*, and a set of *activities*. Each buffer holds jobs that await service. Each activity takes job(s) from at least one of the buffers and requires at least one processor available to process the job(s). A job departing after service from a buffer will next be routed to another buffer, or leave the network, with probabilities depending on the activity taken.

5.3.2 Reducing SFC Scheduling to SPN Scheduling

With an ideal setting, the SFC scheduling problem can be reduced to a variant of the scheduling problem in an SPN.

Buffer. According to our system model, each SFF in the network holds a number of buffers which are used to store packets. All incoming packets of the same packet class at an SFF are stored in the same buffer. We denote by $\mathcal{B}$ the total set of buffers in the system. A packet's class can be determined at an SFF by extracting information encapsulated in the packet header. When an SFF receives a packet, it determines the packet's class and pushes it into the corresponding buffer. Packets in the same buffer are processed in FIFO order.

Processor. Each server in our model corresponds to a processor in an SPN. Each server can process packets belonging to the packet classes handled by its SFIs, regardless of its location. As there can be multiple SFIs for the same SF, multiple servers can process packets from the same buffer.

Activity. We define an activity as the processing of a packet from a buffer $B \in \mathcal{B}$ by an eligible server, *i.e.* a server in $\mathcal{S}$ which contains an SFI of the required SF. The total set of activities can be expressed by $\mathcal{A} = \{B \mapsto S \mid B \in \mathcal{B} \ \wedge \ S \in \mathcal{S}_B\}$, where $\mathcal{S}_B \subseteq \mathcal{S}$ is the set of eligible servers for packets in buffer B. $B \mapsto S$ denotes an activity which processes packets from a buffer B over a server S. We denote by $\mathcal{A}_B$ the set of activities connected to buffer B and by $\mathcal{A}_S$ the set of activities connected to server S. Associated with each activity $A \in \mathcal{A}$ is a processing rate μ_A that determines the rate at which a packet will be processed by this activity. The processing rate depends on the relation of the server, buffer, and SFF. If the server and the buffer are under the control of the same SFF, the processing rate is given by the service rate of the corresponding SFI, *i.e.* $\mu_A = \mu_F$ where F is the SF of the SFI; otherwise, the processing rate of the activity is given by a function $g(\cdot)$ of the latency between the corresponding SFFs and the SFI's service rate, *i.e.* $\mu_A = g(\mu_F, \bar{d}_E)$ where E is the link between the SFF holding the buffer and the SFF controlling the SFI.

Routing. Each packet from a buffer B, once being served by an activity A, changes its packet class and gets injected into buffer B' or leaves the network. We define by $p^A_{BB'}$ the probability that a packet from buffer B is injected into buffer B'. Consequently, $1 - \sum_{B' \in \mathcal{B}} p^A_{BB'}$ is the probability that the packet leaves the network. The packet's class transitions as its processing stage is advanced by one SF after being served by the activity. In our model, $p^A_{BB'}$ has a very simple form. If a buffer B holds packets at the last stage within their SFC, then $p^A_{BB'} = 0$ for all $B' \in \mathcal{B}$. For any other $B \in \mathcal{B}$, there always exists one $B' \in \mathcal{B}$ such that $p^A_{BB'} = 1$, else $p^A_{BB'} = 0$.

Example. To further clarify the above mappings, we take the example in Figure 5.3 and consider two SFCs: $C_1 = (F_1, F_2, F_3)$ and $C_2 = (F_2)$. We have four packet classes: (1) packets with SFC C_1 at their first processing stage (to be processed by F_1); (2) packets with SFC C_1 at their second stage (to be processed by F_2); (3) packets with SFC C_1 at their last stage (to be processed by F_3); (4) packets with SFC C_2 to be processed by F_2. As we have 3 SFFs, we would have a total of 12 buffers as depicted in Figure 5.4. Activities A_1, A_2, and A_3 connect buffer B_1 to S_1, S_3, and S_4, respectively. μ_{A_1}, the processing rate of A_1, is μ_{F_1} – the processing rate of SFI of type F_1; $\mu_{A_2} = g(\mu_{F_1}, \bar{d}_{E_1})$, where E_1 is the link between SFF1 and SFF2; and $\mu_{A_3} = g(\mu_{F_1}, \bar{d}_{E_2})$, with E_2 being the link between SFF1 and SFF3. When a packet in buffer B_1 is served by any of the connected activities, it changes its class and is injected into the corresponding next buffer – B_2 if it is served by A_1, B_6 if served by A_2, B_{10} if served by A_3 etc.

Reduction to SPN. Knowing the topology of $\mathcal{G}$, the set of servers $\mathcal{S}$, the set of SFCs, and the set of SFIs $\mathcal{I}$, we can determine $\mathcal{B}$, $\mathcal{S}$, $\mathcal{A}$, and $p^A_{BB'}$. In the ideal case, we assume that at any time t, all schedulers in the network are aware of the state of all buffers, *i.e.* the buffer utilization level which is given by $\vec{z}(t)$, a vector of size $|\mathcal{B}|$, and also of the state of every $S \in \mathcal{S}$, $q_S(t) = \{0, 1\}$, where $q_S(t) = 0$ if S is idle, else 1. Our SFC scheduling problem is then reducible to the SPN scheduling problem [DL08], aiming at designing a control policy for the activities such that the SPN's throughput is maximized, while ensuring that all the buffers are stabilized.

5.3.3 Integer Allocation Maximum Pressure Policy (IA-MPP)

Dai and Lin [DL08] show that the optimal scheduling can be obtained for SPNs by following the maximum pressure policy (MPP). We prove that a simplified version of MPP, IA-MPP can be applied to the SFC scheduling problem. IA-MPP also achieves optimality, but with much less computation than MPP.

The essential decision we have to make immediately is on the amount of resources allocated to each of the activities at a server when it becomes idle. We denote by a vector $\vec{h}$ of size $|\mathcal{A}|$ an allocation. A feasible allocation has to satisfy $0 \leq h_A \leq 1, A \in \mathcal{A}$. If an activity performs at level h_A, it consumes a fraction of h_A resources of the corresponding

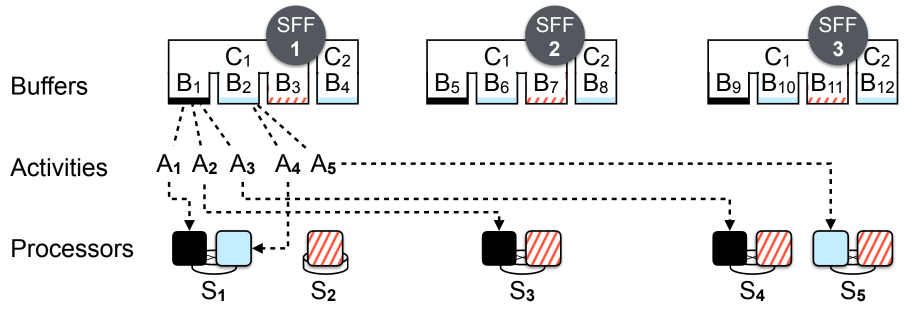


Figure 5.4: An SPN representation of the scenario in Figure 5.3 with two SFCs $C_1 = (F_1, F_2, F_3)$, $C_2 = (F_2)$. Showing only A_1-A_5.

server. Note that $\sum_{A \in \mathcal{A}_S} h_A = 1, \forall S \in \mathcal{S}$. Let $\mathcal{H}(t)$ be the set of all feasible allocations in the network at time t. For each buffer $B \in \mathcal{B}$ and each activity $A \in \mathcal{A}$, we define

$$r_{BA} = \begin{cases} \mu_A & A \in \mathcal{A}_B, \\ -\mu_A & A \in \mathcal{A}_{B'} \text{ and } p^A_{B'B} = 1, \\ 0 & \text{otherwise.} \end{cases} \tag{5.1}$$

$\mathbf{R} = (r_{BA})$ is called the *input-output matrix* of the network. It captures the average processing rates of packets from buffer B consumed by activity A, as introduced in [Har00]. Given a weight vector $\vec{\alpha}$ of size $|\mathcal{B}|$, we define by $\Phi_{\vec{\alpha}}(\vec{h}, \vec{z}(t)) = (\vec{\alpha} \times \vec{z}(t)) \cdot \mathbf{R}\,\vec{h}$ the *network pressure* at time t with parameter $\vec{\alpha}$ under allocation $\vec{h} \in \mathcal{H}(t)$ and buffer utilization level $\vec{z}(t)$. MPP aims to maximize network pressure by picking suitable allocations:

$$\vec{h}^* \in \arg\max_{\vec{h} \in \mathcal{H}(t)} \Phi_{\vec{\alpha}}(\vec{h}, \vec{z}(t)). \tag{5.2}$$

Note that $\mathcal{H}(t)$ is bounded and convex. As $\Phi_{\vec{\alpha}}(\vec{h}, \vec{z}(t))$ is linear in $\vec{h}$, the maximum of $\Phi_{\vec{\alpha}}(\vec{h}, \vec{z}(t))$ will be achieved at one of the extreme points. We can prove that the existence of an extreme allocation for maximum network pressure is ensured.

Lemma 5.1. *For any buffer level $\vec{z}(t)$ ($z_B(t) \geq 0, \forall B \in \mathcal{B}$), there exists an extreme allocation $\vec{h}^* \in \mathcal{H}(t)$ that maximizes the network pressure $\Phi(\vec{h}, \vec{z}(t))$ such that for each constituent buffer B of $\vec{h}^*$, the buffer level $z_B(t)$ is positive.*

Proof. The network we consider is strict Leontief [BW03] as each activity is associated with exactly one buffer. The lemma follows directly if we consider preemptive scheduling [DL08]. With non-preemption, the lemma holds when the network is reversed Leontief. This is (also) the case here as in our model each activity needs exactly one processor to be active. □

Lemma 5.2. *The extreme allocation $\vec{h}^*$ for maximum network pressure is an integer allocation.*

Proof. An allocation $\mathcal{A}$ is called an integer allocation if it satisfies $h_A \in \{0,1\}, \forall A \in \mathcal{A}$.

We assume that when the processor is idle, it takes on a dummy activity A_0. Thus, processor S will be able to take any of the activities in $\mathcal{A}_S^0 = A_0 \cup \mathcal{A}_S$. We now prove the lemma by contradiction. Suppose we are given an extreme allocation $\vec{h}$ where $\exists \tilde{A} \in \mathcal{A}$ such that $h_{\tilde{A}} \in (0,1)$. Let $\tilde{S}$ be the processor that holds activity $\tilde{A}$. Note that activity $\tilde{A}$ requires only one processor due to the fact that our network is reversed Leontief.

For each $A \in \mathcal{A}_{\tilde{S}}^0$, we define a new allocation $\vec{h}'(A)$ by modifying $\vec{h}$ in the following way: We process A with $h_A = 1$ at processor $\tilde{S}$ and keep the allocation on other servers unchanged. It is easy to check that $\vec{h}'(A)$ is a feasible allocation.

It follows that

$$\vec{h} = \sum_{A \in \mathcal{A}_{\tilde{S}}^0} h_A \vec{h}'(A),$$

where we set

$$h_{A_0} = 1 - \sum_{A \in \mathcal{A}_{\tilde{S}}^0, A \neq A_0} h_A.$$

Since

$$\sum_{A \in \mathcal{A}_{\tilde{S}}^0} h_A = 1, \ \tilde{A} \in \mathcal{A}_{\tilde{S}}^0, \text{ and } h_{\tilde{A}} < 1,$$

the summation contains at least two terms. As a result, $\vec{h}$ is a linear combination of at least two feasible allocations and thus, it cannot be an extreme allocation, contradicting the assumption. Hence any extreme allocation must be integer. □

This shows that the allocation produced by MPP in our SFC scheduling problem never splits the processing capacity of a processor. We thus refer to this version of MPP as **IA-MPP**. This property gives us the following network stability result.

Theorem 5.1. *The network operating under a non-preemptive IA-MPP can be stabilized if ever possible.*

Proof. To prove this, we first introduce an auxiliary linear program called *static planning problem* defined by Harrison [Har00]:

$$\min \rho \text{ s.t. } \mathbf{R}\vec{x} = 0; \sum_{A \in \mathcal{A}_S} x_A \leq \rho, \forall S \in \mathcal{S}; x_A > 0, \forall A \in \mathcal{A}.$$

Here $\vec{x}$ is a column vector of size $|\mathcal{A}|$ representing the long-run fraction of time during which each activity is used. The above problem indicates that the long-run input rate to the buffer is equal to the long-run output rate from the buffer. According to Theorem 1 proposed in [DL05], the static planning problem has a feasible solution with $\rho \leq 1$ if the network is stable under some service policy. On the other hand, applying Theorem 9 of the same work [DL05], we can prove that the non-preemptive non-processor-splitting

IA-MPP can stabilize the network if the static planning problem has a feasible solution with $\rho \leq 1$ considering the fact that our network is reversed Leontief. $\square$

Corollary 5.1. *For any $\vec{\alpha} > 0$, IA-MPP with parameter $\vec{\alpha}$ is asymptotically optimal with respect to network throughput.*

Proof. Lemma 5.1 implies that our network model and assumptions satisfy the extreme-allocation-available (EAA) condition. Combined with Theorem 5.1, IA-MPP with parameter $\vec{\alpha}$ is asymptotically efficient according to Theorem 1 in [DL08]. $\square$

Theorem 5.2. *For any given $\varepsilon > 0$, there exists an IA-MPP $\vec{h}^*$ that is asymptotically optimal for a quadratic cost function of the buffer level $\vec{z}(t)$,* i.e. $\sum_{B\in\mathcal{B}} \alpha_B(z_B(t))^2$.

The proof of the above theorem follows from the fact that our network model and assumptions satisfy Assumptions 1-4 in [DL08]. Thus, the same result on asymptotic optimality of quadratic holding cost in Theorem 3 from [DL08] applies here. This result basically provides a theoretical estimation of the buffer level and thus, implies a rough guarantee on network latency since queuing latency is usually the dominant factor during the entire packet processing. We will further validate end-to-end latency for packet processing in the network in Section 5.5.

Following Lemma 5.2, IA-MPP can be simplified as follows. For any $S \in \mathcal{S}$, and any activity $A \in \mathcal{A}_S$, we define

$$\Phi_{AS} = \textstyle\sum_{B\in\mathcal{B}} \alpha_B r_{BA} z_B(t). \tag{5.3}$$

If processor S is in idle state at time t, the scheduler selects

$$A^* \in \arg\max_{A\in\mathcal{A}_S} \Phi_{AS} \tag{5.4}$$

to be served over the server. When more than one allocation attains the maximum, a tie-breaking rule will be applied. Note that the solution space for Equation 5.4 is much smaller than that for Equation 5.2, requiring much less computation as a consequence.

Lemma 5.3. *The IA-MPP scheduler has a time complexity of $O(|\mathcal{V}|)$, with $|\mathcal{V}|$ the total number of SFFs in the network.*

Proof. To find optimal allocation, and for a given $S \in \mathcal{S}$, we need to perform the calculation in Equation 5.3 for all $A \in \mathcal{A}_S$ and then apply Equation 5.4. Note that r_{BA} under the summation has nonzero values for only one or two $B \in \mathcal{B}$ (refer to Equation 5.1). The calculation in Equation 5.3 can be reduced to summation of two terms, and hence has $O(1)$ complexity. The IA-MPP calculation thus has complexity of $O(|\mathcal{B}|)$ as $|\mathcal{A}_S| \leq |\mathcal{B}|$. Furthermore, we have $|\mathcal{B}| = k|\mathcal{V}|$ where k is the total number of packet classes which is a constant for a given network. $\square$

This lemma indicates that time complexity of IA-MPP scales with the number of SFFs, which we expect to be much smaller than the number of servers or the number of SFIs.

5.4 Distributed Scheduling Policy

While scheduling optimally, IA-MPP assumes that schedulers can access each other's state. This can become problematic in distributed, multi-site, setups, when such accesses cannot be synchronized instantly. In this section we thus propose a distributed variant of IA-MPP which takes into account the constraints of a deployable scheduler, disabling cross-scheduler accesses and considering link latencies for scheduling.

5.4.1 STEAM Overview

We now propose multi-site cooperative IA-MPP (STEAM), which is an adaptation of IA-MPP to a distributed setting. In short, with STEAM, each SFF runs its own scheduler using only site-local state, together with an admission control policy (ACP) module. Furthermore scheduling is performed on batches.

Local State. We consider a multi-site setting where a scheduler instance running at SFF $V \in \mathcal{V}$ has only site-local information: the state of (1) buffers at V (*e.g.*, buffer occupancy levels), (2) SFIs of V (*e.g.*, workload), and (3) servers running these SFIs (*e.g.*, busy or idle). Topological information (*e.g.*, where SFIs of other SFFs are running) is static and thus pertains to global information known to all scheduler instances.

Admission Control Policy. For the distributed scheduling problem, an SFF decides whether to serve a packet by an own local SFI, or by a remote SFI. This decision is performed by an ACP module called **STEAM T-valve (SALVE)**. If no SFI of the required SF is available locally, the packet must be forwarded to another SFF. SALVE balances load among SFFs by forwarding packets when local traffic load is too high.

To measure traffic load, SALVE estimates the arrival rates and the service rates for each SF the SFF has SFIs for, using an exponentially weighted moving average estimator with a fixed history length, taking also into account traffic bursts [Ali+14]. Using these estimations, SALVE applies a threshold-based mechanism to decide whether to serve a packet locally or by other SFFs. Specifically, we use a pair of thresholds $\theta_l \leq \theta_h$. We define the traffic load tl as the ratio of the rate estimator of the packets arriving, and the rate estimator of the corresponding service rate. For each incoming packet, SALVE checks the tl of the SF related to the packet's next step and performs the following: if $tl < \theta_l$, the packet is processed locally; if $\theta_l \leq tl \leq \theta_h$, it is processed locally with probability $1 - \frac{load-\theta_l}{\theta_h-\theta_l}$ and forwarded to other SFFs otherwise; if $tl > \theta_h$, it is forwarded to other SFFs. Note that SALVE updates its arrival rate estimation only when handing off a packet to STEAM. To prevent forwarding loops, SALVE keeps track of each packet's detour count, and drops a packet if this number is above a threshold. We thus use the TTL header of NSH, which is intended for loop detection of SFCs and comes at no additional cost [QEP18].

When SALVE decides to detour a packet, it applies a weighted round-robin mechanism to choose among all SFFs which have at least one matching SFI to serve this packet. We use the total processing capacity of each SFF's servers (with matching SFIs) to set the weights. Note that server capacities are static, hence SALVE calculates the weights offline.

Scheduler. STEAM takes the scheduling logic from IA-MPP, but considers the network to consist only of the buffers at the local SFF, local server state, and the activities assigning these buffers to these servers. Whenever a local server is idle, STEAM decides the next activity using a modified Equation 5.3:

$\Phi_{AS} = \sum_{B \in \mathcal{B}} \alpha_B \hat{r}_{BA} \hat{z}_B(t)$. Here $\hat{\vec{z}}(t)$ is the local buffer utilization level and $\hat{\mathbf{R}} = (\hat{r}_{BA})$ is the local input-output matrix, with values $\vec{z}(t)$ and $\mathbf{R}$ for buffers and activities that are local and zero otherwise.

Batch Scheduling. IA-MPP schedules a packet over a server when the server is idle. However, per-packet runtime scheduling may not fit well with large deployable systems mainly for two reasons: (1) Per-packet runtime scheduling introduces a runtime overhead for each packet, resulting in high system load at the SFF even if the scheduling logic is lightweight. (2) Taking a server into account for scheduling only if the server is idle is optimal in theory when link delays are negligible compared with processing delays at the servers. This might however not always be the case in practice.

STEAM thus uses a packet threshold $\phi_{w,S}$ for each of its servers and applies batch scheduling with batch size ϕ_b. The batch size ϕ_b specifies the (maximum) number of packets STEAM sends over to a server at each scheduling round. More precisely, STEAM uses for each of its servers $S \in \mathcal{S}$ a threshold $\phi_{w,S}$ equaling the number of packets the fastest SFI of server S is able to process within the expected round-trip time (RTT) between the server and the SFF. If there are less than $\phi_{w,S}$ packets on the way or queued at a server, STEAM considers this server to be available for taking a scheduling decision, sending up to ϕ_b packets from the selected buffer to this server.

Using ϕ_b and $\phi_{w,S}$ reduces the scheduling granularity to one decision per batch and also reduce the effect of link delays. However, the larger the batches, the fewer possibilities STEAM has for choosing the "best" scheduling decision. Section 5.5.5 investigates the effects of choosing ϕ_b.

Note that with IA-MPP there is no need for a separate resource sharing policy at the server since the share each SFI receives is inherently dictated by the scheduling decision. When batch scheduling is enabled, we employ a round-robin policy at each server. Since $\phi_{w,S}$ and ϕ_b are very small in general, the impact of such a round-robin policy is considered negligible.

5.4.2 STEAM Deployment

While focusing on the theoretical design and concepts of STEAM, we consider practical constraints of an implementation as well. Following Eiffel [Sae+19], which shows feasibility of software packet schedulers running at high packet rates, we implement [Fai19] our STEAM prototype as a software scheduler and show its feasibility in Section 5.5.5. Besides using a software scheduler, we consider white-box switches [Nel+16] and servers with SmartNICs [Fir+18] as possible deployment targets.

5.5 Evaluation

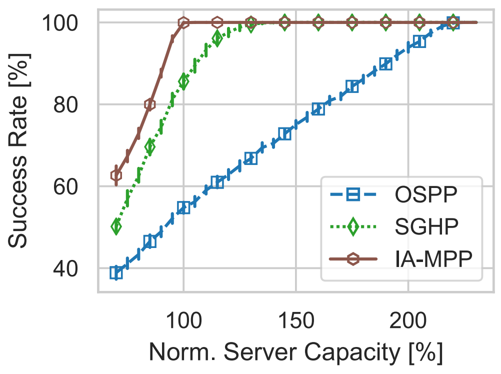
We conducted performance evaluation with large-scale simulations as well as a prototype implementation. Our packet-level discrete event simulator (9K lines of Python code) simulates scenarios in compliance with RFC 7665 [HP15], comprising the network topology including link latencies, packet handling at SFFs, SFIs, and servers, the processing of the SFIs running on servers, and the schedulers.

5.5.1 Algorithms Compared Against

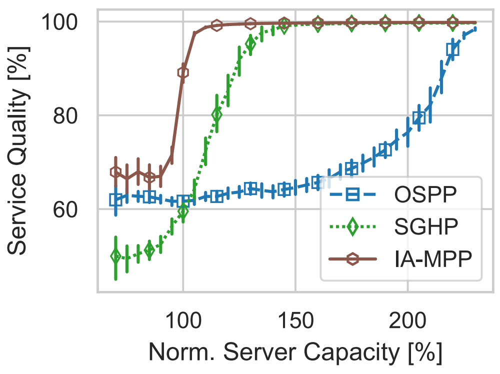
We compare IA-MPP and STEAM with the following two variants of existing static or coarse-grained dynamic algorithms.

OSPP: As a variant of [Sat+18; Fei+18], the offline static planning policy (OSPP) performs offline planning ahead of traffic arrival, but applies runtime load balancing to react to sudden traffic changes. Similarly to these solutions, if multiple SFIs of the same SF are available, OSPP distributes the traffic using service rate of SFIs as weights, while also considering latency between SFFs – favoring higher-capacity SFIs closer to a packet's egress.

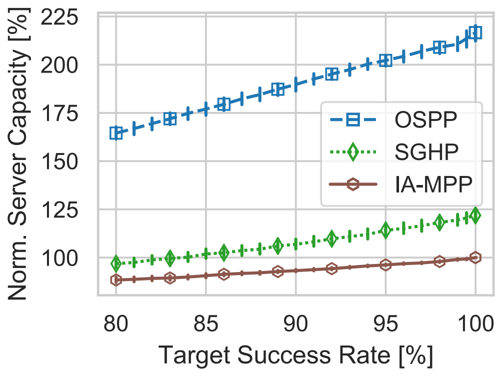
SGHP: The second scheduler, shortened greedy heuristic policy (SGHP), adapts the most recent existing heuristics SGH [Kuo+16] and SPH [ZLZ19] which do not require any *a priori* information like arrival rate or resource demand of a request. Upon receiving a packet, SGHP extends the routing path iteratively and selects the next SFI among all possible *site-local* ones which is likely to provide the shortest delay to serve the packet based on link latency and queue state information. If the load of the local site is too high or if there is no matching local SFI, SGHP starts forwarding to other sites using SALVE.



(a) Success rate over server capacity



(b) Service quality over server capacity



(c) Required capacity over target success rate

Figure 5.5: Single site scenario, running centralized scheduling IA-MPP vs baselines. Varying server capacity c_S to reach full success rate. c_S normalized to IA-MPP's c_S at 100% success rate.

5.5.2 Setup

Unless stated otherwise, STEAM uses $\theta_l = 0.1$, $\theta_h = 1.3$, $\phi_b = 1$. We measure the performance of the schedulers when running the servers at a certain capacity c_s. Sweeping c_s allows to draw conclusions of how effectively the schedulers are able to leverage all available processing power. All scenarios use link latencies following a Poisson distribution with $700\mu s$ for SFI-SFF links and $3000\mu s$ for SFF-SFF links [Guo+15; Cog]. We repeat each experiment with five different seeds.

Metrics. We study two performance metrics: **Success rate** is the ratio of successfully served packets to the total number of arrivals. **Service quality** is one minus the total latency (ingress-egress) of a packet normalized to the QoS deadline of its SFC, also called "average response latency" [ZLZ19]. The higher the values for these metrics, the better the solution.

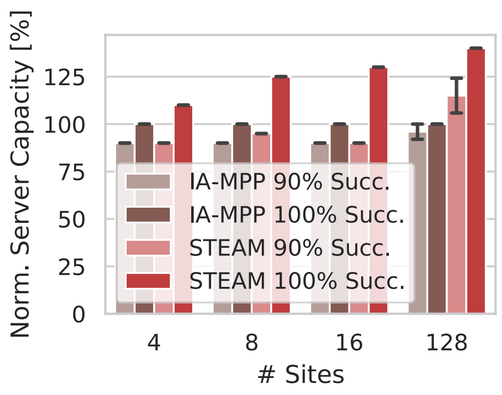
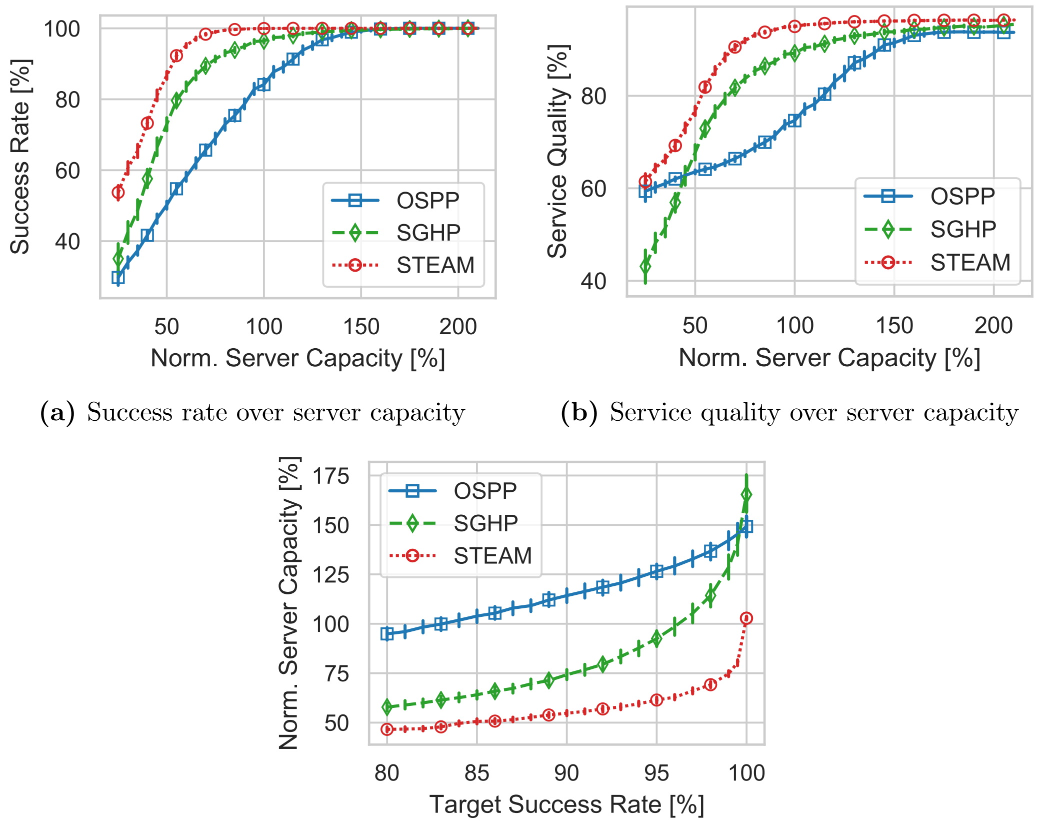


Figure 5.6: From centralized to distributed scheduling, varying #sites. Normalized to IA-MPP.

Workload. Unless stated otherwise, the experiments use a configuration as follows. The flow arrivals are time-varying and bursty. We use a Markov modulated process (MMP) [FM93] to simulate flow arrivals, which is a widely used model [Nee09; Wan+15; Pac+11], with two states – "low" and "high". λ_l and λ_h are the flow arrival rates in these respective states, p_l is the probability of transition from low to high state, and p_h the opposite. We use $p_l = 0.56$, $p_h = 0.4$, $\lambda_h = 1/240\mu s$, $\lambda_l = 1/24\mu s$. We consider the packet arrival process within a flow to be random and independent from other flows, following a Poisson distribution ($\lambda_f = 1/800\mu s$). Flow sizes are also random, following a Poisson distribution ($\lambda_s = 150\mu s$). Each flow randomly selects an existing SFC and a pair of ingress/egress SFFs. Each SFC has a QoS deadline, set as a function of the service rates of involved SFs, which specifies the maximum allowed latency observed by a packet (typically $\ll 100ms$). We consider the SFI processing rates to be similar to the numbers reported for NFVs [Fei+18; Gha+15; Kab+17], in particular to values in the range of $1s/82\mu s$ - $1s/200\mu s$ per resource unit (see Section 5.2).

