Holistic Resource Scheduling for Data Center In-Network Computing

Marcel Blöcher, Lin Wang, *Senior Member, IEEE*, Patrick Eugster, *Member, IEEE*, and Max Schmidt

***Abstract*— The recent trend towards more programmable switching hardware in data centers opens up new possibilities for distributed applications to leverage in-network computing (INC). Literature so far has largely focused on individual application scenarios of INC, leaving aside the problem of coordinating usage of potentially scarce and heterogeneous switch resources among multiple INC scenarios, applications, and users. Alas, the traditional model of resource pools of isolated compute containers does not fit an INC-enabled data center. This paper describes HIRE, a holistic INC-aware resource manager which allows for server-local and INC resources to be coordinated in unison. HIRE introduces a novel flexible resource (meta-)model to address heterogeneity and resource interchangeability, and includes two approaches for INC scheduling: (a) retrofitting existing schedulers; (b) designing a new one. For (a), HIRE presents a retrofitting API and demonstrates it with four state-of-the-art schedulers. For (b), HIRE proposes a flow-based scheduler, cast as a min-cost max-flow problem, where a unified cost model is used to integrate the different costs. Experiments with a workload trace of a 4000 machine cluster show that HIRE makes better use of INC resources by serving 8−30% more INC requests, while simultaneously reducing network detours by 20% and reducing tail placement latency by 50%.**

***Index Terms*— Data center, in-network computing, switch, resource, scheduling, heterogeneity, non-linear.**

I. Introduction

OVER the past decades network appliances have become increasingly programmable. Originally benefitting the prototyping, testing, and deployment of more flexible and novel network(-wide) services and protocols (e.g., routing, congestion control), this trend has been more recently exploited for benefitting more specific applications and services. By supporting certain specific computations "in the network" on the path between data sources and sinks, individual distributed systems concerns like agreement [1] or caching [2], [3], and even high-level application components such as for machine learning [4], [5], [75], can be handled in a much accelerated fashion, ushering in a new era of INC.

Despite the various use cases [6], [7], one main challenge that has been so far overlooked is that of the co-existence of INC-enabled applications, typically known as "multitenancy". Most existing works are focused on isolated scenarios, where network appliances are instrumented for benefitting a single application, and evaluations focus on workloads for that application. Recent work has proposed isolation mechanisms and multitenancy support on a single network appliance [8]–[10], and investigates when to (re-)deploy a task on an INC switch vs.a server [11], but coordinating the usage of network appliances for INC at the network level remains unaddressed. If INC is indeed to establish itself as a paradigm, it is to be expected that INC-enabled applications, or even just several instances of such applications, will compete over resources of network appliances which are clearly limited.

Management of resources considering end hosts/servers in data centers (DCs) without taking into account INC is already a non-trivial problem which has been heavily investigated over the past years [12]–[17], also considering various accelerators [18]–[20]. Throwing network appliance resources — INC resources for short — into the mix adds new challenges and significantly exacerbates known ones (see Fig. 1): (1) Networking components (e.g., programmable application-specific integrated circuits (ASICs)) exhibit **strong heterogeneity** not only w.r.t.processing power, but also programming models supported [21]. A same INC service also has different resource demands when deployed on different switch types [22]. (2) INC resources are scarce, requiring **interchangeable resources** to be specified for fallback as a new scheduling dimension for INC-enabled jobs. (3) INC-enabled jobs impose more **fine-grained locality** constraints w.r.t.the underlying network topology, with dependencies between server and network appliances. (4) INC resources exhibit **non-linear sharing** characteristics; unlike "complete" isolation on servers, partial INC resources (e.g., reconfigurable match table (RMT) stages), can be reused by multiple tenants or INC service(s) on a same switch [8]. These constraints make existing DC resource management frameworks (RMFs) inefficient or inapplicable, calling for new solutions.

This paper presents HIRE (holistic INC-aware resource manager), a new RMF 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

Manuscript received September 28, 2021; revised February 26, 2022 and April 16, 2022; accepted April 18, 2022; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor C. Wu. This work was supported in part by the German Research Foundation (DFG) as part of the projects B2 in the Collaborative Research Center (CRC) under Grant 1053 Multi-Mechanism Adaptation for the Future Internet (MAKI), in part by the Swiss National Science Foundation (SNSF) under Grant 200021_192121, in part by the European Research Council (ERC) under Grant 617805, and in part by NSF under Grant 1618923. *(Corresponding author: Patrick Eugster.)*

Marcel Blöcher was with the Department of Computer Science, Technische Universität Darmstadt, 64289 Darmstadt, Germany. He is now with SAP SE, 69190 Walldorf, Germany.

Lin Wang is with the Department of Computer Science, Vrije Universiteit Amsterdam, 1081 HV Amsterdam, The Netherlands, and also with the Department of Computer Science, Technische Universität Darmstadt, 64289 Darmstadt, Germany.

Patrick Eugster was with the Department of Computer Science, Technische Universität Darmstadt, 64289 Darmstadt, Germany. He is now with the Faculty of Informatics, Università della Svizzera Italiana (USI), 6904 Lugano, Switzerland, and also with Purdue University, West Lafayette, IN 47907 USA (e-mail: eugstp@usi.ch).

Max Schmidt is with the Department of Computer Science, Technische Universität Darmstadt, 64289 Darmstadt, Germany.

Digital Object Identifier 10.1109/TNET.2022.3174783

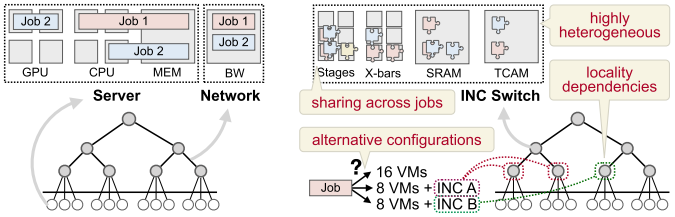


Fig. 1. DC scheduling problem: (left) traditional scheduling with a focus on server and network bandwidth resources, (right) new challenges in scheduling with INC resources.

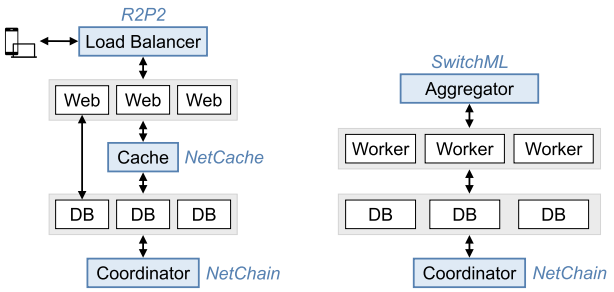


Fig. 2. Example applications with potential INC-enabled components highlighted in blue: (left) typical web application [24] and (right) machine learning training [5].

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 RMF. The polymorphic resource request can also be updated quantitatively at a later time to allow for resource request updates in long-lasting deployments.

HIRE makes scheduling decisions by taking the polymorphic requests and following a scheduling policy. HIRE incorporates two different approaches for the design of such a policy: (a) retrofitting existing schedulers to make them compatible with the resource model and (b) proposing a brand-new HIRE-native scheduler tailored for its resource model. For the former (a), we provide a methodology enabling existing schedulers like Sparrow [16] and YARN [58] to work with polymorphic requests with automatic transformations. Retrofitting existing schedulers provides the benefit of backward compatibility, but comes with the disadvantage that not all the challenges can be successfully addressed due to the design limitations of the existing schedulers. To fully leverage the features offered by the resource model (b), HIRE proposes a novel flow-based scheduler featuring 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 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.

In summary, this paper makes the following contributions. After summarizing prior efforts on INC-enabled applications (§II) and synthesizing the unique set of challenges faced by INC resource scheduling (§III-A) we present

1) HIRE's design (§III-B), including its novel resource model and corresponding interfaces towards applications (§IV).
2) an API and methodology to retrofit existing schedulers, illustrating it through 4 state-of-the-art schedulers (§V).
3) a novel scheduler following the flow-based approach and our unique flow network and cost model designs (§VI).
4) an evaluation of holistic INC-aware resource manager (HIRE) through large-scale simulations with real-world workload traces (§VII). In short, compared to retrofitted state-of-the-art schedulers, HIRE makes better use of INC resources, serving $8-30\%$ more INC requests while at the same time reducing network detours by 20%, and reducing tail placement latency by 50-60%.

§VIII contrasts with related work. §IX draws final conclusions. Details on HIRE's cost model are given in Appendix IX.

This paper extends a previous report [23] by generalizing HIRE's architecture, and introducing APIs and a methodology for retrofitting existing schedulers.

II. Background and Motivation

We present background information on the landscape of INC and DC scheduling, and motivate the need of a unified RMF.

