pre-print author version - to appear at ACM ASPLOS'21

Switches for HIRE: Resource Scheduling for Data Center In-Network Computing

Marcel Blöcher[†] Lin Wang[◇,†] Patrick Eugster[♯,†,‡] Max Schmidt[†]

[†]*TU Darmstadt, Germany* [◇]*Vrije Universiteit Amsterdam, The Netherlands*

[♯]*Università della Svizzera italiana, Switzerland* [‡]*Purdue University, USA*

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. 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 a unified manner. HIRE introduces a novel flexible resource (meta-)model to address heterogeneity, resource interchangeability, and non-linear resource requirements, and integrates dependencies between resources and locations in a unified cost model, cast as a min-cost max-flow problem. In absence of prior work, we compare HIRE against variants of state-of-the-art schedulers retrofitted to handle INC requests. 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 at the same time reducing network detours by* 20%*, and reducing tail placement latency by* 50%*.*

1. 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 [37, 12] or caching [51, 38], and even high-level application components such as for machine learning [83, 66], can be handled in a much accelerated fashion, ushering in a new era of in-network computing (INC).

Despite the various use cases [3, 63], one main challenge that has been so far overlooked is that of the co-existence of INC-enabled applications, typically known as "multitenency". 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 [78, 90, 32, 87, 86], and investigates when to (re-)deploy a task on an INC switch *vs.* a server [74], 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.

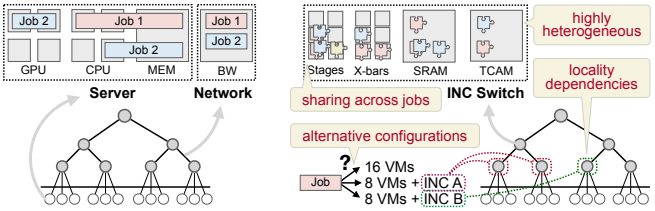

Figure 1: DC scheduling problem: (left) traditional scheduling with a focus on server and network bandwidth resources, (right) new challenges in scheduling with INC resources.

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 [23, 9, 34, 75, 6, 59, 77, 14], also considering GPUs and other accelerators [29, 54, 60, 81, 58]. Throwing network appliance resources — INC resources for short — into the mix adds new challenges and significantly exacerbates existing ones (see Fig. 1): (1) Networking components such as programmable ASICs and NPUs are **highly heterogeneous** in terms of not only processing power, but also programming models supported [19, 72]. The same INC service exhibits different resource demands when deployed on different switch types [39]. (2) INC resources are relatively 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 regarding the underlying network topology, with dependencies between server and network appliances. (4) INC resources exhibit **non-linear sharing** characteristics as, unlike "complete" isolation on servers, partial INC resources (*e.g.*, RMT stages) may be reused by multiple tenants or INC service(s) on the same switch [78]. These constraints render existing

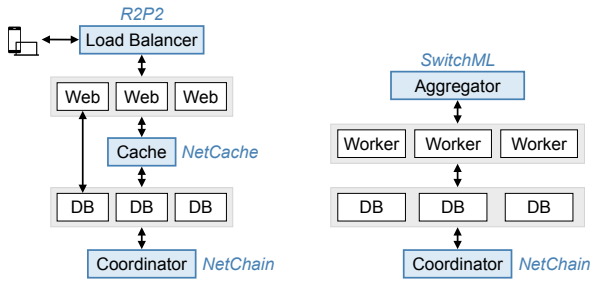

Figure 2: Example applications with potential INC-enabled components highlighted in blue: (left) typical web application [64] and (right) machine learning training [48, 66].

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

In short, this paper makes the following contributions. After summarizing prior efforts on INC-enabled applications (§2) and synthesizing the unique set of challenges faced by INC resource scheduling (§3.1) we

1. present the design of HIRE (§3.2), including its novel model of resources and corresponding interfaces for applications to interact with it (§4).
2. introduce HIRE's novel scheduler following the flow-based approach and our unique designs for the flow network and the cost model (§5).
3. evaluate HIRE through large-scale simulations with real-world workload traces (§6). In short, compared to retrofitted state-of-the-art schedulers, HIRE makes better use of INC resources by serving $8-30\%$ more INC requests, while at the same time reducing network detours by 20%, and reducing tail placement latency by 50-60%.

§7 contrasts with related work. §8 draws final conclusions. Details on HIRE's cost model are given in Appendix A.

2. Background and Motivation

This section presents background information on the landscape of INC as well as DC scheduling, and motivates the need of a unified RMF for both INC and server resources.

In-network computing. Recent advances on programmable data planes have sparked significant interest in (DC) INC [71]. Emerging network hardware like programmable switches and smartNICs with ASICs, FPGAs, and NPUs are becoming increasingly popular. Besides forwarding packet, these devices are capable of performing some logical/arithmetic operations on packets at line rate. Programming languages such as P4 [4] and frameworks like μP4 [72] and Lyra [19] provide programming abstractions for the network data plane, enabling network devices to be customized for application-specific computation.

Apart from networking tasks like monitoring [41, 2, 31, 57] and congestion control [49], INC has been explored for various scenarios including data aggregation, caching, and coordination/replication (examples shown in Fig. 2), achieving gains on performance [63] and/or energy efficiency [74]. In-network data aggregation for instances sets up aggregation overlays on switches to reduce the network traffic for DC jobs (*e.g.*, SQL joins, MapReduce, graph processing, machine learning) that involve partition-aggregate patterns [65, 66, 24]. In-network caching offloads highly-frequent key-value pairs to switches to reduce latency in serving queries to these pairs [38, 51, 52]. In-network coordination provides locking or concurrency control services to distributed (storage) systems on switches [37, 12, 46, 11, 84], which have also been discussed thoroughly in the context of state machine replication [42, 47, 92]. Another recent work uses INC for coordinating/routing remote procedure calls [43].

All these INC services are originally described to run mutually exclusively on a network infrastructure. The management of these INC services on switches is also handled manually or by the network controller. As a result, any co-location of these services on a same switch/network will lead to potential configuration conflicts and especially resource contention. While solutions for running multiple INC services on a single switch have been proposed recently [32, 90, 87, 86, 78], network-global multitenancy support for INC is an open problem.

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 [68]. Both single-resource [59, 23] and multi-resource [22, 25, 8] scheduling have been well studied.

DC resources are typically shared among multiple applications/frameworks (*e.g.*, Spark, Flink, TensorFlow) [33, 77]. To cope with this sharing, early resource managers like Mesos [33] make resource offers to different computing frameworks in rounds. To reduce task placement latency, modern resource managers including Omega [69] and Hydra [10] 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. Resource managers 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 [16, 17, 77, 73, 34, 23]. (b) on the other hand aim to improve scalability by simplifying the scheduler design with distributed randomization techniques [15, 59, 13, 40].

So far, the network is beyond the scope of DC resource managers – except for virtual network embedding algorithms used to reserve network *bandwidth* (only) [1, 30, 62, 82]. Popular DC resource managers are completely agnostic to the status of the network managed by a separate entity — the network controller — thus being unlikely to support INC resources. No resource models, abstractions, or management frameworks are available to manage INC resources, holistically, *i.e.*, jointly with server resources, for a multitenant DC environment.

3. Challenges and System Design

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

3.1. Challenges to DC Scheduling with INC

Presence of INC resources fundamentally changes DC scheduling, further complicating the scheduling problem in four ways. Tab. 1 summarizes how existing schedulers cope with these.

Heterogeneity [HET]**.** Existing resource managers consider single- or multi-resources with feature flags [59, 69, 10], and recently server-accelerators like GPUs [44, 81, 58] 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 [72, 19]. 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 [39, 20, 21, 72, 19]. Upon service requests the resource manager needs to interact with the toolchain of a potential target INC switch to determine resource demands like reconfigurable match tables (RMT) stages (not statically pre-determinable because of non-linear sharing).

Table 1: How existing schedulers cope with INC challenges. ^{P} **performance heterogeneity, but not late binding of exact task resource demands with respect to a target device;** ^{E} **domain specific solution focusing on performance estimates of alternatives;** ^{S} **static alternatives, *i.e.*, alternatives specified in the resource request, not induced by the resource manager;** ^{D} **single device;** ^{A} **few discrete levels or (anti-)affinity constraints, but no built-in support *e.g.* for requesting a tree or a chain of devices.**

Approach	[HET]	[ALT]	[LOC]	[NOL]
HIRE	✓	✓	✓	✓
Heterogeneity-aware resource managers				
Gavel [58]	$(✓)^{P}$	$(✓)^{E,S}$		
AlloX [44]	$(✓)^{P}$	$(✓)^{E,S}$		
Gandiva [81]	$(✓)^{P}$	$(✓)^{E,S}$		
Themis [53]	$(✓)^{P}$	$(✓)^{E,S}$	$(✓)^{A}$	
Tetrished [75]	$(✓)^{P}$	$(✓)^{S}$	$(✓)^{A}$	
Generic resource managers				
Hydra [10]			$(✓)^{A}$	
Omega [69]			$(✓)^{A}$	
Mesos [33]			$(✓)^{A}$	
Yarn [76]			$(✓)^{A}$	
INC switch management				
μP4 [72]		$(✓)^{S}$		
INC on demand [74]		$(✓)^{D,S}$		

This makes the scheduling of heterogeneous resources, discussed more broadly in the light of related work (*cf*. §7), even more complex.