5.5.3 Single-site Experiments

We first consider a single-site topology with 1 SFF, 36 servers, 5 SFs and 80 SFIs. There are five SFCs: $C_1 = (F_1, F_2)$, $C_2 = (F_1, F_3, F_5)$, $C_3 = (F_2, F_4)$, $C_4 = (F_5)$, and $C_5 = (F_3, F_4)$ with QoS deadlines $\{56, 100, 44.4, 28, 56.4\}ms$.
Figure 5.5 shows the results for IA-MPP and the two competing heuristics running a single site, so all schedulers have access to all state, making comparison fair. We normalized server capacities to the capacity required by IA-MPP to achieve full success. We observe that IA-MPP outperforms the baselines, even in a non-distributed scenario. Specifically, we observe from Figure 5.5a and Figure 5.5b that IA-MPP provides the best success rate and quality of service, given a server capacity, while OSPP shows the worst performance. Figure 5.5c depicts the required capacity to achieve success rates above 80%. To achieve 0 packet drops, an OSPP solution needs twice the capacity(!), and SGHP 25% more server



(a) Success rate over server capacity

(b) Service quality over server capacity

(c) Required capacity over target success rate

Figure 5.7: 50 sites running distributed scheduling: STEAM vs baselines. Varying server capacity c_S to reach full success rate. c_S normalized to STEAM's c_S at 100% success rate.

capacity. These results illustrate that using IA-MPP reduces required server capacity to achieve a target success rate, while also providing better packet latency.

5.5.4 Multi-site Experiments

IA-MPP vs STEAM. First we study the effect of distributing the scheduling decisions per SFF. We sweep the number of sites from 4 to 128 (and traffic load accordingly), and use for each site the same configuration as in Section 5.5.3. Figure 5.6 compares the required server capacity to reach 90% and 100% success rate running STEAM vs IA-MPP. The values are normalized within each site to the capacity required by IA-MPP to achieve full success. Note that each STEAM instance uses only site-local state. We observe that the performance gap between STEAM and IA-MPP increases as we increase the size of the network or the required success rate target. For smallest topology, the two perform almost identically, but the gap increases to 40% when using a 32 times

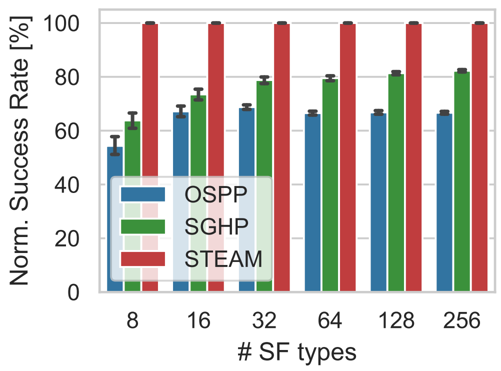


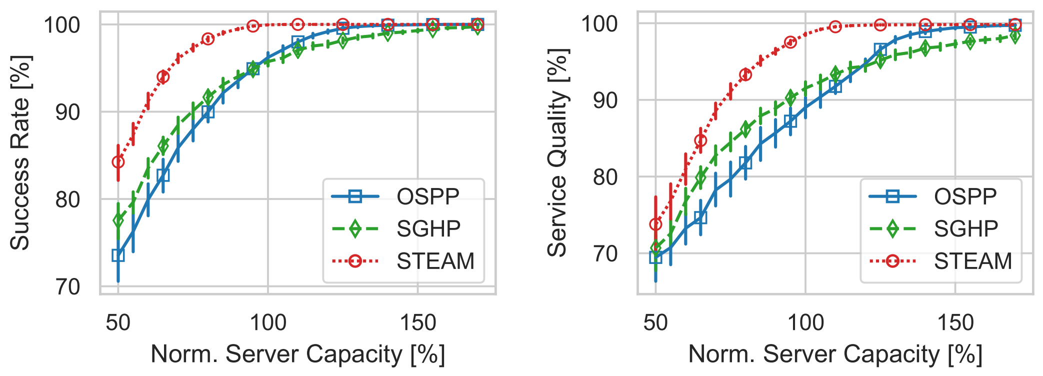
Figure 5.8: 50 sites, varying #SFs, using c_S within 50%-100% of STEAM's c_S with full success.

larger topology, hence this gap grows slower compared with the topology size increase. Nevertheless, STEAM shows great performance, considering the fact that IA-MPP runs global optimization.

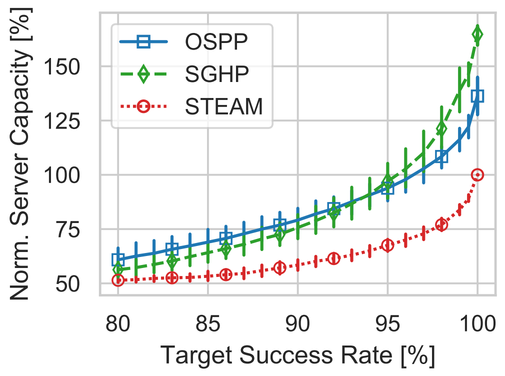
Performance at Scale. Next we consider a topology in the image of publicly available information on data center locations of an Internet service provider (ISP) [Cog]. The topology comprises 50 sites, each with one SFF and 6 to 12 servers. There are 10 SFs in the network with a total of 1600 SFIs across all sites and 30 SFCs each with up to four SFs. We compare STEAM with baseline solutions, all using only site-local state. Figure 5.7 depicts the results. STEAM shows best performance, reaching full success with $50-70\%$ less server capacity. This is as STEAM, driven by our optimal solution, tries to maximize the resource multiplexing in the network and hence can efficiently use available resources. Furthermore, better service quality signals better packet latency with STEAM.

Complexity Increases. Next we vary the number of SFs in order to make the scheduling problem more challenging. The more SFs in the network, the more complex the problem becomes for STEAM, hence scheduling decisions might be negatively affected. We use again the ISP setup with 50 sites and set the number of total SFIs to 20 times the number of SFs, and create 3 times as many SFCs as SFs available. Figure 5.8 shows the average success rates when running servers at capacities between 50% to 100% of the capacity level which STEAM needs to achieve full success. We see that STEAM's gain in success rate over baselines remains always above 20% - 35%.

Trace-driven Workload. In this experiment we use real-world trace files of a related scenario capturing, end-to-end voice and video Skype calls with a total of 484 nodes [TNG]. We consider a topology of 10 sites, 5 SFs, 5 SFCs, 100 SFIs in total, and 4 servers per site, so that each SFF receives the traffic from ~ 48 Skype nodes. For each seed we



(a) Success rate over server capacity **(b)** Service quality over server capacity



(c) Required capacity over target success rate

Figure 5.9: Pcap workload: STEAM vs baselines. Varying server capacity c_S to reach full success rate. c_S normalized to STEAM's c_S at 100% success rate.

take a $10min$ slice from the trace, which we consider to be a reasonable period between two offline planning phases. We consider packets with same source and destination addresses to belong to the same flow and apply the same SFC. Figure 5.9 shows similar results as when running the MMP-based workload. STEAM shows best success rates and service quality at all shown server capacities. Using STEAM reduces the amount of server resources needed for full success by 30% - 70% compared with the baselines. These results indicate that gains are not due to specific tuning of the traffic model, but hold across different traffic patterns.

Batch Scheduling. Finally, we study the effect of batch size ϕ_b on STEAM's performance. Batch scheduling lowers time complexity, but might negatively affect overall scheduling decisions. We study the trade-off. Figure 5.10 shows the required capacity to achieve 90% and 100% success rate when varying ϕ_b of STEAM. Values are normalized to the server capacity required when running STEAM with $\phi_b = 1$ (no batching) and reaching full success. Up to $\phi_b = 64$, there is no significant drawback to batching. With

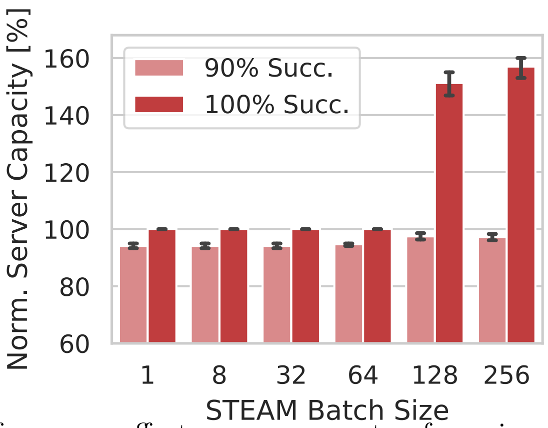


Figure 5.10: Performance effect on success rate of varying batch size running STEAM.

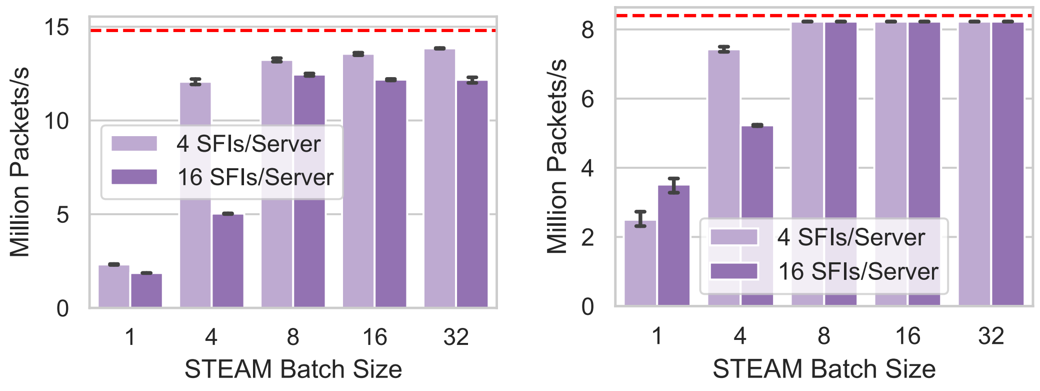
ϕ_b of 128 or 256, 90% target success rate requires slightly more server capacity; to reach full success, we require 50% - 60% more server capacity. Next we show how batching makes runtime scheduling feasible.

5.5.5 Prototype

As described in Section 5.4.2, we have implemented a prototype of STEAM based on DPDK[1], including the NSH protocol [QEP18] to check feasibility running on a standard server with varying bucket sizes ϕ_b. We use two servers (each 2× E5-2630, $128GB$ memory, Intel X520-2 10G SFP+; Linux 4.15.0-48-generic; DPDK 18.11.1) connected via a switch. One of the servers runs our packet generator (a FastClick [BSM15] module), and the other runs STEAM. STEAM uses one core for receiving packets and running SALVE (we set $\theta_l = \theta_h = \infty$, to force all packets going to STEAM), and one core for running STEAM's scheduler sending packets back to the packet generator (intentionally to the SFIs). For each SFF buffer we use a DPDK ring buffer of up to 2048 packets. The system uses 16 hugepages of $1GB$ each, shared among the ring buffers. Note that we did not configure special optimizations, *e.g.*, distributing the buffers across multiple Rx cores.

We run experiments with packets of size $64B$ and $128B$, which corresponds to packet rates of $\sim 14.88 * 10^6/s$ and $\sim 8.45 * 10^6/s$, respectively. The packet generator sends packets at line rate to STEAM and receives packets from STEAM after each scheduling decision. We report the rate at which the traffic generator receives packets from STEAM. To test the effect of scheduling complexity, we run the experiments with 4 and 16 SFIs per server. Figure 5.11 shows the packet rate STEAM can uphold, v.s. the (theoretical) hardware limit of the NIC (red). STEAM reaches almost line rate starting from a batch size of 8, which translates to $2 * 10^6/s$ up to $3.8 * 10^6/s$ scheduling decisions combined

[1] `https://www.dpdk.org`



(a) Goodput at 64B packets workload **(b)** Goodput at 128B packets workload

Figure 5.11: STEAM prototype scheduling performance; varying ϕ_b.

with batching, STEAM hits almost $15 * 10^6/s$ packets. For packets of size $64B$, we do not hit the NIC limit, as we did not apply all possible micro-optimizations.

5.6 Conclusions

This chapter introduced a queuing-based system model to characterize the *runtime traffic scheduling problem* for service function chaining. We presented a throughput-optimal scheduling policy, called IA-MPP. IA-MPP transforms the SFC traffic scheduling problem to an SPN and follows the MPP. When using IA-MPP, no *a priori knowledge* of traffic distribution is required, which enables fast response to sudden traffic changes. We demonstrated the benefits of IA-MPP over state-of-the-art solutions, especially the advantages of performing scheduling decisions with packet-level granularity. Using IA-MPP reduces required server capacity to achieve a target success rate while also providing better packet latency compared with state-of-the-art approaches. However, IA-MPP is a solution that performs global optimization with considering runtime state information of all sites.

We further extended IA-MPP and presented STEAM, a distributed runtime SFC scheduling policy. STEAM inherits the scheduling properties of IA-MPP with respect to local site scheduling, but performs cooperative scheduling among all sites in the system. This makes STEAM independent of global runtime state information of all sites, enabling STEAM deployments of large-scale. We demonstrated the small performance gap of STEAM compared with IA-MPP by running a scale out benchmark with the goal of pedantically 0 packet drops. When increasing the size of the network in the benchmarks by a factor of 32, STEAM requires only 40% more server capacity compared with the global optimum solution of IA-MPP, to achieve 0 packet drops.

STEAM provides mechanisms to vary the scheduling granularity from packet-level to batches of packets, and applies further optimizations to reach the goal of low runtime overhead for scheduling decisions. We presented measurements of our STEAM prototype implementation which achieves 10^6 to $4*10^6$ scheduling decisions per second (with 1 CPU core) when running in per-packet scheduling mode. We demonstrated in this chapter that runtime scheduling of SFC traffic is superior to existing, static or coarse-grained dynamic solutions, which shows the benefit of runtime scheduling solutions that leverage runtime environment information.

Part III

Infrastructure-Level Resource Scheduling

This part focuses on the infrastructure level and presents two alternative solutions to the INC resource scheduling problem for data center resource managers.

Chapter 6 starts with our solution IncSched, which extends existing resource managers to be compatible with INC. IncSched acts as a mediator between the users submitting resource requests, and the plugged scheduler, whereas all INC related logic is handled by IncSched.

Finally, Chapter 7 presents HIRE, a new resource manager for joint server and INC resource management. HIRE extends the introduced concepts of IncSched and presents a new scheduling logic for holistic server and INC scheduling. These advances make HIRE more successful in delivering INC resources to requesting jobs.

6

IncSched: Towards Cluster Resource Scheduling with INC Support

Chapter Outline

This chapter introduces IncSched, a resource management framework for INC-aware Scheduling. IncSched introduces a novel flexible resource model to address heterogeneity, dependencies, and alternatives specific to the INC scheduling problem. We discuss how IncSched can be easily used to plug in existing scheduling policies making them applicable for INC resource scheduling. This democratizes the use of INC switches in data center applications.

6.1 Overview

Over the past decades data center network devices have become increasingly programmable (see Chapter 3). Originally targeting the prototyping and deployment of more flexible and novel network(-wide) services and protocols (*e.g.*, routing), this trend has been more recently exploited for benefiting more specific services and applications. By supporting specific computations through primitives [PN19] "in the network" on the path between data sources and sinks, distributed systems concerns like agreement [Jin+18; Dan+15] or caching [Liu+17; Jin+17] and even high-level application functionality like machine learning [Sap+19; XZ19] can be dealt with in a much accelerated fashion. This trend has ushered in a new era of INC.

As outlined in Chapter 2 and in the following sections, resource management for shared data center INC is missing. IncSched closes this gap with the overarching goal of democratizing the use of switches for INC in a multitenant data center. As its core, IncSched introduces a novel resource model that enables users to submit jobs with server and INC resources, by referring to a data center wide INC store that shows available INC services. A submitted job may contain multiple variants (resource alternatives), for which the user specifies constraints on how these variants can be considered. IncSched translates the submitted jobs to resource requests that can be handled by existing scheduling policies, hiding all INC specific configuration and deployment details. For this purpose, IncSched keeps tracks of all data center resources and provides a set of APIs to ease server and INC scheduling for the plugged scheduling policy.

6.1.1 Design Challenges

IncSched faces three design challenges, which we discuss in this chapter:

Democratize INC in a Multitenant Cluster. If INC is to establish itself as a paradigm, it is to be expected that INC-enabled applications, or even just several users of such applications, will compete over resources on network devices, which are clearly limited in clusters. Existing works focus mostly on isolated scenarios, where network devices are instrumented for benefiting a single application, and evaluations focus on workloads for that application. Some attempts have been made on multiplexing different

INC primitives, but the results are limited to a single network device [ZBH18; HM16; Zha+19; Wan+20]. Considering that an INC primitive usually consists of multiple components (*e.g.*, NetChain [Jin+18] involves a chain of switches) to be deployed on several network devices, there is still a big gap between single-device and network-wide multitenancy.

Existing INC applications typically rely on the network controller for both coordinating the use of INC resources and configuring switches for INC primitives [Jin+18; Liu+17]. We argue that this practice violates the general "single-responsibility principle" [Mar02] and the network controller should stay agnostic to the logic of the INC-enabled applications including their resource demands.

IncSched resolves this issue and consolidates resource management of servers and INC, but leaves INC specific configuration details to the network controller. IncSched democratizes data center INC resources by an analysis of the differences of scheduling INC/server tasks and resources, which builds the foundation for the design of IncSched. For this purpose, IncSched introduces a new resource model, that allows to express server and INC resource demands within a single job request. Furthermore, in order to handle the matching dilemma of request and available resources, the new resource model supports to specify resource alternative, hence IncSched delegates the alternative selection to the plugged scheduling logic.

Match INC Resources. INC does not fit to the generic resource container abstraction running on servers. As modern clusters already run resource managers for allocating server resources [Bur+16], a natural idea is to treat INC switches as additional resource type(s) and let the resource manager deal with their allocation. This handover of responsibility brings clear benefits as resource managers typically already have a global view of server resources, thus enabling holistic decisions for joint server and INC resource allocation supporting properties like data locality and load balancing.

However, management of server resources in clusters without taking into account INC is already a non-trivial problem which has been investigated by many works over the past years [Gog+16; Cur+14; Isa+09; Tum+16; Bou+14; Ous+13; Ver+15; Del+18], some also involving GPUs and other accelerators [Gu+19; Mah+20; Pen+18; Xia+18; Cha+20] (see Chapter 2). Throwing INC resources into the mix adds new challenges and significantly exacerbates existing ones. This raises the question for an appropriate resource model that encodes all INC specific properties.

IncSched introduces the concept of *INC templates* with an *INC template store*, which encapsulates the logic of INC primitives. Users submit jobs by composing INC templates with server resources. This approach hides the complex resource sharing constraints of INC tasks, and gives the user a perspective of isolated compute containers.

Enable Existing Data Center Schedulers for INC. Unfortunately, developing and testing new data center schedulers (and policies) from scratch is non-trivial. This raises the question how to adapt existing data centers schedulers to make them ready for INC resource scheduling.

Existing data center schedulers often provide interfaces for customization, allowing the data center operator to fit a scheduler to its personal preferences. Most dominantly, typical schedulers enable data center resource sharing among multiple frameworks by framework-level customizations [Vav+13; Sch+13; Hin+11], or even supporting customized sub-cluster scheduling policies [Cur+19]. However, exiting schedulers have in common the generic resource abstraction of compute containers running on servers (optionally with feature flags like GPU accelerators). Throwing INC into the scheduling responsibility challenges the schedulers, and raises the question how existing scheduling solutions can be made aware of INC resource scheduling.

IncSched acts as a mediator between the users, who submit jobs by using our new resources model (INC templates with an INC template store), and the plugged scheduling logic of existing resource management frameworks. For this purpose, IncSched internally creates as many (partially overlapping) sub groups of resource nodes as required, in order to encode compatibility constraints of INC and server tasks, but hiding the resource complexity for the plugged scheduler. The plugged schedulers use the provided API to retrieve a list of potential resources nodes (servers or switches) for a specific pending task. This encapsulates all interdependencies of server and INC resources within IncSched.

6.1.2 Related Work

Since Chapter 7 builds on this chapter, we discuss in the following related work relevant for Chapter 6 and Chapter 7. Related works mainly fall into following categories:

Individual INC Services. As mentioned, many individual INC services have been proposed for facilitating networking [Kim+16; Bas+20; Li+19] and other [Jin+18; Jin+17; Liu+19; Yu+20] tasks leveraging the high performance and programmability of network appliances. However, all these efforts focus on single use cases, leaving aside the problem of coordinating usage of potentially scarce and heterogeneous resources on network appliances among multiple INC scenarios, applications, and users.

Multitenancy for Network Appliances. Some recent works [HM16; Wan+20] have explored the potential of enabling multitenancy support on a single network appliance, *e.g.*, a P4 [Bos+14] switch or a smart network interface controller. However, such efforts are limited to device-level sharing, *i.e.*, co-locating multiple INC services on the same network appliance. None of them have considered how to coordinate the use of INC resources on network appliances at a network-global level.

Data Center Resource Models. As discussed in Section 2.3, existing data center resource managers focus mainly on server resources (*e.g.*, CPU, memory). Very few also consider bandwidth reservations between servers [JS14]. These approaches use either a simple list of requested VM resources, or a more complex request model based on, *e.g.*, VCs, VOCs, TAGs, or virtual data centers. All these resource models focus on server resources and bandwidth demands between a group of VMs. As seen in Chapter 2, an resource manager for INC needs to manage not only server resources, but also INC resources, making these models unsuitable. Harmony [Ben19] discusses early ideas to extend the TAG [Bal+11] to encode relative placement constraints of switches to pre-allocated servers. However, Harmony does not consider resource alternatives and automatic translation of topologies and resource demands as IncSched and HIRE do.

Data Center Resource Management. While various aspects of data center resource management have been explored over the last years (centralized *vs.* distributed, or prediction-based *vs.* runtime-agnostic) [JS14], none of the existing approaches tackle the problem of resource management for application requests including INC. The majority of resource management frameworks focus on the scheduler architecture of server-local resource management [Sch+13; Hin+11; Del+15; Ver+15; Vav+13; Cur+19; Wan+19]. Others focus on scheduling policy design [Gho+11; Ous+13; Del+18; ZWY19; Gra+16c; Tum+16; Gra+16b]. Quincy [Isa+09], Firmament [Gog+16], and Aladdin [Wu+19b] use a network flow model for considering data locality of jobs, which allows to consider shared resources of consecutive jobs. HyperSched focuses on machine learning training workloads and enables the automatic exploration of the optimal tradeoff between hyper-parameter configurations and training deadline guarantees [Lia+19]. Decima proposes to use reinforcement learning to generate scheduling decisions from experience [Mao+19]. Besides server-local resources, some approaches consider the scheduling task as a virtual network embedding problem with the goal of providing bandwidth guarantees [Fue+18; Bal+11; Lee+13; Guo+10] between the servers of a job. However, no approach considers the requirements laid out in Section 6.2 and Section 7.2 for INC-aware resource management.

GPU scheduling. With widespread adoption of GPUs for accelerating deep learning, a variety of domain-specific schedulers for GPU clusters have been proposed [Zha+17b; Mah+20; Xia+18; Jeo+19; Gu+19; Pen+18; Pen+19]. These intend to replace general-purpose cluster schedulers by exploiting characteristics of deep learning workloads. In response to the challenge of gang scheduling and tradeoff between locality and GPU utilization, several techniques including trading of locality for waiting time and migrating jobs have been developed [Jeo+19]. Gandiva employs time-slicing and job migration/-packing on GPUs for more fine-grained scheduling [Xia+18]. Allox discusses the task scheduling problem when CPU and GPU resources are interchangeable [Le+20]. INC scheduling is yet more complex due to high heterogeneity, fine-grained locality, and on-device resource sharing.

So far, the network is beyond the scope of most existing resource management frameworks, except for virtual network embedding algorithms specifically used for network bandwidth reservations [Bal+11; Guo+10; Pop+12; Xie+12]. In particular, popular cluster resource managers [Vav+13; Bur+16; Cur+19] are completely agnostic to the status of the network managed by a separate entity – the network controller, thus being unlikely to support INC resources directly.

All these resource models focus on server resources and bandwidth demands between a group of VMs. As we will seen in Section 6.2, an resource management framework for INC needs to manage not only server resources, but also INC resources, making these models unsuitable. In short, our work differs from all previous related work to data center resource scheduling, that IncSched is the *first-of-its-kind INC-aware resource management framework*, with the first resource model that captures both INC and server resources.

6.1.3 Contributions

In this chapter, we make the following contributions:

1. We identify the main challenges in resource management for INC-enabled data centers. In particular, we identify the following three main challenges in achieving efficient INC-aware resource scheduling in a multitenant cluster: (a) INC resources have **high heterogeneity** in terms of both programming models/interfaces and resource availabilities. (b) INC primitives have **complex scheduling dependencies** due to their multiplexing constraints on different network devices combined with dependencies to server resources. (c) INC-enabled applications impose **many demand alternatives**, bringing in a new dimension for scheduling decision-making (Section 6.2).

2. We introduce a *novel resource abstraction* using *INC templates* with an *INC template store*. Our INC templates encapsulate the logic of INC primitives, each exposing interfaces for users to specify their resource demands at a high level. Users submit jobs by composing INC templates with server resources, which will be automatically transformed (using combinatorial constraints given in the request) into actual materialized requests (MatReqs), each treated as a scheduling alternative. Based on the INC templates, we propose the *first-of-its-kind INC-aware resource management framework*, called IncSched. Our framework allows existing scheduling policies to be plugged in with minimal changes through a simple API (Section 6.3).

3. We discuss the proposed resource management framework design (IncSched) by adapting various popular scheduling policies each in a case study, including queue-based (best effort) [Bur+16], dominant resource fairness delay scheduling [Zah+10; Apa19], and power of two choices [Ous+13]. For each case study, we discuss required modifications of the original policy to fit well to the scheduling problem of server and INC resources (Section 6.4).

We evaluate each case study (Section 6.4) based on a discrete-event simulator with a real-world cluster trace. For resource alternative selection, we implement multiple alternative selection strategies and discuss their compatibility and performance with each of the plugged scheduler logic. Our results show the potential effectiveness of our resource management framework in making retrofitted schedulers INC-aware.

6.2 INC Challenges and Key Insights

Multitenancy support for a single programmable network device has been shown recently [ZBH18; HM16; Zha+19; Wan+20]. Considering that an INC primitive usually consists of multiple components (*e.g.*, NetChain [Jin+18] involves a chain of switches) to be deployed on several network devices, there is still a big gap between single-device and network-wide multitenancy. Additional research is clearly required to ultimately *democratize the use of INC switches* in cluster applications. Below we identify three main challenges towards achieving INC-aware resource management, and present insights for tackling them.

6.2.1 High Heterogeneity

Programmable network devices are highly heterogeneous with respect to both programming models and resource availabilities. To enable programmability, emerging switches (*e.g.*, Intel[1] Tofino [Int18] and Intel FlexPipe [Int13]) and network accelerators (*e.g.*, Netronome NFP-6000 [Net18] and FlexNIC [Kau+16]) are equipped with reconfigurable hardware spanning programmable ASICs, NPUs, FPGAs, and general-purpose CPUs. Several of these hardware platforms come with limited programming models and interfaces; *i.e.* programmable network devices exhibit different levels of “programmability”, in contrast to servers which are expected to support general Turing-complete computations. Given the hardware diversity, an INC primitive may be implemented with different programming models targeting different types of devices. With different program synthesis or compilation configurations, the resource demands and performance characteristics of INC primitives can differ significantly. Moreover, resources capacities can largely differ between network devices. For example, programmable ASIC-based switches may have different specifications for processing pipeline stages, SRAMs or TCAMs, and stateful memory (*e.g.*, registers, counters). The high degree of heterogeneity of INC resources makes the cluster resource scheduling problem complex.

To handle such complexities, we introduce a new **resource model** for INC primitives based on the concept of **INC templates** maintained in an **INC template store**. INC templates abstract the specifications of INC primitives, providing tunable knobs for customization, and encapsulating information like setup constraints. Users can submit requests using such high-level INC templates, inheriting their properties. Our resource management framework takes care of the transformation of INC templates to actual jobs that can be tackled by the resource scheduler.

[1] former Barefoot Tofino

6.2.2 Complex Scheduling Dependencies

Taking a scheduling decision for a specific server or switch can strongly impact the value of all other resource candidates. Furthermore, INC primitives can have complex multiplexing dependencies on switches, *i.e.* the scheduling decision for one INC primitive may affect that for others. This can be caused by both runtime environment conflicts of different INC primitives and resource constraints on switches. For instance, a switch running $P4_{14}$ can multiplex different INC primitives implemented in $P4_{14}$, *e.g.*, running NetChain [Jin+18] and R2P2 [Kog+19] on the same switch concurrently [HM16; Wan+20], but it may face compatibility issues with an INC primitive in $P4_{16}$. The number of INC primitives that can be scheduled on a switch is determined by the amount of total switch resources and the INC primitive types. In addition, some switches may not support multitenancy at all, or only to a certain degree. Thus, a switch may have to be locked for one INC primitive(s) throughout the lifetime of the corresponding job(s) once scheduled, even if the switch has extra resources.

Our solution to this issue is to introduce a **state manager** to track dependencies in the resource management framework and expose such information to the scheduling policy so dependencies can be respected.

6.2.3 Many Demand Alternatives

INC resources are relatively scarce in comparison to server resources, *e.g.*, the critical resource of on-chip stateful memory is limited to tens of MB on a typical Tofino switch [Wan+20]. Given this scarcity, one must be prepared for many requests for INC resources to be unsatisfiable within a non-trivial timeframe. Fortunately, INC-accelerated applications can by definition also be accomplished without INC resources. *e.g.*, a partition/aggregate job can run without INC, though probably taking longer, or requiring more servers to execute in the same timeframe. Considering also the aforementioned heterogeneity and dependency factors, an INC-enabled job can be specified by a set of substantially different, interchangeable resource demands with varying performance properties and multiplexing dependencies. Such flexibility adds an extra dimension to the scheduling problem: which resource demand to accept for an INC-enabled job.

Without considering the dependencies challenge, and when runtime estimates of job alternatives are known, previous work has proposed solutions for the plan-ahead scheduling problem [Tum+16]. Recently, a similar concept has been adopted, but it is limited to substituting GPUs with CPUs in job scheduling based on performance samples [Le+20].

We propose to augment existing scheduling policies with an **alternative-selection strategy**. In default, the decision for the mutually exclusive alternatives is made first, based on some preferences, before the tasks from the finally chosen alternative are scheduled following the original scheduling policy.