A. In-Network Computing

Recent advances on programmable data planes have sparked significant interest in (DC) INC [26]. Emerging hardware like programmable switches and smart network interface controllers (smartNICs) with ASICs, field programmable gate arrays (FPGAs), and network processing units (NPUs) are becoming increasingly popular. Besides forwarding packets, these devices are capable of performing some logical/arithmetic operations on packets at line rate. Programming languages such as P4 [27] and frameworks like μP4 [21] provide programming abstractions for the network data plane, enabling network devices to be customized for application-specific computation.

Apart from networking tasks like monitoring [28], [29] and congestion control [30], INC has been explored for various scenarios including data aggregation, caching, and coordination/replication (examples shown in Fig. 2), achieving gains on performance [7] and/or energy efficiency [11]. In-network *data aggregation* for instance sets up aggregation overlays on switches to reduce network traffic for DC jobs (e.g., SQL joins, MapReduce, graph processing, machine learning) that involve partition-aggregate patterns [31], [32]. In-network *caching* offloads highly-frequent key-value pairs to switches to reduce latency in serving queries to these pairs [2], [3], [33]. In-network *coordination* provides locking or concurrency control services to distributed systems on switches [1], [34]–[36], which have also been discussed in the context of state machine replication [37], [38]. Another recent work uses INC for coordinating and *routing* remote procedure calls [40].

All these INC services are originally described to run in exclusivity on a network infrastructure. The management of

these INC services on switches is also handled manually or by the network controller. Consequently, any co-location of these services on a same switch/network can lead to configuration conflicts and especially resource contention. While solutions for running multiple INC services on a single switch have been proposed recently [8]–[10], network-global multitenancy support for INC remains an open problem.

B. DC Scheduling

DC 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 [41]. Both single-resource [12], [16] and multi-resource [42]–[44] scheduling have been well studied.

DC resources are typically shared among multiple applications/frameworks (e.g., Spark, Flink, TensorFlow) [17], [45]. To cope with this sharing, early resource managers (RMs) like Mesos [45] make resource offers to different computing frameworks in rounds. To reduce task placement latency, modern RMs including Omega [46] and Hydra [47] follow a shared-state or federated architecture which provides a shared global view of the cluster for multiple computing frameworks to perform task scheduling simultaneously. RMs employ a scheduling policy for task allocation where both (a) centralized policies and (b) distributed policies have been studied. (a) typically involve sophisticated scheduling algorithms and are known for achieving high resource efficiency [12], [13], [17], [48]–[50]. (b) on the other hand aim to improve scalability by simplifying the scheduler design with distributed randomization techniques [16], [51].

So far, the network is beyond the scope of DC RMs – except for virtual network embeddings used to reserve network *bandwidth* (only) [52]–[55]. Popular DC RMs are completely agnostic to the network status managed by a separate entity, the network controller. No resource models, abstractions, or management frameworks are available to manage INC resources, holistically, i.e., jointly with server resources, for multitenant DC environments.

III. Challenges and System Design

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

A. Challenges to DC Scheduling With INC

Presence of INC resources fundamentally changes DC scheduling, further complicating scheduling. In particular, we identify four major challenges listed in the following. Tab. I summarizes how existing schedulers cope with them.

1) Heterogeneity [HET]: Existing RMs consider single- or multi-resources with feature flags [16], [46], [47], and recently server-accelerators like graphical processing units (GPUs) [19], [20], [56] 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 [21]. 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 [21], [22], [59]. Upon service requests the RM needs to interact with the toolchain of a potential target INC switch to determine resource demands like RMT stages (not statically pre-determinable due to non-linear sharing). This makes the scheduling of heterogeneous resources, discussed more broadly w.r.t.related work in §VIII, even more complex.

TABLE I

How Existing Schedulers Cope With INC Challenges. ^{P} Performance Heterogeneity, but Not Late Binding of Exact Task Resource Demands w.r.t.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 RM; ^{D} Single Device; ^{A} Few Discrete Levels or (Anti-)Affinity Constraints, but No Built-In Support e.g.for a Tree or a Chain of Devices

Approach	[HET]	[INT]	[LOC]	[NOL]
HIRE	✓	✓	✓	✓
Heterogeneity-aware RMs				
Gavel [20], AlloX [56], Gandiva [19]	(✓)^{P}	(✓)^{E,S}		
Themis [57]	(✓)^{P}	(✓)^{E,S}	(✓)^{A}	
Tetrished [14]	(✓)^{P}	(✓)^{S}	(✓)^{A}	
Generic RMs				
Hydra [47], Omega [46], Mesos [45], Yarn [58]			(✓)^{A}	
INC switch management				
μP4 [21]		(✓)^{S}		
INC on demand [11]		(✓)^{D,S}		

2) Interchangeability [INT]: Heterogeneity leads to interchangeable resources pending decisions at runtime. Given the scarcity and diversity of INC resources compared to server resources (e.g., the critical resource of on-chip stateful memory is limited to tens of MB on a Tofino switch [3]), 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 [56]. 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 RMs consider interchangeable resources requiring job runtime estimation [19], [20], [56], single device decisions targeting energy efficiency [11], and time-sliced allocations [21] with pre-specified alternatives—none considers RM-induced alternatives at runtime. Straightforwardly encoding all combinations in existing models yields prohibitive complexity.

3) Locality [LOC]: Most INC services come with locality constraints concerning the underlying network topology (e.g., sticking to top-of-rack (ToR) switches [3], using a chain/tree of switches [1], [31]). Taking a decision for a

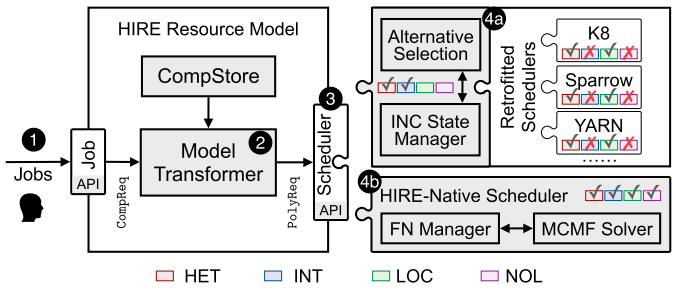


Fig. 3. HIRE system architecture (novelties shown in gray).

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 [3], [31]. Adding extra "detours" via specific appliances may cancel benefits or even worsen performance. Thus INC services possess finer-grained locality requirements than those for pure server jobs where locality is typically described simply with few discrete levels or (anti-)affinity constraints [14], [18], [47], or than relative placement constraints of switches w.r.t.pre-allocated servers [6].

4) Non-Linearity [NOL]: In server-centric RMFs, 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 [8]. 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 resources (e.g., RMT stages [60] in NetCache [3] and HovercRaft [37]) 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. Thus, exact consumption of (scarce) resources depends on co-location of INC services at runtime. Another non-linear resource demand of INC services is packet recirculation. Some INC services [3], [40] leverage packet recirculation for complex configurations, depending on the target switch. For example if the number of registers to compare to in one stage is not sufficient for an INC service. Thus, usage of packet recirculation of an INC service depends on the target switch and specific configuration at runtime. Compute resource sharing may induce memory sharing, e.g., RMTs can have fixed stage-memory mappings [22]. [NOL] may affect multiple resource dimensions.

The above challenges, when combined, make it hard to adapt existing RMFs and corresponding schedulers to work efficiently with INC resources, as we will show in §V.

B. System Design

Targeting all the above challenges, we propose a novel DC scheduler design named HIRE.

1) Overview: A high-level overview of the HIRE architecture is shown in Fig. 3. Tenants ❶ describe their jobs with HIRE's job APIs and submit each job as a *composite resource request* (`CompReq`). We assume the tenants are truthful when specifying their resource requests, as typically done in other works [12], [14], [16]. A `CompReq` is a directed graph of composites (see List. 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`). HIRE then ❸ employs a scheduling policy to make scheduling decisions taking `PolyReq`s as input. HIRE supports two types of policies: 4a retrofitted existing schedulers (e.g., Sparrow, YARN) to work with `PolyReq`s; 4b newly designed HIRE-native schedulers tailored for scheduling `PolyReq`s.

2) HIRE Resource Model (§IV): 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 job APIs, tenants ❶ 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 ❷ turned 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 interchangeable ([INT]) and non-linear ([NOL]) resources.

3) HIRE Retrofitting API (§V): For backward-compatibility HIRE provides 4a an API for retrofitting existing schedulers to work with `PolyReq`s. This is necessary for guaranteeing scheduling correctness as `PolyReq`s contain semantics that are not understandable by existing schedulers. For example, interchangeable resources in a `PolyReq` are mutually exclusive and should not be scheduled simultaneously. HIRE's retrofitting API includes a shim-layer translating `PolyReq`s into normal job requests that can be understood and correctly scheduled by existing schedulers; yet, being designed without INC in mind, these can only partially address the aforementioned challenges.