Alternatives [ALT]**.** 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 [38]), 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 [44]. Such flexibility adds an extra dimension to the scheduling problem: which resource demand to accept for each INC-enabled job at runtime. Existing domain-specific resource managers consider interchangeable resources requiring job runtime estimation [44, 81, 58], single device decisions targeting energy efficiency [74], and time-sliced allocations [72] with pre-specified alternatives – none considers resource manager-induced alternatives at runtime. Straightforwardly encoding all combinations in existing models yields

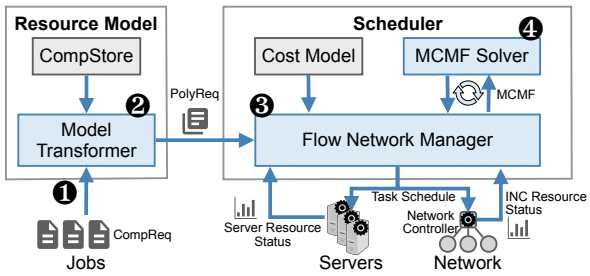

Figure 3: HIRE system architecture.

prohibitive complexity.

Locality [LOC]**.** Most INC services come with locality constraints concerning the underlying network topology, *e.g.*, sticking to top-of-rack (ToR) switches [38] or using a chain/tree of switches [37, 65]. Taking a decision for a specific server or switch strongly impacts the value of all other choices. Furthermore, most benefits in INC scenarios have been shown when INC resources are exploited on communication paths between communicating end points [65, 38]. Adding extra "detours" via specific appliances may cancel benefits or even worsen performance. In short, INC services possess more fine-grained locality requirements than those for pure server jobs where locality is typically described simply with a few discrete levels or (anti-)affinity constraints [75, 10, 54]. Harmony [3] also discusses INC and server placement constraints, but is limited to relative placement constraints of switches to pre-allocated servers.

Non-linearity [NOL]**.** In server-centric 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 [78]. That is, the runtime resource usage of an INC service may depend on a switch's state: if another tenant is using the switch for the same INC service, some INC runtime resources (*e.g.*, RMT stages [5] in NetCache [38] and HovercRaft [42]) 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. Compute resource sharing may induce memory sharing, *e.g.*, RMT can have fixed stage-memory mappings [39]. In general, [NOL] may affect multiple resource dimensions.

These constraints make it hard to adapt existing RMFs and corresponding schedulers to include INC resources.

3.2. System Design

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

Overview. A high-level overview of the HIRE architecture is shown in Fig. 3. Tenants ❶ describe their jobs with HIRE's APIs and submit each job as a *composite resource request* (`CompReq`). A `CompReq` is a directed graph of composites (see 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`) automatically. The HIRE scheduler takes all the `PolyReq`s as input and ❸ generates a flow network embedding all the scheduling constraints and objectives. HIRE then ❹ solves an min-cost max-flow (MCMF) problem instance with the flow network and produces the final scheduling decisions. HIRE also supports incremental submissions of jobs. In particular, tenants can submit a `CompReq` request and indicate its association to a previously submitted one. The scheduler will consider this association and respect the (locality) constraints in scheduling.

HIRE resource model (§4). HIRE features a novel resource model where tenants describe and submit their jobs as `CompReq`s. A `CompReq` consists in a set of composites derived from the composite templates (addressing [HET]) pre-configured in the *composite template store* (`CompStore`). Using HIRE APIs, tenants can specify the configuration for each of the composites in a `CompReq`, and the way composites for a same job are interconnected ([LOC]). Once submitted, `CompReq`s are transformed into `PolyReq`s by the model transformer module. A `PolyReq` considers the different implementation options for the `CompReq`'s composites and provides more detailed resource demands of the job, incorporating resource alternatives ([ALT]) and non-linear resource usage ([NOL]).

HIRE scheduler (§5). HIRE includes a scheduler to find the mapping of `PolyReq`s to physical resources. The scheduling problem differs from the traditional problem chiefly through the alternatives ([ALT]) and non-linear resource sharing ([NOL]) in the `PolyReq`. The HIRE scheduler takes all the `PolyReq`s as input and applies a flow-based scheduling policy. At each scheduling cycle, all newly submitted `PolyReq`s are aggregated and the scheduler generates a flow network by following a carefully designed cost model defining how to translate current 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 alternatives and non-linearity in `PolyReq` on the flow network. We boil the scheduling problem down to a standard MCMF problem for which HIRE employs an efficient MCMF solver, similar to Firmament [23]. In the evaluation (Fig. 7) we test how the modified flow network impacts MCMF solver speed.

4. HIRE Resource Model

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

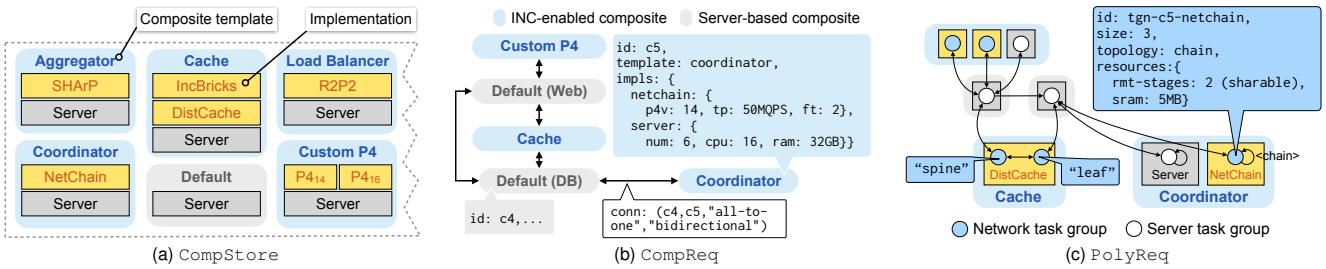

Figure 4: HIRE resource model for the web application scenario of Fig. 2 (left): (a) `CompStore` of HIRE with 6 composite templates, (b) schematic representation of a `CompReq`, and (c) the `PolyReq` derived from the `CompReq` by the model transformer module.