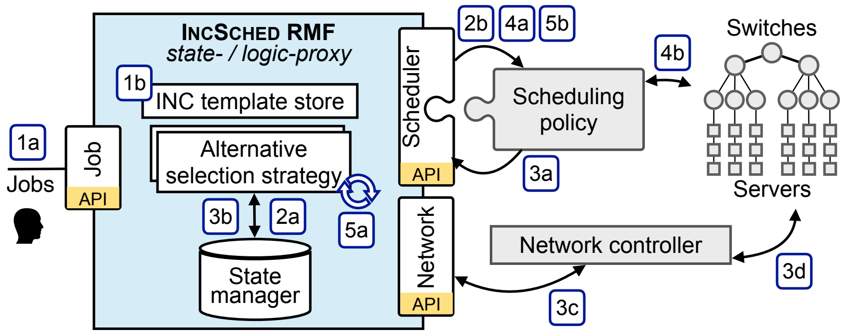


Figure 6.1: Architecture of IncSched.

6.3 IncSched Architecture

We sketch the design of our IncSched resource management framework for INC-aware resource scheduling.

6.3.1 Overview

IncSched (see Figure 6.1) acts as a proxy for state and logic to make existing schedulers applicable for INC resource scheduling by integrating all INC-specific logic into the resource management framework abstractions. A job's life-cycle comprises the following steps:

Step 1: Users submit jobs (1a) to IncSched using its resource model (Section 6.3.2), referring to INC templates. IncSched translates (1b) each job via the INC template store and creates as many MatReqs (materialized requests) as needed to capture all requested demand alternatives.

Step 2: The alternative selection logic (Section 6.3.3) runs for each arriving MatReq of a job (2a) using the up-to-date INC load information provided by the state manager. As a result, MatReqs that are mutually exclusive to the selected one will be withdrawn or temporarily disabled. IncSched records its alternative selection decision in the state manager, and forwards all enabled MatReqs to the scheduling logic (2b).

Step 3: The plugged scheduling policy uses the APIs outlined in Listing 6.2 to check for possible task allocations. It informs IncSched of task allocations (resource claims) or ended/terminated tasks (3a). IncSched checks each task allocation update for possible consequences on its job's alternative selection and updates the state manager accordingly (3b). For all allocation updates IncSched informs the network controller (INC virtualization subsystem, 3c), which accordingly configures INC on switches (3d).

Step 4: After IncSched receives the deployment status of the network controller, the scheduling logic receives a callback for each performed task allocation ($\boxed{4a}$), and further continues its allocation logic as desired ($\boxed{4b}$).

Step 5: Eventually, each job requires an alternative selection. Depending on the actual alternative selection logic, a timeout might kick in ($\boxed{5a}$), which updates the MatReqs of a job. The scheduling policy receives MatReq updates accordingly ($\boxed{5b}$), which might trigger further updates on taken allocations (going back to $\boxed{3}$).

6.3.2 Resource Model

The IncSched resource model defines the JSON user interface (to submit job requests), the INC template store (with all INC templates), the state manager, and MatReqs.

Job Requests. A job request is a list of task groups derived from INC templates. Listing 6.1 shows an example job request. Each task group is associated with a set of resource demands following the configuration keys specified in the target INC template (for INC task groups). A request's `alternatives` field specifies the combinatorial constraints of the given task groups by a set of conditions which all must hold, using & (and), | (or), ˆ (xor), and ! (not).

INC Template Store. IncSched holds all available INC templates in the INC template store. The INC template store serves as a resource matching reference for IncSched, and as a setup reference for the network controller. Each template in the INC template store defines resource demand interfaces and encodes resource constraints (*e.g.*, P4$_{14}$). When a `getMachineCandidates` API call is issued, IncSched checks the involved INC templates for dependency information.

MatReq. IncSched converts each submitted job request to a set of materialized requests (MatReqs), each representing one (mutually exclusive) alternative of the job, by utilizing transformation rules of the INC template store. A MatReq is a combination of the task groups in a job request, which satisfies all expressions given in the `alternatives` field. The example in Listing 6.1 results in two MatReqs.

```
{id: 17, ..., taskGroups: [
    {id: tg1, type: server,  count: 4, core: 2, ...},
    {id: tg2, type: inc, primitiveType: netchainV1,
        cfg: {keySize: 16B, keys: 8000, replication: 3}},
    {id: tg3, type: server, count: 6, core: 2, ...}],
 alternatives: ['tg1', 'tg2^tg3']}
```

Listing 6.1: Example job request (in JSON) of a user.

State Manager. IncSched encapsulates all INC related information and stores state/load information in the state manager, so that a plugged scheduling policy sees the scheduling problem of INC resources like *just another server resource scheduling problem*. For each INC primitive, the state manager keeps track of the total resource capacity and resource reservations on switches with jobs involving the INC primitive. *e.g.*, when a new INC primitive gets "activated" on a switch (3a), the state manager updates both the total capacity and reservation information for the new INC primitive and the reservation information for all other INC primitives running on that switch. Furthermore, the state manager keeps track of the number of switches which have free resources to activate new INC primitives. Listing 6.2 shows the API calls to access these cluster-wide INC load estimates.

```
// retrieve load/state information
getIncLoad( IncPrimitive ) : Double
getIncActivation( IncPrimitive ) : Double
getTotalIncActivation : Double

// check resource compatibility
getMachineCandidates( TaskGroup ) : List [ MachineId ]
getMaxContainer( TaskGroup ) : TaskCount

// update state manager
claimResources( TaskGroup , MachineId , TaskCount )
freeResources( TaskGroup , MachineId , TaskCount )
rejectRequest( MatReq )
```

Listing 6.2: The IncSched scheduler API (in Scala).

6.3.3 Alternative Selection

Each job refers to one or multiple (mutually exclusive) MatReqs, but the plugged scheduling policy may expect only one MatReq per job. The alternative selection strategy in IncSched is in charge to coordinate the set of enabled/disabled MatReqs. We propose the following strategies:

Concurrent lets the plugged scheduling policy take a decision, so IncSched passes it all MatReqs. Upon each allocation (3b), IncSched checks whether the pending allocation belongs to (i) a newly selected MatReq, or (ii) an already chosen one. With (i), the state manager remembers the newly selected MatReq for this job and IncSched disables all other MatReqs of the same job (5b).

Cluster load selects a MatReq based on the cluster-wide load estimates (2a) for each new job (see Listing 6.2). Using these metrics the strategy applies a threshold-based selection (we use 98%), choosing the INC MatReq showing the lowest maximum load of the affected INC primitives, considering all switch resource dimensions. If any of the affected resource dimensions of all INC MatReqs is above the threshold, the strategy selects a server-based MatReq.

Timeout allows the user to take a decision by submitting a job without alternatives, *i.e.* the `alternative` expression of the request allows only one MatReq. Intentionally, the user submits the preferred alternative. When the requested INC resources are not available within an acceptable time period (we use 10% of a job's runtime), the user triggers a fallback and submits a job request with fewer or no INC resources.

Starvation Prevention. In some cases a strategy might take a wrong alternative selection—the load of the affected (INC) resources increases—leading to unfulfilled job requests and starvation. In such cases the strategy triggers a check (5a) of the necessity of a fallback for the alternative selection.

6.4 Case Studies

This section studies how three existing scheduler designs fit the INC resource scheduling problem using IncSched.

6.4.1 Methodology

Due to the lack of a multi-tenant/shared data center testbed for INC, we perform large-scale simulations. We built a cluster scheduling simulator (7K lines of Scala code) similar to that of Omega [Sch+13], but with support for the IncSched components shown in Figure 6.1, INC resources, and multi-path network topologies. Each case study runs the same scheduling problem focusing on one plugged scheduler logic, and reports the following metrics to test how well the three selection strategies of Section 6.3.3 perform to meet the design goals:

Satisfied INC jobs represents the ratio of MatReqs scheduled with INC out of the total number of jobs with INC.

Preempted tasks is the ratio of the number of tasks preempted to the total number of tasks started.

Placement latency is the time between job requests and tasks start processing on a machine. We report both median and tail (95th percentile) placement latencies.

We replay 48 hours of a public production workload trace from a 4000 machine Alibaba cluster [Gro18], which contains jobs of two priority classes. To best fit the 4000 servers we use a Fat-Tree (see Section 2.2) topology with $k = 26$, holding 4394 servers and 845 switches. For the switches we define three resource dimensions, namely reserved recirculation capacity, stages (48), and SRAM (22MB), in order to roughly estimate INC resource demands referring to INC processing overhead, program complexity, and storage, respectively [Jos+15].

Name	|Switches|	Requirements
SHArP [Gra+16a]	$\lceil \log \|G\| \rceil$	SHArP ASIC
IncBricks [Liu+17]	$\mathsf{max}(3, \lceil \log \|G\| \rceil)$	OpenFlow + Accelerator
NetCache [Jin+17]	$\mathsf{max}(3, \lceil \log \|G\| \rceil)$	$P4_{14}$
DistCache [Liu+19]	$\mathsf{max}(3, \lceil \log \|G\| \rceil)$	$P4_{14}$
NetChain [Jin+18]	$\mathsf{max}(3, 3\|G\|/10^3)$	$P4_{14}$
Harmonia [Zhu+19]	$\lceil \|G\|/9000 \rceil$	$P4_{14}$
HovercRaft [KB20]	$\lceil \|G\|/9000 \rceil$	$P4_{14}$
R2P2 (JBSQ) [Kog+19]	$\lceil \|G\|/9000 \rceil$	$P4_{14}$

Table 6.1: Resource constraints for INC approaches used in evaluation.

Name	Recirculation capacity	Stages	SRAM (MB)
SHArP [Gra+16a]	/	/	$[1, 8]$ MB
IncBricks [Liu+17]	$[0, 40]\%$	$[4, 8]$	$[3, 12]$ MB
NetCache [Jin+17]	$[0, 10]\%$	$[0, 8]$	$[6, 12]$ MB
DistCache [Liu+19]	$[0, 10]\%$	$[0, 8]$	$[6, 12]$ MB
NetChain [Jin+18]	$[0, 10]\%$	$[0, 8]$	$[6, 12]$ MB
Harmonia [Zhu+19]	0	$[0, 3]$	$[768, 2048]$ KB
HovercRaft [KB20]	$[0, 10]$	$[0, 18]$	$[0, 128]$ KB
R2P2 (JBSQ) [Kog+19]	$[0, 30]\%$	$[0, \|G\|]$	$[1, 64]$ KB

Table 6.2: INC approaches used in evaluation. Each column gives resource demand per switch.

Table 6.1 lists 8 INC primitives we add to the INC template store: NetChain [Jin+18], SHArP [Gra+16a], IncBricks [Liu+17], NetCache [Jin+17], DistCache [Liu+19], Harmonia [Zhu+19], HovercRaft [KB20], and R2P2 [Kog+19]. We set resource demand ranges according to numbers reported, and communicated to us, by the authors. Table 6.2 shows the required resources for each INC primitive as a constant or range. In the latter case, the simulator draws a random configuration for each task group.

To discuss the limitations of INC multiplexing we run two setups, limiting the number of active INC primitive types per switch to 1 and 3. Each experiment sets the target ratio μ of jobs with INC requests, thus randomly selecting jobs of the trace and applying, for 1/4th of a selected job's task groups, any of the INC primitives to create a job alternative (adding entries to the `alternative` field of a request). To capture savings of required servers and reduced processing time of a job using INC, we reduce both by 10%.

The schedulers use algorithms of different runtime complexity, hence they have different think times for solving the same scheduling problem. For queue-based schedulers, typical reported numbers [Sch+13; Tir+20; Cur+19] are in the range $0.4 - 7.2$ ms per allocation. For fair comparison we set each scheduler's think time to match these numbers for an idle cluster state.

We run each experiment—characterized by plugged scheduler, alternative selection, μ, INC multiplexing—with 3 seed.

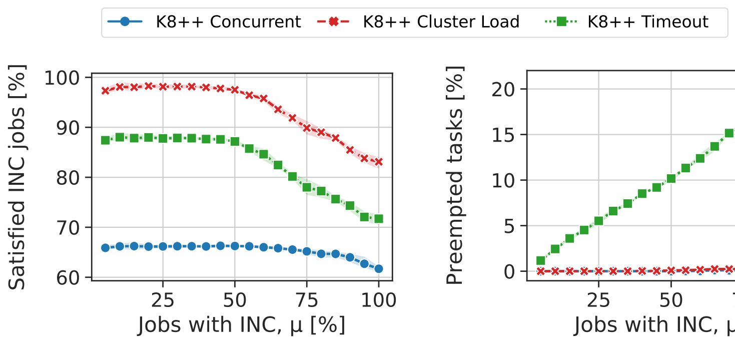
6.4.2 Queue-based Scheduling Using Best Effort

We first study a common [Bur+16] scheduler design using queues to hold pending jobs, and a two-part scheduling policy performing I. *feasibility checking* and II. *scoring.* Consider Kubernetes (K8): put simply, it schedules at the granularity of pods and uses priority queues for *active*, *back-off*, and *unschedulable* pods. Before scheduling the next pod, K8 checks whether back-off pods must be pushed to active queues. If all active queues are empty, K8 also checks the unschedulable queue. Similarly to Omega and Borg [Bur+16], K8 (I.) iterates over all machines in a round-robin fashion to find at least 10% of all machines (5% for $\geq$ 5000 machines) which are capable to serve the current pod. Then, K8 (II.) scores the candidates to find the best machine for serving the pod. When considering the next pod, I. resumes at the position where it stopped before. K8 pushes a pod to the back-off queue with an increasing pause time if no machine is found, or to the unschedulable queue after several failed attempts.

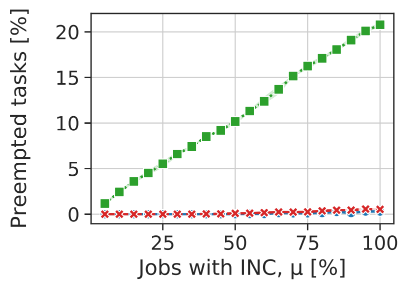
K8++. IncSched+K8 (K8++) is a K8-inspired scheduling logic plugged into IncSched using K8's multi-dimensional resource model. Each pending pod in K8++ holds a reference to its MatReq. Our API (Listing 6.2) makes K8++ aware of INC resources with few changes. For the strategies Concurrent and Cluster load, K8++ needs a server fallback trigger, similar to the 5a trigger, for cases when the INC MatReq decision "turns out to be bad". Before pushing an INC pod to the back-off queue, K8++ checks whether the MatReq decision could be reverted *without* performing task preemptions of the given job—if no task of this MatReq is started yet—and invokes the API. When an INC pod would be pushed to the unschedulable queue, K8++ always invokes `rejectRequest`.

Results. Figure 6.2 (a-f) shows the results when running K8++ with INC multiplexing. The highest success rate of delivering INC resources is achieved with the Cluster load policy, whereas Concurrent performs worst, staying almost constant at 65%. Only the Timeout policy triggers many preemptions (Figure 6.2b). As a consequence, as seen in Figure 6.2d, only that policy retains low tail latency (at the cost of more preemptions).

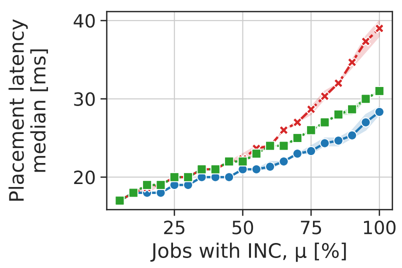
We also show the results for no INC multiplexing in Figure 6.2e for K8++, as it is the only case study whose performance trends differ when running without multiplexing—



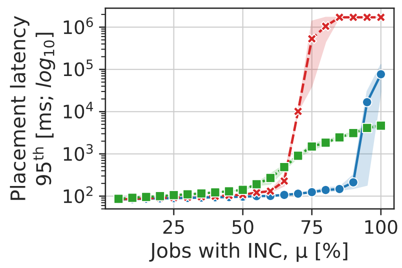
(a) Satisfied INC jobs



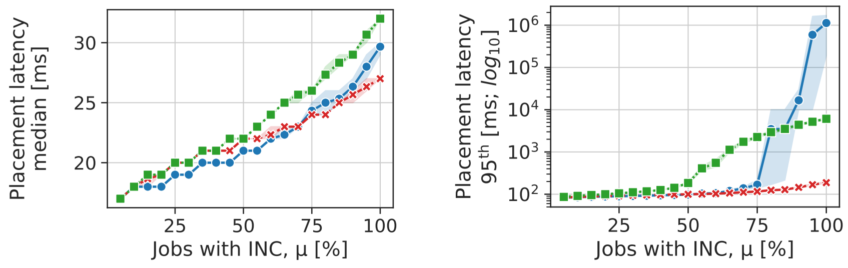
(b) Preempted tasks



(c) Placement latency, median



(d) Placement latency, 95th



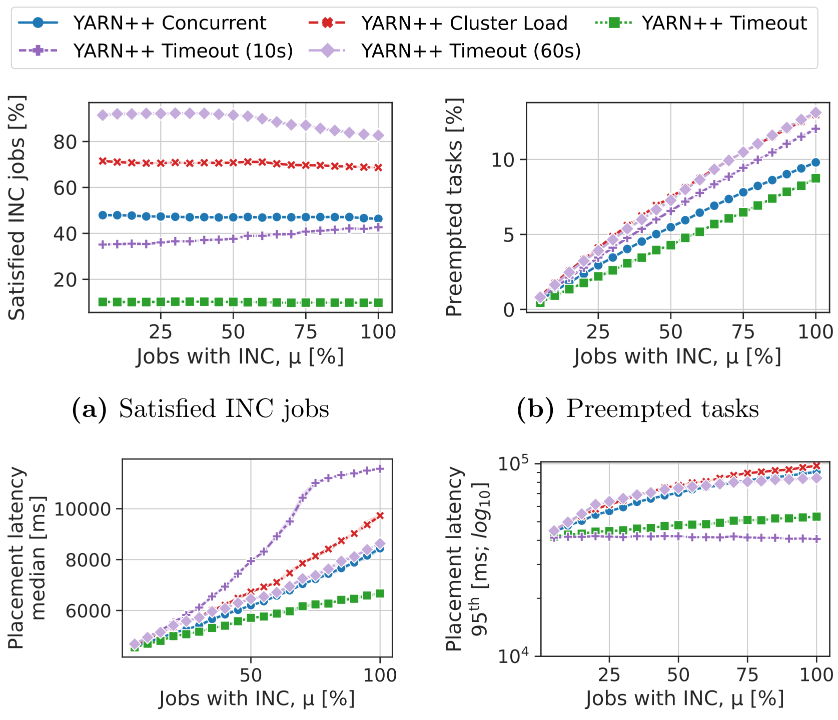
(e) Placement latency with disabled multiplexing of INC primitives

Figure 6.2: K8++ scheduling performance with INC multiplexing (a-d) and without (e), as function of μ (INC demand).

revealing better performance of the Cluster load policy in that case. We interpret this result as a hint that the INC cluster load estimates (Listing 6.2) require more details for scenarios with multiplexing.

6.4.3 Delay Scheduling Using DRF

Next we investigate a queue-based delay scheduler [Zah+10] inspired by the Yarn [Vav+13] capacity scheduler [Apa19]. The main difference to the previously discussed scheduler design is that the scheduling logic only selects a new candidate machine for a task group if several re-checks fail to start a task.



(a) Satisfied INC jobs

(b) Preempted tasks

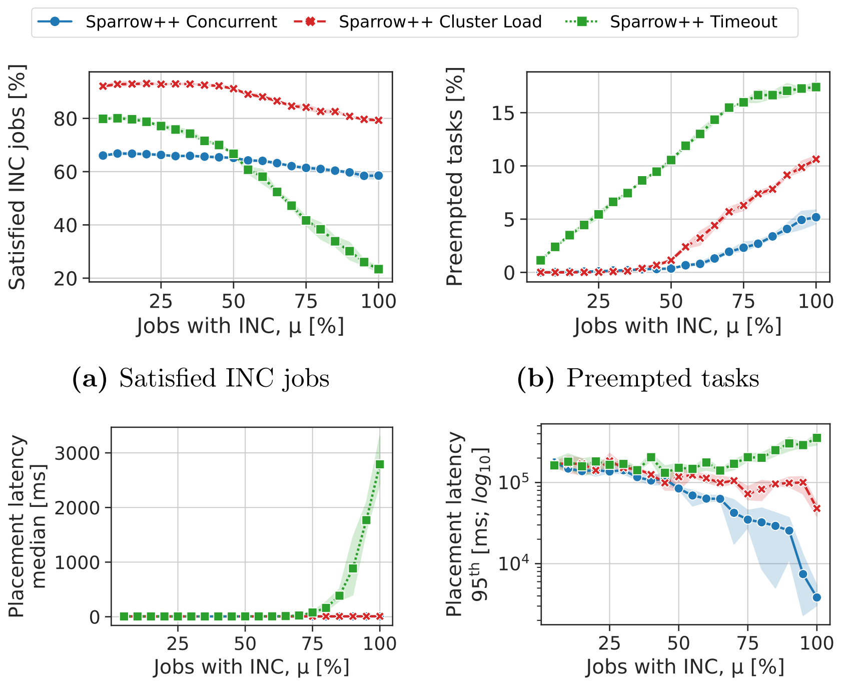
(c) Placement latency, median

(d) Placement latency, 95th

Figure 6.3: Yarn++ scheduling performance with INC multiplexing, as function of μ (INC demand).

Yarn++. IncSched+Yarn (Yarn++) applies dominant resource fairness (DRF) for ranking machine candidates. We set re-check delay to 50ms, and the machine candidate selection delay to 100ms. For the strategies Concurrent and Cluster load, Yarn++ uses a timeout of 60s when no candidate machine is found to select a fallback MatReq using the API.

Results. Figure 6.3 shows results not only for the 3 strategies, but also with two additional Timeout policies with fixed timeouts (10s and 60s). Differently from the results of K8++, Yarn++ shows lower success rates of delivering INC resources which are almost unaffected by the demands of requesting jobs. However, increasing the number of requesting jobs still negatively affects placement latency and task preemptions. The two additional policies with custom timeouts show potential for achieving higher INC success rates, but at the cost of more preemptions. Taking placement latency into account shows that delay scheduling is very sensitive to this approach of selecting alternative MatReqs, because the scheduling logic needs some queuing time for proper scheduling.



(a) Satisfied INC jobs

(b) Preempted tasks

(c) Placement latency, median

(d) Placement latency, 95th

Figure 6.4: Sparrow++ scheduling performance with INC multiplexing, as function of μ (INC demand).

6.4.4 Power of Two Choices

The last case study is based on Sparrow [Ous+13], a scheduler using a variant of power of two choices [Mit01] with batch sampling and late binding. For each pending job with some unscheduled tasks, Sparrow draws $2 \times m$ machines randomly for m pending tasks and enqueues the tasks to the service- or batch queue of the machines. Each time a machine has enough spare resources, it pops the next task to start locally, if this task was not already started on another machine.

Sparrow++. IncSched+Sparrow (Sparrow++) uses a similar approach, but adds a re-check timer for all not fully allocated MatReq. We observed very high placement latency (almost starvation), especially for INC MatReqs, when switches hit their multiplexing limit, and for task groups with few tasks only (which leads to very few machine samples). The re-check timer of Sparrow++ kicks in for every MatReq and checks whether its number of samples is below a threshold. If so, Sparrow++ adds another round of samples. We observed stable results for re-check of 200ms and a 50% threshold.

Results. Figure 6.4 shows placement latency is not negatively affected for the Concurrent and Cluster load policies, which also show stable success rates for delivering INC resources.

These results are achieved by Sparrow++ at the cost of task preemptions, and high RPC/API call load. However, the results also show that Sparrow++ does not fit well the Timeout policy, achieving lowest success rate in delivering INC resources as the number of requesting MatReqs increases, with highest preemption rates and placement latency among all policies.

6.5 Conclusions

This chapter showed how to make existing cluster scheduler designs ready for the INC resource scheduling problem, by proposing an resource management framework called IncSched. Our simulations indicate that using IncSched with existing scheduling policies tackles the scheduling problem in a satisfactory way, achieving high success rates of serving INC job requests, while keeping placement latency low. Some limitations remain, which motivate several challenging research directions.

Scheduling Logic. Our case study evaluation shows great performance for most combinations of plugged scheduling policies and alternative selection strategies. Queuing-based policies (K8++, Yarn++) seem to be overall more sensitive to chosen parameters for specific setups. K8++ shows drastically different placement latency when INC multiplexing enters the picture, and Yarn++ unveils the problem of using delay scheduling for a timeout-driven MatReq decision strategy. Optimistic Concurrent scheduling policies like Sparrow++ seem to better fit INC scheduling, but only as long as IncSched takes care of choosing the MatReq. We see further avenues for improvement on the Concurrent selection strategy, especially how the load estimates (through the API) are integrated into the plugged schedulers, *e.g.*, scoring of machine candidates, or by using scheduling policies acting as global optimizations [Tum+16; Le+20; Gog+16].

Complex Multiplexing. It is an open question how the INC scheduling problem changes with more templates added to the INC template store. Furthermore, future work on INC virtualization techniques might introduce more constraints on INC multiplexing, which might require the network controller to be *in the loop* when checking these constraints.

Complex INC Templates. Thus far we considered all INC templates without their communication patterns, *e.g.*, Net-Chain [Jin+18] is treated similarly to Daiet [Sap+17] (except for switch resource constraints). Intuitively, a scheduler should be able to take better decisions when considering how INC tasks communicate. Yet, considering the INC scheduling problem as a classic virtual network embedding problem is likely to impose algorithms of high complexity.

INC Resource Sharing. IncSched acts as a mediator between jobs with INC resource demand and existing scheduling policies that do not consider INC resources. A limitation of this approach is the way how IncSched translates INC tasks and resources, in order to make them applicable for the plugged scheduler: INC tasks and resources are yet another (isolated) group of server tasks and resources. This restricts expressiveness and heterogeneity that can be encoded for INC scheduling. Most dominantly, the non linear resource usage of INC tasks which we will discuss in the following chapter.

The next chapter presents HIRE, a full-fledged resource scheduler solution. HIRE addresses several open challenges mentioned above.

7

HIRE: A Cluster Resource Manager for INC and Server Resources

Chapter Outline

This chapter introduces HIRE, a new data center scheduler for holistic INC and server resource management. HIRE extends the resource model introduced by IncSched (Chapter 6) to enable automatic resource alternative mappings and to support non-linear resource usage of INC services. Furthermore, HIRE considers all resource alternative of all jobs for server and INC resources in its scheduling logic for holistic scheduling goals. These techniques enable HIRE to perform scheduling with INC in mind, resulting in better placement decisions and higher INC serving rates than state-of-the-art solutions backed by IncSched.

7.1 Overview

The previous chapter discussed the gap of resource managers and INC. To summarize, despite the proliferation of exciting INC applications, running multiple INC applications on the same network, or even having multiple tenants using the same INC application remains under-explored. Existing works focus mostly on isolated scenarios, where network devices are instrumented for benefiting a single application, and evaluations focus on workloads for that application.

This chapter presents HIRE, a Holistic INC-aware Resource managEr supporting INC-enabled applications. HIRE features novel designs aiming at addressing the aforementioned challenges. More specifically, HIRE introduces a novel *resource model* with which jobs are described by *composite requests* specifying both server and INC resource demands. The new resource model also allows for expressing scheduling *alternatives* that will be scheduled mutually exclusively at runtime. HIRE then uses a set of transformation rules to "translate" the composite request of every job into a new form called *polymorphic request*, based on a notion of *composite templates* capturing different target INC platforms accessible to the resource management framework. The polymorphic resource request can also be updated quantitatively at a later time to allow for resource request updates in long-lasting deployments.

HIRE proposes a novel flow-based scheduler to achieve efficient resource allocation leveraging its resource model. Our scheduler features a set of unique designs for the flow network and the cost model. In particular, the flow network incorporates a *shadow network* in addition to the physical network topology to encode both the server and INC resources in the same network, with locality constraints respected through the propagation of the cost model on the network. In addition, the flow network introduces several types of *shortcut edges* to support the selection of scheduling alternatives. The cost model takes into account the non-linear resource sharing behavior and ensures it is respected in the scheduling process. Despite these new features, our scheduler maintains the same scheduling complexity as other flow-based schedulers.

7.1.1 Design Challenges

HIRE builds on IncSched and faces three design challenges, which we discuss in this chapter:

Adopt IncSched Properties. First of all, since HIRE builds on IncSched (Chapter 6), HIRE implicitly tackles all challenges IncSched considers. IncSched presented a new resource model which covers INC and server resources. This was developed with the design goal of being compatible with existing scheduling policies. Even though this has the advantage of easily plugging existing scheduling policies, it has decisive drawbacks (see Section 6.5), most dominantly: (1) an alternative selection strategy that is separated from the plugged scheduling logic, (2) very coarse-grained resource alternatives (job level), and (3) isolated scheduling of each group of resources.
In this chapter, HIRE mitigates these limitations. HIRE is a new scheduler with a scheduling logic that directly integrates server and INC resources, resource alternatives at the level of task groups, and the alternative selection strategy. HIRE follows the path of flow-based schedulers [Isa+09; Gog+16], but with extensions to the flow network, the cost model, and the update mechanisms to support this set of features.

Incorporate all INC Characteristics. Throwing INC resources into the mix adds new challenges and significantly exacerbates existing ones, as already identified and discussed in the previous chapter (Section 6.2) with respect to heterogeneity, dependencies, and alternatives. We now look into the dependency challenge and do a further break down, with a total of four challenges: (1) INC resources such as programmable ASICs and NPUs are highly heterogeneous [het] in terms of not only processing power, but also programming models [Gao+20a; Son+20]; (2) INC resources are relatively scarce compared with server resources, demanding interchangeable resources [alt] as fallbacks; (3) INC-enabled jobs impose fine-grained locality [loc] constraints regarding the underlying network topology, with dependencies between servers and INC appliances; (4) INC resources exhibit non-linear sharing [nol], as contrary to “complete” isolation on servers, partial INC resources may be shared by multiple tenants or INC service(s) [Wan+20].
These new challenges, especially [nol], render the existing resource management frameworks inapplicable, including IncSched, motivating a new resource management framework design for INC-enabled data centers. HIRE tackles these challenges by making the `CompStore` (an extension of the IncSched INC template store) aware of non-linear usage of resources. HIRE leverages this information in its scheduling logic for feasibility match-

ing and scoring of potential resource mappings. This enables HIRE to target holistic scheduling goals.

Automatic Resource Alternatives. IncSched introduced a resource model for combined server and INC resources, with job requests that combine resources with alternatives. This gives users a tool to encode resource alternatives, however, IncSched requires the user to identify potential alternatives. Furthermore, the way how IncSched translates alternatives, to be compatible with existing schedulers, limits the flexibility of scheduling decisions, making resource alternatives coarse-grained and likely to be not effective.
HIRE extends the IncSched INC template store, to make INC templates become *composite templates* (with intertwined server and INC parts), that define resource alternatives along other properties. A user optionally restricts potential resource alternatives selections, but HIRE takes care of automatically transforming jobs into fine-grained alternative descriptions.