4) HIRE-Native Scheduler (§VI): To fully address the scheduling challenges with INC, we propose a 4b HIRE-native scheduler to find the mapping of `PolyReq`s to physical resources directly. The scheduling problem differs from the traditional problem chiefly through the alternatives ([INT]) and non-linear resource sharing ([NOL]) in the `PolyReq`. The HIRE-native 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 (FN) by following a carefully designed cost model defining how to translate current DC 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 interchangeable resources and non-linearity in `PolyReq` on the flow network. We reduce the scheduling problem to a standard min-cost max-flow (MCMF) problem for which HIRE employs an efficient MCMF solver, similar to Firmament [12]. In the evaluation (Fig. 6) we test how the modified flow network impacts MCMF solver speed.

IV. HIRE Resource Model

HIRE introduces a new resource model to unify server and INC resources and address [HET] and [INT]. In particular, HIRE introduces the key concept of *composite*, which is

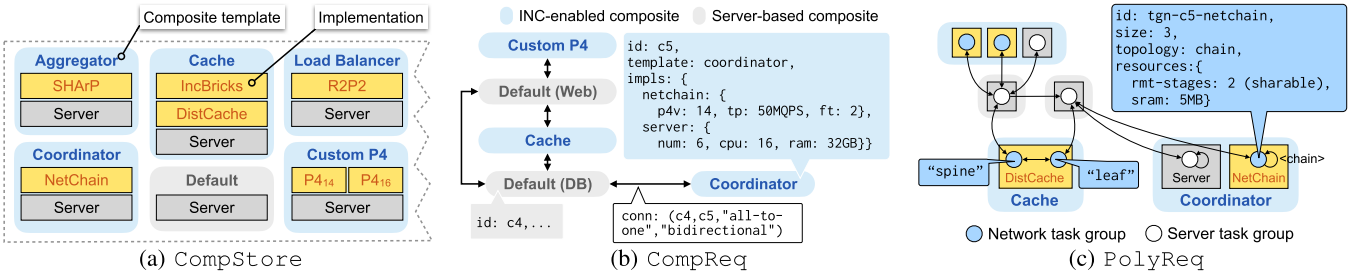


Fig. 4. HIRE resource model for the application in Fig. 2(left): (a) `CompStore` of HIRE with 6 composite templates, (b) schematic representation of a `CompReq`, and (c) the `PolyReq` derived from the `CompReq` by the model transformer.

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 [INT]. For the sake of simplicity, we chose three resource dimensions for INC switches (recirculation capacity, RMT stages, SRAM; cf. §VII-A) and two for servers (CPU, memory). Note that this can be configured by the user and HIRE is not thusly limited, e.g., arithmetic logic units (ALUs) and crossbar units could be considered.

A. 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., 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 Fig. 4a). In addition to pre-configured composites, tenants can expand `default` and `custom-p4` templates for custom ones.

B. 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 List. 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.

```
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)
}
```

List. 1. API for an application master to send a `CompReq`.

Fig. 4b shows a `CompReq` with 5 composites for the typical web application shown in Fig. 2a. 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 configuration map `{p4v:14,tp:50MQPS,ft:2}` capturing the following requirements: 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.

The configuration of inter-composite connectivity between composites "`c4`" and "`c5`" is also shown in Fig. 4b. 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.

C. Polymorphic Resource Requests

HIRE transforms each submitted `CompReq` into a *polymorphic resource request* (`PolyReq`) which is more amenable as input for the scheduler. `CompReqs` expose a friendly performance-oriented interface for tenants to submit job requests without requiring them to specify the exact resource demands explicitly whenever possible. This is especially helpful for INC requests since obtaining their resource

demands requires deep domain knowledge of the switch hardware specification. However, CompReqs have to be transformed to a form that includes detailed resource demands to meet the tenant-specified performance goals, i.e., PolyReqs, before they can be understood by the resource 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.

Fig. 4c depicts the PolyReq that is transformed from the CompReq shown in Fig. 4b. 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 implementation's configuration map.

The PolyReq consumes partial resources on physical switches, which is based on the assumption that the switch runtime can support multiple tenants to be co-located on the same programmable switch simultaneously. Although runtime multi-tenancy support for programmable switches is still a developing research area, we have seen quite promising solutions for resource virtualization of programmable switches [76], [77].

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 mutually exclusive. To support this, PolyReq introduces the concept of *resource flavor* ([INT]), 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 §VI-C.

The CompStore holds graph transformation rules to transform a CompReq to a PolyReq. 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]).

D. Adding Services

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 DC, 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.

V. HIRE Retrofitting API

For backward-compatibility, HIRE provides an API for retrofitting existing schedules. This is essential for ensuring correctness when using existing schedulers to schedule PolyReqs since PolyReqs contain semantics (e.g., alternative selection) that are not understandable by existing schedulers. HIRE['s] retrofitting API (see Fig. 3) 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. Retrofitting Workflow

A job's life-cycle comprises the following steps:

Step 1: HIRE translates each incoming ❸ PolyReq and creates as many materialized requests (MatReqs) as needed (according to the alternative selection strategy), each representing one (mutually exclusive) alternative of a PolyReq. A MatReq is a combination of the task groups that satisfies all constraints of its ancestor PolyReq.

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

Step 3: The plugged scheduler uses the APIs outlined in List. 2 to check for possible task allocations. It informs HIRE of task allocations (resource claims) or ended/terminated tasks. HIRE checks each task allocation update for consequences for its PolyReq['s] alternative selection and updates the INC state manager accordingly. For all INC[-related] allocation updates HIRE informs the network controller accordingly.

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

Lis. 2. The HIRE retrofitting scheduler API (in scala).

Step 4: After HIRE receives the deployment status of the network controller, the scheduling logic receives a callback for each performed task allocation, and continues its allocation logic as desired.

Step 5: Eventually each `PolyReq` needs an alternative selection. Depending on the actual alternative selection logic, a timeout can kick in, which updates the MatReqs of a `PolyReq`. The scheduler receives MatReq updates accordingly, which can trigger further updates on allocations made.

B. INC State Manager

HIRE encapsulates all INC related information and stores state/load information in the INC state manager, so that a plugged scheduler sees the INC resource scheduling problem like *just another server resource scheduling problem*. For each INC service, the INC state manager keeps track of the total resource capacity and resource reservations on switches with jobs involving the INC service. e.g., when a new INC service gets "activated" on a switch, the INC state manager updates both the total capacity and reservation information for the new INC service and the reservation information for all other INC services running on that switch. Furthermore, the INC state manager keeps track of the number of switches which have free resources to activate new INC services. List. 2 shows the API calls to access these cluster-wide INC load estimates. For distributed schedulers like Sparrow, the INC state manager introduces a central point inevitably. This can only be avoided by modifying Sparrow to support alternative selection logic in its design, removing the necessity of the INC state manager.

C. Retrofitting Existing Schedulers

We have retrofitted four existing schedulers, used in the experiments shortly, namely: Kubernetes (K8), Yarn, Sparrow and CoCo (Firmament). In summary, the limitations of the retrofitted schedulers in the face of INC challenges are mitigated 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.

Thus we treat 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, INC compatibility, and INC multiplexing constraints. Each baseline runs each experiment with two *modes* of the alternative selection strategy 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 (10% of a job's duration). We implement the four baselines as follows:

Yarn++: A queuing-based delay scheduler [61] inspired by the Yarn [58] capacity scheduler with two queues (batch/service jobs) with FIFO ordering using task submission times. Yarn++ uses a 1min timeout in concurrent mode, which reverts a job flavor INC decision to prevent starvation. In addition, 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] [62] default configuration, with two active and one backoff queue(s). Similarly to Omega and Borg [62], (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 [63] (Firmament [12]), 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. 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 [64] with batch sampling and late binding inspired by Sparrow [16]. 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.

VI. HIRE SCHEDULER

(The) HIRE (scheduler) has multiple scheduling problems to solve: (1) which flavor to take for each `PolyReq` ([INT]), (2) which server takes which server task, and (3) which switch

TABLE II
NOTATION FOR HIRE

Symbol	Description
J	Job request
T	Task
M^s, M^n	Server node, INC node
G^s, G^n	Server task group, INC task group
$\vec{f}_G$	Flavor vector of task group G
$\vec{x}_J$	Active flavor vector of job J
Z	Task group type
$\vec{d}_G$	Resource demand vector of task group G
$\vec{e}_{Z,M}$	Aggreg. resource demands of task groups 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

takes which INC task. The decision for each problem influences the available options ([NOL]) and possible scheduling quality for the other problems ([LOC]). Tab. II lists notations used.

A. Problem Modeling

INC scheduling can be considered as a variant of the general multi-dimensional bin packing problem [65]. 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 ([INT], [NOL]).