fined as functional unit with a mix of candidate INC and server implementations. HIRE provides composite *templates* together with their implementation details in the `CompStore`, which masks complexity caused by [HET]. In addition, composites allow tenants to specify implementation alternatives to be scheduled at runtime, addressing the [ALT] challenge. For the sake of simplicity, we chose three resource dimensions for INC switches, namely recirculation capacity, RMT stages, and SRAM (cf. §6.2) and two dimensions for servers (CPU and memory). Note that this can be configured by the user and HIRE is not thusly limited, *e.g.*, ALUs and crossbar units could be considered.

```
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`.

4.1. Composite Templates

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

4.2. Composite Resource Requests

Tenants submit jobs in the form of *composite resource requests* (`CompReq`s). 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.

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 following configuration map: `{p4v:14,tp:50MQPS,ft:2}` which instructs the requirements that the INC nodes for `netchain` have to support $P4_{14}$, the throughput has to be at least 50MQPS, and the setup should be able to tolerate up to 2 concurrent node failures. In addition, locality constraints (*e.g.*, `locality:tor`) can also be specified with the configuration map. For the implementation `server`, a configuration map with detailed resource demands is specified by the tenant as 6 servers (*e.g.*, containers) each equipped with 16 CPU cores and 32GB of RAM. HIRE also allows tenants to specify multiple versions for the same implementation in a composite template by supplying different configuration maps.

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.

4.3. Polymorphic Resource Requests

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

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 configuration map of the implementation.

As the implementations specified in a `CompReq` for each composite are alternatives to each other, *i.e.*, only one will be actually scheduled at runtime, the corresponding task groups for these implementations in `PolyReq` are also exclusive to each other. To support this, `PolyReq` introduces the concept of *resource flavor* ([ALT]), and assigns each task group a *flavor vector* $\vec{f}$. The size of $\vec{f}$ equals the total number of decision variables required to encode the `CompReq`, which in most cases is smaller than the total number of task groups. Each element in the flavor vector of a task group represents the relationship of this task group to others and has three possible states: "`0`" (mutually exclusive), "`1`" (concurrent), and "`x`" (ignorable). All $\vec{f}$ of a `PolyReq` are of same length (or padded with "`x`" entries). For example, in the "cache" composite, the flavor vector for the "spine" task group for the `distcache` implementation is $\langle$`xxxx11xxx`$\rangle$, 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 $\langle$`xxxxxxx01`$\rangle$, and for the `server` implementation is $\langle$`xxxxxxx10`$\rangle$, 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 §5.3.

Table 2: Notation for HIRE.

Symbol	Description
	Resource model §4.3
J	Job request
T	Task
M^s and M^n	Server and INC node
G^s and G^n	Server and INC task group
$\vec{f}_G$	Flavor vector of task group G
	Problem modeling §5.1
$\vec{x}_J$	Active flavor vector of job J
Z	Task group type
$\vec{d}_G$	Resource demand vector of task group G
$\vec{e}_{Z,M}$	Aggregated resource demands of ↪ task groups of type Z on M
$a_{T,M}$	Allocation of task T on node M
s_G	Flavor selector for task group G
$\vec{r}_M$	Available resource vector of node M
$\vec{q}_Z$	Sharing degree vector of a task group type Z
y_J	Scheduling decision for job J
	Flow network §5.3
F_J	Flavor selector node of job J
P_J	Unscheduled node of job J
N^s,N^n	Auxiliary topology nodes of server part N^s ↪ and INC (shadow network) part N^n

4.4. Model Transformation

The `CompStore` holds information on how to transform a `CompReq` to a `PolyReq`, by applying graph transformation rules. This allows HIRE to build more complex topologies for specific implementations of a composite template, and allows to hide INC service specific implementation details from the user ([HET]). Our HIRE prototype uses Scala code to describe transformation rules in the `CompStore`, but we could also use a graph domain specific-language (DSL) like GraphIt [89].

4.5. Limitations

HIRE utilizes the information of composite templates for translating resource requests, creating alternatives, and unwrapping resource sharing constraints. To ensure correct deployment profiles of new INC services, especially for all heterogeneous switches of a 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.

5. HIRE Scheduler

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

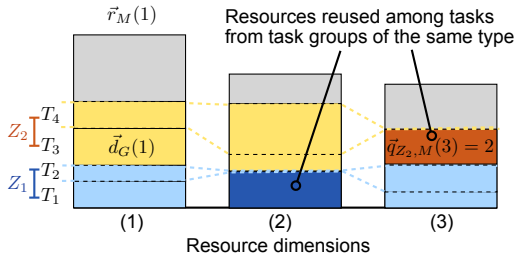

Figure 5: Non-linear resource sharing example.

which INC task. The decision for each of these problems influences the available options ([NOL]) and possible scheduling quality to reach for each other problem ([LOC]). Tab. 2 lists the used notations.

5.1. Problem Modeling

The scheduling problem can be considered as a variant of the general multi-dimensional bin packing problem [70]. We formalize a simplified version of it to highlight the new challenges mentioned above. This formalization is not comprehensive, but captures the most important factors ([ALT], [NOL]).

The scheduling problem concerns determining the flavor of each job and mapping the correct task groups in every `PolyReq` onto 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 $Z(G)$ the type of task group G. Fig. 5 shows an example where all tasks in task groups of type Z_1 share the resources on the second resource dimension while it is the third resource dimension for type Z_2. For any Z, the total number of tasks that are assigned to node M is given by

$$n_{Z,M} = \sum_J \sum_{G \in J:Z(G)=Z} \sum_{T \in G} y_J s_G a_{T,M}. \quad (1)$$

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

$$\vec{e}_{Z,M} = \sum_J \sum_{G \in J:Z(G)=Z} \sum_{T \in G} y_J s_G a_{T,M} \vec{d}_G. \quad (2)$$

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

$$\max \sum_J y_J \quad s.t. \quad (3)$$

$$\sum_Z \vec{e}_{Z,M} \oslash \vec{q}_{Z,M} \leq \vec{r}_M, \forall M \quad (4)$$

$$\prod_{G \in J} \prod_{T \in G} \sum_M s_G a_{T,M} = y_J, \forall J \quad (5)$$

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

5.2. Flow-based Scheduling Approach

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

Approach overview. Flow-based scheduling, first introduced with Quincy [34], 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 [67, §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 [67, §C.4.2]. HIRE extends flow-based scheduling with unique features to meet its requirements (§3.1) as follows.

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

Resource locality ([LOC]): Both server and INC resources need to be integrated in a single flow network so HIRE can schedule resources jointly. When doing so, we must ensure that no flow of a server task can reach nodes referring to INC resources, and vice versa. HIRE achieves this by *having two representations of the 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 for the HIRE cost model to reflect server and INC locality constraints, also jointly (*i.e.*, across the two parts of the flow network).

INC heterogeneity ([HET]), non-linearity ([NOL]): INC services not only consume resources of a "multi-dimensional" resource vector, but have complex dependencies, *e.g.*, the need of a switch feature. Furthermore, when (de)allocating an INC task on a switch, the number of running INC service instances may change depending on the sharing nature of involved services. HIRE keeps track of these dependencies by *propagating status information along the network*, so all possible flows in the flow network end in valid allocations. More importantly, the propagated, cached, status information of the flow network allows HIRE to quickly find matching resources for requesting tasks, respecting heterogeneity and non-linearity.

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

5.3. HIRE Flow Network Structure

We show how to use the above novelties to build a HIRE flow network. Fig. 6 shows an example for the `PolyReq` of Fig. 4c.

Nodes. The flow network holds nodes of following types: one super flavor selector node (**S**); tasks group nodes (**G**) including server task groups ($\mathbf{G}^s$), and network task groups ($\mathbf{G}^n$) according to the `PolyReq`s which are further categorized

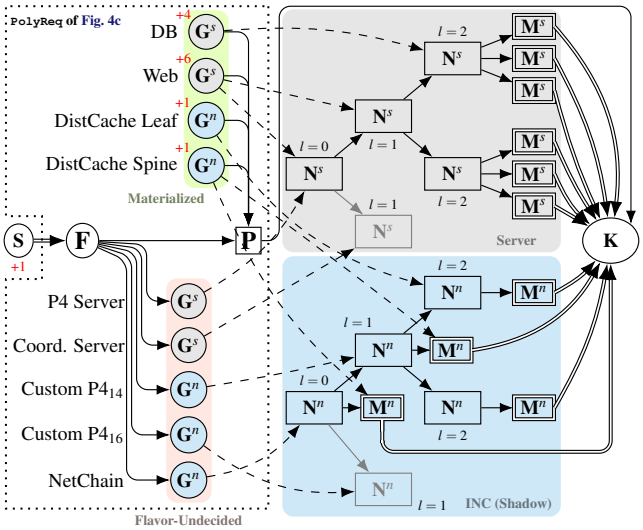

Figure 6: HIRE flow network for Fig. 4c. Double edges have capacity of 1. Dashed edges are shortcut edges. Numbers in red are positive supplies. l denotes node depth in the topology.

into flavor-undecided and materialized (with flavors decided) ones; one postponing node (**P**) for each job; one flavor selector node (**F**) for each job that has alternatives; DC resource nodes (**M**) including server resource nodes ($\mathbf{M}^s$) and INC resource nodes[1] ($\mathbf{M}^n$); auxiliary nodes (**N**) for the shadow network (for brevity only half is shown in Fig. 6); one sink node (**K**).

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

Fig. 6 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

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

by the number of available resource nodes ($\mathbf{M}^n$ and $\mathbf{M}^s$) for serving tasks (resource nodes have edges of capacity 1). In general, HIRE can perform as many decisions in the flavor-undecided part, as jobs take part of the flavor-undecided part, but at most 1 decision per job (the **S** node connects to all flavor nodes **F**, each with an edge of capacity of 1).

Cost model. The HIRE cost model is summarized as follows (for more details please see Appendix A). There are two sources of positive supplies in the HIRE flow network: (1) supply of **S** is given by the number of **F** nodes or a customized upper-bound to limit the number of flavor decisions per scheduling round and (2) supply of a materialized **G** node equals the number of tasks in the task group. The capacity of all edges $\mathbf{S} \rightarrow \mathbf{F}$ is set to one since we allow only one flavor decision per job in one scheduling round. All edges $\mathbf{M} \rightarrow \mathbf{K}$ also have a capacity of one where only one decision is allowed for each resource node in one round. The costs on edges are assigned as follows. For edges $\mathbf{M} \rightarrow \mathbf{K}$ 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 [67], on the server nodes we propagate two numerical vectors of lower and upper bounds of the available resources for the shortcut edge construction. For INC nodes, in addition to the numerical vectors, three bit vectors of size of the number of INC services are used for flagging whether at least one node in its subtree supports the INC service, an INC service is active on all nodes, and an INC service is active on at least one node, respectively. Moreover, each **N** node maintains a map containing a counter for the running tasks of a task group in the subtree rooted at **N**; this map is propagated in the flow network via a gossip-like protocol.

Flow network updates. The flow network is updated upon job arrivals and completions. When jobs arrive, HIRE starts to prepare the next scheduling round by adding or updating the 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.* §5.1) and adds the job's postpone node **P** to the flow network. For each (new) task group of the job, HIRE compares $\vec{f}_G$ with $\vec{x}_J$ and adds a **G** node either to the flavor-undecided part or to the materialized part of the job. If all decision variables of $\vec{f}$ (except `x`) are equal to $\vec{x}$, then **G** belongs to the materialized part. If there is at least one contradiction (`0`$\neq$`1`), the task group is not in the job (anymore). In all other cases ($\vec{x}$ has `x` overlapping with $\vec{f}$), **G** belongs to the flavor-undecided part. Finally, a **P** node is added for the job and each new task group is connected to **P**. The edge costs are updated following our cost model. Upon job completions, the flow network is not immediately updated. Instead, a special flag is assigned to the nodes/edges that are affected. The flow network is updated at the beginning of each scheduling round using the flags on nodes/edges.

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

6. Evaluation

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

RQ1 How successful is HIRE at fulfilling INC requests as overall demands for INC increase (§6.3)?

RQ2 How well does HIRE handle resource sharing and INC server locality dependencies (§6.4)?

RQ3 What is the impact of INC resource heterogeneity on the scheduling problem (§6.5)?

RQ4 How well does HIRE handle resource contention to improve on tail placement latency (§6.6)?

6.1. Retrofitting Existing Schedulers

All experiments compare HIRE against retrofitted variants of four existing schedulers, 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.

More detailed, for these baselines, 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 constraints, INC compatibility, and INC multiplexing constraints. Each baseline runs each experiment with two *modes* for handling job alternatives (INC *vs.* server): ***concurrent*** submits all INC-enabled jobs simultaneously as a server-only and a strict INC job variant, and withdraws the job counterpart on the first allocation that does not fit both

Table 3: INC approaches used in evaluation. Last 3 columns give resource demand per switch (before |), and per INC service instance (after |).

Name	|Switches|	`PolyReq`	Requirements	Res. recirc. cap.	Stages	SRAM (MB)
SHArP [24]	$\lceil \log \lvert G \rvert \rceil$	Tree	SHArP ASIC	/	/	0 | [1,8]MB
IncBricks [51]	$\max(3, \lceil \log \lvert G \rvert \rceil)$	Single	OF + Accel	0 | [0,40]%	0 | [4,8]	0 | [3,12]MB
NetCache [38]	$\max(3, \lceil \log \lvert G \rvert \rceil)$	Single (ToR)	$P4_{14}$	0 | [0,10]%	8 | [0,8]	0 | [6,12]MB
DistCache [52]	$\max(3, \lceil \log \lvert G \rvert \rceil)$	*cf*. Fig. 4c	$P4_{14}$	0 | [0,10]%	8 | [0,8]	0 | [6,12]MB
NetChain [37]	$\max(3, 3\lvert G \rvert / 10^3)$	*cf*. Fig. 4c	$P4_{14}$	0 | [0,10]%	8 | [0,8]	0 | [6,12]MB
Harmonia [92]	$\lceil \lvert G \rvert / 9000 \rceil$	Single	$P4_{14}$	0 | 0	3 | [0,3]	0 | [768,2048]KB
HovercRaft [42]	$\lceil \lvert G \rvert / 9000 \rceil$	Single	$P4_{14}$	0 | [0,10]	18 | [0,18]	0 | [0,128]KB
R2P2 (JBSQ) [43]	$\lceil \lvert G \rvert / 9000 \rceil$	Single	$P4_{14}$	0 | [0,30]%	0 | [0,|G|]	0 | [1,64]KB

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 four baselines:

Yarn++: A queuing-based delay scheduler [85] inspired by the Yarn [76] 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 [7] default configuration, with two active and one backoff queue(s). Similarly to Omega and Borg [7], (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 [67] (Firmament [23]), 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 [56] with batch sampling and late binding inspired by Sparrow [59]. 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.

6.2. Methodology

Due to the lack of a multi-tenant/shared data center testbed for INC, we perform large scale simulations. We built a cluster scheduling simulator (13K lines of Scala code) similar to that of Omega [69], but with support for the HIRE components shown in Fig. 3, INC resources, and multi-path network topologies. The source code of the simulator with all schedulers is publicly available at GitHub[2] (*cf*. Appendix B). Each experiment — characterized by ⟨plugged scheduler, target ratio μ of jobs requesting INC resources, INC heterogeneity (yes/no)⟩ — runs with three seeds; we report the following metrics:

Satisfied INC jobs: Ratio of `PolyReq`s with INC getting scheduled with INC (Figs. 8a and 8f). For HIRE we also report ratio of scheduled INC task groups (Figs. 8b and 8g).

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. 8c and 8h).

Switch load: Amount of resources per dimension allocated among all switches, measured in a time interval for the whole simulation time (Figs. 8d and 8i).