7.1.2 Related Work

So far, no work (except IncSched, see Chapter 6) has studied the resource management problem for INC on a data center wide level. We refer to Section 6.1.2 for general related work, and to Section 7.2.1 for a detailed comparison of HIRE's features with related work.

7.1.3 Contributions

In this chapter, we make the following contributions:

1. We present the design of HIRE (Section 7.2.2), including its novel model of resources and corresponding interfaces for applications to interact with it (Section 7.3).

2. We introduce HIRE's novel scheduler following the flow-based approach and our unique designs for the flow network (Section 7.4) and the cost model (Section 7.5).

3. We evaluate HIRE through large-scale simulations with real-world workload traces (Section 7.6). In short, compared with retrofitted state-of-the-art schedulers, HIRE makes better use of INC resources by serving $8 - 30\%$ more INC requests, while at the same time reducing network detours by 20%, and reducing tail placement latency by 50-60%.

7.2 Challenges and System Design

In this section, we first identify the specific challenges to data center scheduling with INC and present our system design.

Approach	**[het]**	**[alt]**	**[loc]**	**[nol]**
HIRE	✓	✓	✓	✓
Heterogeneity-aware resource managers				
Gavel [Nar+20]	(✓)^P	(✓)^{E,S}		
AlloX [Le+20]	(✓)^P	(✓)^{E,S}		
Gandiva [Xia+18]	(✓)^P	(✓)^{E,S}		
Themis [Mah+20]	(✓)^P	(✓)^{E,S}	(✓)^A	
Tetrished [Tum+16]	(✓)^P	(✓)^S	(✓)^A	
Generic resource managers				
Hydra [Cur+19]			(✓)^A	
Omega [Sch+13]			(✓)^A	
Mesos [Hin+11]			(✓)^A	
Yarn [Vav+13]			(✓)^A	
INC switch management				
μP4 [Son+20]		(✓)^S		
INC on demand [Tok+19]		(✓)^{D,S}		

Table 7.1: How existing schedulers cope with INC challenges.
^{P} performance heterogeneity, but not late binding of exact task resource demands with respect to a target device; ^{E} domain-specific solution focusing on performance estimates of alternatives; ^{S} static alternatives, *i.e.*, alternatives specified in the resource request, not induced by the resource manager; ^{D} single device; ^{A} few discrete levels or (anti-)affinity constraints, but no built-in support *e.g.*, for requesting a tree or a chain of devices.

7.2.1 Challenges to Data Center Scheduling with INC

Presence of INC resources fundamentally changes data center scheduling, further complicating the scheduling problem in four ways. Table 7.1 summarizes how existing schedulers cope with these.

Heterogeneity [het]. Existing resource managers consider single- or multi-resources with feature flags [Ous+13; Sch+13; Cur+19], and recently server-accelerators like GPUs [Le+20; Xia+18; Nar+20] with performance heterogeneity. INC resources extend performance heterogeneity: Programmable network appliances are composed of various reconfigurable hardware components, *e.g.*, programmable ASICs, FPGAs, NPUs, in addition to general-purpose CPUs. Several of these components come with limited programming models and interfaces [Son+20; Gao+20a]. Programmable network appliances hence exhibit different levels of "programmability", in contrast to servers which are expected to support general Turing-complete computations. An INC service may thus be implemented following different programming models targeting different appliances. Changing compilation/program synthesis approach can considerably alter resource requirements and performance characteristics of INC services [Jos+15; Gao+19; Gao+20b; Son+20; Gao+20a]. Upon service requests the resource manager needs to interact with the toolchain of a potential target INC switch to determine resource demands like RMT stages (not statically pre-determinable because of non-linear sharing). This makes the scheduling of heterogeneous resources, discussed more broadly in the light of related work (see Section 7.1.2), even more complex.

Alternatives [ALT]. Heterogeneity leads to interchangeable resources pending decisions at runtime. Given the scarcity and diversity of INC resources compared with server resources (*e.g.*, the critical resource of on-chip stateful memory is limited to tens of MB on a Tofino switch [Jin+17]), one must be prepared for many requests for INC resources to be unsatisfiable within a non-trivial timeframe. Fortunately, INC-enabled applications by definition can also be accomplished without INC resources. For example, a partition/aggregate job can go without INC, but will probably run longer, or need more servers to run in the same timeframe. More generally, an INC-enabled job can be specified by a set of substantially different, *interchangeable* resource demands with varying performance properties [Le+20]. Such flexibility adds an extra dimension to the scheduling problem: which resource demand to accept for each INC-enabled job at runtime. Existing domain-specific resource managers consider interchangeable resources requiring job runtime estimation [Le+20; Xia+18; Nar+20], single device decisions targeting energy efficiency [Tok+19], and time-sliced allocations [Son+20] with pre-specified alternatives—none considers resource manager-induced alternatives at runtime. Straightforwardly encoding all combinations in existing models yields prohibitive complexity.

Locality [LOC]. Most INC services come with locality constraints concerning the underlying network topology, *e.g.*, sticking to ToR switches [Jin+17] or using a chain/tree of switches [Jin+18; Sap+17]. Taking a decision for a specific server or switch strongly impacts the value of all other choices. Furthermore, most benefits in INC scenarios have been shown when INC resources are exploited on communication paths between communicating end points [Sap+17; Jin+17]. Adding extra "detours" via specific appliances may cancel benefits or even worsen performance. In short, INC services possess more fine-grained locality requirements than those for pure server jobs where locality is typically described simply with a few discrete levels or (anti-)affinity constraints [Tum+16; Ous+13; Mah+20]. Harmony [Ben19] also discusses INC and server placement constraints, but is limited to relative placement constraints of switches to pre-allocated servers.

Non-linearity [NOL]. In server-centric resource management framework solutions, the underlying assumptions are that all resource requests can be easily made piece-meal, entirely separated from others, and corresponding resources can be easily (de-)commissioned. This may not hold straightforwardly with INC, as the sharing of INC resources often exhibits non-linear behavior [Wan+20]. That is, the runtime resource usage of an INC service may depend on a switch's state: if another tenant is using the switch for the same INC service, some INC runtime resources (*e.g.*, RMT stages [Bos+13] in NetCache [Jin+17] and HovercRaft [KB20]) can be shared among tenants. This means that the first tenant to use an INC service on a switch has to consume extra resources for registering the shared runtime resources for the service. Meanwhile, each tenant still consumes other resources (*e.g.*, SRAM for tenant-specific key-value pairs in NetCache) separately.

These constraints make it hard to adapt existing resource management frameworks and corresponding schedulers to include INC resources.

7.2.2 System Design

Aiming to address all the above challenges, we propose a novel data center scheduler design named HIRE.

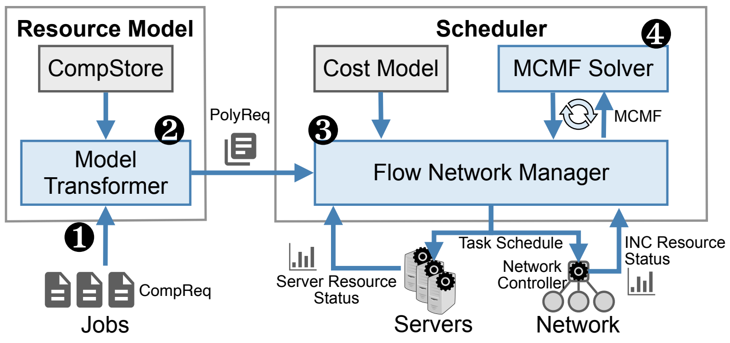


Figure 7.1: HIRE system architecture.

Overview. A high-level overview of the HIRE architecture is shown in Figure 7.1. Tenants ❶ describe their jobs with HIRE's APIs and submit each job as a *composite resource request* (`CompReq`). A `CompReq` is a directed graph of composites (see Listing 7.1 for an example). Once a job is submitted, it goes through the *model transformer* module which ❷ transforms the `CompReq` into a *polymorphic resource request* (`PolyReq`) automatically. The HIRE scheduler takes all the `PolyReq`s as input and ❸ generates a flow network embedding all the scheduling constraints and objectives. HIRE then ❹ solves an min-cost max-flow (MCMF) problem instance with the flow network and produces the final scheduling decisions. HIRE also supports incremental submissions of jobs. In particular, tenants can submit a `CompReq` request and indicate its association to a previously submitted one. The scheduler will consider this association and respect the (locality) constraints in scheduling.

HIRE Resource Model (§7.3). HIRE features a novel resource model where tenants describe and submit their jobs as `CompReq`s. A `CompReq` consists in a set of composites derived from the composite templates (addressing [HET]) pre-configured in the *composite* *template store* (`CompStore`). Using HIRE APIs, tenants can specify the configuration for each of the composites in a `CompReq`, and the way composites for a same job are interconnected ([LOC]). Once submitted, `CompReq`s are transformed into `PolyReq`s by the model transformer module. A `PolyReq` considers the different implementation options for the `CompReq`'s composites and provides more detailed resource demands of the job, incorporating resource alternatives ([ALT]) and non-linear resource usage ([NOL]).

HIRE Scheduler (§7.4). HIRE includes a scheduler to find the mapping of `PolyReq`s to physical resources. The scheduling problem differs from the traditional problem chiefly through the alternatives ([ALT]) and non-linear resource sharing ([NOL]) in the `PolyReq`. The HIRE scheduler takes all the `PolyReq`s as input and applies a flow-based scheduling policy. At each scheduling cycle, all newly submitted `PolyReq`s are aggregated and the scheduler generates a flow network by following a carefully designed cost model defining how to translate current data center resource status, resource demands in `PolyReq`s, and the scheduling objectives into a flow network with costs on arcs. The challenge is to design a cost model that represents not only the scheduling constraints but also the alternatives and non-linearity in `PolyReq` on the flow network. We boil the scheduling problem down to a standard MCMF problem for which HIRE employs an efficient MCMF solver, similar to Firmament [Gog+16]. In the evaluation (Figure 7.12) we test how the modified flow network impacts MCMF solver speed.

7.3 HIRE Resource Model

HIRE introduces a new resource model to unify server and INC resources and address [HET] and [ALT]. In particular, HIRE introduces the key concept of *composite*, which is defined as functional unit with a mix of candidate INC and server implementations. HIRE provides composite *templates* together with their implementation details in the `CompStore`, which masks complexity caused by [HET]. In addition, composites allow tenants to specify implementation alternatives to be scheduled at runtime, addressing the [ALT] challenge. For the sake of simplicity, we chose three resource dimensions for INC switches, namely recirculation capacity, RMT stages, and SRAM (see Section 7.6) and two dimensions for servers (CPU and memory). Note that this can be configured by the user and HIRE is not thusly limited, *e.g.*, ALUs and crossbar units could be considered. Table 7.2 summarizes all notation introduced in the following sections.

7.3.1 Composite Templates

The *composite template* yields the foundation for tenants to construct the different functionalities required by a job. For a target functionality, a composite template provides the APIs for tenants to specify candidate implementations and their requirements. For example, using the `coordinator` composite template a tenant can specify coordination functionality with either or both of the two candidate implementations: INC-based (*e.g.*,

Symbol	Description
	Resource model Section 7.3.3
J	Job request
T	Task
M^s and M^n	Server and INC node
G^s and G^n	Server and INC task group
$\vec{f}_G$	Flavor vector of task group G
	Problem modeling Section 7.4.1
$\vec{x}_J$	Active flavor vector of job J
Z	Task group type
$\vec{d}_G$	Resource demand vector of task group G
$\vec{e}_{Z,M}$	Aggregated resource demands of $\hookrightarrow$ task groups of type Z on M
$a_{T,M}$	Allocation of task T on node M
s_G	Flavor selector for task group G
$\vec{r}_M$	Available resource vector of node M
$\vec{q}_Z$	Sharing degree vector of a task group type Z
y_J	Scheduling decision for job J

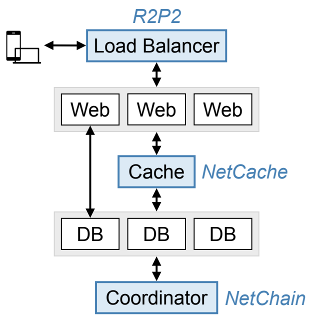
Table 7.2: Notation for HIRE.

NetChain) and server-based. Each of the implementations in a composite template provides an API in the form of a configuration map which the tenant can use to specify the required hardware and software properties. For INC-based implementations, the composite template also holds the semantics as well as the performance profiles of the implementation. This way, tenants can specify the properties for an INC-based implementation at a high level (*e.g.*, throughput of 50MQPS in NetChain), and without having to understand the (usually complex) internals of the implementation to configure it properly in a heterogeneous environment ([HET]). Server-based implementations, however, allow the tenant to provide a detailed configuration map with specific resource demands.

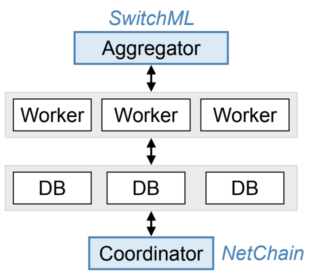
Composite templates are hosted in the `CompStore` (see Figure 7.3a). In addition to pre-configured composites, tenants can expand default and custom-p4 templates for customized ones.

7.3.2 Composite Resource Requests

Tenants submit jobs in the form of *composite resource requests* (`CompReqs`). A `CompReq` is a directed graph of composites specified using HIRE APIs (see Listing 7.1 for an example). Each composite in the `CompReq` is derived from a composite template in the `CompStore`. The directed edges connecting the composites in the `CompReq` indicate their dependencies and serve as input for setting up the inter-composite routing policy.



(a) Typical web application [QCB18].



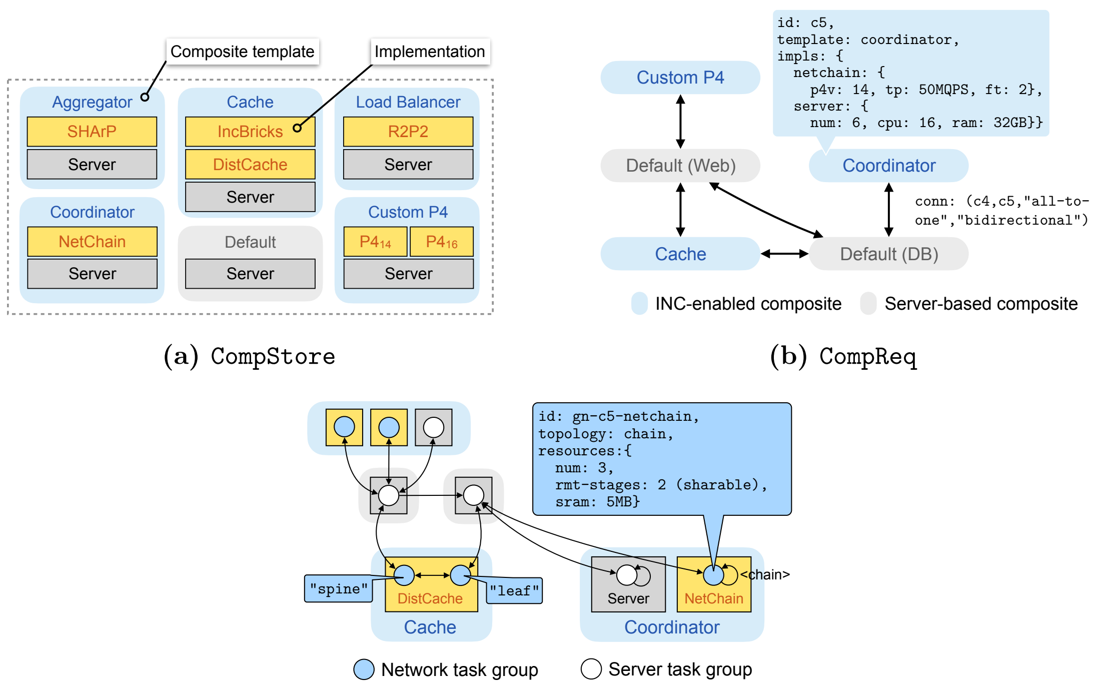
(b) Machine learning training [Li+14; Sap+19].

Figure 7.2: Example applications with potential INC-enabled components highlighted in blue.

```
def setupSendCompositeRequest() {
  val c4 = Composite('c4', CompStore.lookup('Server',
    properties='{cpu:16, mem:8.5, instances:12}'))
  val coordi = CompStore.lookup('Coordinator',
    filterImpl=None, properties='{tp:50MQPS, ft:2}')
  coordi.impl.foreach(impl => { /* custom modify req. */})
  val c5 = Composite('c5', coordi)
  val composites = c4 :: c5 :: /* ... */ :: Nil
  val connections = Connect(c4, c5, Connect.Bidi) :: Nil
  val prio = Priority(requestPriority)
  ComReq(prio, composites, connections)
}
```

Listing 7.1: API for an application master to send a `CompReq`.

Figure 7.3b shows a `CompReq` with 5 composites for the typical web application shown in Figure 7.2. As an example, the composite `c5` (see the code snippet) is derived from the `coordinator` template in the `CompStore` and two implementations are specified by the tenant. The implementation `netchain` is specified with the following configuration map: `{p4v:14,tp:50MQPS,ft:2}` which instructs the requirements that the INC nodes for `netchain` have to support P4$_{14}$, the throughput has to be at least 50MQPS, and the setup should be able to tolerate up to 2 concurrent node failures. In addition, locality constraints (*e.g.*, `locality:tor`) can also be specified with the configuration map. For the implementation `server`, a configuration map with detailed resource demands is specified by the tenant as 6 servers (*e.g.*, containers) each equipped with 16 CPU cores and 32GB of RAM. HIRE also allows tenants to specify multiple versions for the same implementation in a composite template by supplying different configuration maps.



```
id: c5,
template: coordinator,
impls: {
  netchain: {
    p4v: 14, tp: 50MQPS, ft: 2},
  server: {
    num: 6, cpu: 16, ram: 32GB}}
```

```
conn: (c4,c5,"all-to-
one","bidirectional")
```

```
id: gn-c5-netchain,
topology: chain,
resources:{
  num: 3,
  rmt-stages: 2 (sharable),
  sram: 5MB}
```

(a) `CompStore` **(b)** `CompReq` **(c)** `PolyReq`

Figure 7.3: HIRE resource model for the web application scenario of Figure 7.2a: (a) `CompStore` of HIRE with 6 composite templates, (b) schematic representation of a `CompReq`, and (c) the `PolyReq` derived from the `CompReq` by the model transformer module.

The configuration of inter-composite connectivity between composites "`c4`" and "`c5`" is also shown in Figure 7.3b. Here, the connection type is all of "`c4`" to one of "`c5`" and is bidirectional. The `CompReq` could be easily extended to support also bandwidth requirements by annotating the directed edges in the `CompReq` with bandwidth demands and/or latency constraints, although this is not in focus of this work.

7.3.3 Polymorphic Resource Requests

HIRE transforms each submitted `CompReq` into a *polymorphic resource request* (`PolyReq`) which is more amenable as input for the scheduler. A `PolyReq` is specified by a set of connected task groups. Each task group G represents a bundle of identical tasks that require the same resources indicated by a demand vector $\vec{d}$. The task groups in a `PolyReq` may have two types—a server task group G^s runs on server nodes and a network task group G^n runs on INC nodes.

Figure 7.3c depicts the `PolyReq` that is transformed from the `CompReq` shown in Figure 7.3b. The composite coordinator is transformed into two task groups, each for one of the alternative implementations. In some cases, an implementation may be transformed into multiple task groups, such as the DistCache implementation for the `cache` composite where two task groups "spine" and "leaf" are generated. The task group for the implementation `netchain` (shown in the code snippet) has a size of 3 and is accompanied by the following resource demands: `{rmt-stages:2(sharable),sram:5MB}`. The "`sharable`" label after the resource quantity indicates that this resource can be shared among multiple tenants involving the same implementation. This sharing behavior will be taken into account by the HIRE scheduler ([NOL]). The topology of this task group is specified as a chain, meaning that all the tasks in this task group will be traversed sequentially. The resource demands of the task group for the implementation `server` is derived directly from the configuration map of the implementation.

As the implementations specified in a `CompReq` for each composite are alternatives to each other, *i.e.* only one will be actually scheduled at runtime, the corresponding task groups for these implementations in `PolyReq` are also exclusive to each other. To support this, `PolyReq` introduces the concept of *resource flavor* ([ALT]), and assigns each task group a *flavor vector* $\vec{f}$. The size of $\vec{f}$ equals the total number of decision variables required to encode the `CompReq`, which in most cases is smaller than the total number of task groups. Each element in the flavor vector of a task group represents the relationship of this task group to others and has three possible states: "`0`" (mutually exclusive), "`1`" (concurrent), and "`x`" (ignorable). All $\vec{f}$ of a `PolyReq` are of same length (or padded with "`x`" entries). For example, in the "cache" composite, the flavor vector for the "spine" task group for the `distcache` implementation is ⟨`xxxx11xxx`⟩, meaning that the "leaf" task group will have to be scheduled concurrently with "spine". In contrast, in the "coordinator" composite the flavor vector for the task group for the `netchain` implementation is ⟨`xxxxxxx01`⟩, and for the `server` implementation is ⟨`xxxxxxx10`⟩, meaning that only one of the task groups for the `netchain` and `server` implementations in the "coordinator" composite will be scheduled. We will explain how the scheduler uses the flavor vector to track mutually exclusive implementations in Section 7.4.

7.3.4 Model Transformation

The `CompStore` holds information on how to transform a `CompReq` to a `PolyReq`, by applying graph transformation rules. This allows HIRE to build more complex topologies for specific implementations of a composite template, and allows to hide INC service specific implementation details from the user ([HET]). Our HIRE prototype uses Scala code to describe transformation rules in the `CompStore`, but we could also use a graph domain-specific language (DSL) like GraphIt [Zha+18].

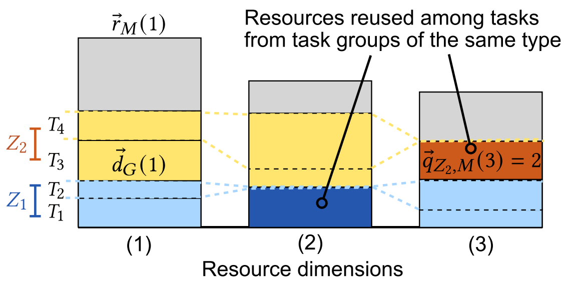


Figure 7.4: Non-linear resource sharing example.

7.3.5 Limitations

HIRE utilizes the information of composite templates for translating resource requests, creating alternatives, and unwrapping resource sharing constraints. To ensure correct deployment profiles of new INC services, especially for all heterogeneous switches of a data center, new INC services must first be added to the `CompStore` (*e.g.*, by the INC service implementer), before users can use them in a `CompReq`. We do not consider this to be a limitation of the expressiveness or flexibility of HIRE, rather it leads to a more reliable operation of INC services. New (feature) flags/dimensions of future INC services can be added in a backward-compatible manner, since the HIRE resource model builds on directed graphs with configuration dictionaries for composites and their connections.

7.4 HIRE Scheduler

HIRE has multiple scheduling problems to solve: (1) which flavor to take for each of the `PolyReq` ([ALT]), (2) which server takes which server task, and (3) which switch takes which INC task. The decision for each of these problems influences the available options ([NOL]) and possible scheduling quality to reach for each other problem ([LOC]). Table 7.2 lists the used notations.

7.4.1 Problem Modeling

The scheduling problem can be considered as a variant of the general multi-dimensional bin packing problem [SSS11]. We formalize a simplified version of it to highlight the new challenges mentioned above. This formalization is not comprehensive, but captures the most important factors ([ALT], [NOL]).

The scheduling problem concerns determining the flavor of each job and mapping the correct task groups in every `PolyReq` onto data center resources, with the goal of maximizing job success rate (and/or other goals), while respecting resource capacity constraints. We use binary indicator y_J to represent the scheduling decision for job J where $y_J = 1$ if J is scheduled and 0 otherwise. Assume after scheduling $\vec{x}_J$ produces the final selected flavor of job J. The status $s_G \in \{0,1\}$ of task group G in the final scheduling decision is given by $s_G = (||\vec{f}_G \wedge \vec{x}_J||_1 > 0)$ where $s_G = 1$ means G is selected and 0 otherwise. Note that the elements with value "`x`" in $\vec{f}$ are skipped in the "$\wedge$" operation since they stand for ignorable states. A job is successfully scheduled if all its selected task groups, *i.e.* those having $s_G = 1$ in its `PolyReq`, are successfully scheduled. This refers to the gang-scheduling problem where we do not allow partial scheduling of a job. We use matrix $[a_{T,M}]$ to denote the task-to-node mapping decisions; $a_{T,M} = 1$ indicates task T is mapped to node M and $a_{T,M} = 0$ otherwise. To model non-linear resource sharing, we assume task groups are categorized into types, and tasks in task groups of the same type can share resources on the resource dimensions specified with the "`sharable`" flag in the `PolyReq`. Z denotes a task group type and $Z(G)$ the type of task group G. Figure 7.4 shows an example where all tasks in task groups of type Z_1 share the resources on the second resource dimension while it is the third resource dimension for type Z_2. For any Z, the total number of tasks that are assigned to node M is given by

$$n_{Z,M} = \textstyle\sum_J \sum_{G\in J:Z(G)=Z} \sum_{T\in G} y_J s_G a_{T,M}. \tag{7.1}$$

Combined with the "`sharable`" flag, we define a sharing-degree vector $\vec{q}_{Z,M}$ which has the same size as the resource demand vector. An element in $\vec{q}_{Z,M}$ is equal to $n_{Z,M}$ if the corresponding resource dimension is sharable and 1 otherwise. The aggregate amount of resources demanded by all tasks from task groups of type Z on node M is given by

$$\vec{e}_{Z,M} = \textstyle\sum_J \sum_{G\in J:Z(G)=Z} \sum_{T\in G} y_J s_G a_{T,M} \vec{d}_G. \tag{7.2}$$

Our scheduling problem can be characterized as an integer program (IP):

$$\max \textstyle\sum_J y_J \quad s.t. \tag{7.3}$$
$$\textstyle\sum_Z \vec{e}_{Z,M} \oslash \vec{q}_{Z,M} \le \vec{r}_M, \forall M \tag{7.4}$$
$$\textstyle\prod_{G\in J} \prod_{T\in G} \sum_M s_G a_{T,M} = y_J, \forall J \tag{7.5}$$

"$\oslash$" stands for Hadamard division which is applied element-wise between two vectors. The first constraint guarantees that the resource capacities are respected on all nodes, which also takes into account non-linear sharing behavior. The idea is to divide the total resource consumptions by the sharing degree captured by $\vec{q}_{Z,M}$ on the sharable resource dimensions for each task group type Z. The second constraint is a combination of non-linear constraints and ensures that a job is scheduled only if all tasks in all its tasks groups with $s_G = 1$ are scheduled. The IP formulation shows that the search space is extremely large. An exact solution is likely to be impractical due to scalability issues, especially when we consider data centers with thousands of servers and INC nodes. Thus we present a heuristic that can achieve high efficiency and scale to large scenarios.

7.4.2 Flow-based Scheduling Approach

Our heuristic leverages graph theory. In particular, we transform the scheduling problem into a MCMF problem.

Approach Overview. Flow-based scheduling, first introduced with Quincy [Isa+09], uses a flow network to take scheduling decisions on servers. In the basic variant (for slot-based scheduling), each task spawns a unitary flow which could either pass by a node corresponding to a server resource, or by an "unscheduled" node before reaching the sink. After applying an MCMF solver, the scheduler extracts for each flow the server resource node (a valid allocation) or the unscheduled node (postponed allocation). When considering multi-dimensional resources (heterogenous tasks), the flow network must ensure that each flow of a task node can only reach servers with matching available resources. Existing approaches (*e.g.*, CoCo [Sch15, §7.3]) enforce multi-dimensional resource constraints of servers by assigning each edge from a server to the sink a capacity of one, and by connecting each task node to the flow network so only servers with matching available resources or the unscheduled node are reachable. This way, at most one additional task is allocated on each server during a scheduling attempt. An alternative flow network with vector-based flows could allocate multiple tasks on the same server in one attempt, but solving vector-based MCMF problems is unlikely to become feasible within reasonable time [Sch15, §C.4.2]. HIRE extends flow-based scheduling with unique features to meet its requirements (Section 7.2.1) as follows.

Capturing INC Constraints. We propose the following novel designs to handle the following INC constraints.

Resource locality ([loc]): Both server and INC resources need to be integrated in a single flow network so HIRE can schedule resources jointly. When doing so, we must ensure that no flow of a server task can reach nodes referring to INC resources, and vice versa. HIRE achieves this by *having two representations of the data center topology in the flow network*, one for server and a shadow one for INC resources. HIRE knows which of the flow network nodes refers to which location in the topology, so it can transfer locality and cost term information from the server to the INC part and vice versa, without letting flows of server nodes pass INC resources. We propose two algorithms for the HIRE cost model to reflect server and INC locality constraints, also jointly (*i.e.* across the two parts of the flow network).

INC heterogeneity ([het]), non-linearity ([nol]): INC services not only consume resources of a "multi-dimensional" resource vector, but have complex dependencies, *e.g.*, the need of a switch feature. Furthermore, when (de)allocating an INC task on a switch, the number of running INC service instances may change depending on the sharing nature of involved services. HIRE keeps track of these dependencies by *propagating status information along the network*, so all possible flows in the flow

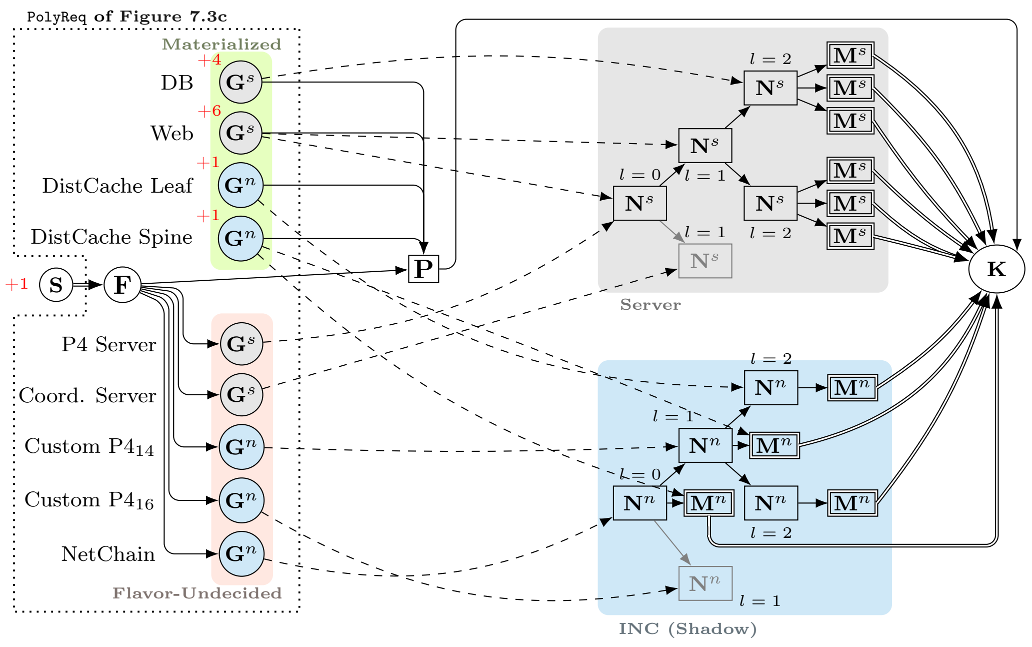


Figure 7.5: HIRE flow network for Figure 7.3c. Double edges have capacity of 1. Dashed edges are shortcut edges. Numbers in red are positive supplies. l denotes node depth in the topology.

network end in valid allocations. More importantly, the propagated, cached, status information of the flow network allows HIRE to quickly find matching resources for requesting tasks, respecting heterogeneity and non-linearity.