The scheduling problem concerns determining the flavor of each job and mapping the correct task groups in every `PolyReq` onto DC 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 $Z(G)$ task group G's type. For any Z, the total number of tasks that are assigned to node M is given by

$$n_{Z,M} = \sum\nolimits_J \sum\nolimits_{G \in J: Z(G)=Z} \sum\nolimits_{T \in G} y_J s_G a_{T,M}.$$

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

$$\vec{e}_{Z,M} = \sum\nolimits_J \sum\nolimits_{G \in J: Z(G)=Z} \sum\nolimits_{T \in G} y_J s_G a_{T,M} \vec{d}_G.$$

Our scheduling problem can be cast as an integer program:

$$\max \sum_J y_J \ s.t. \ \sum\nolimits_Z \vec{e}_{Z,M} \oslash \vec{q}_{Z,M} \leq \vec{r}_M, \quad \forall M$$

$$\prod\nolimits_{G \in J} \prod\nolimits_{T \in G} \sum\nolimits_M s_G a_{T,M} = y_J, \quad \forall J$$

"$\oslash$" 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 task groups with $s_G = 1$ are scheduled. The integer program 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 DCs with thousands of servers and INC nodes. Thus we present a heuristic that can achieve high efficiency while scaling well.

B. Flow-Based Scheduling Approach

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

1) Approach Overview: Flow-based scheduling, first introduced with Quincy [13], 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 [63, §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 [63, §C.4.2]. HIRE extends flow-based scheduling with unique features to meet its requirements (§III-A) as follows.

2) Capturing INC Constraints: We propose several novelties.

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 DC 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 to reflect server and INC locality constraints, also jointly (i.e., across both flow network parts).

Heterogeneity ([HET]) and 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 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 ([INT]): 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.

HIRE can support fairness with a similar approach as in Quincy and Firmament [63, §6.3.2], by adapting the maximum and minimum number of running tasks for each job.

C. HIRE Flow Network Structure

We show how we build a flow network using these novelties.

1) Nodes: The flow network holds nodes of following types: one super flavor selector node ($\mathbf{S}$); tasks group nodes ($\mathbf{G}$) 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 ($\mathbf{P}$) for each job; one flavor selector node ($\mathbf{F}$) for each job that has alternatives; DC resource nodes ($\mathbf{M}$) including server resource nodes ($\mathbf{M}^s$) and INC resource nodes[1] ($\mathbf{M}^n$); auxiliary nodes ($\mathbf{N}$) for the shadow network (for brevity only half is shown in Fig. 5); one sink node ($\mathbf{K}$).

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

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

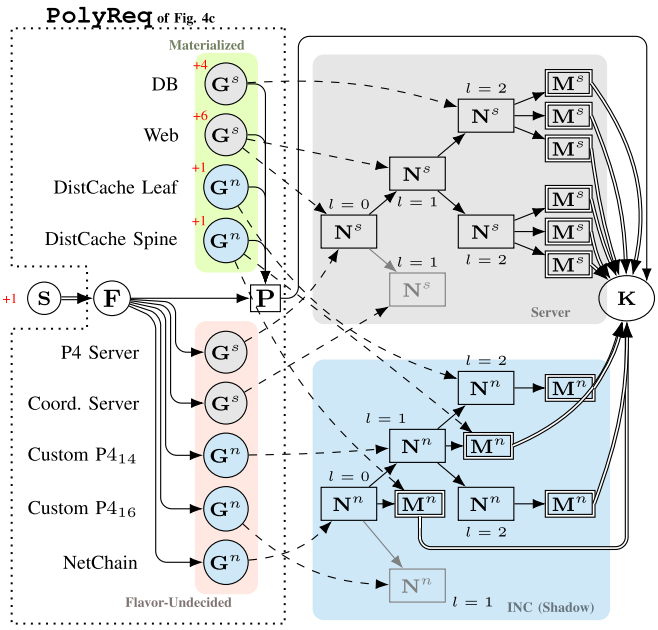


Fig. 5. HIRE flow network for Fig. 4a. Double edges have capacity of 1. Dashed edges are shortcut edges. Numbers in red are positive supplies. l denotes node depth in the topology.

Fig. 5 shows an example flow network for the `PolyReq` of Fig. 4c (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 in the flavor-undecided part, but at most 1 decision per job (the $\mathbf{S}$ node connects to all flavor nodes $\mathbf{F}$, each with an edge of capacity of 1).

3) Cost Model: The HIRE cost model is summarized as follows (for more details please see Appendix IX). There are two sources of positive supplies in the HIRE flow network: (1) supply of $\mathbf{S}$ is given by the number of $\mathbf{F}$ nodes or a customized upper-bound to limit the number of flavor decisions per scheduling round and (2) supply of a materialized $\mathbf{G}$ node equals the number of tasks in the task group. The capacity of all edges $\mathbf{S} \rightarrow \mathbf{F}$ is set to 1 since we allow only 1 flavor decision per job in 1 scheduling round. All edges $\mathbf{M} \rightarrow \mathbf{K}$

also have a capacity of 1 where only one 1 is allowed for each resource node in 1 round. The costs on edges are assigned as follows. For edges $\mathbf{M} \rightarrow \mathbf{K}$ 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 $\mathbf{F}/\mathbf{G} \rightarrow \mathbf{P}$ the cost is proportional to the job queueing time and the number of scheduled tasks of the job. Shortcut edges $\mathbf{G} \rightarrow \mathbf{M}/\mathbf{N}$ 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 $\mathbf{F} \rightarrow \mathbf{G}$ edges is an approximation of the total cost in the corresponding flavor.

Similarly to CoCo [63], 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 $\mathbf{N}$ node maintains a map containing a counter for the running tasks of a task group in the subtree rooted at $\mathbf{N}$; this map is propagated in the flow network via a gossip-like protocol.

4) 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 jobs in the flow network. For a new job J, HIRE initializes the current selected flavor $\vec{x}_J = \langle \mathtt{x} \ldots \mathtt{x} \rangle$ (cf.§VI-A) and adds the job's postpone node $\mathbf{P}$ to the flow network. For each (new) task group of the job, HIRE compares $\vec{f}_G$ with $\vec{x}_J$ and adds a $\mathbf{G}$ node either to the flavor-undecided part or to the materialized part of the job. If all decision variables of $\vec{f}$ (except $\mathtt{x}$) are equal to $\vec{x}$, then $\mathbf{G}$ 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 $\mathtt{x}$ overlapping with $\vec{f}$), $\mathbf{G}$ belongs to the flavor-undecided part. Finally, a $\mathbf{P}$ node is added for the job and each new task group is connected to $\mathbf{P}$. 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 $\mathbf{G}$ nodes of the flavor-undecided part trigger updates of the corresponding $\vec{x}$, i.e., overwriting $\mathtt{x}$ values with $\mathtt{0/1}$. Before moving to the next scheduling round, HIRE checks all $\mathbf{G}$ 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. Appendix IX-B discusses HIRE's complexity of solving the MCMF instance.

VII. Evaluation

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

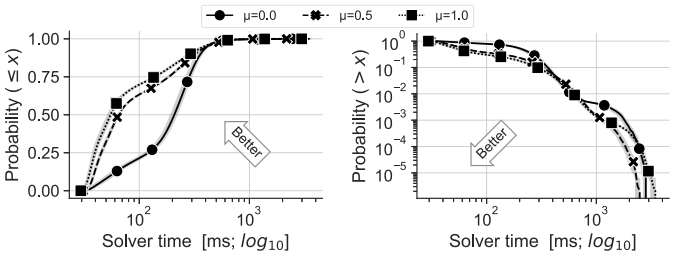


Fig. 6. HIRE MCMF solver speed (CDF and CCDF) at different ratios of `PolyReq`s with INC (from no INC to all INC).

RQ1 How successful is HIRE at fulfilling INC requests as overall demands for INC increase (§VII-B)?
RQ1 How well does HIRE handle resource sharing and INC server locality dependencies (§VII-C)?
RQ1 What is the impact of INC resource heterogeneity on the scheduling problem (§VII-D)?
RQ1 How well does HIRE handle resource contention to improve on tail placement latency (§VII-E)?