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

We replay 36 hours of a public production workload trace from a 4000 machine Alibaba cluster [28], which contains jobs of two priority classes. To best fit the 4000 servers we use a fat tree topology with $k = 26$, holding 4394 servers and 845 switches. For the switches we define three resource dimensions, namely reserved recirculation capacity, stages (48), and SRAM size (22MB), in order to roughly estimate INC resource demands referring to INC processing overhead, program complexity, and storage, respectively [39].

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

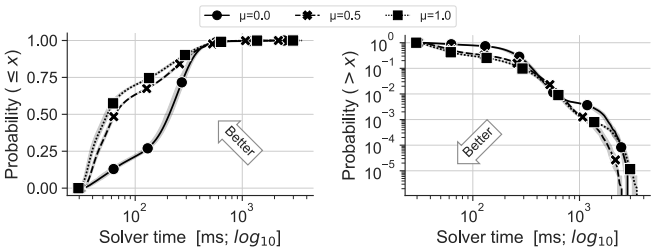

Figure 7: HIRE MCMF solver speed (CDF and CCDF) at different ratios of `PolyReq`s with INC (from no INC to all INC).

We add 9 INC services to the `CompStore` (listed in Tab. 3) — NetChain [37], SHArP [24], IncBricks [51], NetCache [38], DistCache [52], Harmonia [92], HovercRaft [42], and R2P2 [43] — and set resource demand ranges (with resources sharing) according to numbers reported and communicated to us by the authors. To discuss the effects of INC heterogeneity (§3.1) we run two setups, one with all switches of homogeneous capabilities (supporting all INC services) and one with randomly choosing two compatible INC services per switch. To achieve the target ratio μ of jobs requesting INC resources, jobs of the trace are selected randomly, and for up to 1/3rd of a selected job's task groups, any of the INC composites are applied to create a job alternative (adding entries to the `alternative` field of a request). To capture savings of required servers and reduced processing time of a job using INC, we reduce both by 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 $10x$ or higher, depending on usage pattern [43, 92, 37].)

For HIRE, we set parameters of the cost model (*cf*. Appendix A) as follows: Φ_{pref} uses $500ms, 2000ms$ for lower/upper. The upper threshold also sets the timeout for preempting a flavor decision, in case of congested resources. Φ_w uses $500ms$. HIRE is set to perform up to 250 INC flavor decisions per scheduling round. HIRE and CoCo++ limit the number of requesting task groups in the graph to 800 at any time, by using a backlog of "postponed" task groups using FIFO with a task group's submission time. This helps to prevent situations where the MCMF solver runs too long (*cf*. Fig. 7). HIRE and CoCo++ add up to 50 shortcut edges per task group in the graph.

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 [69, 73, 10] 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 [23], but we also benchmark HIRE to validate the assumption that it runs at similar speed as [23]. Fig. 7 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.

Fig. 8 shows the results for experiments with homogeneous switches (8a-8e) and heterogeneous switches (8f-8j), which we discuss in the following.

6.3. Satisfying INC Requests (RQ1)

The primary goal of HIRE is to serve INC requests. We report the ratio of satisfied INC jobs and run experiments where we increase the overall ratio of jobs with INC demands in Fig. 8a. We find HIRE serves more than 92% of all jobs when demand is highest, about 30% more than the best baseline (K8++ concurrent) 69%. For cases with fewest INC demands (only 5% of all jobs ask for INC resources), the improvement is above 8% for all baselines. To further analyze HIRE's performance, we let it run with a simplified flavor logic – decide only once for each job whether to serve the whole `PolyReq` with INC or without. Even with this simplified logic HIRE achieves better results than all baselines, falling below 11% behind normal HIRE. Fig. 8b 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. 8a by simply rejecting the majority of each job's INC part. We note the reported numbers correspond to the success rates in Fig. 8a, hence HIRE does not sacrifice fairness among jobs.

6.4. Cluster Resource Efficiency (RQ2)

We gauge HIRE's ability to use cluster resources efficiently in two ways – by considering (*i*) the switch detour metric and (*ii*) resource load of the switches. (*i*) tests to what extent the scheduler's placement decisions affect DC fabric east-west traffic (lower is better). Fig. 8c shows detour values for the experiments of Fig. 8a: 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. 8d 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

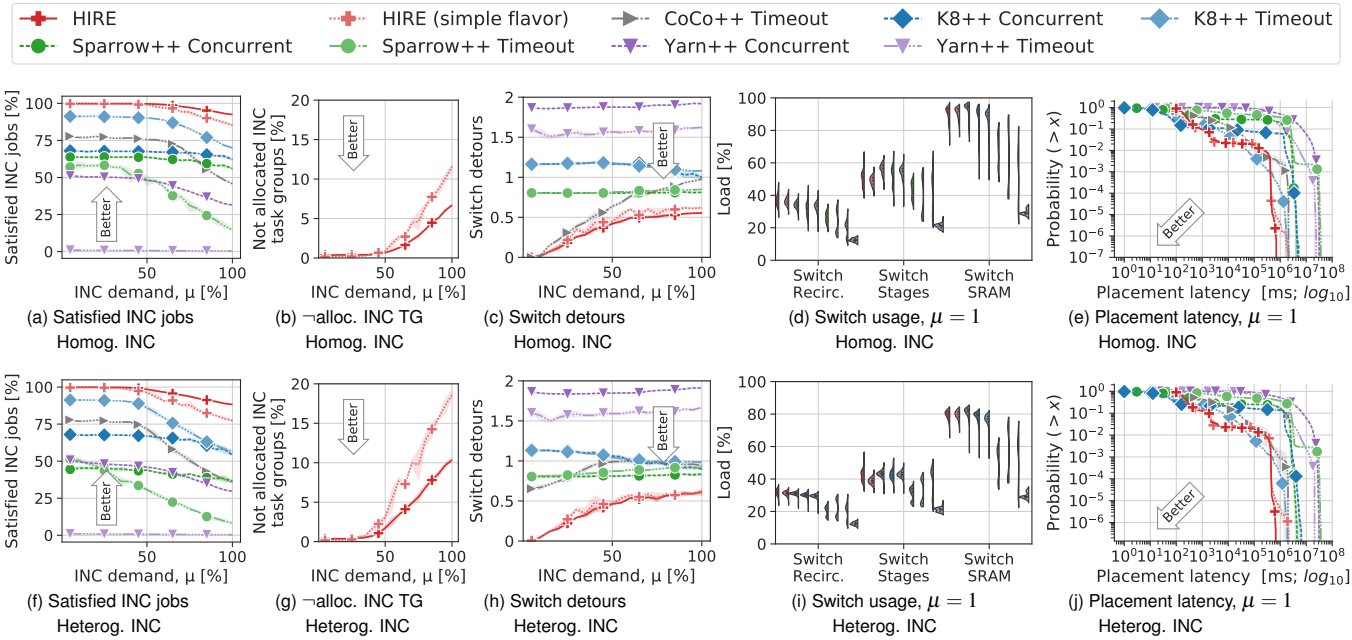

Figure 8: Scheduling performance as a function of μ (ratio of jobs requesting INC) for experiments with homogeneous switches (8a-8e) and with heterogeneous switches (8f-8j). The last two plots in each row (8d, 8i, 8e, 8j) focus on $\mu = 1$.

tasks (and jobs). We attribute this to HIRE's ability to exploit resource sharing of co-located INC services.

6.5. Scheduling Under High INC Heterogeneity (RQ3)

We are particularly interested in understanding the effect of INC resource heterogeneity on scheduling performance. Thus we compare the results with two cluster setups – with (a) homogeneous switches (Figs. 8a-8e) and (b) heterogeneous switches (Figs. 8f-8j). 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. 8g still validates that HIRE serves INC task groups corresponding to the success rate in Fig. 8f. For switch detours (Fig. 8h), we note similar trends but HIRE shows higher values for $\mu \leq 0.5$ than in (a). Switch resource load (Fig. 8i) unveils the difficulties of resource packing, but the overall trends remain the same – HIRE needs less INC resources whilst at the same time serving more jobs with INC.

6.6. Preventing Resource Contention (RQ4)

Another side-effect of resource heterogeneity is potential resource contention which may lead to long tail placement latencies. Figs. 8e and 8j show the complementary CDFs of placement latency when $\mu = 1$. HIRE shows the best tail latency, $50-60\%$ shorter than the best baselines in both scenarios. While making more efficient use of INC resources, HIRE schedules 90% of all allocations with latencies $< 1s$.

7. Related Work

DC resource models. Existing DC resource managers (RMs) focus mainly on server resources (*e.g.*, CPU, memory); very few also consider bandwidth reservations between servers [35]. 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. All these resource models focus on server resources and bandwidth demands between a group of VMs. As seen in §2, an RM for INC needs to manage not only server resources, but also INC resources, making these models unsuitable. Harmony [3] focuses on intertwining the network controller and the application orchestrator and proposes to extend the *tenant application graph* [1], to encode relative placement constraints of switches to pre-allocated servers. Harmony does not consider resource alternatives and automatic translation of topologies and resource demands like HIRE does.

DC RMFs. While various aspects of DC resource management have been explored over the last years (centralized *vs.* distributed, or prediction-based *vs.* runtime-agnostic) [35], none of the existing approaches tackle the problem of resource management for application requests including INC. The majority of RMFs focus on the scheduler architecture of server-local resource management [69, 33, 15, 77, 76, 10, 79]. Others focus on scheduling policy design [22, 59, 14, 91, 27, 25, 26]. Quincy [34], Firmament [23], and Aladdin [80] use a network flow model for considering data locality of jobs, which allows to consider shared resources of consecutive jobs. HyperSched focuses on machine learning training workloads and enables

the automatic exploration of the optimal tradeoff between hyper-parameter configurations and training deadline guarantees [50]. Decima proposes to use reinforcement learning to generate scheduling decisions from experience [55]. Besides server-local resources, some approaches consider the scheduling task as a virtual network embedding problem with the goal of providing bandwidth guarantees [18, 1, 45, 30] between the servers of a job. However, no approach considers the requirements laid out in §3 for INC-aware RM.

GPU scheduling. With widespread adoption of GPUs for accelerating deep learning, a variety of domain-specific schedulers for GPU clusters have been proposed [88, 54, 81, 36, 29, 60, 61]. 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 [36]. Gandiva employs time-slicing and job migration/packing on GPUs for more fine-grained scheduling [81]. Allox discusses the task scheduling problem when CPU and GPU resources are interchangeable [44]. INC scheduling is yet more complex due to high heterogeneity, fine-grained locality, and on-device resource sharing.

8. Conclusions

HIRE provides a resource management solution for data center in-network computing by introducing (a) a resource model which captures user requests through high-level APIs, transformed automatically into logical requests with resource alternatives specified, and (b) a novel scheduler design tailored for joint scheduling of server and in-network computing resources under resource alternatives. In large scale simulations, HIRE clearly outperforms non-trivial retrofitted variants of existing data center schedulers, demonstrating the need for novel solutions in this space. HIRE does rely on other works for the in-network computing compiling/programming toolchain [20, 39, 21, 72, 74, 19] and methods for combining different in-network computing services [78, 90, 32, 87, 86] on switches. We see a need for further research for a full-fledged integrated solution of HIRE running in a data center with in-network computing – there is the need for a full implementation of in-network computing resource sharing, with support of partial reconfiguration of runtime re-allocation.

Acknowledgements

We thank the reviewers and Lizy K John for their feedback. This work has been co-funded by the German Research Foundation (DFG) as part of the projects B2 and C7 in the Collaborative Research Center (CRC) 1053 "MAKI" and DFG grant 392046569 (61761136014 for NSFC), the SNSF grant 200021_192121, ERC grant 617805, and NSF grant 1618923.

Table 4: 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. 5
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

A. HIRE Cost Model

Tab. 2 summarizes the notation used in the appendix.

The cost model of HIRE, together with the flow network, offers the following properties: (1) balanced switch and server utilization, (2) co-located, if possible, in-network computing (INC) service instances of same INC service to maximize resource sharing benefits and keeping the set of active INC services (per switch) small, (3) informed flavor selection where the scheduler always tries to select the "cheapest" flavor with respect to the task counts in task groups and the aggregate flavor costs of tasks in all the task groups belonging to the flavor, and (4) locality-aware scheduling of tasks on machines close to (or covered by 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 min-cost max-flow (MCMF) problem. To this end, we flatten $\vec{\sigma}$ by applying a weighted average. Such weights can be used to model priorities or other custom policies. Tab. 5 summarizes all cost terms of $\vec{\sigma}$, and refers to sub-cost functions Φ specified below.

Job-independent costs. The first two edge types in Tab. 5, *i.e.*, $\mathbf{M}^s/\mathbf{M}^n \to \mathbf{K}$, are job-independent and evaluate machine resource utilization and balance. Costs are lower for machines with low utilization, and with higher variation among the load of all resources dimensions. Furthermore, $\mathbf{M}^n \to \mathbf{K}$ considers the network level in the topology and the number of different active INC services, so that it is less attractive to choose a switch for INC that is not close to a server or which combines more different INC services on the same $\mathbf{M}^n$. More specifically, we define two cost functions:

- $\Phi_{\lfloor \mathbf{P} \rfloor}$ – A cost term proportional to the number of active INC services on an $\mathbf{M}^n$, normalized to the maximum number of INC services that could run on a particular $\mathbf{M}^n$.
- $\Phi_{\mathbf{ToR}}$ – A cost term inversely proportional to the number of network hops an $\mathbf{M}^n$ node is away from its closest $\mathbf{M}^s$ node, normalized to the largest possible distance.

Table 5: HIRE cost model uses multi-dimensional cost vectors for each edge as specified in the table. 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	$\mathsf{avg}(\vec{u})$	$\mathsf{avg}(\vec{u})$	$\mathsf{avg}(\vec{d} \oslash \vec{r})$	$\mathsf{avg}(\vec{d} \oslash \vec{r})$	-	$\Phi_{\hat{\vec{x}}}$	-	-
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	-	Φ_{ToR}	Φ_{loc}	Φ_{loc}	-	-	-	-
Interference	-	$\Phi_{\lfloor \text{P} \rfloor}$	1	Φ_{new}	-	-	-	-
Priority	-	-	Φ_{prio}	Φ_{prio}	Φ_{delay}	-	Φ_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	Φ_{pref}	3	1

Job-dependent cost. The remaining columns of Tab. 5 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, with respect to current load, in each resource dimension. Furthermore, we define the following cost functions for locality, resource interference, and priority. The high-level goal of these cost functions is to co-locate tasks (Φ_{loc}), leverage INC resource sharing (Φ_{new}), and prioritize long waiting G (Φ_{delay}).

$\Phi_{\textbf{new}}$ – 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, $1/(x+1)$ with x the number of active INC services on a switch divided by the max possible.

$\Phi_{\textbf{loc}}$ – For server tasks, HIRE prefers subtrees which already host tasks of the same or a directly connected task group of the same *polymorphic resource request* (`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. 6) for the server part of flow network (with $\mathbf{M}^s$ and $\mathbf{N}^s$), and Γ (Alg. 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 $\mathbf{G} \to \mathbf{N}$, Φ_{loc} checks the two nodes $\mathbf{N}^n, \mathbf{N}^s$ that correspond to the same location in the data center (DC), and returns the combination of norm$(\Gamma_{N^n,G})$ and $\Upsilon_{N^s,G}$, respectively. (b) For $\mathbf{G} \to \mathbf{M}^n$, Φ_{loc} simply considers the corresponding $\mathbf{N}^n$ to calculate the costs as per (a). (c) For $\mathbf{G} \to \mathbf{M}^s$, Φ_{loc} considers a simplified version of Eq. 6 to evaluate the number of tasks running on $\mathbf{M}$, but considers Γ and Υ of the parent $\mathbf{N}^s$ for the connected $\mathbf{G}^s$. Υ is recursively defined:

$$\Upsilon_{N_1^s,G} = \frac{\sum_{N_2^s \in children(N_1^s)} \begin{cases} \frac{|G| \text{ not running on } N_2^s}{|\mathscr{T}_G|} & N_2^s \in \{M^s\} \\ \Upsilon_{N_2^s,G} & N_2^s \in \{N^s\} \end{cases}}{|children(N_1^s)|} \tag{6}$$

$\Phi_{\textbf{pref}}$ – 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, Φ_{pref} returns 1, if its above, it returns 0, otherwise $3 \times (-\tanh(ratio \times 3 - 3))$, 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 G, also the other $G's$ costs are considered. $\Phi_{\hat{\vec{x}}}$ depends on $G \to M/N$ and $\vec{f}$ for estimating the overall cost of a flavor. While updating all shortcuts ($G \to M/N$), HIRE updates an approximate cost estimate of each of the involved flavors of F as follows. The cheapest shortcut edge $G \to M/N$ of each G multiplied with $|G|$ gives the total cost estimate for G. The cost estimate for a flavor is the sum of all involved G estimates. $\Phi_{\hat{\vec{x}}}$ returns for each flavor a cost proportional to the ratio of the estimated flavor cost term compared to the largest flavor cost term.

$\Phi_{\textbf{prio}}$ – Proportional to job priority: 0 (highest), 1 (lowest).

$\Phi_{\textbf{delay}}$ – 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 $w_J \times e^{|G| \text{ scheduled}}/|G| / (\max w \times e)$.

Φ_w – Postpone the flavor decision, if there are only very expensive options available. Φ_w uses a threshold and returns 1 if w_J is above the threshold, or $0.5 \times \cos((ratio - 1.0) \times \pi) + 0.5$, with *ratio* equals w_J divided by the threshold.

B. Artifact Appendix

B.1. Abstract

Our artifact consists of three parts. (1) the source code of the HIRE simulator, including the implementations of Yarn++, Sparrow++, K8++, and CoCo++; (2) the runner tool

Algorithm 1: INC Locality Propagation