Resource alternatives ([ALT]): Scheduling decisions for resource alternatives require joint consideration of server and INC resources, so that all parts of a flavor take resource availability into account. HIRE resolves this problem by *adding a flavor selector node for each corresponding job to the flow network.* HIRE connects the task groups belonging to the flavor-undecided part of a job to the job's flavor, and sets their own supply to 0. The HIRE cost model ensures that each possible flow of the flavor selector considers the joint cost of a flavor, so that a MCMF solver selects the flavor which fits best the current cluster utilization, considering all alternatives of all jobs simultaneously.

7.4.3 HIRE Flow Network Structure

We show how to use the above novelties to build a HIRE flow network. Figure 7.5 shows an example for the `PolyReq` of Figure 7.3c.

Nodes. The flow network holds nodes of following types: one super flavor selector node (**S**); tasks group nodes (**G**) including server task groups ($\mathbf{G}^s$), and network task groups ($\mathbf{G}^n$) according to the `PolyReq`s which are further categorized into flavor-undecided and materialized (with flavors decided) ones; one postponing node (**P**) for each job; one flavor selector node (**F**) for each job that has alternatives; data center resource nodes (**M**) including server resource nodes ($\mathbf{M}^s$) and INC resource nodes[1] ($\mathbf{M}^n$); auxiliary nodes (**N**) for the shadow network (for brevity only half is shown in Figure 7.5); and one sink node (**K**).

Edges. The **S** node connects to all flavor nodes **F** in the graph (with edges of capacity 1). A **G** node has a connection from **F** if it belongs to the flavor-undecided part of the job. A **G** node is also connected to **M**/**N** nodes via *shortcut* edges (dashed lines in the figure). We call them shortcut edges since there can be several of them to encode scheduling preferences. An edge $\mathbf{G} \rightarrow \mathbf{M}$ indicates that **M** contains enough resources to run at least one task in **G**, while an edge $\mathbf{G} \rightarrow \mathbf{N}$ indicates that all resource nodes that can be reached via **N** can run at least one task in **G**. An edge $\mathbf{G} \rightarrow \mathbf{P}$ allows the flow network to postpone the scheduling of **G**. All **M** and **N** nodes are interconnected following the physical network topology. All resource nodes **M** and the postponing node **P** connect to the sink node **K**.

Figure 7.5 shows an example flow network for the `PolyReq` of Figure 7.3c (this example shows a single job, but HIRE holds all pending jobs and task groups in a single large flow network). In this example, the flavor of 5 task groups is not yet decided, so these task groups belong to the flavor-undecided part of the job. All other task groups of this job with flavors decided (4 task groups) belong to the materialized part. Their supply equals the number of remaining tasks to start. If this example graph shows the whole flow network HIRE is working on in the ongoing scheduling round, HIRE can allocate up to 12 tasks ($4 + 6 + 1 + 1$) in the materialized part, and up to 1 task allocation in the flavor-undecided part, but in total limited by the number of available resource nodes ($\mathbf{M}^n$ and $\mathbf{M}^s$) for serving tasks (resource nodes have edges of capacity 1). In general, HIRE can perform as many decisions in the flavor-undecided part, as jobs take part of the flavor-undecided part, but at most 1 decision per job (the **S** node connects to all flavor nodes **F**, each with an edge of capacity of 1).

7.5 HIRE Cost Model

Table 7.3 lists all notation used and introduced by the HIRE cost model. The HIRE cost model is summarized as follows.

There are two sources of positive supplies in the HIRE flow network:

[1] Each switch has an **N** node and if it provides INC resources, an $\mathbf{M}^n$ node is attached next to it for the INC part.

Symbol	Description
E	Edge in the flow network
$\|G\|$	Number of tasks in task group G
σ_E, $\vec{\sigma}_E$	Cost (σ_E) of edge E, summarizes $\vec{\sigma}_E$
$\top$	Cost value for normalization (no upper bound cost term)
Φ_i	Cost function i used in Table 7.4 and Table 7.5
w_J	Waiting time of job J
$\vec{u}_M$	Utilization vector of node M
c_E	Capacity of edge E
χ	Parameter: Level of detail for shortcut edge
γ	Parameter: INC locality gain
ξ	Parameter: Decay factor for γ propagation
$\Gamma_{N,G}$	INC locality gain of task group G and machine M
$\Upsilon_{N,G}$	VM locality gain of task group G and machine M

Table 7.3: List of notation used for HIRE's cost model.

1. supply of **S** is given by the number of **F** nodes or a customized upper-bound to limit the number of flavor decisions per scheduling round and
2. supply of a materialized **G** node equals the number of tasks in the task group.

The capacity of all edges **S** $\rightarrow$ **F** is set to one since we allow only one flavor decision per job in one scheduling round. All edges **M** $\rightarrow$ **K** also have a capacity of one where only one decision is allowed for each resource node in one round. The costs on edges are assigned as follows. For edges **M** $\rightarrow$ **K** in the server part, the cost is proportional to the node utilization and balance level of resource dimensions computed as the standard deviation of the utilizations of all resource dimensions, while for the INC shadow part, the cost is proportional to the node depth in the topology and the number of active INC services that are already running on the INC node. For edges **F/G** $\rightarrow$ **P** the cost is proportional to the job queueing time and the number of scheduled tasks of the job. Shortcut edges **G** $\rightarrow$ **M/N** have costs proportional to the utilization and the balance level of the corresponding resource nodes (in the subtree). Job priority and non-linear resource sharing behavior are also encoded in the cost of shortcut edges. The cost for **F** $\rightarrow$ **G** edges is an approximation of the total cost in the corresponding flavor.

Similarly to CoCo [Sch15], on the server nodes we propagate two numerical vectors of lower and upper bounds of the available resources for the shortcut edge construction. For INC nodes, in addition to the numerical vectors, three bit vectors of size of the number of INC services are used for flagging whether at least one node in its subtree supports the INC service, an INC service is active on all nodes, and an INC service is active on at least one node, respectively. Moreover, each **N** node maintains a map containing a counter for the running tasks of a task group in the subtree rooted at **N**; this map is propagated in the flow network via a gossip-like protocol.

Flow Network Updates. The flow network is updated upon job arrivals and completions. When jobs arrive, HIRE starts to prepare the next scheduling round by adding or updating the job in the flow network. For a new job J, HIRE initializes the current selected flavor $\vec{x}_J = \langle \mathtt{x} \dots \mathtt{x} \rangle$ (see Section 7.4.1) and adds the job's postpone node **P** to the flow network. For each (new) task group of the job, HIRE compares $\vec{f}_G$ with $\vec{x}_J$ and adds a **G** node either to the flavor-undecided part or to the materialized part of the job. If all decision variables of $\vec{f}$ (except `x`) are equal to $\vec{x}$, then **G** belongs to the materialized part. If there is at least one contradiction ($\mathtt{0} \neq \mathtt{1}$), the task group is not in the job (anymore). In all other cases ($\vec{x}$ has `x` overlapping with $\vec{f}$), **G** belongs to the flavor-undecided part. Finally, a **P** node is added for the job and each new task group is connected to **P**. The edge costs are updated following our cost model. Upon job completions, the flow network is not immediately updated. Instead, a special flag is assigned to the nodes/edges that are affected. The flow network is updated at the beginning of each scheduling round using the flags on nodes/edges.

When HIRE processes the result of an MCMF instance, allocations of **G** nodes of the flavor-undecided part trigger updates of the corresponding $\vec{x}$, *i.e.* overwriting `x` values with `0/1`. Before moving to the next scheduling round, HIRE checks all **G** nodes of the flavor-undecided part (of updated $\vec{x}$) to see whether they still belong to the flavor-undecided/materialized part or are not relevant for the job anymore.

Cost Model Goals. The cost model of HIRE, together with the flow network, offers the following properties:

1. balancing switch and server utilization,
2. co-locating, if possible, INC service instances of the same INC service to maximize resource sharing benefits and keep the set of active INC services (per switch) small,
3. informed flavor selection where the scheduler always tries to select the "cheapest" flavor with respect to the task counts in task groups and the aggregate flavor costs of tasks in all the task groups belonging to the flavor, and
4. locality-aware scheduling of tasks on machines close to (or covered by the same network topology tree) the running tasks of the same or directly connected task groups.

HIRE uses a multi-dimensional cost vector $\vec{\sigma}$ for each edge in the flow network. We further transform $\vec{\sigma}$ to a scalar cost value σ, so that HIRE can run the MCMF problem. To this end, we flatten $\vec{\sigma}$ by applying a weighted sum. The weights can be used to model priorities or other custom policies. Table 7.4 and Table 7.5 summarize all cost terms of $\vec{\sigma}$, and refer to sub-cost functions Φ specified below.

$\vec{\sigma}$ **elements**	$\mathbf{N \rightarrow M/N}$	$\mathbf{M}^s \rightarrow \mathbf{K}$	$\mathbf{M}^n \rightarrow \mathbf{K}$	$\mathbf{S \rightarrow F}$
Utilization	-	avg($\vec{u}$)	avg($\vec{u}$)	-
Multiplexing	-	1-stddv($\vec{u}$)	1-stddv($\vec{u}$)	-
Locality	-	-	Φ_{ToR}	-
Interference	-	-	$\Phi_{\lfloor \text{P} \rfloor}$	-
Priority	-	-	-	-
Flattening	-	avg($\vec{\sigma}$)	avg($\vec{\sigma}$)	-
Penalty	-	-	-	$\top$

Table 7.4: Job-independent costs: multi-dimensional cost vectors for each edge as specified in the table. Before sending the graph to the solver, HIRE flattens $\vec{\sigma}$ as shown in the second last row into the range $[0, \top]$, and for some edges we add a penalty (last row).

Job-independent Costs. Table 7.4 shows job-independent edge costs for evaluating machine resource utilization and resource balancing. Costs are lower for machines with lower utilization, and with higher variation among the load of all resources dimensions. Furthermore, $\mathbf{M}^n \rightarrow \mathbf{K}$ considers the network level in the topology and the number of different active INC services, so that it is less attractive to choose a switch for INC that is not close to a server or which combines more different INC services on the same $\mathbf{M}^n$. More specifically, we define two cost functions:

$\Phi_{\lfloor \mathbf{P} \rfloor}$ – A cost term proportional to the number of active INC services on an $\mathbf{M}^n$, normalized to the maximum number of INC services that could run on a particular $\mathbf{M}^n$.

$\Phi_{\mathbf{ToR}}$ – A cost term inversely proportional to the number of network hops an $\mathbf{M}^n$ node is away from its closest $\mathbf{M}^s$ node, normalized to the largest possible distance.

Job-dependent Cost. Table 7.5 lists all job-dependent edge costs. The first two rows (utilization and multiplexing) define cost terms so that HIRE prefers allocations for which the resource demand matches better the available resources. More specifically, the cost is smaller if the task group uses a similar portion, with respect to current load, in each resource dimension. Furthermore, we define the following cost functions for locality, resource interference, and priority. The high-level goal of these cost functions is to co-locate o tasks (Φ_{loc}), leverage INC resource sharing (Φ_{new}), and prioritize long waiting task groups (Φ_{delay}).

$\Phi_{\mathbf{loc}}$ – For server tasks, HIRE prefers subtrees which already host tasks of the same or a directly connected task group of the same `PolyReq`. For INC tasks, HIRE prefers switches that are close (in terms of network hops) to other switches involved

$\vec{\sigma}$ **elements**	$\mathbf{F} \to \mathbf{G}$	$\mathbf{F} \to \mathbf{P}$	$\mathbf{G}^s \to \mathbf{N}^s/\mathbf{M}^s$	$\mathbf{G}^n \to \mathbf{N}^n/\mathbf{M}^n$	$\mathbf{G} \to \mathbf{P}$
Utilization	$\Phi_{\hat{\vec{x}}}$	-	$\mathsf{avg}(\vec{d} \oslash \vec{r})$	$\mathsf{avg}(\vec{d} \oslash \vec{r})$	-
Multiplexing	-	-	$\mathsf{stddv}(\vec{d} \oslash \vec{r})$	$\mathsf{stddv}(\vec{d} \oslash \vec{r})$	-
Locality	-	-	Φ_{loc}	Φ_{loc}	-
Interference	-	-	$\top$	Φ_{new}	-
Priority	-	Φ_w	Φ_{prio}	Φ_{prio}	Φ_{delay}
Flattening	$\mathsf{avg}(\vec{\sigma})$	$\mathsf{avg}(\vec{\sigma})$	$\mathsf{avg}(\vec{\sigma})$	$\mathsf{avg}(\vec{\sigma})$	$\mathsf{avg}(\vec{\sigma})$
Penalty	Φ_{pref}	$3\top$	-	-	$5\top$

Table 7.5: Job-dependent costs: multi-dimensional cost vectors for each edge as specified in the table. Before sending the graph to the solver, HIRE flattens $\vec{\sigma}$ as shown in the second last row using a weighted average function into the range $[0, \top]$, and for some edges we add a penalty (last row). $\oslash$ refers to the element wise division (Hadamard division).

in the same or connected task group. We combine the two locality preferences so that switches consider servers and vice versa, simply by checking both flow network parts (server and shadow) for the same node in the topology for calculating the cost term. More specifically, we define two locality metrics, Υ (Equation 7.6) for the server part of flow network (with $\mathbf{M}^s$ and $\mathbf{N}^s$), and Γ (Figure 7.6) for the INC shadow network (with $\mathbf{M}^n$ and $\mathbf{N}^n$). HIRE takes the weighted average (using task counts) of Υ and Γ and normalizes the value afterwards.
There are three cases to consider: (a) For $\mathbf{G} \to \mathbf{N}$, Φ_{loc} checks the two nodes $\mathbf{N}^n, \mathbf{N}^s$ that correspond to the same location in the data center, and returns the combination of $\text{norm}(\Gamma_{N^n,G})$ and $\Upsilon_{N^s,G}$, respectively. (b) For $\mathbf{G} \to \mathbf{M}^n$, Φ_{loc} simply considers the corresponding $\mathbf{N}^n$ to calculate the costs as per (a). (c) For $\mathbf{G} \to \mathbf{M}^s$, Φ_{loc} considers a simplified version of Equation 7.6 to evaluate the number of tasks running on $\mathbf{M}$, but considers Γ and Υ of the parent $\mathbf{N}^s$ for the connected $\mathbf{G}^s$. Υ is recursively defined:

$$\Upsilon_{N_1^s,G} = \frac{\sum_{N_2^s \in children(N_1^s)} \begin{cases} \frac{|G| \text{ not running on } N_2^s}{|G|} & N_2^s \in \{M^s\} \\ \Upsilon_{N_2^s,G} & N_2^s \in \{N^s\} \end{cases}}{|children(N_1^s)|} \tag{7.6}$$

$\mathbf{\Phi_{new}}$ – Prefer switches with matching INC service already active, and switches with more active INC services. If a $\mathbf{G}^n$ node uses an INC service that is already active on a switch, return 0, otherwise, $1/(\delta + 1)$ with δ the number of active INC services on a switch divided by the max possible.

$\mathbf{\Phi_{pref}}$ – This term adds a penalty cost according to the job's waiting time, using two configuration parameters for lower and upper bound. If waiting time is below the

```
procedure IncLocProp ($N_{start}$, $G$, $\gamma$)
    $\mathcal{N}_{visited} \leftarrow \emptyset$
    $\mathcal{N}_{visit} \leftarrow \{N_{start}\}$
    while $\gamma > 0$ and $\mathcal{N}_{visit} \neq \emptyset$ do
        $\mathcal{N}_{next} \leftarrow \emptyset$
        forall $N \in \mathcal{N}_{visit} \setminus \mathcal{N}_{visited}$ do
            $\Gamma_{N,G} \leftarrow \Gamma_{N,G} + \gamma$                // propagate
            $\mathcal{N}_{visited} \leftarrow \mathcal{N}_{visited} \cup \{N\}$
            $\mathcal{N}_{next} \leftarrow \mathcal{N}_{next} \cup$ neighbors($N$)
        $\mathcal{N}_{visit} \leftarrow \mathcal{N}_{next} \setminus \mathcal{N}_{visited}$
        $\gamma \leftarrow \lfloor \gamma/\xi \rfloor$                   // decay propagation
```

Figure 7.6: INC Locality Propagation.

thresholds, Φ_{pref} returns $\top$, if its above, it returns 0, otherwise $3\top \times (-\mathsf{tanh}(ratio \times 3 - 3))$, with *ratio* the linearly scaled inverse waiting time within the range.

$\Phi_{\hat{\vec{x}}}$ – HIRE uses a total cost estimate for each possible flavor, so that when selecting any of the possible task groups, also the costs of other tasks groups are considered. $\Phi_{\hat{\vec{x}}}$ depends on $\mathbf{G} \rightarrow \mathbf{M/N}$ and $\vec{f}$ for estimating the overall cost of a flavor. While updating all shortcuts ($\mathbf{G} \rightarrow \mathbf{M/N}$), HIRE updates an approximate cost estimate of each of the involved flavors of $\mathbf{F}$ as follows. The cheapest shortcut edge $\mathbf{G} \rightarrow \mathbf{M/N}$ of each task group multiplied by $|G|$ gives the total cost estimate for G. The cost estimate for a flavor is the sum of all involved G estimates. $\Phi_{\hat{\vec{x}}}$ returns for each flavor a cost proportional to the ratio of the estimated flavor cost term compared with the largest flavor cost term.

$\Phi_{\mathbf{prio}}$ – Proportional to job priority: 0 (highest), $\top$ (lowest).

$\Phi_{\mathbf{delay}}$ – Prefer placement of tasks with longer waiting time and with fewer tasks remaining. w_J compared with other jobs, considering number of scheduled tasks of the given G, using $w_J \times e^{|G|_{\text{ scheduled }}/|G|}/(\mathsf{max}\ w \times e)$.

Φ_w – Postpone the flavor decision, if there are only very expensive options available. Φ_w uses a threshold and returns $\top$ if w_J is above the threshold, or $\top \times (0.5 \times \mathsf{cos}((ratio - 1.0) \times \pi) + 0.5)$, with *ratio* equals w_J divided by the threshold.

7.6 Evaluation

We use a workload trace of a 4000 machine cluster to run large-scale experiments to address following questions:

RQ1 How successful is HIRE at fulfilling INC requests as overall demands for INC increase (Section 7.6.3)?

RQ2 How well does HIRE handle resource sharing and INC server locality dependencies (Section 7.6.4)?

RQ3 What is the impact of INC resource heterogeneity on the scheduling problem (Section 7.6.5)?

RQ4 How well does HIRE handle resource contention to improve on tail placement latency (Section 7.6.6)?

7.6.1 Retrofitting Existing Schedulers

All experiments compare HIRE against retrofitted variants of four existing schedulers, using IncSched (Chapter 6). In summary, IncSched mitigates the limitations of the retrofitted schedulers in the face of INC challenges as follows: (1) cannot handle interchangeable INC resources → transform requests with alternatives beforehand by creating two variants for each job; (2) cannot suitably capture topological constraints → ignore topologies; (3) cannot track actual resource reuse among co-located INC services → ignore sharing, *i.e.*, INC services do not benefit from reusing resources; (4) no runtime dependency support → substitute retrofitted scheduler's own device list with our simulator API that filters for feasible nodes, *i.e.*, borrowing semantics from HIRE.
More detailed, for these baselines, IncSched treats switches like a distinct group of servers: when a baseline policy wants to iterate over all possible switches for a specific INC service, the simulator returns only those machines (switches) matching resource constraints, INC compatibility, and INC multiplexing constraints. Each baseline runs each experiment with two *modes* for handling job alternatives (INC vs server):

Concurrent submits all INC-enabled jobs simultaneously as a server-only and a strict INC job variant, and withdraws the job counterpart on the first allocation that does not fit both variants.

Timeout submits only the INC variant of each job, but submits the server fallback variant if the INC variant is not served within a timeout. The Timeout mode uses 10% of a job's duration as a timeout for each job. For Yarn++, we also use other Timeout configurations in Section 7.6.7.

We do not compare HIRE *vs.* baselines using the Timeout and Concurrent modes of IncSched, but not IncSched's Cluster load policy (IncSched's Cluster load is not compatible with [nol]). We implement four baselines:

Yarn++: A queuing-based delay scheduler [Zah+10] inspired by the Yarn [Vav+13] capacity scheduler with two queues (batch/service jobs) with FIFO ordering using task submission times. Yarn++ applies rack-aware scheduling to improve locality (delays: 50ms re-check; 100ms rack-preference).

K8++: A queue-based best-effort policy inspired by K8's [Bur+16] default configuration, with two active and one backoff queue(s). Similarly to Omega and Borg [Bur+16], (1) K8++ iterates over all machines in a round-robin fashion to find at least 5% of the total machines which are capable to serve the current request. Then, (2) K8++ checks this machine subset to find the best candidate for serving the request and allocates the resources. For the next request, (1) resumes where it stopped before. We use the default multi-dimensional cost model, and a sample size of 10%.

CoCo++: A flow-based scheduler with a flow network and cost model inspired by CoCo [Sch15] (Firmament [Gog+16]), using the same MCMF solver as HIRE. CoCo++ considers INC resources by adding one virtual rack for each INC service, each connecting to all compatible switches at the time of scheduling. This enables CoCo++ to incorporate INC compatibility checks, however, without considering [NOL] and [LOC]. Furthermore, CoCo++ cannot handle job alternatives within a scheduling round, thus CoCo++ runs only in `Timeout` mode.

Sparrow++: A distributed scheduler using a variant of power of two choices [Mit01] with batch sampling and late binding inspired by Sparrow [Ous+13]. For each pending job with some unscheduled tasks, Sparrow++ draws $2 \times m$ machines randomly for m pending tasks and enqueues the tasks to the service- or batch queue of the machines. Each time a machine (server or switch) has enough spare resources, its Sparrow++ agent checks the next task to start locally, via RPCs to a central Sparrow++ instance. We observed very high placement latency (almost starvation), especially for INC `PolyReq`s, when switches hit their resource limit, and for small task groups (leading to very few machine samples). Sparrow++ mitigates this issue by using a re-check timer, which kicks in for every `PolyReq` and checks whether its number of samples is below a threshold. If so, Sparrow++ adds another round of samples. We observed stable results for a re-check timer of 200ms and a 50% threshold.

7.6.2 Methodology

We extended the cluster scheduling simulator built for IncSched (Chapter 6), now a codebase of 13K lines of Scala code, with support for the HIRE components shown in Figure 7.1, INC resources with [NOL] characteristics, and multi-path network topologies. The source code of the simulator with all schedulers is publicly available at GitHub[2]. Each experiment, characterized by ⟨plugged scheduler, target ratio μ of jobs requesting INC resources, INC heterogeneity (yes/no)⟩, runs with three seeds.
We report the following metrics:

[2] `https://github.com/mblo/hire-cluster-simulator`

Satisfied INC jobs: Ratio of `PolyReq`s with INC getting scheduled with INC (Figs. 7.7a and 7.7d). For HIRE we also report ratio of scheduled INC task groups (Figs. 7.7b and 7.7e).

Preempted tasks: Ratio of the number of tasks preempted to the total number of tasks started (Figs. 7.7c and 7.7f).

Switch detours: Number of additional levels in the switch topology required to cover all involved servers with the set of involved switches for a job (Figs. 7.8a and 7.8b).

Switch load: Amount of resources per dimension allocated among all switches, measured in a time interval for the whole simulation time (Figs. 7.9a and 7.9b).

Placement latency: Time between a task group of a `PolyReq` arrives until all its tasks start processing on machines.

We replay 36 hours of a public production workload trace from a 4000 machine Alibaba cluster [Gro18], which contains jobs of two priority classes. To best fit the 4000 servers we use a Fat-Tree (see Section 2.2) topology with $k = 26$, holding 4394 servers and 845 switches. For the switches we define three resource dimensions, namely reserved recirculation capacity, stages (48), and SRAM size (22MB), in order to roughly estimate INC resource demands referring to INC processing overhead, program complexity, and storage, respectively [Jos+15].

Name	|Switches|	`PolyReq`	Requirements
SHArP [Gra+16a]	$\lceil \log \|G\| \rceil$	Tree	SHArP ASIC
IncBricks [Liu+17]	$\mathsf{max}(3, \lceil \log \|G\| \rceil)$	Single	OF + Accelerator
NetCache [Jin+17]	$\mathsf{max}(3, \lceil \log \|G\| \rceil)$	Single (ToR)	$P4_{14}$
DistCache [Liu+19]	$\mathsf{max}(3, \lceil \log \|G\| \rceil)$	see Figure 7.3c	$P4_{14}$
NetChain [Jin+18]	$\mathsf{max}(3, 3\|G\|/10^3)$	see Figure 7.3c	$P4_{14}$
Harmonia [Zhu+19]	$\lceil \|G\|/9000 \rceil$	Single	$P4_{14}$
HovercRaft [KB20]	$\lceil \|G\|/9000 \rceil$	Single	$P4_{14}$
R2P2 (JBSQ) [Kog+19]	$\lceil \|G\|/9000 \rceil$	Single	$P4_{14}$

Table 7.6: Transformation rules and resource constraints for building `PolyReq`s for INC approaches used in evaluation.

Table 7.6 shows 8 INC services we configure in the `CompStore`. This is the same set of INC services used for the evaluation of IncSched (Chapter 6), but now considering also the topology of the INC services. For each INC service, we list parameters how to build the `PolyReq` with respect to number of required switches and topology how the switches communicate. Most INC services depend on $P4_{14}$, but some require a custom ASIC and an attached accelerator. To discuss the effects of INC heterogeneity (Section 7.2.1) we run two setups, one with all switches of homogeneous capabilities (supporting all INC services) and one with randomly choosing two compatible INC services per switch.

Table 7.7 lists the resource demand for each INC service. We set resource demand ranges (with resources sharing) according to numbers reported and communicated to us by the authors. For each task group with INC, the simulator draws actual resource demand values based on these ranges using the seed of the simulation.

To achieve the target ratio μ of jobs requesting INC resources, jobs of the trace are selected randomly, and for up to 1/3rd of a selected job's task groups, any of the INC composites are applied to create a job alternative (adding entries to the `alternative` field of a request). To capture savings of required servers and reduced processing time of a job using INC, we reduce both by 10%. We chose 10% as an upper bound to keep saving effects as a non-dominant source for performance effects. However, some INC services exhibit savings like $10x$ or higher, depending on usage pattern [Kog+19; Zhu+19; Jin+18].

The schedulers use algorithms of different runtime complexities, hence they have different think times for solving the same scheduling problem. For queue-based schedulers, typical reported numbers [Sch+13; Tir+20; Cur+19] are in the range $0.4 - 7.2$ ms per allocation. For fair comparison we set each scheduler's think time to match these numbers for an idle cluster state. For HIRE and CoCo++, we set think time as a function of flow network statistics using numbers reported in [Gog+16], but we also benchmark HIRE to validate the assumption that it runs at similar speed as Firmament [Gog+16]. Section 7.6.8 sheds light on MCMF solver speed when running HIRE.

For HIRE, we set parameters of the cost model as follows: Φ_{pref} uses $500ms, 2000ms$ for lower/upper. The upper threshold also sets the timeout for preempting a flavor decision, in case of congested resources. Φ_w uses $500ms$. HIRE is set to perform up to 250 INC flavor decisions per scheduling round. HIRE and CoCo++ limit the number of requesting task groups in the graph to 800 at any time, by using a backlog of "postponed" task groups using FIFO with the submission time. This helps to prevent situations where the MCMF solver runs too long (see Figure 7.12). HIRE and CoCo++ add up to 50 shortcut edges per task group in the graph.

7.6.3 Satisfying INC Requests (RQ1)

The primary goal of HIRE is to serve INC requests. We report the ratio of satisfied INC jobs and run experiments where we increase the overall ratio of jobs with INC demands in Figure 7.7a. We find HIRE serves more than 92% of all jobs when demand is highest, about 30% more than the best baseline (K8++ Concurrent) 69%. For cases with fewest INC demands (only 5% of all jobs ask for INC resources), the improvement is above 8% for all baselines. To further analyze HIRE's performance, we let it run with a simplified flavor logic – decide only once for each job whether to serve the whole `PolyReq` with INC or without. Even with this simplified logic HIRE achieves better results than all baselines, falling below 11% behind normal HIRE. Yarn++ with Timeout mode shows

Name	Res. recirc. cap.	Stages	SRAM (MB)
SHArP [Gra+16a]	/	/	0 | [1, 8]MB
IncBricks [Liu+17]	0 | [0, 40]%	0 | [4, 8]	0 | [3, 12]MB
NetCache [Jin+17]	0 | [0, 10]%	8 | [0, 8]	0 | [6, 12]MB
DistCache [Liu+19]	0 | [0, 10]%	8 | [0, 8]	0 | [6, 12]MB
NetChain [Jin+18]	0 | [0, 10]%	8 | [0, 8]	0 | [6, 12]MB
Harmonia [Zhu+19]	0 | 0	3 | [0, 3]	0 | [768, 2048]KB
HovercRaft [KB20]	0 | [0, 10]	18 | [0, 18]	0 | [0, 128]KB
R2P2 (JBSQ) [Kog+19]	0 | [0, 30]%	0 | [0, $\|G\|$]	0 | [1, 64]KB

Table 7.7: INC approaches used in evaluation. Each column gives resource demand per switch (before |), and per INC service instance (after |).