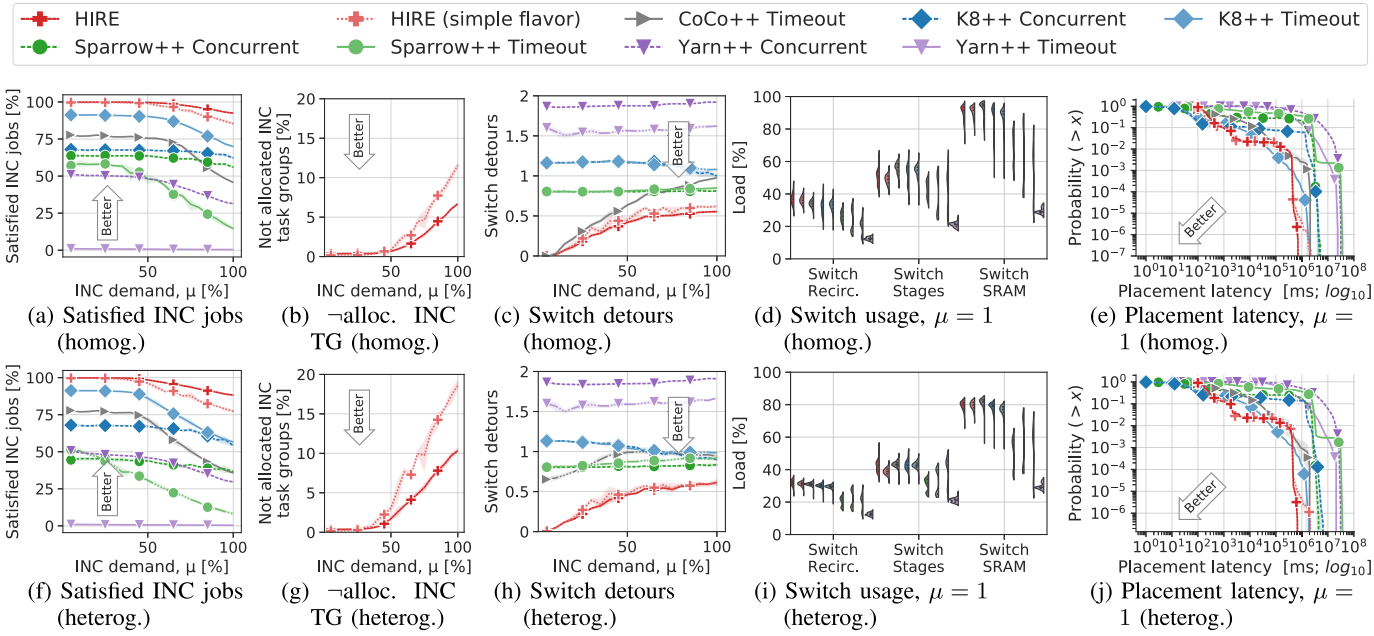
A. Methodology

HIRE's implementation can be largely based on existing data center schedulers, except that a control channel has to be established from the scheduler to each switch for updating switch programs. We focus our evaluation on performance of scheduling policy. Due to lack of a multi-tenant/shared data center testbed for INC, we perform large scale simulations. We built a cluster scheduling simulator (13K lines of Scala code) akin to that of Omega [46], but with support for the HIRE components shown in Fig. 3, INC resources, and multi-path network topologies. The source code is publicly available with all schedulers including retrofitted ones (§V-C) used as baselines.[2] Each experiment — characterized by $\langle$ scheduler, target ratio μ of jobs requesting INC resources, INC heterogeneity (yes/no)$\rangle$ — runs with three seeds; we report:

Satisfied INC jobs: Ratio of `PolyReq`s with INC getting scheduled with INC (Figs. 7a and 7f). For HIRE we also report ratio of scheduled INC task groups (Figs. 7b and 7g).
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. 7c and 7h).
Switch load: Amount of resources per dimension allocated among all switches, measured in a time interval for the whole simulation time (Figs. 7d and 7i).
Placement latency: Time between a task group of a `PolyReq` arrives until all its tasks start processing on machines. As an example, if a task group arrives unluckily right after HIRE started a scheduling round, the task group will be considered in the next scheduling round, which adds to the placement latency of that task group.

We replay 36 hours of a public production workload trace from a 4000 machine Alibaba cluster [66], which contains jobs of two priority classes. For best match we use a fat tree topology with $k = 26$, holding 4394 servers and 845 switches. For switches we define three resource dimensions — reserved recirculation capacity, stages (48), and SRAM size (22MB)

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



(a) Satisfied INC jobs (homog.) (b) ¬alloc. INC TG (homog.) (c) Switch detours (homog.) (d) Switch usage, $\mu = 1$ (homog.) (e) Placement latency, $\mu = 1$ (homog.)

(f) Satisfied INC jobs (heterog.) (g) ¬alloc. INC TG (heterog.) (h) Switch detours (heterog.) (i) Switch usage, $\mu = 1$ (heterog.) (j) Placement latency, $\mu = 1$ (heterog.)

Fig. 7. Scheduling performance as a function of μ (ratio of jobs requesting INC) for experiments with *homog*eneous switches (7a-7e) and with *heterog*eneous switches (7f-7j). The last two plots in each row (7d, 7i, 7e, 7j) focus on $\mu = 1$.

TABLE III
USED INC APPROACHES. LAST 3 COLUMNS GIVE RESOURCE DEMAND PER SWITCH (BEFORE |), AND PER INC INSTANCE (AFTER |)

Name	|Switches|	`PolyReq`	Requirements	Res. recirc. cap.	Stages	SRAM (MB)
SHArP [32]	$\lceil \log \lvert G \rvert \rceil$	Tree	SHArP ASIC	/	/	0 | [1, 8]MB
IncBricks [2]	$\max(3, \lceil \log \lvert G \rvert \rceil)$	Single	OF + Accel	0 | [0, 40]%	0 | [4, 8]	0 | [3, 12]MB
NetCache [3]	$\max(3, \lceil \log \lvert G \rvert \rceil)$	Single (ToR)	$P4_{14}$	0 | [0, 10]%	8 | [0, 8]	0 | [6, 12]MB
DistCache [33]	$\max(3, \lceil \log \lvert G \rvert \rceil)$	cf. 4c	$P4_{14}$	0 | [0, 10]%	8 | [0, 8]	0 | [6, 12]MB
NetChain [1]	$\max(3, 3\lvert G \rvert / 10^3)$	cf. 4c	$P4_{14}$	0 | [0, 10]%	8 | [0, 8]	0 | [6, 12]MB
Harmonia [39]	$\lceil \lvert G \rvert / 9000 \rceil$	Single	$P4_{14}$	0 | 0	3 | [0, 3]	0 | [768, 2048]KB
HovercRaft [37]	$\lceil \lvert G \rvert / 9000 \rceil$	Single	$P4_{14}$	0 | [0, 10]	18 | [0, 18]	0 | [0, 128]KB
R2P2 (JBSQ) [40]	$\lceil \lvert G \rvert / 9000 \rceil$	Single	$P4_{14}$	0 | [0, 30]%	0 | [0, $\lvert G \rvert$]	0 | [1, 64]KB

— to roughly estimate INC resource demands referring to INC processing overhead, program complexity, and storage, respectively [22].

We add 9 INC services to the `CompStore` (listed in Tab. III) — NetChain [1], SHArP [32], IncBricks [2], NetCache [3], DistCache [33], Harmonia [39], HovercRaft [37], and R2P2 [40] — and set resource demand ranges (with resources sharing) according to numbers reported and communicated to us by the respective authors. Existing INC papers do not offer enough details to simulate the network traffic with INC tasks in a sophisticated manner (e.g., with PCAP traces). Therefore, our simulation is more high-level than packet-level simulation. Specifically, we simulate the processing times and resource usages of the tasks that run on switches and servers, which takes into account the impact of network bandwidth and latency implicitly. To discuss the effects of INC heterogeneity (§III-A) 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. 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 a request's `alternative` field). To capture savings of required servers and reduced processing time of a job using INC, we reduce both by up to 10%. (We chose 10% as an upper bound to keep saving effects as a non-dominant source for performance effects - some INC services exhibit savings like $10\times$ or higher, depending on usage pattern [1], [39], [40].)

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 [46], [47], [50] 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 [12], but we also benchmark HIRE to validate the assumption that it runs at similar speed as [12]. Fig. 6 shows median solver speed, when HIRE runs at different levels μ (this benchmark runs on an AMD EPYC 7542). The MCMF solver speed is positively affected by increased INC demand—potentially due to the smaller number of switches vs.number of servers.

For HIRE, we set parameters of the cost model (cf. Appendix IX) as follows: $\S\Phi_{\text{pref}}$ uses 500ms, 2000ms for lower/upper. The upper threshold also sets the timeout for preempting a flavor decision, in case of congested resources. $\S\Phi_w$ 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 for too long (cf. Fig. 6). HIRE and CoCo++ add up to 50 shortcut edges per task group in the graph. All these parameters are set empirically and the cost model should be customized for different scheduling goals.

B. 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 Fig. 7a. We find HIRE serves more than 92% of all jobs when demand is highest, over 30% more than the best baseline of 69% (K8++ concurrent). 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. Fig. 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 Fig. 7a by simply rejecting the majority of each job's INC part. We note the reported numbers correspond to the success rates in Fig. 7a, hence HIRE does not sacrifice fairness among jobs.

C. 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 DC fabric east-west traffic (lower is better). Fig. 7c shows detour values for the experiments of Fig. 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. HIRE prefers placement decisions (server and INC) of the same sub-tree in the network topology. (ii) Switch resource load in Fig. 7d 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.

D. Scheduling Under High INC Heterogeneity (RQ3)

To understand the effect of INC resource heterogeneity on scheduling performance we compare the results with two cluster setups – with (a) homogeneous switches (Figs. 7a-7e) and (b) heterogeneous switches (Figs. 7f-7j). 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. Fig. 7g still validates that HIRE serves INC task groups corresponding to the success rate in Fig. 7f. For switch detours (Fig. 7h), we note similar trends but HIRE shows higher values for $\mu \leq 0.5$ than in (a). Switch resource load (Fig. 7i) unveils the difficulties of resource packing, but the overall trends remain the same – HIRE needs less INC resources all the while serving more jobs with INC.