```
1  procedure IncLocProp(N_start, G, γ)
2      N_visited ← ∅
3      N_visit ← {N_start}
4      while γ > 0 and N_visit ≠ ∅ do
5          N_next ← ∅
6          forall N ∈ N_visit \ N_visited do
7              Γ_{N,G} ← Γ_{N,G} + γ            // propagate
8              N_visited ← N_visited ∪ {N}
9              N_next ← N_next ∪ neighbors(N)
10         N_visit ← N_next \ N_visited
11         γ ← ⌊γ/ξ⌋                          // decay propagation
```

(a Python3 program) that runs the experiments with the configurations presented in the paper and plotting scripts; and (3) Docker configurations to ease the setup.

Users can reproduce all simulation results (Fig. 8 and Fig. 7). Furthermore, the artifact can be easily extended/modified to benchmark other schedulers, INC configurations, and workloads.

B.2. Artifact Checklist (Meta-Information)

- **Program:** HIRE simulator (discrete event-based simulator in Scala) and tooling (Python3)
- **Compilation:** Scala 2.13, Sbt, Python3
- **Run-time environment:** JVM $\geq$ 11 (and Python3)
- **Hardware:** A server/workstation $\geq$ 32GB RAM, x86_64 CPU with Linux or MacOS (Windows not tested). We used an AMD EPYC 7542 with 512 GB RAM with Ubuntu 20.04.
- **Execution:** Automated by tooling.
- **Metrics:** All metrics discussed in the paper (placement latency, switch resource load, switch detours, satisfied INC jobs). The simulator supports more metrics.
- **Experiments:** (Deterministic[3]) discrete-event simulation; parameter sweeps: one out of 9 schedulers, μ (number of jobs requesting INC) from 5% – 100% (20 configs), with two switch setups, with 3 seeds = in total 1080 experiments
- **How much disk space required (approx.)?:** less than 160GB (Docker 1.7GB, workload trace 1.5GB, experiments $\leq$150GB, depending on verbosity level)
- **How much time is needed to prepare workflow (approx.)?:** less than 1 hour, mostly automated.
- **How much time is needed to complete experiments (approx.)?:** usually 1-16 hours (up to 48 hours) per experiment. Parallelism recommended. Depends strictly on JVM and CPU performance. (AMD EPYC 7542 with 512 GB RAM finishes all experiments in $\sim$ 3 weeks).
- **Publicly available?:** Yes. Open-source on Zenodo[4] and GitHub[5].
- **Code licenses (if publicly available)?:** Apache v2.
- **Workflow framework used?:** Custom tooling.
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.4446702.

[3] Some results may vary when the parallel MCMF solver is used. MCMF solvers find solutions of same cost but maybe with different flows. Schedulers take the result of the fastest MCMF solver. As a result, some simulations are not strictly deterministic.

B.3. Description

B.3.1. How to access: The artifact (source code, scripts, and instructions) is available on Zenodo[4] and on GitHub[5].

The experiments of this paper use a cluster trace which is available on Zenodo[4] and Github[6] (setup scripts give more details on this).

B.3.2. Hardware dependencies: We recommend a x86_64 server with 512GB RAM and 64 cores (for running simulations in parallel). A minimum of 32GB RAM is sufficient for non-parallel simulations.

B.3.3. Software dependencies: *Automated setup:* the setup scripts use Docker on a x86_64 Linux host system (no further dependencies). *Manual setup:* We used Ubuntu 20.04, Python 3.7.5, Scala 2.13.4, SBT 1.4.4, and OpenJDK GraalVM CE 20.1.0 (build 11.0.7+10-jvmci-20.1-b02).

B.4. Installation

Please check the README.md of the artifact for a step-by-step guide. In short, the following steps are required:

1. Download this artifact from Zenodo[4] or GitHub[5]. For the traces, either download the pre-compiled version[4] or follow the steps described in the *README.md* and download the traces manually[6]. (Please note, the simulations do not use the full trace of 48GB)
2. Prepare simulator *jar*
   For the Docker setup:

```
docker␣build␣--build-arg␣HOST_USER=$(id␣-u␣-n)␣--build
↪ -arg␣HOST_UID=$(id␣-u)␣-t␣asplos21-hire/runner:
↪ latest␣.
```

   Alternatively, without Docker:

```
#␣dependencies:␣Python3,␣JDK␣11,␣and␣SBT
pip3␣install␣-r␣requirements.txt
#␣or␣using␣a␣virtual␣environment...
#␣python3␣-m␣venv␣./py-env
#␣source␣"./py-env/bin/activate"
#␣pip3␣install␣-r␣requirements.txt
sbt␣assembly␣
```

3. Prepare workload traces (see *README.md* for details)

B.5. Experiment Workflow

The suggested workflow is described in the *README.md* file of this artifact, linked above. In short, *./src/main/evaluation/experiments/* contains experiment configurations for all experiments showed in the paper. A typical workflow includes:

1. Update/create an experiment configuration in *./src/main/evaluation/experiments/*. The experiment runner creates a Cartesian product of all possible parameter sweeps (separated by colon) of an experiment, and queues simulations accordingly. For example, *seed=0:1:2* and *mu-inp=0.05:1.0:0.5* will end up in 9 simulations. You may want to set the number of parallel simulations with *–worker XX*.

[4] https://doi.org/10.5281/zenodo.4446702

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

[6] https://github.com/alibaba/clusterdata/tree/master/cluster-trace-v2018

2. Check the experiment configuration before starting the simulations using an interactive bash script, keyboard shortcuts are printed.