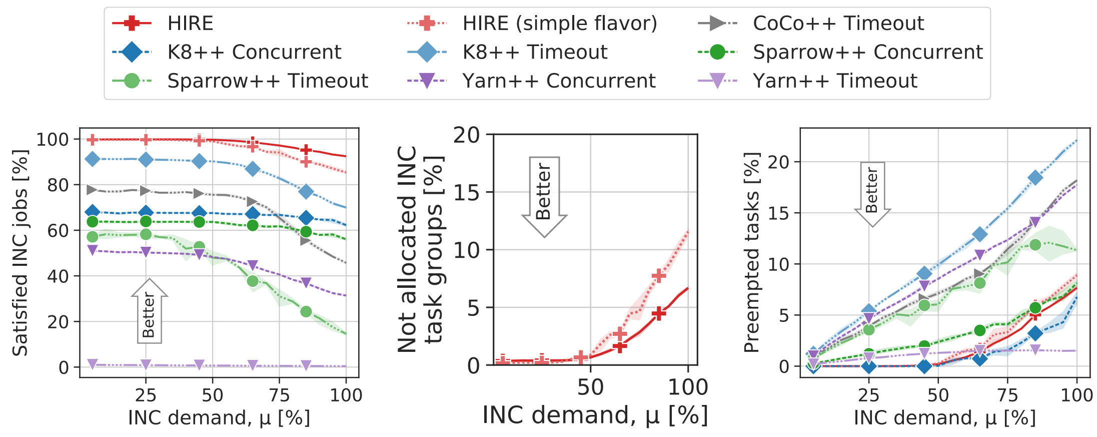
the worst success in satisfying INC resources. We investigate this performance drop in Section 7.6.7.

Figure 7.7b shows for the same experiments the ratio of unserved INC task groups when running HIRE (for better scaling, we only show numbers for HIRE). This metric serves as a test to check whether HIRE achieves a high success rate in Figure 7.7a by simply rejecting the majority of each job's INC part. We note the reported numbers correspond to the success rates in Figure 7.7a, hence HIRE does not sacrifice fairness among jobs.

Figure 7.7c shows the ratio of preempted tasks to the total number of tasks started. In general, the lower the ratio of preempted tasks, the better. Nonetheless, a scheduler use task preemptions to improve on other metrics. All schedulers show a similar pattern, the ratio of preempted tasks grows with increasing INC demand. HIRE shows almost no preemption until approximately $\mu = 50\%$, and then steadily grows to 8% preemptions at $\mu = 100\%$. Except for Yarn++, all retrofitted schedulers with Concurrent show lower preemption rations compared with their Timeout counterpart.

7.6.4 Cluster Resource Efficiency (RQ2)

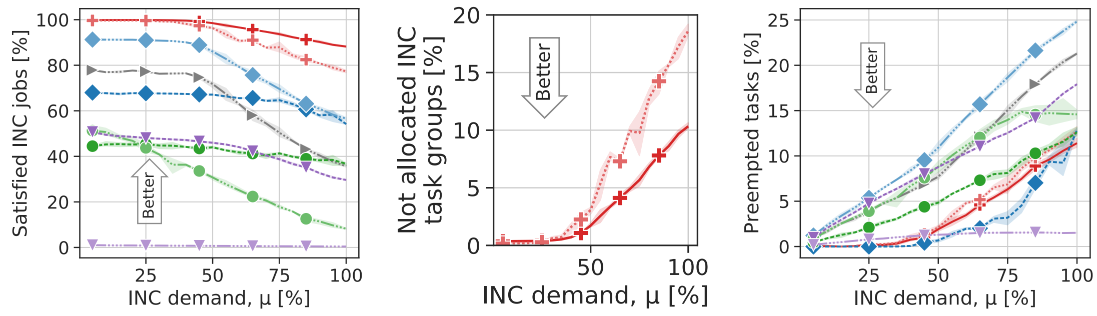
We gauge HIRE's ability to use cluster resources efficiently in two ways – by considering (*i*) the switch detour metric and (*ii*) resource load of the switches. (*i*) tests to what extent the scheduler's placement decisions affect data center fabric east-west traffic (lower is better). Figure 7.8a shows detour values for the experiments of Figure 7.7a: HIRE performs best, requiring on average less than 0.6 additional switch levels per job to cover all traffic – an improvement by at least 24% over all baselines (which serve fewer INC jobs). We also note very high values for Yarn++; this indicates a problem of rack-aware server task placement in combination with locality-unaware INC placement. The results of CoCo++ allow the assumption that the good values for HIRE can be attributed to its cost model and flow network which intertwines server and INC resources.



(a) Satisfied INC jobs; Homogeneous switches

(b) Not allocated INC task groups; Homog. switches

(c) Preempted tasks; Homogeneous switches



(d) Satisfied INC jobs; Heterogeneous switches

(e) Not allocated INC task groups; Heterog. switches

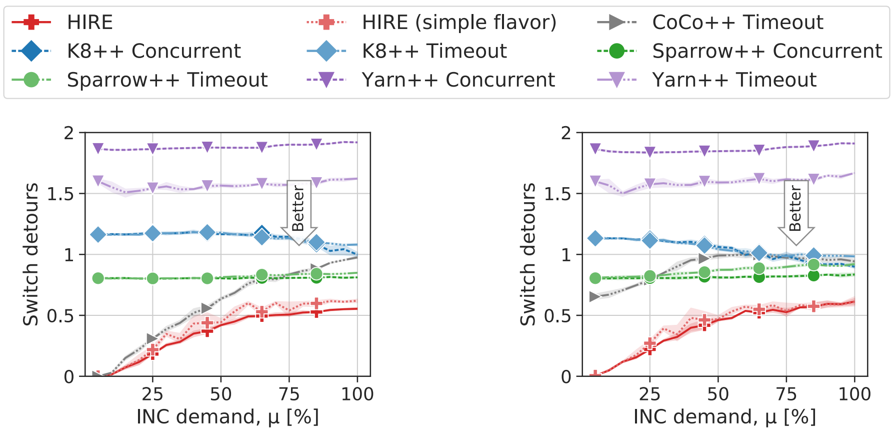
(f) Preempted tasks; Heterogeneous switches

Figure 7.7: Scheduling performance as function of μ (ratio of jobs requesting INC) for experiments with homogeneous and heterogeneous switches.

(*ii*) Switch resource load in Figure 7.9a focuses on the experiments with highest INC demand ($\mu = 1$) and reports the load of all switches over the whole simulation time. We clearly identify SRAM as the bottleneck resource dimension of the experiments. More importantly, HIRE shows lower values for usage of switch stages, all the while serving more INC tasks (and jobs). We attribute this to HIRE's ability to exploit resource sharing of co-located INC services. HIRE prefers placement decisions (server and INC) of the same sub-tree in the network topology.

7.6.5 Scheduling Under High INC Heterogeneity (RQ3)

We are particularly interested in understanding the effect of INC resource heterogeneity on scheduling performance. Thus we compare the results with two cluster setups – with



(a) Switch detours; Homogeneous switches

(b) Switch detours; Heterogeneous switches

Figure 7.8: Switch detours as a function of μ (ratio of jobs requesting INC) for experiments with homogeneous and heterogeneous switches.

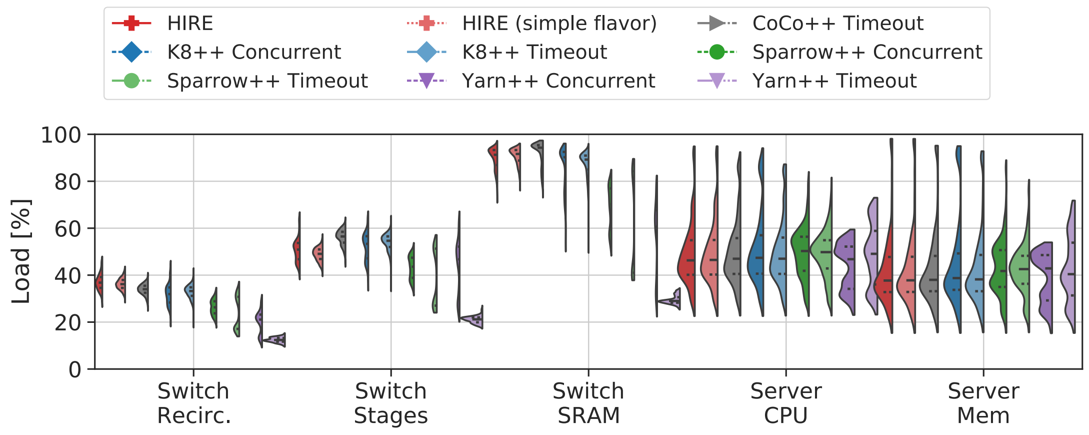
(a) homogeneous switches (Figs. 7.7a-7.10a) and (b) heterogeneous switches (Figs. 7.7d-7.10b). With (b) HIRE still achieves best results in delivering INC resources, serving 88% of all jobs with INC resources when all jobs ask for INC. The best baselines drop to 57%. Furthermore, we observe that the performance gap to HIRE grows from 11% (a) to 18% (b) when deactivating the flexible flavor logic. Figure 7.7e still validates that HIRE serves INC task groups corresponding to the success rate in Figure 7.7d. For switch detours (Figure 7.8b), we note similar trends but HIRE shows higher values for $\mu \leq 0.5$ than in (a). Switch resource load (Figure 7.9b) unveils the difficulties of resource packing, but the overall trends remain the same—HIRE needs less INC resources whilst at the same time serving more jobs with INC.

7.6.6 Preventing Resource Contention (RQ4)

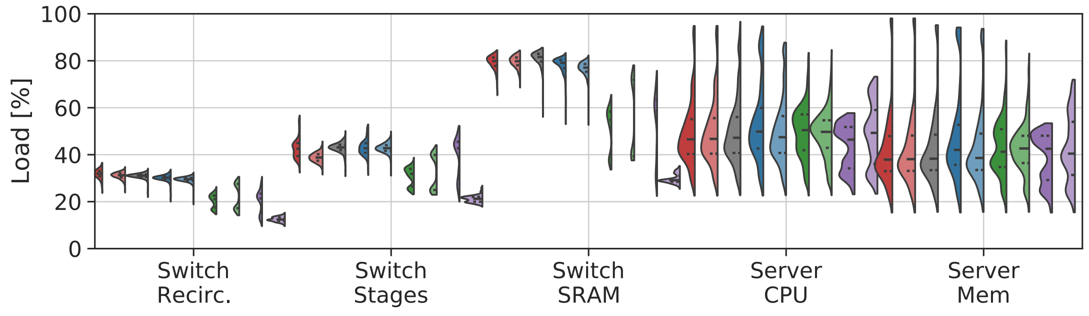
Another side-effect of resource heterogeneity is potential resource contention which may lead to long tail placement latencies. Figs. 7.10a and 7.10b show the complementary CDFs of placement latency when $\mu = 1$. HIRE shows the best tail latency, $50 - 60\%$ shorter than the best baselines in both scenarios. While making more efficient use of INC resources, HIRE schedules 90% of all allocations with latencies $< 1s$.

7.6.7 Yarn++ Parameter Stability

The experiments of IncSched with Yarn (see Section 6.4.3) showed Yarn's sensitivity to the Timeout configuration. To further investigate the performance of Yarn++, we run Yarn++ with different Timeout configurations. Figure 7.11 shows the result of Yarn++ with three additional configurations: Yarn++ with a static timeout of 60 seconds, 10



(a) Resource usage; Homogeneous switches



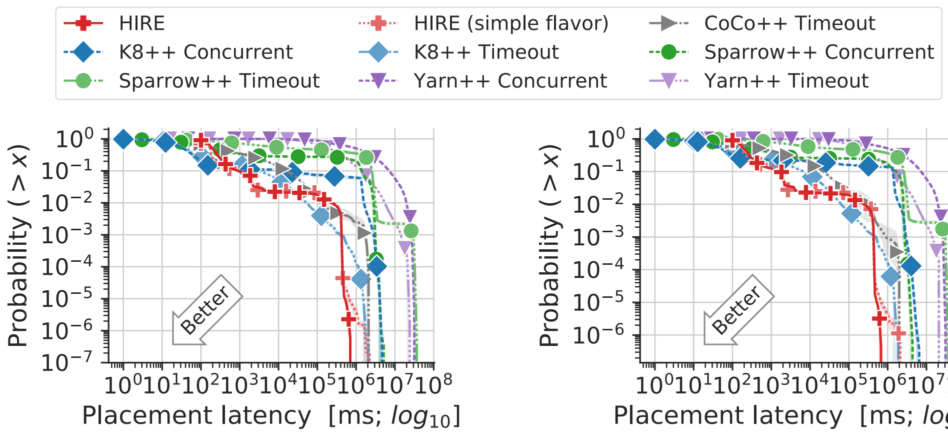
(b) Resource usage; Heterogeneous switches

Figure 7.9: Cluster resource efficiency as a function of μ (ratio of jobs requesting INC) for experiments with homogeneous and heterogeneous switches.

minutes, and 15 minutes. Regardless of the job durations, the new Yarn++ configurations always use the statically configured timeout to trigger a server fallback. Figure 7.11a shows the highest success in delivering INC for Yarn++ with $10m$ and $15m$. With a static timeout of $60s$, Yarn++ shows a slightly higher success than the normal Timeout mode (which uses 10% of a job's duration). The placement latency in Figure 7.11c shows a different pattern. Yarn++ with a static timeout of $60s$ shows higher latencies than static timeouts of $10m$ and $15m$ and the normal Timeout mode.

7.6.8 MCMF Solver Speed

As described in Section 7.6.2, our evaluation runs in an discrete-event simulator written in Scala. This requires to set scheduler think time as a function of the problem input size, in order to perform fair comparison of different schedulers. To validate the parameters used for HIRE, we run a performance benchmark on a server with an AMD EPYC 7542. We are interested in the effects on runtime performance of the MCMF solver, when INC



(a) Placement latency; Homogeneous switches

(b) Placement latency; Heterogeneous switches

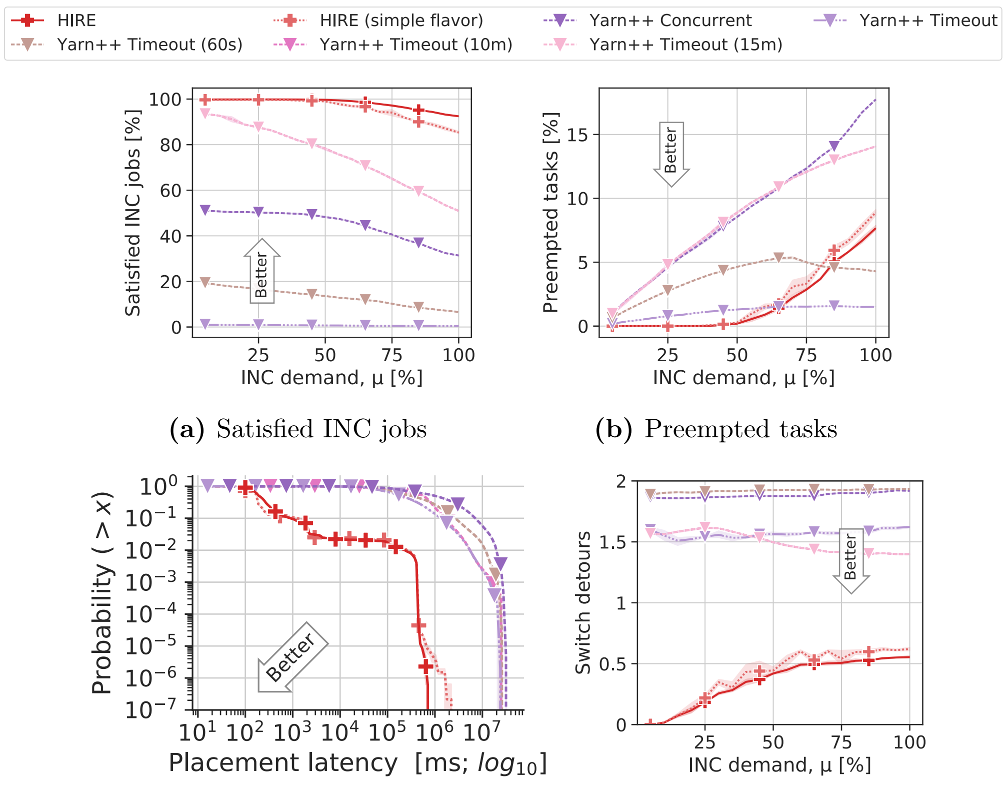
Figure 7.10: Placement latency as a complementary CDF when $\mu = 1$, *i.e.* when all jobs ask for INC resources for experiments with homogeneous and heterogeneous switches.

enters the picture. Figure 7.12 shows MCMF solver speed, when HIRE runs at different levels μ, from server-only workload ($\mu = 0$) to all jobs with INC ($\mu = 1$). For all other parameters, this benchmark uses the same configuration of Figure 7.7a.

Figure 7.12a shows a median MCMF solver speed of approximately 200ms for a workload with only server jobs, and a median of approximately 60ms for $\mu \geq 0.5$. Figure 7.12b shows tail latency of MCMF solver speeds. The MCMF solver speed is positively affected by increased INC demand, potentially due to the smaller number of switches vs. number of servers in the flow network. Another reason could be the flavor selection of HIRE, which causes to split a job request into different scheduling rounds, depending on the number of different flavors of a job. However, we note no significant difference in tail latencies (in the range of 2500ms to 3500ms).

7.7 Conclusions

This chapter introduced a new data center resource scheduler, called HIRE. HIRE provides a resource management solution for data center INC by introducing (a) a resource model which captures user requests through high-level APIs, transformed automatically into logical requests with resource alternatives specified, and (b) a novel scheduler design tailored for joint scheduling of server and INC resources under resource alternatives. HIRE clearly outperforms non-trivial retrofitted variants of existing data center schedulers using our solution IncSched (Chapter 6), demonstrating the need for novel solutions in



(a) Satisfied INC jobs (b) Preempted tasks

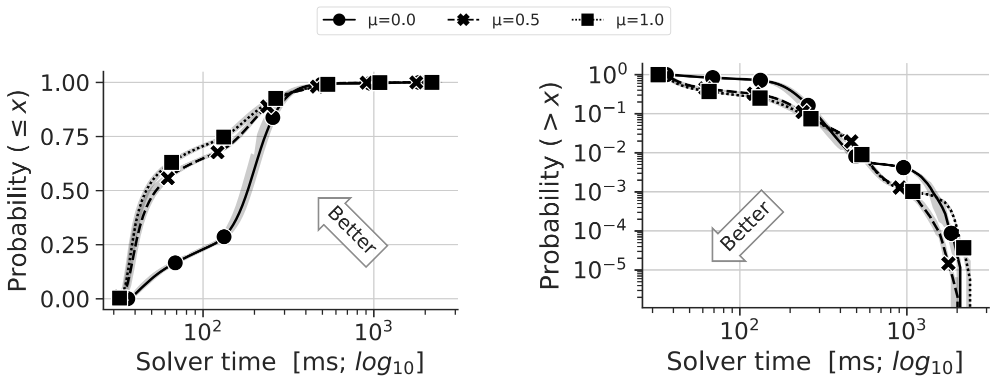
(c) Placement latency (d) Switch detours

Figure 7.11: Scheduling performance using different configurations of Yarn++ for experiments with homogeneous switches.

this space.

HIRE follows the path of flow-based schedulers [Gog+16; Isa+09], but introduces a flow network that considers server and INC resources at the same time, and integrates resource allocations with alternative selection (to decide for each job, which of the resource alternatives to choose). For this purpose, HIRE integrates a INC shadow network into the "normal" server flow network, and adds special node supplies for the resource alternative selection. This design satisfy the feasibility constraints for an MCMF solver, but enables to take decisions on multi-dimensional resource allocations (for server and INC tasks) and resource alternative selections, and provides means to encode holistic scheduling preferences *e.g.*, for cross-resource locality and priority.

We demonstrated in this chapter that data center resource scheduling with a scheduling logic that jointly considers server and INC resources is superior to existing, retrofitted schedulers using a non-trivial resource management framework solution (IncSched, Chapter 6), with respect to common scheduler performance metrics including placement



(a) CDF of MCMF solver speed **(b)** CCDF of MCMF solver speed

Figure 7.12: HIRE MCMF solver speed (CDF and CCDF) at different ratios of `PolyReq`s with INC (from no INC to all INC).

latency, but also INC specific performance metrics like switch detours and satisfied INC jobs. A key enabler for this novel solution is a resource model that builds on automatic, fine-grained resource alternative mappings, supporting [NOL] resource usages with all INC implementation and setup details stores in the scheduler.

Part IV

Epilogue

The last part of this dissertation concludes with a summary of our contributions and a discussion of potential future research directions. Lastly, we provide an outlook.

8 Conclusion

Chapter Outline

In this dissertation, we presented four holistic runtime scheduling solutions for the distributed computing landscape to improve distributed systems' performance and the efficiency of the underlying resources. In the following sections, we summarize our contributions, discuss future work, and finally provide an outlook.

8.1 Summary

We have presented four solutions in the field of holistic runtime scheduling for the distributed computing landscape. Our contributions work towards the goal of a holistically organized distributed computing landscape. The motivation of our contributions is twofold. First, the ever-increasing demand for Internet services fosters the necessity to bring runtime scheduling to the application level for every possible use case. Second, the highly heterogeneous computing infrastructure and especially the emergence of INC, pushed the demand for holistic resource scheduling, *i.e.*, resource scheduling that considers both compute resources on servers and on switches. This dissertation comprises three main parts that made the following contributions.

Part I In summary, Part I laid the foundation of all other parts with an overview of resource scheduling in the distributed computing landscape. We introduced a classification of INC that contributes to the general discussion of offloaded application logic in the distributed computing landscape. This classification sets three characteristics that classify an offloaded function as belonging to INC, namely physical, semantic, and logical characteristics.

Part II The second part discussed two application scenarios and highlighted the importance of runtime resource scheduling for the efficient usage of the distributed computing landscape. Chapter 4 discussed the scenario of data aggregation of big data applications. We proposed the runtime scheduling solution ROME, which automatically optimizes the resource usage to reduce total aggregation latency. ROME works standalone and in tandem with well-known systems Flink and Spark. We demonstrated the performance of ROME with these two systems for iterative and classic batch workloads. Especially ROME's automatic mode shows great potential to improve an application's aggregation plan at runtime. Chapter 5 discussed the second scenario and presented a runtime scheduling solution for the traffic scheduling problem of distributed service function chains, with a variant using a centralized policy (IA-MPP) and a fully distributed variant (STEAM). IA-MPP and STEAM reduce required resources and improve delivered service quality compared with state-of-the-art solutions. STEAM does not require a global view, works without traffic estimates, and operates at packet-level granularity, which are the main advantages over other schedulers for this problem.
Part II presented resource scheduling solutions on the application level for two scenarios. These solutions could be integrated with an existing or one of our proposed (Part III) data center resource scheduling solutions. The scenarios show the advantage of runtime scheduling solutions to achieve better resource efficiency and increase service quality, even though these two scenarios are not representative to serve as a generalization for all application-level scheduling solutions.

Part III The third part focused on the infrastructure level and presented two data center resource scheduling solutions for managing server and INC resources. Chapter 6 presented the resource management framework IncSched with a new resource model and INC scheduling logic. IncSched can be used jointly with existing infrastructure-level resource schedulers to make these for the first time compatible with INC resources. We evaluated IncSched in combination with three schedulers.

These cover centralized and fully distributed scheduler architectures and three scheduler designs, namely queue-based, dominant resource fairness delay scheduling, and power of two choices. The widespread applicability of IncSched demonstrates its flexibility, which is mainly driven by the fact that IncSched encapsulates the complexity of INC-specific implementation details. Lastly, Chapter 7 presented the new resource manager HIRE, which expands upon IncSched with an extended resource model and a holistic scheduling logic for joint server and INC resource scheduling. HIRE considers resource alternatives, non-linear resource usages, and INC specific side-effects and constraints within the same scheduling logic. In the evaluation, we compared HIRE with IncSched solutions and showed the advantage of HIRE's holistic scheduling logic. HIRE serves more resource requests with INC and performs scheduling decisions of better quality concerning INC resource sharing and reduced network detours of INC-server communication.
Part III presented the first data center resource scheduling solutions for shared INC on a data center level, with a solution that retrofits existing schedulers and a solution with an aligned design for better resource allocations. These solutions improve resource efficiency by serving more resource requests with INC in a shared data center setup.

To evaluate the hypothesis of this dissertation, we built four systems as summarized above. Our solutions at the application level demonstrate for two scenarios how runtime resource scheduling achieves better resource efficiency and application performance. The infrastructure-level solutions show how to democratize INC resources on a data center wide perspective and how holistic resource scheduling better accounts for resource interdependencies, thus serving more INC resource requests.

8.2 Future Work

In this section, we outline some open questions and possible future research with high relevance for which our work lays important directions.

8.2.1 Pushing INC to More Application Scenarios

The solutions on the application level focus on two application scenarios and exemplify the advantages of runtime scheduling solutions. We choose the scenarios of big data aggregation systems and traffic scheduling of distributed service function chains. The proposed systems in Chapter 4 and Chapter 5 focus on the runtime scheduling problem, using widespread available servers as a deployment target. The next logical step is to integrate these solutions with INC accelerated setups. For example, by integrating these systems with any recently proposed INC solutions for INC data aggregation [Sap+17; Gra+16a; Mai+14] and INC service function chains [Wu+19a].

Data Aggregation. Adding INC to the data aggregation scenario with ROME opens up several challenges. First, from an engineering perspective, the communication layer must be re-designed based on UDP instead of TCP. Second, INC switches have less memory than servers (switch memory is typically limited to tens of MB). Hence, additional logic is required to classify aggregation operators as potential candidates for offloading. These changes must be done for the ROME middleware system, but also for Spark and Flink, if used jointly. Lastly, from a conceptual perspective, ROME optimizes the aggregation overlay depending on the number of available aggregation nodes and the aggregation function's data stream input/output ratio. This optimization assumes that each node's aggregation time depends strictly on the data input size. However, with INC, this assumption might not hold anymore. INC switches typically operate at the network ports' line-rate, with some exceptions that require packet recirculation (which reduces the effective throughput of the switch). We see demand for further research to exploit potential optimizations of aggregation overlays with a combined usage of server and INC nodes.

Service Function Chaining. The traffic scheduling problem of distributed SFCs with STEAM shows two potential variants of how to integrate INC. First, INC could be the deployment target of the service functions. STEAM could be set up jointly with a system that offloads service functions on an INC switch, *e.g.*, Dejavu [Wu+19a]. STEAM should work out of the box with a setup using Dejavu. The main reasons for this are as follows. STEAM considers stateless functions, performs queuing at the forwarders, and considers all functions to be installed ahead of scheduling. These properties match perfectly to a setup with INC service functions. In summary, a setup with STEAM and Dejavu could provide more insights into how STEAM performs with INC-based service functions.

The second variant of how to integrate INC with STEAM is an even more challenging setup. In our evaluation, we run STEAM on a server using a DPDK implementation. Even though the performance satisfied our expectations, a promising future direction is to run the STEAM scheduling logic on a switch using INC. This setup brings the benefit of offloading a service function forwarder's scheduling logic to a switch, *i.e.*, a device whose original purpose is packet forwarding. As we have shown in Section 5.3.3, STEAM's scheduling logic requires constant processing time to the number of packet classes. Furthermore, STEAM has a small memory footprint (2 registers per service function), making STEAM likely to be a valid INC candidate. However, STEAM follows the concept of packet queuing at the service function forwarder, which might be an issue for INC—switch memory might not be sufficient for STEAM's queuing demand. STEAM's batch mode reduces the queuing demand at the service function forwarder, however, the general problem remains. One solution to this could be an INC switch that offers more buffer capacity. Alternatively, STEAM's scheduling logic could perform the majority of queuing at the servers. Related work for INC load balancing [Kog+19] shows promising solutions for efficient implementation of custom queuing policies, which could be a starting point for an INC variant of STEAM.

8.2.2 Stateful Service Function Chaining

STEAM considers stateless SFCs, *i.e.*, service functions without a state of packets belonging together. Although this is a realistic assumption in many scenarios [Kab+17; ARI+18], a logical next step is to explore how to apply STEAM for stateful SFCs. A possible extension to stateful processing with STEAM is to apply its scheduling logic not on a packet level granularity, but more coarse-grained. For example, by using a database for storing previous scheduling decisions. When packets arrive at a STEAM instance, before handing over a packet to STEAM's scheduling, the adapted STEAM solution could check the database. Upon an entry exists for the packet's flow, STEAM forwards the packet immediately according to the previous decision. Also, for each service function forwarder's queue, STEAM must continuously check the queue's head packet if an entry in the database exists. The database could also fade out entries after a particular time, similar to flowlet routing [Kan+07; Kat+16b] in data center networks. The outlined extension of STEAM enables stateful SFCs. However, it cannot guarantee strict packet order consistency within flows. For strict packet order consistency additional logic is required, *e.g.*, to check the queues for awaiting packets belonging to the same flow of an ongoing scheduling decision.

8.2.3 INC Data Center Benchmarks

Due to the lack of a multi-tenant/shared data center testbed for INC, we performed large-scale simulations to evaluate IncSched and HIRE. These benchmarks do not consider real-world performance benefits of applications when using INC. Furthermore, the benchmarks used approximate resource demands for INC solutions, based on numbers reported by the authors of such solutions. We designed the evaluation to match the reported numbers. However, simulations cannot substitute benchmarks using a real-world INC testbed or even a data center with INC. Such benchmarks could provide more insights into the performance of HIRE and IncSched. Furthermore, a real deployment could also give more insights on the advantages of HIRE's locality awareness, which aims to prevent network detours when using INC.

8.2.4 INC Switch Runtime

HIRE does rely on other works for the INC compiling and programming toolchain [Gao+19; Jos+15; Gao+20b; Son+20; Tok+19; Gao+20a] and methods for combining different INC services [Wan+20; ZBH18; HM16; Zha+19; Zha+17a] on switches. We see a demand for further research to fully implement a combined system of HIRE and an INC runtime system that supports resource sharing and (partial) runtime-reallocation. A one-size-fits-all

INC virtualization layer with an INC switch runtime system with these features is not yet available.