E. Preventing Resource Contention (RQ4)

Another side-effect of resource heterogeneity is potential resource contention which may lead to long tail placement latencies. Figs. 7e and 7j 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 $< 1s$.

VIII. Related Work

A. DC Resource Models

Existing DC RMs focus mainly on server resources (e.g., CPU, memory); some also consider bandwidth reservations between servers [67]. These approaches use either a simple list of requested virtual machine (VM) resources, or a more complex request model based on, e.g., virtual clusters, virtual oversubscribed clusters, tenant application graphs, or virtual data centers. Yet, all these resource models focus on server resources and bandwidth demands between a group of VMs. As seen in §II, an RM for INC needs to manage not only server resources, but also INC resources, making these models unsuitable. Harmony [6] focuses on intertwining the network controller and the application orchestrator and proposes to extend the *tenant application graph* [52], to encode relative placement constraints of switches to pre-allocated servers. Unlike HIRE, Harmony does not consider resource alternatives and automatic translation of topologies and resource demands.

B. DC Resource Management Frameworks

While various aspects of DC resource management have been explored over the last years (centralized vs.distributed, or prediction-based vs.runtime-agnostic) [67], no existing work tackles the problem of resource management for application requests including INC. The majority of RMFs focus on the scheduler architecture of server-local resource management [17], [45]–[47], [58]. Others focus on scheduling policy design [16], [42], [43], [68], [69]. Quincy [13], Firmament [12], and Aladdin [70] use a network flow model for considering data locality of jobs, which allows to consider shared resources of consecutive jobs. HIRE is inspired by the network flow model used by these schedulers, but contains multiple innovations to capture the INC-specific constraints as discussed in detail in Section VI-B.2. Several approaches consider scheduling as a virtual network embedding problem with the goal of providing bandwidth guarantees [52], [53], [71], [72] between theservers of a job, however ignoring the requirements laid out in §III for an INC-aware RM.

TABLE IV
LIST OF NOTATION USED IN THE COST MODEL

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 $\vec{\sigma}_E$
Φ_i	Cost function i used in Tab. V
w_J	Waiting time of job J
$\vec{u}_M$	Utilization vector of node M
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

C. GPU Scheduling

With widespread adoption of GPUs for accelerating deep learning, a variety of domain-specific schedulers for GPU clusters have been proposed [18], [19], [73], [74]. 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 [73]. Gandiva employs time-slicing and job migration/packing on GPUs for more fine-grained scheduling [19]. Allox discusses the task scheduling problem when CPU and GPU resources are interchangeable [56]. But INC scheduling is yet more complex due to high heterogeneity, fine-grained locality, and on-device resource sharing.

IX. CONCLUSION

HIRE provides a resource management solution for data center INC by introducing (1) a resource model which captures user requests through high-level APIs, transformed automatically to specify resource alternatives, and (2) a novel scheduler design tailored for joint scheduling of server and INC resources under resource alternatives. While supporting retrofitting of existing data center schedulers agnostic to INC, HIRE clearly outperforms four such retrofitted schedulers in large scale simulations. HIRE does rely on other works for compiling/programming toolchain [11], [21], [22], [59] combining INC services [8]–[10] on switches. We see a need for further research for a full-fledged integrated solution of HIRE running in a data center with INC – there is the need for a full implementation of INC resource sharing, with support of partial reconfiguration of runtime re-allocation.

APPENDIX
A HIRE COST MODEL

We use the notation (see Tab. II) of the paper in the appendix. Tab. IV summarizes the notation used in the cost model.

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 w.r.t.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. Tab. V summarizes all cost terms of $\vec{\sigma}$, and refers to sub-cost functions Φ specified below.

A. Job-Independent Costs

The first two edge types in Tab. V, i.e., $\mathbf{M}^s/\mathbf{M}^n \rightarrow \mathbf{K}$, are job-independent and evaluate machine resource utilization and balance. 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:

- $\Phi_{\lfloor \mathrm{P} \rfloor}$ – 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 $\mathbf{M}^n$.
- Φ_{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.

B. Job-Dependent Cost

The remaining columns of Tab. V are 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, w.r.t.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 tasks ($\S\Phi_{\mathrm{loc}}$), leverage INC resource sharing ($\S\Phi_{\mathrm{new}}$), and prioritize long waiting task groups ($\S\Phi_{\mathrm{delay}}$).

- Φ_{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 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 (Eq. 1) for the server part of flow network (with $\mathbf{M}^s$ and $\mathbf{N}^s$), and Γ (algorithm 1) 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 $\mathbf{G} \rightarrow \mathbf{N}$, $\S\Phi_{\mathrm{loc}}$ checks the two nodes $\mathbf{N}^n, \mathbf{N}^s$ that correspond

TABLE V

HIRE COST MODEL USES MULTI-DIMENSIONAL COST VECTORS FOR EACH EDGE. OTHER EDGES HAVE $\sigma = 0$. 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 $[0, 1]$, AND FOR SOME EDGES WE ADD A PENALTY (LAST ROW). $\oslash$ REFERS TO THE ELEMENT WISE DIVISION (HADAMARD DIVISION)

$\vec{\sigma}$ elements	$\mathbf{M}^s \to \mathbf{K}$	$\mathbf{M}^n \to \mathbf{K}$	$\mathbf{G}^s \to \mathbf{N}^s/\mathbf{M}^s$	$\mathbf{G}^n \to \mathbf{N}^n/\mathbf{M}^n$	$\mathbf{G} \to \mathbf{P}$	$\mathbf{F} \to \mathbf{G}$	$\mathbf{F} \to \mathbf{P}$	$\mathbf{S} \to \mathbf{F}$
Utilization	$\mathsf{avg}(\vec{u})$	$\mathsf{avg}(\vec{u})$	$\mathsf{avg}(\vec{d} \oslash \vec{r})$	$\mathsf{avg}(\vec{d} \oslash \vec{r})$	-	$\S\Phi_{\hat{\vec{x}}}$	-	-
Multiplexing	1-$\mathsf{stddv}(\vec{u})$	1-$\mathsf{stddv}(\vec{u})$	$\mathsf{stddv}(\vec{d} \oslash \vec{r})$	$\mathsf{stddv}(\vec{d} \oslash \vec{r})$	-	-	-	-
Locality	-	$\S\Phi_{\text{ToR}}$	$\S\Phi_{\text{loc}}$	$\S\Phi_{\text{loc}}$	-	-	-	-
Interference	-	$\S\Phi_{\lfloor \text{P} \rfloor}$	1	$\S\Phi_{\text{new}}$	-	-	-	-
Priority	-	-	$\S\Phi_{\text{prio}}$	$\S\Phi_{\text{prio}}$	$\S\Phi_{\text{delay}}$	-	$\S\Phi_w$	-
Flattening	$\mathsf{avg}(\vec{\sigma})$	$\mathsf{avg}(\vec{\sigma})$	$\mathsf{avg}(\vec{\sigma})$	$\mathsf{avg}(\vec{\sigma})$	$\mathsf{avg}(\vec{\sigma})$	$\mathsf{avg}(\vec{\sigma})$	$\mathsf{avg}(\vec{\sigma})$	-
Penalty	-	-	-	-	5	$\S\Phi_{\text{pref}}$	3	1

Algorithm 1 INC Locality Propagation

```
procedure IncLocProp(N_start, G, γ)
    N_visited ← ∅
    N_visit ← {N_start}
    while γ > 0 and N_visit ≠ ∅ do
        N_next ← ∅
        forall the N ∈ N_visit \ N_visited do
            Γ_{N,G} ← Γ_{N,G} + γ                    // propagate
            N_visited ← N_visited ∪ {N}
            N_next ← N_next ∪ neighbors(N)
        N_visit ← N_next \ N_visited
        γ ← ⌊γ/ξ⌋                                    // decay propagation
```

to the same location in the DC, and returns the combination of norm($\Gamma_{N^n,G}$) and $\Upsilon_{N^s,G}$, respectively. (b) For $\mathbf{G} \to \mathbf{M}^n$, $\S\Phi_{\text{loc}}$ simply considers the corresponding $\mathbf{N}^n$ to calculate the costs as per (a). (c) For $\mathbf{G} \to \mathbf{M}^s$, $\S\Phi_{\text{loc}}$ considers a simplified version of Eq. 1 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:

$$\Upsilon_{N_1^s,G} = \frac{\sum_{N_2^s \in children(N_1^s)} \begin{cases} \frac{|G| \text{ not running on } N_2^s}{|G|} & N_2^s \in \{M^s\} \\ \Upsilon_{N_2^s,G} & N_2^s \in \{N^s\} \end{cases}}{|children(N_1^s)|} \tag{1}$$

Φ_{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.