```
docker run -it -v $PWD:/app --rm asplos21-hire/runner
↪ ./src/main/evaluation/experiments/exp-asplos-
↪ baselines-k8.sh --dry
```

3. Run simulations with

```
docker run -it -v $PWD:/app --rm asplos21-hire/runner
↪ ./src/main/evaluation/experiments/exp-asplos-
↪ baselines-k8.sh
```

This will write all data to the output folder specified in the experiment script (*–output XX*).

The experiment runner creates for each simulation configuration a folder structure like *./exp-rerun-asplos-baselines-k8/run-0/*, *./exp-rerun-asplos-baselines-k8/run-1/* ...
Each simulation folder (*./exp-rerun-asplos-baselines-k8/run-0/*) contains a *config.json* and a *cmd* file with the configuration of the simulation, and a *stats.zip* with csv files. The statistics files (csv) are described in more detail in *Statistics.md*.

Once all simulations finished running, all result data must be located in the local directory. */src/main/evaluation/evals/* contains plotting scripts (Python3), each creating multiple plots (PDFs).

- To run a single plotting scripts, (*e.g.*, Fig. 8a) with results from K8++ and HIRE run

```
docker run -it -v $PWD:/app --rm asplos21-hire/runner
↪ python3 ./src/main/evaluation/evals/
↪ evaluate_paper_inc_success.py -e exp-rerun-asplos-
↪ hire -e exp-rerun-asplos-baselines-k8 -o .. --sweep
↪ mu-inp
```

This example reads experiment data from two folders. The output is relative to the first experiment folder.

- To run all plotting scripts of Fig. 8 with these two schedulers (K8++ and HIRE), run

```
docker run -it -v $PWD:/app --rm asplos21-hire/runner
↪ ./src/main/evaluation/evals/run-paper-eval.sh -e exp
↪ -rerun-asplos-hire -e exp-rerun-asplos-baselines-k8
↪ -o .. --sweep mu-inp
```

B.6. Evaluation and Expected Result

The following commands run the entire paper evaluation. The Docker setup must be prepare before (*cf*. §B.4).