8.2.5 Integrating Dynamic INC Availability

Our systems IncSched and HIRE do not consider the dynamic availability of INC resources, *i.e.*, HIRE does not consider to automatically inform running applications about newly available (INC) resources. Even though HIRE considers updated resource requests, *i.e.*, users can ask for (additional) INC resources even after a job has started. For example, if an application benefits from an INC service, but at the time of submission of the resource request, no INC resource was available. The application may still benefit if it gets access to INC later while the application is still running. An extension of HIRE could provide a push-based interface, that informs running applications about previously not satisfied resource alternatives, that have now become available. HIRE could be extended with this push-based interface, without any conceptual changes to the scheduling logic. A straightforward adaption of HIRE could re-submit failed INC requests to the scheduler using the lowest priority. If any of these re-submitted requests get satisfied, the linked job can be informed.

8.2.6 Coupling INC Demand with Plan-Ahead

IncSched and HIRE consider resource requests without plan-ahead, *i.e.*, resource requests do not contain information for how long a resource is required or if and when the request will be updated later on. Even though this is a very generic and often used resource request model [Bur+16; Vav+13; JS14], it limits the possibilities the scheduler can take into account. HIRE already supports updates on resource requests. Users can add new task groups to previously submitted resource requests or update existing task groups. Furthermore, a user can set the requested task count of a task group to 0, indicating that there will be an update on the task count eventually. We refer to this as an implicit notion of upcoming resource demands. However, HIRE does not take advantage of this information (task groups with task count equals zero) for scheduling. A possible extension could consider this information for increased plan-ahead.

Another extension could require explicit information on lifetime. Similarly to (domain-specific) schedulers that consider estimated finish times of jobs [Xia+18; Mah+20; Le+20; Tum+16], a possible extension to HIRE could consider the lifetime of resource requests or parts of it. The resource model could be extended with explicit information on how long a resource is required. For example, by introducing two categories of task group lifetime, namely *job-lifetime* and *short-time*. Furthermore, a callback that returns the estimated progress of a resource request could be beneficial. These extensions to HIRE bring new opportunities for better resource packing.

8.3 Outlook

The advent of in-network computing is one aspect of the ever-growing heterogeneity of data center hardware. Domain-specific hardware like GPUs and TPUs make servers more heterogeneous but also more powerful for some applications. Many scenarios expose INC to the user (see Section 3.3), which requires resource scheduling solutions as we have presented. In other scenarios, domain-specific accelerators are used only on the infrastructure level and are not fully exposed to the applications [Fir+18]. Despite the many application scenarios that benefit from INC, there is still an open discussion of how INC should be used [McC+19b; Ben19; PN19; Alo+19; Son+20]. We are just beginning to see how INC is used and how it will influence distributed systems. We expect to see more domain-specific accelerators become available in the distributed computing landscape, at servers, in the network fabric, and at the edge. The resource heterogeneity and the demand for omnipresent Internet services will drive the need for a holistically organized distributed computing landscape that utilizes all resources most efficiently. Several factors exacerbate the demand for holistic runtime solutions, including the cost of moving data, latency-sensitive applications, and the seemingly infinite data growth. We have been working towards this goal and demonstrated the benefit of holistic runtime scheduling solutions across the stack—from application to infrastructure levels. Our solutions ROME and STEAM have exemplified the power of runtime scheduling at the application level, especially for scenarios where not all information is available ahead of runtime. The infrastructure-level solutions IncSched and HIRE have demonstrated the advantage of a holistic resource model and scheduling logic for INC-aware resource management. Hopefully, our ideas will be used for the next generation of holistic runtime resource scheduling solutions for the distributed computing landscape.

This work was funded by the German Federal Ministry of Education and Research (BMBF) Software Campus grant 01IS17050 “dynSFC” and by the German Research Foundation (DFG) as part of the project B2 in the Collaborative Research Center (CRC) 1053 “MAKI”.

Bibliography

[Abu+10] Hussam Abu-Libdeh, Paolo Costa, Antony Rowstron, Greg O'Shea, and Austin Donnelly. "Symbiotic routing in future data centers". In: *SIGCOMM*. 2010.

[Add+15] Bernardetta Addis, Dallal Belabed, Mathieu Bouet, and Stefano Secci. "Virtual network functions placement and routing optimization". In: *ACM CloudNet*. 2015, pp. 171–177.

[Ale+14] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, et al. "The Stratosphere Platform for Big Data Analytics". In: *The VLDB Journal* 23.6 (2014), pp. 939–964.

[Ali+14] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, et al. "CONGA: Distributed congestion-aware load balancing for datacenters". In: *CCR*. Vol. 44. 4. 2014, pp. 503–514.

[Ali+16] Abdul Alim, Richard G Clegg, Luo Mai, Lukas Rupprecht, Eric Seckler, Paolo Costa, Peter Pietzuch, Alexander L Wolf, Nik Sultana, Jon Crowcroft, et al. "FLICK: developing and running application-specific network services". In: *Annual Technical Conference (ATC)*. USENIX, 2016, pp. 1–14.

[Alo+19] Gustavo Alonso, Carsten Binnig, Ippokratis Pandis, Kenneth Salem, Jan Skrzypczak, Ryan Stutsman, Lasse Thostrup, Tianzheng Wang, Zeke Wang, and Tobias Ziegler. "DPI: The Data Processing Interface for Modern Networks". In: *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. 2019.

[Anw+15] Bilal Anwer, Theophilus Benson, Nick Feamster, and Dave Levin. "Programming Slick Network Functions". In: *SIGCOMM*. 2015, p. 14.

[Apa11] Apache Software Foundation. *Flink*. 2011. URL: `http://flink.apache.org`.

[Apa14] Apache Software Foundation. *Spark*. `http://spark.apache.org`. 2014.

[Apa19] Apache Software Foundation. *Apache Hadoop: Capacity Scheduler*. 2019. URL: `https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html`.

[ARI+18] ARIB, ATIS, CCSA, ETSI, TSDSI, TTA, and TTC. *Technical Realization of Service Based Architecture; Stage 3*. Technical Specification (TS) TS29.500. Version 0.4.0. 3GPP, 2018.

[Arm+10] Michael Armbrust, Armando Fox, Rean Griffith, et al. "A View of Cloud Computing". In: *Communications of the ACM (CACM)* 53.4 (Apr. 2010), pp. 50–58.

[AWE19] Alexey Andreyev, Xu Wang, and Alex Eckert. Mar. 2019. URL: `https://engineering.fb.com/2019/03/14/data-center-engineering/f16-minipack/`.

[Bal+11] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Antony I. T. Rowstron. "Towards predictable datacenter networks". In: *ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*. 2011, pp. 242–253.

[Bal+20] Hitesh Ballani, Paolo Costa, Raphael Behrendt, et al. "Sirius: A Flat Datacenter Network with Nanosecond Optical Switching". In: *Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM)*. ACM, 2020, pp. 782–797. DOI: `10.1145/3387514.3406221`.

[Bas+20] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minlan Yu, and Michael Mitzenmacher. "PINT: Probabilistic In-band Network Telemetry". In: *ACM SIGCOMM*. ACM, 2020, pp. 662–680.

[BCH13] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. "The datacenter as a computer: An introduction to the design of warehouse-scale machines". In: *Synthesis lectures on computer architecture* 8.3 (2013), pp. 1–154. DOI: `10.2200/S00516ED2V01Y201306CAC024`.

[Ben19] Theophilus A. Benson. "In-Network Compute: Considered Armed and Dangerous". In: *ACM Workshop on Hot Topics in Operating Systems (HotOS)*. 2019, pp. 216–224.

[Ber14] David Bernstein. "Containers and Cloud: From LXC to Docker to Kubernetes". In: *Cloud Computing* 1.3 (2014), pp. 81–84.

[Bha+11] Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A. Acar, and Rafael Pasquin. "Incoop: MapReduce for Incremental Computations". In: *SOCC*. 2011.

[Bha+17] Deval Bhamare, Mohammed Samaka, Aiman Erbad, Raj Jain, Lav Gupta, and H. Anthony Chan. "Optimal virtual network function placement in multi-cloud service function chaining architecture". In: *Computer Communications* 102 (2017), pp. 1–16.

[Blö+17] **Marcel Blöcher**, Malte Viering, Stefan Schmid, and Patrick Eugster. "The Grand CRU Challenge". In: *Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems (HotConNet)*. ACM, 2017, pp. 7–11. DOI: `10.1145/3094405.3094407`.

[Blö+18] **Marcel Blöcher**, Tobias Ziegler, Carsten Binnig, and Patrick Eugster. "Boosting Scalable Data Analytics with Modern Programmable Networks". In: *Proceedings of the 14th International Workshop on Data Management on New Hardware (DAMON)*. ACM, 2018, pp. 1–3. DOI: `10.1145/3211922.3211923`.

[Blö+19] **Marcel Blöcher**, Matthias Eichholz, Pascal Weisenburger, Patrick Eugster, Mira Mezini, and Guido Salvaneschi. "GRASS: Generic Reactive Application-Specific Scheduling". In: *Proceedings of the 6th SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS)*. ACM, Oct. 2019, pp. 21–30. DOI: `10.1145/3358503.3361274`.

[Blö+20a] **Marcel Blöcher**, Emilio Coppa, Pascal Kleber, Patrick Eugster, William Culhane, and Masoud Ardekani Saeida. "ROME: All Overlays Lead to Aggregation, but Some Are Faster than Others". Submitted for publication. Mar. 2020.

[Blö+20b] **Marcel Blöcher**, Ramin Khalili, Lin Wang, and Patrick Eugster. "Letting off STEAM: Distributed Runtime Traffic Scheduling for Service Function Chaining". In: *Proceedings of the 39th Conference on Computer Communications (INFOCOM)*. IEEE, Aug. 2020, pp. 824–833. DOI: `10.1109/INFOCOM41043.2020.9155404`.

[Blö+21] **Marcel Blöcher**, Lin Wang, Patrick Eugster, and Max Schmidt. "Switches for HIRE: Resource Scheduling for Data Center In-Network Computing". In: *Proceedings of the 26th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2021, pp. 268–285. DOI: `10.1145/3445814.3446760`.

[Bos+13] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN". In: *ACM SIGCOMM Computer Communication Review (CCR)*. Vol. 43. 4. 2013, pp. 99–110.

[Bos+14] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. "P4: Programming protocol-independent packet processors". In: *ACM SIGCOMM Computer Communication Review (CCR)* 44.3 (2014), pp. 87–95.

[Bou+14] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. "Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing". In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2014, pp. 285–300.

[BSM15] Tom Barbette, Cyril Soldani, and Laurent Mathy. "Fast userspace packet processing". In: *IEEE/ACM ANCS*. 2015, pp. 5–16.

[Bur+16] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. "Borg, Omega, and Kubernetes: Lessons Learned from Three Container-Management Systems over a Decade". In: *Queue* 14.1 (Jan. 2016), pp. 70–93. ISSN: 1542-7730. DOI: 10.1145/2898442.2898444.

[BW03] Maury Bramson and R. J. Williams. "Two Workload Properties for Brownian Networks". In: *Queueing Systems* 45.3 (2003), pp. 191–221.

[BZ97] Jon C. R. Bennett and Hui Zhang. "Hierarchical Packet Fair Queueing Algorithms". In: *TON* 5.5 (1997), pp. 675–689.

[CB02] Brian Carpenter and Scott Brim. *Middleboxes: Taxonomy and issues.* RFC 3234. 2002.

[Cha+10] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. "Introducing OpenSHMEM: SHMEM for the PGAS community". In: *ACM Conference on Partitioned Global Address Space Programming Model.* 2010, p. 2.

[Cha+20] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. "Balancing efficiency and fairness in heterogeneous GPU clusters for deep learning". In: *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys)*. ACM, 2020, pp. 1–16.

[Che+11] Yanpei Chen, A. Ganapathi, R. Griffith, and R. Katz. "The Case for Evaluating MapReduce Performance Using Workload Suites". In: *MASCOTS*. 2011.

[Cho+16] Mosharaf Chowdhury, Zhenhua Liu, Ali Ghodsi, and Ion Stoica. "HUG: Multi-Resource Fairness for Correlated and Elastic Demands". In: *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2016, pp. 407–424.

[Chu+17] Pavel Chuprikov, Alex Davydow, Kirill Kogan, Sergey I. Nikolenko, and Alexander V. Sirotkin. “Planning in compute-aggregate problems as optimization problems on graphs”. In: *ICNP*. 2017, pp. 1–2.

[Chu+18] Pavel Chuprikov, Alex Davydow, Kirill Kogan, Sergey I. Nikolenko, and Alexander Sirotkin. “Formalizing Compute-Aggregate Problems in Cloud Computing”. In: *Structural Information and Communication Complexity - 25th International Colloquium, SIROCCO 2018, Revised Selected Papers*. 2018, pp. 377–391.

[Cis] Cisco. *Best Practices in Core Network Capacity Planning*. Tech. rep.

[Clo53] Charles Clos. “A study of non-blocking switching networks”. In: *The Bell System Technical Journal* 32.2 (1953), pp. 406–424. DOI: 10.1002/j.1538-7305.1953.tb01433.x.

[CNC19] Cloud Native Computing Foundation CNCF. *containerd – An industry-standard container runtime with an emphasis on simplicity, robustness and portability*. Feb. 2019. URL: https://containerd.io.

[Cog] Cogent Communications. *Cogent Network Map*. URL: http://cogentco.com/en/network/network-map.

[Coh+15] Rami Cohen, Liane Lewin-Eytan, Joseph Seffi Naor, and Danny Raz. “Near optimal placement of virtual network functions”. In: *INFOCOM*. 2015, pp. 1346–1354.

[Cos+12] Paolo Costa, Austin Donnelly, Antony Rowstron, and Greg O’Shea. “Camdoop: exploiting in-network aggregation for big data applications”. In: *USENIX Conference on Networked Systems Design and Implementation (NSDI)*. 2012, pp. 3–3.

[CR90] Y.C. Cheng and T.G. Robertazzi. “Distributed computation for a tree network with communication delays”. In: *Aerospace and Electronic Systems, IEEE Transactions on* 26.3 (1990), pp. 511–516. ISSN: 0018-9251. DOI: 10.1109/7.106129.

[CT00] Jae-Hwan Chang and L. Tassiulas. “Energy conserving routing in wireless ad-hoc networks”. In: *INFOCOM*. 2000. DOI: 10.1109/INFCOM.2000.832170.

[Cul+14] William Culhane, Kirill Kogan, Chamikara Jayalath, and Patrick Eugster. “LOOM: Optimal Aggregation Overlays for In-Memory Big Data Processing”. In: *6th Workshop on Hot Topics in Cloud Computing, (HotCloud)*. USENIX, 2014. URL: https://www.usenix.org/conference/hotcloud14/workshop-program/presentation/culhane.

[Cul+15] William Culhane, Kirill Kogan, Chamikara Jayalath, and Patrick Eugster. “Optimal communication structures for big data aggregation”. In: *Conference on Computer Communications, INFOCOM*. IEEE, 2015, pp. 1643–1651. DOI: 10.1109/INFOCOM.2015.7218544.

[Cul15] William Culhane. “Optimal "Big Data" Aggregation Systems – From Theory to Practical Application”. https://docs.lib.purdue.edu/cgi/viewcontent.cgi?article=1218&context=open_access_dissertations. PhD thesis. Purdue University, USA, 2015.

[Cur+14] Carlo Curino, Djellel Eddine Difallah, Chris Douglas, Subru Krishnan, Raghu Ramakrishnan, and Sriram Rao. “Reservation-based Scheduling: If You’re Late Don’t Blame Us!” In: *ACM Symposium on Cloud Computing (SoCC)*. 2014, pp. 1–14.

[Cur+19] Carlo Curino, Subru Krishnan, Konstantinos Karanasos, et al. “Hydra: a federated resource manager for data-center scale analytics”. In: *Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX, 2019, pp. 177–192.

[Dan+15] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. “NetPaxos: Consensus at Network Speed”. In: *Symposium on Software Defined Networking Research (SOSR)*. ACM, 2015.

[Dan+16] Huynh Tu Dang, Marco Canini, Fernando Pedone, and Robert Soulé. “Paxos made switch-y”. In: *ACM SIGCOMM Computer Communication Review (CCR)* 46.1 (2016), pp. 18–24.

[Del+15] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. “Hawk: Hybrid Datacenter Scheduling”. In: *USENIX Annual Technical Conference (ATC)*. 2015, pp. 499–510.

[Del+18] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. “Kairos: Preemptive Data Center Scheduling Without Runtime Estimates”. In: *ACM Symposium on Cloud Computing (SoCC)*. 2018, pp. 135–148.

[DG08] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: simplified data processing on large clusters”. In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. URL: http://doi.acm.org/10.1145/1327452.1327492.

[DL05] J. G. Dai and Wuqin Lin. “Maximum Pressure Policies in Stochastic Processing Networks”. In: *Operations Research* 53.2 (2005), pp. 197–218.

[DL08] J. G. Dai and Wuqin Lin. “Asymptotic optimality of maximum pressure policies in stochastic processing networks”. In: *The Annals of Applied Probability* 18.6 (2008), pp. 2239–2299.

[Duf+99] Nick G. Duffield, Pawan Goyal, Albert G. Greenberg, Partho Pratim Mishra, K. K. Ramakrishnan, and Jacobus E. van der Merwe. "A Flexible Model for Resource Management in Virtual Private Networks". In: *Proceedings of the 1999 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM).* ACM, 1999, pp. 95–108. DOI: `10.1145/316188.316209`.

[Era+17] Vincenzo Eramo, Emanuele Miucci, Mostafa Ammar, and Francesco Giacinto Lavacca. "An Approach for Service Function Chain Routing and Virtual Function Network Instance Migration in Network Function Virtualization Architectures". In: *TON* 25.4 (2017), pp. 2008–2025.

[FA13] Nathan Farrington and Alexey Andreyev. "Facebook's data center network architecture". In: *Optical Interconnects Conference.* IEEE, 2013, pp. 49–50.

[Fac18] Facebook. June 2018. URL: `https://engineering.fb.com/data-center-engineering/twine/`.

[Fai19] Daniel Failing. "Entwicklung einer Testumgebung in CloudLab für dynamisches Scheduling von SFC". Bachelor-Thesis. TU Darmstadt, Germany, 2019.

[Fei+18] Xincai Fei, Fangming Liu, Hong Xu, and Hai Jin. "Adaptive VNF Scaling and Flow Routing with Proactive Demand Prediction". In: *INFOCOM.* 2018.

[Fir+18] Daniel Firestone, Andrew Putnam, Sambrama Mundkur, et al. "Azure accelerated networking: SmartNICs in the public cloud". In: *NSDI.* 2018, pp. 51–66.

[Fir17] Daniel Firestone. "VFP: A Virtual Switch Platform for Host SDN in the Public Cloud". In: *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017.* USENIX Association, 2017, pp. 315–328.

[FM93] Wolfgang Fischer and Kathleen S. Meier-Hellstern. "The Markov-Modulated Poisson Process Cookbook". In: *Perform. Eval.* 18.2 (1993), pp. 149–171.

[Fue+18] Carlo Fuerst, Stefan Schmid, Lalith Suresh, and Paolo Costa. "Kraken: Online and Elastic Resource Reservations for Cloud Datacenters". In: *IEEE/ACM Transactions on Networking (ToN)* 26.1 (2018), pp. 422–435.

[Gao+19] Xiangyu Gao, Taegyun Kim, Aatish Kishan Varma, Anirudh Sivaraman, and Srinivas Narayana. "Autogenerating Fast Packet-Processing Code Using Program Synthesis". In: *ACM Workshop on Hot Topics in Networks (HotNets).* 2019, pp. 150–160.

[Gao+20a] Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. "Lyra: A Cross-Platform Language and Compiler for Data Plane Programming on Heterogeneous ASICs". In: *ACM SIGCOMM.* 2020, pp. 435–450.

[Gao+20b] Xiangyu Gao, Taegyun Kim, Michael D. Wong, Divya Raghunathan, Aatish Kishan Varma, Pravein Govindan Kannan, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. “Switch Code Generation Using Program Synthesis”. In: *ACM SIGCOMM*. 2020, pp. 44–61.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. “The Google File System”. In: *SOSP*. 2003.

[Gha+15] Milad Ghaznavi, Aimal Khan, Nashid Shahriar, Khalid Alsubhi, Reaz Ahmed, and Raouf Boutaba. “Elastic virtual network function placement”. In: *ACM CloudNet*. 2015, pp. 255–260.

[Gho+11] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. “Dominant Resource Fairness: Fair Allocation of Multiple Resource Types”. In: *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2011.

[Gho+17] Soudeh Ghorbani, Zibin Yang, P. Brighten Godfrey, Yashar Ganjali, and Amin Firoozshahian. “DRILL: Micro Load Balancing for Low-latency Data Center Networks”. In: *SIGCOMM*. 2017, pp. 225–238.

[Gog+16] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. “Firmament: Fast, Centralized Cluster Scheduling at Scale”. In: *Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, 2016, pp. 99–115.

[Gra+14] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. “Multi-resource packing for cluster schedulers”. In: *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*. ACM, 2014, pp. 455–466. doi: 10.1145/2619239.2626334.

[Gra+16a] Richard L Graham, Devendar Bureddy, Pak Lui, Hal Rosenstock, Gilad Shainer, Gil Bloch, Dror Goldenerg, Mike Dubman, Sasha Kotchubievsky, Vladimir Koushnir, et al. “Scalable hierarchical aggregation protocol (SHArP): a hardware architecture for efficient data reduction”. In: *International Workshop on Communication Optimizations in HPC (COMHPC)*. IEEE, 2016, pp. 1–10.

[Gra+16b] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. “Altruistic Scheduling in Multi-Resource Clusters”. In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2016, pp. 65–80.

[Gra+16c] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. “GRAPHENE: Packing and Dependency-Aware Scheduling for Data-Parallel Clusters”. In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2016, pp. 81–97.

[Gro18] Alibaba Group. *Alibaba Cluster Trace Program 2018*. Version 23c0b40. 2018. URL: `https://github.com/alibaba/clusterdata/tree/23c0b40`.

[Gu+19] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Harry Liu, and Chuanxiong Guo. "Tiresias: A GPU Cluster Manager for Distributed Deep Learning". In: *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2019, pp. 485–500.

[Guo+10] Chuanxiong Guo, Guohan Lu, Helen J. Wang, Shuang Yang, Chao Kong, Peng Sun, Wenfei Wu, and Yongguang Zhang. "SecondNet: a data center network virtualization architecture with bandwidth guarantees". In: *ACM Conference on Emerging Networking Experiments and Technology (CoNEXT)*. 2010, p. 15.

[Guo+15] Chuanxiong Guo, Lihua Yuan, Dong Xiang, et al. "Pingmesh: A large-scale system for data center network latency measurement and analysis". In: *CCR*. Vol. 45. 4. 2015, pp. 139–152.

[Han+17] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wójcik. "Re-architecting datacenter networks and stacks for low latency and high performance". In: *SIGCOMM*. 2017, pp. 29–42.

[Har00] J. Michael Harrison. "Brownian models of open processing networks: canonical representation of workload". In: *The Annals of Applied Probability* 10.1 (2000), pp. 75–103.

[Heu+18a] Jens Heuschkel, Rick Vogel, **Marcel Blöcher**, and Max Mühlhäuser. "Blow up the CPU Chains! OpenCL-assisted Network Protocols". In: *Proceedings of the 43rd Conference on Local Computer Networks (LCN)*. IEEE, 2018, pp. 657–665. DOI: `10.1109/LCN.2018.8638096`.

[Heu+18b] Jens Heuschkel, Lin Wang, Erik Fleckstein, Michael Ofenloch, **Marcel Blöcher**, Jon Crowcroft, and Max Mühlhäuser. "VirtualStack: Flexible Cross-layer Optimization via Network Protocol Virtualization". In: *Proceedings of the 43rd Conference on Local Computer Networks (LCN)*. IEEE, 2018, pp. 519–526. DOI: `10.1109/LCN.2018.8638106`.

[Hin+11] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy H. Katz, Scott Shenker, and Ion Stoica. "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center". In: *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2011.

[HM16] David Hancock and Jacobus E. van der Merwe. “HyPer4: Using P4 to Virtualize the Programmable Data Plane”. In: *International on Conference on emerging Networking EXperiments and Technologies (CoNEXT)*. ACM, 2016, pp. 35–49.

[Hop00] Christian E. Hopps. “Analysis of an Equal-Cost Multi-Path Algorithm”. In: *RFC* 2992 (2000), pp. 1–8.

[HP15] Joel M. Halpern and Carlos Pignataro. *Service Function Chaining (SFC) Architecture*. RFC 7665. 2015.

[HP19] John L. Hennessy and David A. Patterson. “A New Golden Age for Computer Architecture”. In: *Communications of the ACM (CACM)* 62.2 (Jan. 2019), pp. 48–60.

[Hue+03] Ryan Huebsch, Joseph M. Hellerstein, Nick Lanham, Boon Thau Loo, Scott Shenker, and Ion Stoica. “Querying the Internet with PIER”. In: *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*. VLDB ’03. VLDB Endowment, 2003, pp. 321–332. isbn: 0-12-722442-4. url: `http://dl.acm.org/citation.cfm?id=1315451.1315480`.

[Hun+10] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. “ZooKeeper: Wait-free Coordination for Internet-scale Systems”. In: *ATC*. 2010.

[IBS08] Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman. “A survey of top-k query processing techniques in relational database systems”. In: *ACM Computing Surveys* 40.4 (2008).

[Int13] Intel. *Intel Ethernet Switch FM6000 Series, white paper*. 2013.

[Int18] Intel. *Intel Tofino 2*. Previously Barefoot Networks Tofino 2, acquired by Intel in 2019. 2018. url: `https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html`.

[Isa+09] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew V. Goldberg. “Quincy: fair scheduling for distributed computing clusters”. In: *Symposium on Operating Systems Principles (SOSP)*. ACM, 2009, pp. 261–276.

[Ist+16] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. “Consensus in a Box: Inexpensive Coordination in Hardware”. In: *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2016, pp. 425–438.

[Jai+07] Navendu Jain, Dmitry Kit, Prince Mahajan, Praveen Yalagandula, Mike Dahlin, and Yin Zhang. “STAR: Self-tuning Aggregation for Scalable Monitoring”. In: *VLDB*. 2007.

[Jeo+19] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. "Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads". In: *USENIX Annual Technical Conference (ATC)*. 2019, pp. 947–960.

[Jep+18] Theo Jepsen, Masoud Moshref, Antonio Carzaniga, Nate Foster, and Robert Soulé. "Life in the fast lane: A line-rate linear road". In: *ACM Symposium on SDN Research (SOSR)*. 2018, p. 10.

[Jin+17] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. "NetCache: Balancing Key-Value Stores with Fast In-Network Caching". In: *Symposium on Operating Systems Principles (SOSP)*. ACM, 2017, pp. 121–136.

[Jin+18] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. "NetChain: Scale-Free Sub-RTT Coordination". In: *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2018, pp. 35–49.

[Jos+15] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. "Compiling packet programs to reconfigurable switches". In: *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2015, pp. 103–115.

[JS14] Brendan Jennings and Rolf Stadler. "Resource management in clouds: Survey and research challenges". In: *Journal of Network and Systems Management* (2014), pp. 1–53.

[Jua+16] Yuchin Juan, Yong Zhuang, Wei-Sheng Chin, and Chih-Jen Lin. "Field-aware factorization machines for CTR prediction". In: *Proceedings of the 10th ACM Conference on Recommender Systems*. ACM. 2016, pp. 43–50.

[Jyo+16] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, et al. "Morpheus: Towards Automated SLOs for Enterprise Clusters". In: *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI*. 2016, pp. 117–134.

[Kab+17] Murad Kablan, Azzam Alsudais, Eric Keller, and Franck Le. "Stateless network functions: Breaking the tight coupling of state and processing". In: *NSDI*. 2017, pp. 97–112.

[Kan+07] Srikanth Kandula, Dina Katabi, Shantanu Sinha, and Arthur Berger. "Dynamic load balancing without packet reordering". In: *ACM SIGCOMM Computer Communication Review* 37.2 (2007), pp. 51–62.

[Kat+16a] Georgios P Katsikas, Marcel Enguehard, Maciej Kuźniar, Gerald Q Maguire Jr, and Dejan Kostić. "SNF: Synthesizing high performance NFV service chains". In: *PeerJ Computer Science* 2 (2016), e98.

[Kat+16b] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. “HULA: Scalable Load Balancing Using Programmable Data Planes”. In: *Proc. ACM Symposium on SDN Research*. 2016.

[Kat+18] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Rebecca Steinert, and Gerald Q. Maguire Jr. “Metron: NFV Service Chains at the True Speed of the Underlying Hardware”. In: *NSDI*. 2018, pp. 171–186.

[Kau+16] Antoine Kaufmann, Simon Peter, Naveen Kr. Sharma, Thomas E. Anderson, and Arvind Krishnamurthy. “High Performance Packet Processing with FlexNIC”. In: *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2016, pp. 67–81.

[KB20] Marios Kogias and Edouard Bugnion. “HovercRaft: Achieving Scalability and Fault-Tolerance for Microsecond-Scale Datacenter Services”. In: *ACM European Conference on Computer Systems (EuroSys)*. 2020, pp. 1–17.

[Ke+15] Huan Ke, Peng Li, Song Guo, and I. Stojmenovic. “Aggregation on the fly: reducing traffic for big data in the cloud”. In: *Network, IEEE* 29.5 (Sept. 2015), pp. 17–23. issn: 0890-8044. doi: `10.1109/MNET.2015.7293300`.

[Kim+16] Changhoon Kim, Parag Bhide, E Doe, H Holbrook, A Ghanwani, D Daly, M Hira, and B Davie. *In-band Network Telemetry (INT) Dataplane Specification*. `https://p4.org/assets/INT-current-spec.pdf`. 2016.

[KJL96] Hyoung-Joong Kim, Gyu-In Jee, and Jang-Gyu Lee. “Optimal load distribution for tree network processors”. In: *Aerospace and Electronic Systems, IEEE Transactions on* 32.2 (1996), pp. 607–612. issn: 0018-9251. doi: `10.1109/7.489505`.

[Kle17] Pascal Kleber. “Fault Tolerance in Optimal Aggregation Overlays for Big Data Applications”. Master-Thesis. TU Darmstadt, Germany, 2017.

[Kog+19] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. “R2P2: Making RPCs first-class datacenter citizens”. In: *Annual Technical Conference (ATC)*. USENIX, 2019, pp. 863–880.

[Kul+15] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. “Twitter Heron: Stream Processing at Scale”. In: *SIGMOD*. 2015.

[Kul+17] Sameer G. Kulkarni, Wei Zhang, Jinho Hwang, Shriram Rajagopalan, K. K. Ramakrishnan, Timothy Wood, Mayutan Arumaithurai, and Xiaoming Fu. “NFVnice: Dynamic Backpressure and Scheduling for NFV Service Chains”. In: *SIGCOMM*. 2017, pp. 71–84.