Φ_{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 thresholds, $\S\Phi_{\text{pref}}$ returns 1, if its above, it returns 0, otherwise $3 \times (-\mathsf{tanh}(ratio \times 3 - 3))$, 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. $\S\Phi_{\hat{\vec{x}}}$ depends on $\mathbf{G} \to \mathbf{M}/\mathbf{N}$ and $\vec{f}$ for estimating the overall cost of a flavor. While updating all shortcuts ($\mathbf{G} \to \mathbf{M}/\mathbf{N}$), HIRE updates an approximate cost estimate of each of the involved flavors of $\mathbf{F}$ as follows. The cheapest shortcut edge $\mathbf{G} \to \mathbf{M}/\mathbf{N}$ 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. $\S\Phi_{\hat{\vec{x}}}$ returns for each flavor a cost proportional to the ratio of the estimated flavor cost term compared to the largest flavor cost term.

Φ_{prio} – Proportional to job priority: 0 (highest), 1 (lowest).

Φ_{delay} – Prefer placement of tasks with longer waiting time and with fewer tasks remaining. w_J compared to 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. $\S\Phi_w$ uses a threshold and returns 1 if w_J is above the threshold, or $0.5 \times \mathsf{cos}((ratio - 1.0) \times \pi) + 0.5$, with $ratio$ equals w_J divided by the threshold.

APPENDIX B
FLOW NETWORK SIZE ANALYSIS

The time complexity of the algorithms for solving the MCMF problem highly depends on the size (i.e., the number of nodes and edges) of the flow network (cf. Table 1 in [12]). We now analyze the size increase of the HIRE flow network when compared with that of Firmament [12] using CoCo [63, §7.3]. The Firmament flow network has a node for each server, for each task, and auxiliary nodes for cluster/rack/request aggregators that are much smaller in number than servers. The number of edges is bounded by the square of the number of nodes. In the HIRE flow network, the server part has one node for each server and for each switch, while the INC shadow part has two nodes for each switch. HIRE also has auxiliary nodes for jobs, task groups, and task-group flavors, which are small in number compared with server and task nodes. Consider a fat-tree network with k switch ports. The number of switches is given by $5k^2/4$ and servers by $k^3/4$; the number of switches is $5/k$ times that of the servers. Compared with Firmament, the number of nodes in the flow network increases roughly by $15/k$ times the number of servers, which is a small constant given that k is typically larger than 24 in data centers. However, the performance impact of these additional nodes on solving the MCMF is limited, since flows of server task groups ($\mathbf{G}^s$) can reach only the server part, and flows of network task groups ($\mathbf{G}^n$) only the INC shadow part. Overall, solving the MCMF problem for HIRE has a constant factor higher complexity than that for Firmament.

REFERENCES

[1] X. Jin *et al.*, "NetChain: Scale-free sub-RTT coordination," in *Proc. USENIX NSDI*, 2018, pp. 35–49.

[2] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, and K. Atreya, "Incbricks: Toward in-network computation with an in-network cache," in *Proc. ACM ASPLOS*, 2017, pp. 795–809.

[3] X. Jin *et al.*, "Netcache: Balancing key-value stores with fast in-network caching," in *Proc. ACM SOSP*, 2017, pp. 121–136.

[4] Z. Xiong and N. Zilberman, "Do switches dream of machine learning? Toward in-network classification," in *Proc. ACM HotNets*, 2019, pp. 25–33.

[5] A. Sapio *et al.*, "Scaling distributed machine learning with in-network aggregation," in *Proc. USENIX NSDI*, 2021, pp. 785–808.

[6] T. A. Benson, "In-network compute: Considered armed and dangerous," in *Proc. ACM HotOS*, 2019, pp. 216–224.

[7] D. R. K. Ports and J. Nelson, "When should the network be the computer?" in *Proc. Workshop Hot Topics Operating Syst.*, May 2019, pp. 209–215.

[8] T. Wang *et al.*, "Multitenancy for fast and programmable networks in the cloud," in *Proc. USENIX HotCloud*, 2020.

[9] P. Zheng, T. Benson, and C. Hu, "P4visor: Lightweight virtualization and composition primitives for building and testing modular programs," in *Proc. ACM CoNEXT*, 2018, pp. 98–111.

[10] D. Hancock and J. E. van der Merwe, "Hyper4: Using P4 to virtualize the programmable data plane," in *Proc. ACM CoNEXT*, 2016, pp. 35–49.

[11] Y. Tokusashi, H. T. Dang, F. Pedone, R. Soulé, and N. Zilberman, "The case for in-network computing on demand," in *Proc. 14th EuroSys Conf.*, Mar. 2019, pp. 1–16.

[12] I. Gog, M. Schwarzkopf, A. Gleave, R. N. Watson, and S. Hand, "Firmament: Fast, centralized cluster scheduling at scale," in *Proc. USENIX OSDI*, 2016, p. 99.

[13] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: Fair scheduling for distributed computing clusters," in *Proc. ACM SOSP*, 2009, pp. 261–276.

[14] A. Tumanov, T. Zhu, J. W. Park, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger, "TetriSched: Global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters," in *Proc. ACM EuroSys*, 2016, p. 35:1–35:16.

[15] E. Boutin *et al.*, "Apollo: Scalable and coordinated scheduling for cloud-scale computing," in *Proc. USENIX OSDI*, 2014, pp. 285–300.

[16] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: Distributed, low latency scheduling," in *Proc. ACM SOSP*, 2013, pp. 69–84.

[17] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proc. 10th Eur. Conf. Comput. Syst.*, Apr. 2015, p. 18.

[18] K. Mahajan *et al.*, "Themis: Fair and efficient GPU cluster scheduling for machine learning workloads," in *Proc. USENIX NSDI*, 2020, pp. 289–304.

[19] W. Xiao *et al.*, "Gandiva: Introspective cluster scheduling for deep learning," in *Proc. USENIX OSDI*, 2018, pp. 595–610.

[20] D. Narayanan, K. Santhanam, F. Kazhamiaka, A. Phanishayee, and M. Zaharia, "Heterogeneity-aware cluster scheduling policies for deep learning workloads," in *Proc. USENIX OSDI*, 2020, pp. 481–498.

[21] H. Soni, M. Rifai, P. Kumar, R. Doenges, and N. Foster, "Composing dataplane programs with μP4," in *Proc. ACM SIGCOMM*, 2020, pp. 329–343.

[22] L. Jose, L. Yan, G. Varghese, and N. McKeown, "Compiling packet programs to reconfigurable switches," in *Proc. USENIX NSDI*, 2015, pp. 103–115.

[23] M. Blöcher, L. Wang, P. Eugster, and M. Schmidt, "Switches for HIRE: Resource scheduling for data center in-network computing," in *Proc. 26th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Apr. 2021, pp. 268–285.

[24] C. Qu, R. N. Calheiros, and R. Buyya, "Auto-scaling web applications in clouds: A taxonomy and survey," *ACM Comput. Surv.*, vol. 51, no. 4, p. 73, 2018.

[25] M. Li *et al.*, "Scaling distributed machine learning with the parameter server," in *Proc. USENIX OSDI*, 2014, pp. 583–598.

[26] A. Sivaraman, T. Mason, A. Panda, R. Netravali, and S. A. Kondaveeti, "Network architecture in the age of programmability," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 50, no. 1, pp. 38–44, Mar. 2020.

[27] P. Bosshart *et al.*, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014.

[28] R. B. Basat, S. Ramanathan, Y. Li, G. Antichi, M. Yu, and M. Mitzenmacher, "PINT: Probabilistic in-band network telemetry," in *Proc. ACM SIGCOMM*, 2020, pp. 662–680.

[29] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger, "Sonata: Query-driven streaming network telemetry," in *Proc. ACM SIGCOMM*, 2018, pp. 357–371.

[30] Y. Li *et al.*, "HPCC: High precision congestion control," in *Proc. ACM SIGCOMM*, 2019, pp. 44–58.

[31] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis, "In-network computation is a dumb idea whose time has come," in *Proc. 16th ACM Workshop Hot Topics Netw.*, Nov. 2017, pp. 150–156.

[32] R. L. Graham *et al.*, "Scalable hierarchical aggregation protocol (SHArP): A hardware architecture for efficient data reduction," in *Proc. 1st Int. Workshop Commun. Optimizations HPC (COMHPC)*, Nov. 2016, pp. 1–10.

[33] Z. Liu *et al.*, "DistCache: Provable load balancing for large-scale storage systems with distributed caching," in *Proc. USENIX FAST*, 2019, pp. 143–157.

[34] J. Li, E. Michael, and D. R. Ports, "Eris: Coordination-free consistent transactions using in-network concurrency control," in *Proc. ACM SOSP*, 2017, pp. 104–120.

[35] H. T. Dang, M. Canini, F. Pedone, and R. Soulé, "Paxos made Switchy," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 46, no. 2, pp. 18–24, Apr. 2016.

[36] Z. Yu, Y. Zhang, V. Braverman, M. Chowdhury, and X. Jin, "Netlock: Fast, centralized lock management using programmable switches," in *Proc. ACM SIGCOMM*, 2020, pp. 126–138.

[37] M. Kogias and E. Bugnion, "HovercRaft: Achieving scalability and fault-tolerance for microsecond-scale datacenter services," in *Proc. ACM EuroSys*, 2020, pp. 1–17.

[38] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. Ports, "Just say NO to Paxos overhead: Replacing consensus with network ordering," in *Proc. USENIX NSDI*, 2016, pp. 467–483.

[39] H. Zhu *et al.*, "Harmonia: Near-linear scalability for replicated storage with in-network conflict detection," *VLDB Endowment*, vol. 13, no. 3, pp. 376–389, 2019.

[40] M. Kogias, G. Prekas, A. Ghosn, J. Fietz, and E. Bugnion, "R2P2: Making RPCs first-class datacenter citizens," in *Proc. USENIX ATC*, 2019, pp. 863–880.

[41] M. Schwarzkopf and P. Bailis, "Research for practice: Cluster scheduling for datacenters," *Commun. ACM*, vol. 61, no. 5, pp. 50–53, 2018.

[42] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *Proc. USENIX NSDI*, 2011, p. 24.

[43] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource packing for cluster schedulers," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 455–466, 2014.

[44] M. Chowdhury, Z. Liu, A. Ghodsi, and I. Stoica, "HUG: Multi-resource fairness for correlated and elastic demands," in *Proc. USENIX NSDI*, 2016, pp. 407–424.

[45] B. Hindman *et al.*, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proc. USENIX NSDI*, vol. 11, 2011, p. 22.

[46] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: Flexible, scalable schedulers for large compute clusters," in *Proc. ACM EuroSys*, 2013, pp. 351–364.

[47] C. Curino *et al.*, "Hydra: A federated resource manager for data-center scale analytics," in *Proc. USENIX NSDI*, 2019, pp. 177–192.

[48] C. Delimitrou and C. Kozyrakis, "Paragon: QoS-aware scheduling for heterogeneous datacenters," in *Proc. ACM ASPLOS*, 2013, pp. 77–88.

[49] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and QoS-aware cluster management," in *Proc. ACM ASPLOS*, vol. 2014, pp. 127–144.

[50] M. Tirmazi *et al.*, "Borg: The next generation," in *Proc. ACM EuroSys*, 2020, pp. 1–14.

[51] P. Delgado, D. Didona, F. Dinu, and W. Zwaenepoel, "Job-aware scheduling in eagle: Divide and stick to your probes," in *Proc. ACM SoCC*, 2016, pp. 497–509.

[52] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, "Towards predictable datacenter networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, pp. 242–253, Aug. 2011.

[53] C. Guo *et al.*, "SecondNet: A data center network virtualization architecture with bandwidth guarantees," in *Proc. ACM CoNEXT*, 2010, p. 15.

[54] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica, "FairCloud: Sharing the network in cloud computing," in *Proc. ACM SIGCOMM*, 2012, pp. 187–198.

[55] D. Xie, N. Ding, Y. C. Hu, and R. R. Kompella, "The only constant is change: Incorporating time-varying network reservations in data centers," in *Proc. ACM SIGCOMM*, 2012, pp. 199–210.

[56] T. N. Le, X. Sun, M. Chowdhury, and Z. Liu, "AlloX: Compute allocation in hybrid clusters," in *Proc. ACM EuroSys*, 2020, p. 31.

[57] K. Mahajan *et al.*, "Themis: Fair and efficient GPU cluster scheduling," in *Proc. USENIX NSDI*, 2020, pp. 289–304.

[58] V. K. Vavilapalli *et al.*, "Apache Hadoop YARN: Yet another resource negotiator," in *Proc. ACM SoCC*, 2013, p. 5.

[59] X. Gao, T. Kim, A. K. Varma, A. Sivaraman, and S. Narayana, "Autogenerating fast packet-processing code using program synthesis," in *Proc. 18th ACM Workshop Hot Topics Netw.*, Nov. 2019, pp. 150–160.

[60] P. Bosshart *et al.*, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 99–110, Oct. 2013.

[61] M. Zaharia, K. Elmeleegy, D. Borthakur, S. Shenker, J. S. Sarma, and I. Stoica, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *Proc. ACM EuroSys*, 2010, pp. 265–278.

[62] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *Commun. ACM*, vol. 59, no. 5, pp. 50–57, 2016.

[63] M. Schwarzkopf, "Operating system support for warehouse-scale computing," Ph.D. dissertation, Comput. Lab., Univ. Cambridge, Cambridge, U.K., 2015.

[64] M. Mitzenmacher, "The power of two choices in randomized load balancing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, no. 10, pp. 1094–1104, Oct. 2001.

[65] M. Sindelar, R. K. Sitaraman, and P. Shenoy, "Sharing-aware algorithms for virtual machine colocation," in *Proc. 23rd ACM Symp. Parallelism Algorithms Architectures (SPAA)*, 2011, pp. 367–378.

[66] A. Group. (2018). *Alibaba Cluster Trace Program 2018*. [Online]. Available: https://github.com/alibaba/clusterdata/tree/23c0b40

[67] B. Jennings and R. Stadler, "Resource management in clouds: Survey and research challenges," *J. Netw. Syst. Manage.*, vol. 23, pp. 567–619, Mar. 2014.

[68] W. Zhou, K. P. White, and H. Yu, "Improving short job latency performance in hybrid job schedulers with dice," in *Proc. 48th Int. Conf. Parallel Process.*, Aug. 2019, p. 56.

[69] R. Grandl, M. Chowdhury, A. Akella, and G. Ananthanarayanan, "Altruistic scheduling in multi-resource clusters," in *Proc. USENIX OSDI*, 2016, pp. 65–80.

[70] H. Wu *et al.*, "Aladdin: Optimized maximum flow management for shared production clusters," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2019, pp. 696–707.

[71] C. Fuerst, S. Schmid, L. Suresh, and P. Costa, "Kraken: Online and elastic resource reservations for cloud datacenters," *IEEE/ACM Trans. Netw.*, vol. 26, no. 1, pp. 422–435, Feb. 2018.

[72] J. Lee *et al.*, "CloudMirror: Application-aware bandwidth reservations in the cloud," in *Proc. USENIX HotCloud*, 2013.

[73] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang, "Analysis of large-scale multi-tenant GPU clusters for DNN training workloads," in *Proc. USENIX ATC*, 2019, pp. 947–960.

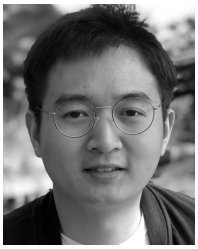
[74] H. Zhao *et al.*, "HiveD: Sharing a GPU cluster for deep learning with guarantees," in *Proc. USENIX OSDI*, 2020, pp. 515–532.

[75] C. Lao *et al.*, "ATP: In-network aggregation for multi-tenant learning," in *Proc. USENIX NSDI*, 2021, pp. 741–761.

[76] R. Stoyanov and N. Zilberman, "MTPSA: Multi-tenant programmable switches," in *Proc. EuroP4@CoNEXT*, 2020, pp. 43–48.

[77] H. Zhu, T. Wang, Y. Hong, D. Ports, A. Sivaraman, and X. Jin, "NetVRM: Virtual register memory for programmable networks," in *Proc. USENIX NSDI*, 2022, pp. 155–170.

Marcel Blöcher received the master's and Ph.D. degrees in computer science from Technische Universität Darmstadt, Germany, in 2015 and 2021, respectively. He is currently working at the SAP Innovation Office within the Global Cloud Services. His research interests include resource scheduling of data center resources and distributed applications running across multiple regions.



Lin Wang (Senior Member, IEEE) received the Ph.D. degree from the Institute of Computing Technology, Chinese Academy of Sciences, in 2015. He held positions at Technische Universität Darmstadt, SnT Luxembourg, and IMDEA Networks Institute. He is currently an Assistant Professor with the Computer Systems Section, Vrije Universiteit Amsterdam. His research interests include networking and distributed systems. He received an Athene Young Investigator Award from Technische Universität Darmstadt in 2018 and a Google Research Scholar Award in 2022.

Patrick Eugster (Member, IEEE) received the M.S. and Ph.D. degrees from the École Polytechnique fédérale de Lausanne (EPFL), Switzerland, in 1998 and 2001, respectively. He is currently a Full Professor of computer science at the Università della Svizzera Italiana (USI) and an Adjunct Faculty Member at Purdue University. He is interested in distributed systems and programming languages. He in particular in the intersection of the two. He was a recipient of an NSF CAREER Award in 2007 and an ERC Consolidator Award in 2012.

Max Schmidt received the bachelor's degree in computer science from Technische Universität Darmstadt, Germany, in 2021, where he is currently pursuing the bachelor's degree in biology and the master's degree in autonomous systems. In the context of computer science, he is primarily interested in distributed systems, machine learning, and robotics.