[Kum+16] Gautam Kumar, Ganesh Ananthanarayanan, Sylvia Ratnasamy, and Ion Stoica. “Hold ’em or Fold ’Em? Aggregation Queries under Performance Variations”. In: *EuroSys*. Association for Computing Machinery, 2016. ISBN: 9781450342407. DOI: `10.1145/2901318.2901351`. URL: `https://doi.org/10.1145/2901318.2901351`.

[Kuo+16] Tung-Wei Kuo, Bang-Heng Liou, Kate Ching-Ju Lin, and Ming-Jer Tsai. “Deploying chains of virtual network functions: On the relation between link and server usage”. In: *INFOCOM*. 2016, pp. 1–9.

[Le+20] Tan N. Le, Xiao Sun, Mosharaf Chowdhury, and Zhenhua Liu. “AlloX: Compute Allocation in Hybrid Clusters”. In: *ACM European Conference on Computer Systems (EuroSys)*. 2020.

[LeB10] Jean-Yves LeBoudec. *Performance Evaluation of Computer and Communication Systems*. EPFL Press, Lausanne, Switzerland, 2010.

[Lee+13] Jeongkeun Lee, Myungjin Lee, Lucian Popa, Yoshio Turner, Sujata Banerjee, Puneet Sharma, and Bryan Stephenson. “CloudMirror: Application-Aware Bandwidth Reservations in the Cloud”. In: *5th Workshop on Hot Topics in Cloud Computing (HotCloud)*. USENIX Association, 2013.

[Lei85] Charles E. Leiserson. “Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing”. In: *Transactions on Computers* 34.10 (1985), pp. 892–901. DOI: `10.1109/TC.1985.6312192`.

[Ler+19] Alberto Lerner, Rana Hussein, Philippe Cudre-Mauroux, and U eXascale Infolab. “The Case for Network-Accelerated Query Processing”. In: *Biennial Conference on Innovative Data Systems Research (CIDR)*. 2019, pp. 13–16.

[LHM11] Y. Liu, Z. Hu, and K. Matsuzaki. “Towards Systematic Parallel Programming over MapReduce”. In: *Euro-Par*. 2011.

[Li+14] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. “Scaling Distributed Machine Learning with the Parameter Server”. In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2014, pp. 583–598.

[Li+16a] Jialin Li, Ellis Michael, Naveen Kr Sharma, Adriana Szekeres, and Dan RK Ports. “Just say NO to paxos overhead: Replacing consensus with network ordering”. In: *USENIX Symposium on Operating Systems Design and Implementation (NSDI)*. 2016, pp. 467–483.

[Li+16b] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G Andersen, and Michael J Freedman. “Be fast, cheap and in control with SwitchKV”. In: *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2016, pp. 31–44.

[Li+19] Yuliang Li, Rui Miao, Hongqiang Harry Liu, et al. “HPCC: high precision congestion control”. In: *ACM SIGCOMM.* ACM, 2019, pp. 44–58.

[Lia+19] Richard Liaw, Romil Bhardwaj, Lisa Dunlap, Yitian Zou, Alexey Tumanov, Joseph Gonzalez, and Ion Stoica. “HyperSched: Dynamic Resource Reallocation for Model Development on a Deadline”. In: *ACM Symposium on Cloud Computing (SoCC).* 2019.

[Liu+17] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. “IncBricks: Toward In-Network Computation with an In-Network Cache”. In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).* ACM, 2017, pp. 795–809.

[Liu+19] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. “DistCache: Provable Load Balancing for Large-Scale Storage Systems with Distributed Caching”. In: *USENIX Conference on File and Storage Technologies (FAST).* 2019, pp. 143–157.

[LMP17] Jialin Li, Ellis Michael, and Dan RK Ports. “Eris: Coordination-free consistent transactions using in-network concurrency control”. In: *ACM Symposium on Operating Systems Principles (SOSP).* 2017, pp. 104–120.

[LRS18] Marcelo Caggiani Luizelli, Danny Raz, and Yaniv Sa’ar. “Optimizing NFV Chain Deployment Through Minimizing the Cost of Virtual Switching”. In: *INFOCOM.* 2018, pp. 2150–2158.

[Mad+05] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. “TinyDB: An Acquisitional Query Processing System for Sensor Networks”. In: *ACM Trans. Database Syst.* 30.1 (Mar. 2005), pp. 122–173. issn: 0362-5915. doi: `10.1145/1061318.1061322`. url: `http://doi.acm.org/10.1145/1061318.1061322`.

[Mah+20] Kshiteej Mahajan, Arjun Singhvi, Arjun Balasubramanian, Varun Batra, Surya Teja Chavali, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. “Themis: Fair and Efficient GPU Cluster Scheduling for Machine Learning Workloads”. In: *USENIX Symposium on Network Systems Design and Implementation (NSDI).* 2020.

[Mai+14] Luo Mai, Lukas Rupprecht, Abdul Alim, Paolo Costa, Matteo Migliavacca, Peter Pietzuch, and Alexander L Wolf. “NetAgg: Using Middleboxes for Application-specific On-path Aggregation in Data Centres”. In: *ACM International on Conference on Emerging Networking Experiments and Technologies (CoNEXT).* 2014, pp. 249–262.

[Mao+19] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. “Learning scheduling algorithms for data processing clusters”. In: *ACM Special Interest Group on Data Communication (SIGCOMM)*. 2019, pp. 270–288.

[Mar+14] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. “ClickOS and the art of network function virtualization”. In: *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association. 2014, pp. 459–473.

[Mar+15] Barbara Martini, Federica Paganelli, Paola Cappanera, Stefano Turchi, and Piero Castoldi. “Latency-aware composition of virtual functions in 5G”. In: *NetSoft*. 2015, pp. 1–6.

[Mar02] Robert C Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall, 2002.

[McC+19a] James McCauley, Aurojit Panda, Arvind Krishnamurthy, and Scott Shenker. “Thoughts on load distribution and the role of programmable switches”. In: *ACM SIGCOMM Computer Communication Review (CCR)* 49.1 (2019), pp. 18–23.

[McC+19b] James McCauley, Aurojit Panda, Arvind Krishnamurthy, and Scott Shenker. “Thoughts on load distribution and the role of programmable switches”. In: *ACM SIGCOMM Computer Communication Review (CCR)* 49.1 (2019), pp. 18–23.

[Men+18] Zili Meng, Jun Bi, Chen Sun, Haiping Wang, and Hongxin Hu. “CoCo: Compact and Optimized Consolidation of Modularized Service Function Chains in NFV”. In: *ICC*. 2018.

[MGZ16] Marouen Mechtri, Chaima Ghribi, and Djamal Zeghlache. “A scalable algorithm for the placement of service function chains”. In: *TNSM* 13.3 (2016), pp. 533–546.

[Mic20] Marco Micera. “Data center resource management for in-network processing”. Master-Thesis. Polytechnic University of Turin, Italy, 2020.

[Mij+15] Rashid Mijumbi, Joan Serrat, Juan-Luis Gorricho, Niels Bouten, Filip De Turck, and Steven Davy. “Design and evaluation of algorithms for mapping and scheduling of virtual network functions”. In: *NetSoft*. 2015, pp. 1–9.

[Mit01] Michael Mitzenmacher. “The power of two choices in randomized load balancing”. In: *IEEE Transactions on Parallel and Distributed Systems* 12.10 (2001), pp. 1094–1104.

[MK16] Antonio Marotta and Andreas Kassler. “A power efficient and robust virtual network functions placement problem”. In: *ITC*. Vol. 1. 2016, pp. 331–339.

[Mor+09] A. Morihata, K. Matsuzaki, Z. Hu, and M. Takeichi. “The Third Homomorphism Theorem on Trees: Downward & Upward lead to Divide-and-Conquer”. In: *POPL*. Jan. 2009.

[Mur+13] Derek Gordon Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. “Naiad: a timely dataflow system”. In: *SIGOPS 24th Symposium on Operating Systems Principles, SOSP*. ACM, 2013, pp. 439–455. DOI: `10.1145/2517349.2522738`. URL: `https://doi.org/10.1145/2517349.2522738`.

[Mus+19] Craig Mustard, Fabian Ruffy, Anny Gakhokidze, Ivan Beschastnikh, and Alexandra Fedorova. “Jumpgate: In-Network Processing as a Service for Data Analytics”. In: *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*. 2019.

[MW13] Dmitriy Morozov and Gunther Weber. “Distributed merge trees”. In: *PPoPP*. 2013.

[Nar+20] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. “Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads”. In: *14th Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Nov. 2020, pp. 481–498.

[Nat18] National Science Foundation. *US-EU Internet Core & Edge Technologies (ICE-T)*. May 2018. URL: `https://www.nsf.gov/pubs/2018/nsf18535/nsf18535.pdf`.

[Nee09] M. J. Neely. “Delay Analysis for Maximal Scheduling With Flow Control in Wireless Networks With Bursty Traffic”. In: *TON* 17.4 (2009), pp. 1146–1159.

[Nel+16] Tim Nelson, Nicholas DeMarinis, Timothy Adam Hoff, Rodrigo Fonseca, and Shriram Krishnamurthi. “Switches are Monitors Too!: Stateful Property Monitoring as a Switch Design Criterion”. In: *HotNets*. 2016, pp. 99–105.

[Net18] Netronome. *Netronome NFP-6000 Intelligent Ethernet Controller Family*. 2018. URL: `https://www.netronome.com/m/documents/PB_NFP-6000_.pdf`.

[New89] Peter Newman. “Fast Packet Switching for Integrated Services”. PhD thesis. University of Cambridge, 1989.

[NN10] Ron Nadiv and Tzvika Naveh. “Wireless Backhaul Topologies: Analyzing Backhaul Topology Strategies”. In: *White Paper Ceragon* (2010).

[Ous+13] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. “Sparrow: distributed, low latency scheduling”. In: *ACM Symposium on Operating Systems Principles (SOSP)*. 2013, pp. 69–84.

[Pac+11] S. Pacheco-Sanchez, G. Casale, B. Scotney, S. McClean, G. Parr, and S. Dawson. "Markovian Workload Characterization for QoS Prediction in the Cloud". In: *IEEE CLOUD*. 2011.

[Pal+15] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. "E2: a framework for NFV applications". In: *SOSP*. 2015, pp. 121–136.

[Pen+18] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. "Optimus: an efficient dynamic resource scheduler for deep learning clusters". In: *ACM European Conference on Computer Systems (EuroSys)*. 2018, pp. 1–14.

[Pen+19] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. "A generic communication scheduler for distributed DNN training acceleration". In: *ACM Symposium on Operating Systems Principles (SOSP)*. 2019, pp. 16–29.

[PN19] Dan R. K. Ports and Jacob Nelson. "When Should The Network Be The Computer?" In: *Workshop on Hot Topics in Operating Systems (HotOS)*. ACM, 2019, pp. 209–215.

[Pon+19] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, et al. "Flowblaze: Stateful packet processing in hardware". In: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 2019.

[Pop+12] Lucian Popa, Gautam Kumar, Mosharaf Chowdhury, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. "FairCloud: sharing the network in cloud computing". In: *ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*. 2012, pp. 187–198.

[QAS16] Long Qu, Chadi Assi, and Khaled Shaban. "Delay-aware scheduling and resource optimization with network function virtualization". In: *TCOM* 64.9 (2016), pp. 3746–3758.

[QCB18] Chenhao Qu, Rodrigo N. Calheiros, and Rajkumar Buyya. "Auto-Scaling Web Applications in Clouds: A Taxonomy and Survey". In: *ACM Computing Surveys* 51.4 (2018), 73:1–73:33.

[QEP18] P. Quinn, U. Elzur, and C. Pignataro. *Network Service Header (NSH)*. RFC 8300. 2018.

[Rec+11] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. "Hogwild: A lock-free approach to parallelizing stochastic gradient descent". In: *Advances in neural information processing systems*. 2011, pp. 693–701.

[Ren+14] Jing Ren, Wen Qi, Cedric Westphal, Jianping Wang, Kejie Lu, Shucheng Liu, and Sheng Wang. “Magic: A distributed max-gain in-network caching strategy in information-centric networks”. In: *Computer Communications Workshops (INFOCOM WKSHPS), 2014 IEEE Conference on*. IEEE. 2014, pp. 470–475.

[Rza+20] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, et al. “Autopilot: Workload Autoscaling at Google”. In: *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys)*. ACM, 2020.

[Sae+19] Ahmed Saeed, Yimeng Zhao, Nandita Dukkipati, Ellen W. Zegura, Mostafa H. Ammar, Khaled Harras, and Amin Vahdat. “Eiffel: Efficient and Flexible Software Packet Scheduling”. In: *NSDI*. 2019, pp. 17–32.

[Sap+17] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. “In-Network Computation is a Dumb Idea Whose Time Has Come”. In: *Workshop on Hot Topics in Networks (HotNets)*. ACM, 2017, pp. 150–156.

[Sap+19] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan R. K. Ports, and Peter Richtárik. “Scaling Distributed Machine Learning with In-Network Aggregation”. In: *CoRR* abs/1903.06701 (2019).

[Sat+18] Argawal Satyam, Malandrino Francesco, Carla Fabiana Chiasserini, and De Swedes. “Joint VNF Placement and CPU Allocation in 5G”. In: *INFOCOM*. 2018.

[SB17] Giuseppe Siracusano and Roberto Bifulco. “Is it a SmartNIC or a Key-Value Store?: Both!” In: *Proceedings of the SIGCOMM Posters and Demos*. ACM. 2017, pp. 138–140.

[SB18] Malte Schwarzkopf and Peter Bailis. “Research for practice: cluster scheduling for datacenters”. In: *Communications of the ACM* 61.5 (2018), pp. 50–53.

[Sch+13] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. “Omega: flexible, scalable schedulers for large compute clusters”. In: *ACM European Conference on Computer Systems (EuroSys)*. 2013, pp. 351–364.

[Sch15] Malte Schwarzkopf. “Operating system support for warehouse-scale computing”. PhD thesis. PhD thesis. University of Cambridge Computer Laboratory, 2015.

[Shv+10] K. Shvachko, Hairong Kuang, S. Radia, and R. Chansler. “The Hadoop Distributed File System”. In: *MSST*. 2010.

[SIG] SIGKDD. *KDD CUP 2012 data set*. URL: `https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html#kdd2012`.

[Sin+15] Arjun Singh, Joon Ong, Amit Agarwal, et al. “Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google’s Datacenter Network”. In: *Proceedings of the 2015 Conference on Special Interest Group on Data Communication (SIGCOMM).* ACM, 2015, pp. 183–197. DOI: `10.1145/2785956.2787508`.

[Son+20] Hardik Soni, Myriana Rifai, Praveen Kumar, Ryan Doenges, and Nate Foster. “Composing Dataplane Programs with μP4”. In: *ACM SIGCOMM.* 2020, pp. 329–343.

[SSS11] Michael Sindelar, Ramesh K. Sitaraman, and Prashant J. Shenoy. “Sharing-aware algorithms for virtual machine colocation”. In: *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA).* 2011, pp. 367–378.

[SSZ98] Ion Stoica, Scott Shenker, and Hui Zhang. “Core-stateless Fair Queueing: Achieving Approximately Fair Bandwidth Allocations in High Speed Networks”. In: *CCR* 28.4 (1998), pp. 118–130.

[Sto+15] Radu Stoenescu, Vladimir Andrei Olteanu, Matei Popovici, et al. “In-Net: in-network processing for the masses”. In: *EuroSys.* 2015, p. 23.

[The] The Computational Biology and Functional Genomics Laboratory at DF-CI/Harward. *TGI Database: DNA sequences.* `http://compbio.dfci.harvard.edu/tgi/`.

[Tir+20] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. “Borg: The next Generation”. In: *ACM European Conference on Computer Systems (EuroSys).* 2020.

[TJP16] Fung Po Tso, Simon Jouet, and Dimitrios P. Pezaros. “Network and server resource management strategies for data centre infrastructures: A survey”. In: *Computer Networks* 106 (2016), pp. 209–225. DOI: `10.1016/j.comnet.2016.07.002`.

[TK03] Hüseyin Özgür Tan and Ibrahim Körpeoğlu. “Power Efficient Data Gathering and Aggregation in Wireless Sensor Networks”. In: *SIGMOD Rec.* 32.4 (Dec. 2003), pp. 66–71. ISSN: 0163-5808. DOI: `10.1145/959060.959072`. URL: `http://doi.acm.org/10.1145/959060.959072`.

[TMZ18] Y. Tokusashi, H. Matsutani, and N. Zilberman. “LaKe: The Power of In-Network Computing”. In: *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig).* Dec. 2018, pp. 1–8.

[TNG] Telecommunications Networks Group - Politecnico di Torino TNG. *Traces from Real Internet Traffic.* URL: `http://tstat.polito.it/traces-skype.shtml`.

[Tok+19] Yuta Tokusashi, Huynh Tu Dang, Fernando Pedone, Robert Soulé, and Noa Zilberman. “The Case For In-Network Computing On Demand”. In: *ACM European Conference on Computer Systems (EuroSys)*. 2019.

[Tum+16] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. “TetriSched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters”. In: *European Conference on Computer Systems (EuroSys)*. ACM, 2016, pp. 1–16.

[Val+12] José Valerio, Pascal Felber, Martin Rajman, and Etienne Riviere. “CADA: Collaborative Auditing for Distributed Aggregation”. In: *2012 Ninth European Dependable Computing Conference, Sibiu, Romania, May 8-11, 2012*. 2012, pp. 1–12.

[Van+02] Robbert Van Renesse, Kenneth Birman, Dan Dumitriu, and Werner Vogels. “Scalable Management and Data Mining Using Astrolabe”. In: *Peer-to-Peer Systems*. Springer, 2002, pp. 280–294.

[Vav+13] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. “Apache hadoop yarn: Yet another resource negotiator”. In: *ACM Symposium on Cloud Computing (SoCC)*. 2013, p. 5.

[Ven+13] Shivaram Venkataraman, Erik Bodzsar, Indrajit Roy, Alvin AuYoung, and Robert S Schreiber. “Presto: Distributed Machine Learning and Graph Processing with Sparse Matrices”. In: *EuroSys*. 2013.

[Ver+15] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. “Large-scale cluster management at Google with Borg”. In: *ACM European Conference on Computer Systems (EuroSys)*. 2015, pp. 1–17.

[Wan+15] Kai Wang, Minghong Lin, Florin Ciucu, Adam Wierman, and Chuang Lin. “Characterizing the Impact of the Workload on the Value of Dynamic Resizing in Data Centers”. In: *Perform. Eval.* 85 (2015), pp. 1–18.

[Wan+16] Luhan Wang, Zhaoming Lu, Xiangming Wen, Raymond Knopp, and Rohit Gupta. “Joint optimization of service function chaining and resource allocation in network function virtualization”. In: *IEEE Access* 4 (2016), pp. 8084–8094.

[Wan+19] Zhijun Wang, Huiyang Li, Zhongwei Li, Xiaocui Sun, Jia Rao, Hao Che, and Hong Jiang. “Pigeon: An Effective Distributed, Hierarchical Datacenter Job Scheduler”. In: *ACM Symposium on Cloud Computing (SoCC)*. 2019, pp. 246–258.

[Wan+20] Tao Wang, Hang Zhu, Fabian Ruffy, Xin Jin, Anirudh Sivaraman, Dan R. K. Ports, and Aurojit Panda. "Multitenancy for Fast and Programmable Networks in the Cloud". In: *Symposium on Hot Topics in Cloud Computing (HotCloud)*. USENIX, 2020.

[WD96] David W Walker and Jack J Dongarra. "MPI: A standard message passing interface". In: *Supercomputer* 12 (1996), pp. 56–68.

[Wik] Wikipedia. *Pageviews hourly statistics dumps.* `https://wikitech.wikimedia.org/wiki/Analytics/Data/Pagecounts-raw`.

[Wil16] Ruth J. Williams. "Stochastic Processing Networks". In: *Annual Review of Statistics and Its Application* 3.1 (2016), pp. 323–345.

[Wu+19a] Dingming Wu, Ang Chen, TS Eugene Ng, Guohui Wang, and Haiyong Wang. "Accelerated Service Chaining on a Single Switch ASIC". In: *ACM Workshop on Hot Topics in Networks (HotNets)*. 2019, pp. 141–149.

[Wu+19b] Heng Wu, Wenbo Zhang, Yuanjia Xu, Hao Xiang, Tao Huang, Haiyang Ding, and Zheng Zhang. "Aladdin: Optimized Maximum Flow Management for Shared Production Clusters". In: *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2019, pp. 696–707.

[Xia+18] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, et al. "Gandiva: Introspective Cluster Scheduling for Deep Learning". In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2018, pp. 595–610.

[Xia+19] Yikai Xiao, Qixia Zhang, Fangming Liu, Jia Wang, Miao Zhao, Zhongxing Zhang, and Jiaxing Zhang. "NFVdeep: adaptive online service function chain deployment with deep reinforcement learning". In: *IEEE/ACM IWQoS*. 2019, 21:1–21:10.

[Xie+12] Di Xie, Ning Ding, Y. Charlie Hu, and Ramana Rao Kompella. "The only constant is change: incorporating time-varying network reservations in data centers". In: *ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*. 2012, pp. 199–210.

[XZ19] Zhaoqi Xiong and Noa Zilberman. "Do Switches Dream of Machine Learning?: Toward In-Network Classification". In: *ACM Workshop on Hot Topics in Networks (HotNets)*. 2019, pp. 25–33.

[Yan+07] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker. "Map-reduce-merge: simplified relational data processing on large clusters". In: *SIGMOD*. 2007.

[YD04] Praveen Yalagandula and Mike Dahlin. "SDIMS: A scalable distributed information management system". In: *SIGCOMM*. 2004.

[YGI09] Yuan Yu, Pradeep Kumar Gunda, and Michael Isard. “Distributed aggregation for data-parallel computing: interfaces and implementations”. In: *SOSP*. 2009.

[Yu+20] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. “NetLock: Fast, Centralized Lock Management Using Programmable Switches”. In: *ACM SIGCOMM*. 2020, pp. 126–138.

[Zah+10] Matei Zaharia, Khaled Elmeleegy, Dhruba Borthakur, Scott Shenker, Joydeep Sen Sarma, and Ion Stoica. “Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling”. In: *European Conference on Computer Systems (EuroSys)*. ACM, 2010, pp. 265–278.

[Zah+12] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. “Resilient Distributed Datasets: a Fault-Tolerant Abstraction for In-memory Cluster Computing”. In: *NSDI*. 2012.

[ZBH18] Peng Zheng, Theophilus Benson, and Chengchen Hu. “P4Visor: lightweight virtualization and composition primitives for building and testing modular programs”. In: *ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*. 2018, pp. 98–111.

[Zen+20] Lior Zeno, Dan RK Ports, Jacob Nelson, and Mark Silberstein. “SwiShmem: Distributed Shared State Abstractions for Programmable Switches”. In: *Workshop on Hot Topics in Networks (HotNets)*. ACM, 2020.

[Zha+15] Zhi-Hui Zhan, Xiao-Fang Liu, Yue-Jiao Gong, Jun Zhang, Henry Shu-Hung Chung, and Yun Li. “Cloud Computing Resource Scheduling and a Survey of Its Evolutionary Approaches”. In: *ACM Computing Surveys* 47.4 (July 2015).

[Zha+17a] Cheng Zhang, Jun Bi, Yu Zhou, Abdul Basit Dogar, and Jianping Wu. “HyperV: A high performance hypervisor for virtualization of the programmable data plane”. In: *IEEE International Conference on Computer Communication and Networks (ICCCN)*. 2017, pp. 1–9.

[Zha+17b] Haoyu Zhang, Logan Stafman, Andrew Or, and Michael J. Freedman. “SLAQ: quality-driven scheduling for distributed machine learning”. In: *ACM Symposium on Cloud Computing (SoCC)*. ACM, 2017, pp. 390–404.

[Zha+18] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman P. Amarasinghe. “GraphIt: a high-performance graph DSL”. In: *Proceedings of the ACM on Programming Languages* 2 (2018), 121:1–121:30.

[Zha+19] Cheng Zhang, Jun Bi, Yu Zhou, and Jianping Wu. “HyperVDP: High-Performance Virtualization of the Programmable Data Plane”. In: *Journal on Selected Areas in Communications (JSAC)* 37.3 (2019), pp. 556–569.

[Zhu+19] Hang Zhu, Zhihao Bai, Jialin Li, Ellis Michael, Dan RK Ports, Ion Stoica, and Xin Jin. “Harmonia: Near-linear scalability for replicated storage with in-network conflict detection”. In: *Proceedings of the VLDB Endowment* 13.3 (2019), pp. 376–389.

[ZLZ19] Qixia Zhang, Fangming Liu, and Chaobing Zeng. “Adaptive Interference-Aware VNF Placement for Service-Customized 5G Network Slices”. In: *INFOCOM*. 2019, pp. 2449–2457.

[ZWY19] Wei Zhou, K Preston White, and Hongfeng Yu. “Improving Short Job Latency Performance in Hybrid Job Schedulers with Dice”. In: *International Conference on Parallel Processing (ICPP)*. 2019, p. 56.

List of Abbreviations

`CompReq` *composite resource request* 123–129, 192

`CompStore` *composite template store* 119, 123–129, 141, 192

`PolyReq` *polymorphic resource request* 123, 124, 127–130, 132, 133, 136, 140–142, 149, 192, 193, 195

ACP admission control policy 74, 83

ASIC application-specific integrated circuit 28, 32

CDF commulative distribution function 147, 193

DNN deep neural network 8

DPI deep packet inspection 28, 31

DRF dominant resource fairness 112

DSA domain specific architecture 2

DSL domain-specific language 128

FPGA field-programmable gate array 6, 7, 9, 20, 29

GPU graphics processing unit 2, 6, 7, 9, 20, 159

HIRE Holistic INC-aware Resource managEr vi, viii, 5, 10, 12, 14, 95, 101, 115, 117–121, 123–129, 131–149, 155, 157–159, 192, 193, 195

IncSched INC-aware Scheduling vi, viii, 4, 5, 10–14, 95, 97–102, 105–108, 110, 114, 115, 117–120, 139–141, 145, 147, 148, 154, 155, 157–159, 188, 189, 192, 195

IA-MPP integer allocation maximum pressure policy 4, 10–13, 35, 69–72, 74, 75, 79, 81–89, 92, 154, 192, 195

IaaS Infrastructure as a Service 24

INC in-network computing v–viii, 2–14, 17, 21, 26–33, 67, 95, 97–115, 117–149, 154–159, 192, 193, 195

IP integer program 130

ISP Internet service provider 89

K8 Kubernetes 110, 140, 188

K8++ IncSched+K8 110–112, 114, 192

MatReq materialized request 102, 105–108, 110, 112–114

MCMF min-cost max-flow 12, 123, 124, 131, 132, 135, 140, 142, 146–149, 193

ML machine learning 2, 6, 23

MMP Markov modulated process 87, 90

MPP maximum pressure policy 11, 12, 71, 79–81, 92

NAT network address translation 28, 29

NFV network function virtualization 27–30, 87

NIC network interface card 28, 29

NPU network processing unit 29

NSH network service header 77, 84, 91

OPEX operating expenses 2

OSPP offline static planning policy 85, 87

PaaS Platform as a Service 24

QoS quality of service 8, 77, 86, 87

RMT reconfigurable match tables 28, 33, 122

ROME *Robust Aggregation Overlays Minimizing Execution Time* vi, viii, 4, 5, 10–13, 35, 37–40, 42, 45–49, 51–67, 154, 156, 159, 191, 195

RTT round-trip time 84

SaaS Software as a Service 24

SALVE STEAM T-valve 72, 74, 75, 83–85, 91

SDN software defined network 32, 33

SF service function 70, 73–79, 83, 85, 87, 89, 92, 192

SFC service function chain 1, 4, 7, 10, 11, 13, 35, 67, 69–75, 77–81, 84, 86, 87, 89, 90, 92, 93, 154–157, 191, 192

SFF service function forwarder 75–79, 82–89, 91, 192

SFI service function instance 7, 70, 71, 73, 75–79, 83–87, 89, 91, 192

SGHP shortened greedy heuristic policy 85, 87

Sparrow++ IncSched+Sparrow 113, 114, 192

SPN stochastic processing network 11, 12, 71, 74, 75, 77–80, 92, 192

SR-IOV Single Root I/O Virtualization 20

STEAM multi-site cooperative IA-MPP vi, viii, 4, 5, 10–13, 35, 69–72, 74, 75, 83–86, 88–93, 154, 156, 157, 159, 192, 195

TAG tenant application graph 23, 101

ToR top-of-rack 21, 122, 141

TPU tensor processing unit 2, 6, 7, 20, 159

VC virtual cluster 23, 101

VM virtual machine 2, 20, 24, 101, 102, 134

VNF virtualized network function 73, 74

VOC virtual oversubscribed cluster 23, 101

Yarn++ IncSched+Yarn 112, 114, 192

List of Figures

List of Tables

A Curriculum Vitæ

Gemäß §8 Abs. 1 lit. a der Promotionsordnung der TU Darmstadt

Wissenschaftlicher Werdegang

2008 – 2012	Bachelorstudium der Informatik an der TU Darmstadt
2012 – 2015	Masterstudium der Informatik an der TU Darmstadt
2015 – 2021	Wissenschaftlicher Mitarbeiter am Fachgebiet Programmierung verteilter Systeme des Fachbereichs Informatik an der TU Darmstadt

B

Erklärung laut Promotionsordnung

Gemäß der Promotionsordnung der TU Darmstadt,

§8 Abs. 1 lit. c Promotionsordnung
Ich versichere hiermit, dass die elektronische Version mit der schriftlichen Version übereinstimmt.

§8 Abs. 1 lit. d Promotionsordnung
Ich versichere hiermit, dass zu einem vorherigen Zeitpunkt noch keine Promotion versucht wurde. In diesem Fall sind nähere Angaben über Zeitpunkt, Hochschule, Dissertationsthema und Ergebnis dieses Versuches mitzuteilen.

§9 Abs. 1 Promotionsordnung
Hiermit erkläre ich, die vorgelegte Arbeit zur Erlangung des akademischen Grades "Doktor-Ingenieur (Dr.-Ing.)" mit dem Titel "Holistic Runtime Scheduling for the Distributed Computing Landscape" selbständig und ausschließlich unter Verwendung der

angegebenen Hilfsmittel erstellt zu haben.

§9 Abs. 2 Promotionsordnung
Die vorgelegte Arbeit hat bisher noch nicht zu Prüfungszwecken gedient.

Darmstadt, 01. Februar 2021

Marcel Blöcher