```
# pass "--dry" to check configuration first
# run yarn
docker run -it -v $PWD:/app --rm asplos21-hire/runner ./
↪ src/main/evaluation/experiments/exp-asplos-baselines-
↪ yarn.sh
# run coco
docker run -it -v $PWD:/app --rm asplos21-hire/runner ./
↪ src/main/evaluation/experiments/exp-asplos-baselines-
↪ coco.sh
# run k8
docker run -it -v $PWD:/app --rm asplos21-hire/runner ./
↪ src/main/evaluation/experiments/exp-asplos-baselines-
↪ k8.sh
# run sparrow
docker run -it -v $PWD:/app --rm asplos21-hire/runner ./
↪ src/main/evaluation/experiments/exp-asplos-baselines-
↪ sparrow.sh
# run hire
docker run -it -v $PWD:/app --rm asplos21-hire/runner ./
↪ src/main/evaluation/experiments/exp-asplos-hire.sh
# run hire speed benchmark
docker run -it -v $PWD:/app --rm asplos21-hire/runner ./
↪ src/main/evaluation/experiments/exp-asplos-speed-
↪ benchmark.sh
# create plots Fig 6
docker run -it -v $PWD:/app --rm asplos21-hire/runner ./
↪ src/main/evaluation/evals/run-paper-eval.sh -e exp-
↪ rerun-asplos-hire -e exp-rerun-asplos-baselines-k8 -e
↪ exp-rerun-asplos-baselines-sparrow -e exp-rerun-asplos
↪ -baselines-yarn -e exp-rerun-asplos-baselines-coco -o
↪ .. --sweep mu-inp --ignore time-it:shared-resource-
↪ mode:useSimpleTwoStateInpServerFlavorOptions
# create plots Fig 7
docker run -it -v $PWD:/app --rm asplos21-hire/runner
↪ python3 src/main/evaluation/evals/
↪ evaluate_paper_solver.py -e exp-rerun-asplos-hire-
↪ speed-benchmark -o . . --sweep mu-inp
```

B.7. Experiment Customization

The artifact contains the configurations of all experiments reported in this paper. There are a bunch of parameters and flags for each experiment that modify the behavior of the scheduler, the workload, and the cluster configuration. Additionally, the code is well-documented and written to be easily extensible.

B.8. Notes

For a quick evaluation, the artifact contains an experiment configuration (*exp-asplos-quick-test.sh*) which runs a small subset of the full paper evaluation. The *README.md* has more information on this.

B.9. Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

References

[1] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Towards predictable datacenter networks. In *ACM SIGCOMM Computer Communication Review (CCR)*, volume 41, pages 242–253, 2011.

[2] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minlan Yu, and Michael Mitzenmacher. PINT: probabilistic in-band network telemetry. In *ACM SIGCOMM*, pages 662–680. ACM, 2020.

[3] Theophilus A. Benson. In-network compute: Considered armed and dangerous. In *ACM Workshop on Hot Topics in Operating Systems (HotOS)*, pages 216–224, New York, NY, USA, 2019.

[4] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George

Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review (CCR)*, 44(3):87–95, 2014.

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

[6] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 285–300, 2014.

[7] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *Communications of the ACM*, 59(5):50–57, 2016.

[8] Mosharaf Chowdhury, Zhenhua Liu, Ali Ghodsi, and Ion Stoica. HUG: multi-resource fairness for correlated and elastic demands. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 407–424, 2016.

[9] Carlo Curino, Djellel Eddine Difallah, Chris Douglas, Subru Krishnan, Raghu Ramakrishnan, and Sriram Rao. Reservation-based scheduling: If you're late don't blame us! In *ACM Symposium on Cloud Computing (SoCC)*, pages 2:1–2:14, 2014.

[10] Carlo Curino, Subru Krishnan, Konstantinos Karanasos, Sriram Rao, Giovanni M. Fumarola, Botong Huang, Kishore Chaliparambil, Arun Suresh, Young Chen, Solom Heddaya, Roni Burd, Sarvesh Sakalanaga, Chris Douglas, Bill Ramsey, and Raghu Ramakrishnan. Hydra: a federated resource manager for data-center scale analytics. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 177–192, 2019.

[11] Huynh Tu Dang, Marco Canini, Fernando Pedone, and Robert Soulé. Paxos made switch-y. *ACM SIGCOMM Computer Communication Review (CCR)*, 46(1):18–24, 2016.

[12] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. Netpaxos: Consensus at network speed. In *ACM Symposium on Software Defined Networking Research (SOSR)*, 2015.

[13] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. Job-aware scheduling in eagle: Divide and stick to your probes. In *ACM Symposium on Cloud Computing (SoCC)*, pages 497–509, 2016.

[14] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. Kairos: Preemptive data center scheduling without runtime estimates. In *ACM Symposium on Cloud Computing (SoCC)*, pages 135–148, 2018.

[15] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. Hawk: hybrid datacenter scheduling. In *USENIX Annual Technical Conference (ATC)*, pages 499–510, 2015.

[16] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 77–88, 2013.

[17] Christina Delimitrou and Christos Kozyrakis. Quasar: resource-efficient and qos-aware cluster management. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 127–144, 2014.

[18] Carlo Fuerst, Stefan Schmid, Lalith Suresh, and Paolo Costa. Kraken: Online and elastic resource reservations for cloud datacenters. *IEEE/ACM Transactions on Networking (ToN)*, 26(1):422–435, 2018.

[19] Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. Lyra: A cross-platform language and compiler for data plane programming on heterogeneous asics. In *ACM SIGCOMM*, pages 435–450, 2020.

[20] Xiangyu Gao, Taegyun Kim, Aatish Kishan Varma, Anirudh Sivaraman, and Srinivas Narayana. Autogenerating fast packet-processing code using program synthesis. In *ACM Workshop on Hot Topics in Networks (HotNets)*, pages 150–160, 2019.

[21] Xiangyu Gao, Taegyun Kim, Michael D. Wong, Divya Raghunathan, Aatish Kishan Varma, Pravein Govindan Kannan, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. Switch code generation using program synthesis. In *ACM SIGCOMM*, pages 44–61, 2020.

[22] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 24–24, 2011.

[23] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert NM Watson, and Steven Hand. Firmament: fast, centralized cluster scheduling at scale. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, page 99, 2016.

[24] Richard L Graham, Devendar Bureddy, Pak Lui, Hal Rosenstock, Gilad Shainer, Gil Bloch, Dror Goldenerg, Mike Dubman, Sasha Kotchubievsky, Vladimir Koushnir, et al. Scalable hierarchical aggregation protocol (sharp): a hardware architecture for efficient data reduction. In *IEEE International Workshop on Communication Optimizations in HPC (COMHPC)*, pages 1–10, 2016.

[25] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. *ACM SIGCOMM Computer Communication Review (CCR)*, 44(4):455–466, 2014.

[26] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 65–80, 2016.

[27] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. GRAPHENE: Packing and dependency-aware scheduling for data-parallel clusters. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 81–97, 2016.

[28] Alibaba Group. Alibaba cluster trace program 2018. https://github.com/alibaba/clusterdata/tree/23c0b40, 2018.

[29] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 485–500, Boston, MA, 2019.

[30] Chuanxiong Guo, Guohan Lu, Helen J. Wang, Shuang Yang, Chao Kong, Peng Sun, Wenfei Wu, and Yongguang Zhang. Secondnet: a data center network virtualization architecture with bandwidth guarantees. In *ACM Conference on Emerging Networking Experiments and Technology (CoNEXT)*, page 15, 2010.

[31] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: query-driven streaming network telemetry. In *ACM SIGCOMM*, pages 357–371, 2018.

[32] David Hancock and Jacobus E. van der Merwe. Hyper4: Using P4 to virtualize the programmable data plane. In *ACM International on Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*, pages 35–49, 2016.

[33] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, volume 11, pages 22–22, 2011.

[34] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: fair scheduling for distributed computing clusters. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 261–276, 2009.

[35] Brendan Jennings and Rolf Stadler. Resource management in clouds: Survey and research challenges. *Journal of Network and Systems Management*, pages 1–53, 2014.

[36] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of large-scale multi-tenant GPU clusters for DNN training workloads. In Dahlia Malkhi and Dan Tsafrir, editors, *USENIX Annual Technical Conference (ATC)*, pages 947–960, 2019.

[37] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 35–49, 2018.

[38] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 121–136, 2017.

[39] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. Compiling packet programs to reconfigurable switches. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 103–115, 2015.

[40] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Matteo Fumarola, Solom Heddaya, Raghu Ramakrishnan, and Sarvesh Sakalanaga. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *USENIX Annual Technical Conference (ATC)*, pages 485–497, 2015.

[41] Changhoon Kim, Parag Bhide, E Doe, H Holbrook, A Ghanwani, D Daly, M Hira, and B Davie. In-band Network Telemetry (INT) Dataplane Specification. https://p4.org/assets/INT-current-spec.pdf, 2016.

[42] Marios Kogias and Edouard Bugnion. Hovercraft: Achieving scalability and fault-tolerance for microsecond-scale datacenter services. In *ACM European Conference on Computer Systems (EuroSys)*, 2020.

[43] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2p2: Making rpcs first-class datacenter citizens. In *USENIX Annual Technical Conference (ATC)*, pages 863–880, 2019.

[44] Tan N. Le, Xiao Sun, Mosharaf Chowdhury, and Zhenhua Liu. Allox: Compute allocation in hybrid clusters. In *ACM European Conference on Computer Systems (EuroSys)*, 2020.

[45] Jeongkeun Lee, Myungjin Lee, Lucian Popa, Yoshio Turner, Sujata Banerjee, Puneet Sharma, and Bryan Stephenson. Cloudmirror: Application-aware bandwidth reservations in the cloud. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2013.

[46] Jialin Li, Ellis Michael, and Dan RK Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 104–120, 2017.

[47] Jialin Li, Ellis Michael, Naveen Kr Sharma, Adriana Szekeres, and Dan RK Ports. Just say NO to paxos overhead: Replacing consensus with network ordering. In *USENIX Symposium on Operating Systems Design and Implementation (NSDI)*, pages 467–483, 2016.

[48] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 583–598, 2014.

[49] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. HPCC: high precision congestion control. In *ACM SIGCOMM*, pages 44–58. ACM, 2019.

[50] Richard Liaw, Romil Bhardwaj, Lisa Dunlap, Yitian Zou, Alexey Tumanov, Joseph Gonzalez, and Ion Stoica. HyperSched: Dynamic resource reallocation for model development on a deadline. In *ACM Symposium on Cloud Computing (SoCC)*, 2019.

[51] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. Incbricks: Toward in-network computation with an in-network cache. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 795–809, 2017.

[52] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. Distcache: Provable load balancing for large-scale storage systems with distributed caching. In Arif Merchant and Hakim Weatherspoon, editors, *USENIX Conference on File and Storage Technologies (FAST)*, pages 143–157, 2019.

[53] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient GPU cluster scheduling. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[54] Kshiteej Mahajan, Arjun Singhvi, Arjun Balasubramanian, Varun Batra, Surya Teja Chavali, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient GPU cluster scheduling for machine learning workloads. In *USENIX Symposium on Network Systems Design and Implementation (NSDI)*, 2020.

[55] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. In *ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 270–288, 2019.

[56] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

[57] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-directed hardware design for network performance monitoring. In *ACM SIGCOMM*, pages 85–98, 2017.

[58] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-aware cluster scheduling policies for deep learning workloads. In *14th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 481–498. USENIX Association, 11 2020.

[59] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: distributed, low latency scheduling. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 69–84, 2013.

[60] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In Rui Oliveira, Pascal Felber, and Y. Charlie Hu, editors, *ACM European Conference on Computer Systems (EuroSys)*, pages 3:1–3:14, 2018.

[61] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed DNN training acceleration. In Tim Brecht and Carey Williamson, editors, *ACM Symposium on Operating Systems Principles (SOSP)*, pages 16–29, 2019.

[62] Lucian Popa, Gautam Kumar, Mosharaf Chowdhury, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. Faircloud: sharing the network in cloud computing. In *ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 187–198, 2012.

[63] Dan R. K. Ports and Jacob Nelson. When should the network be the computer? In *ACM Workshop on Hot Topics in Operating Systems (HotOS)*, pages 209–215, 2019.

[64] Chenhao Qu, Rodrigo N. Calheiros, and Rajkumar Buyya. Auto-scaling web applications in clouds: A taxonomy and survey. *ACM Computing Surveys*, 51(4):73:1–73:33, 2018.

[65] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. In-network computation is a dumb idea whose time has come. In *ACM Workshop on Hot Topics in Networks (HotNets)*, 2017.

[66] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan R. K. Ports, and Peter Richtárik. Scaling distributed machine learning with in-network aggregation. *CoRR*, abs/1903.06701, 2019.

[67] Malte Schwarzkopf. *Operating system support for warehouse-scale computing*. PhD thesis, PhD thesis. University of Cambridge Computer Laboratory, 2015.

[68] Malte Schwarzkopf and Peter Bailis. Research for practice: cluster scheduling for datacenters. *Communications of the ACM*, 61(5):50–53, 2018.

[69] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *ACM European Conference on Computer Systems (EuroSys)*, pages 351–364, 2013.

[70] Michael Sindelar, Ramesh K. Sitaraman, and Prashant J. Shenoy. Sharing-aware algorithms for virtual machine colocation. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 367–378, 2011.

[71] Anirudh Sivaraman, Thomas Mason, Aurojit Panda, Ravi Netravali, and Sai Anirudh Kondaveeti. Network architecture in the age of programmability. *ACM SIGCOMM Computer Communication Review (CCR)*, 50(1), 2020.

[72] Hardik Soni, Myriana Rifai, Praveen Kumar, Ryan Doenges, and Nate Foster. Composing dataplane programs with μp4. In *ACM SIGCOMM*, pages 329–343, 2020.

[73] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: The next generation. In *ACM European Conference on Computer Systems (EuroSys)*, 2020.

[74] Yuta Tokusashi, Huynh Tu Dang, Fernando Pedone, Robert Soulé, and Noa Zilberman. The case for in-network computing on demand. In *ACM European Conference on Computer Systems (EuroSys)*, New York, NY, USA, 2019.

[75] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. Tetrisched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *ACM European Conference on Computer Systems (EuroSys)*, pages 35:1–35:16, 2016.

[76] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *ACM Symposium on Cloud Computing (SoCC)*, page 5, 2013.

[77] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *ACM European Conference on Computer Systems (EuroSYs)*, page 18, 2015.

[78] Tao Wang, Hang Zhu, Fabian Ruffy, Xin Jin, Anirudh Sivaraman, Dan R. K. Ports, and Aurojit Panda. Multitenancy for fast and programmable networks in the cloud. In *USENIX Symposium on Hot Topics in Cloud Computing (HotCloud)*, 2020.

[79] Zhijun Wang, Huiyang Li, Zhongwei Li, Xiaocui Sun, Jia Rao, Hao Che, and Hong Jiang. Pigeon: An effective distributed, hierarchical datacenter job scheduler. In *ACM Symposium on Cloud Computing (SoCC)*, pages 246–258, New York, NY, USA, 2019.

[80] Heng Wu, Wenbo Zhang, Yuanjia Xu, Hao Xiang, Tao Huang, Haiyang Ding, and Zheng Zhang. Aladdin: Optimized maximum flow management for shared production clusters. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 696–707, 2019.

[81] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In Andrea C. Arpaci-Dusseau and Geoff Voelker, editors, *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 595–610, 2018.

[82] Di Xie, Ning Ding, Y. Charlie Hu, and Ramana Rao Kompella. The only constant is change: incorporating time-varying network reservations in data centers. In *ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 199–210, 2012.

[83] Zhaoqi Xiong and Noa Zilberman. Do switches dream of machine learning? toward in-network classification. In *ACM Workshop on Hot Topics in Networks (HotNets)*, pages 25–33, 2019.

[84] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. Netlock: Fast, centralized lock management using programmable switches. In *ACM SIGCOMM*, pages 126–138, 2020.

[85] Matei Zaharia, Khaled Elmeleegy, Dhruba Borthakur, Scott Shenker, Joydeep Sen Sarma, and Ion Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *In Proc. EuroSys*, 2010.

[86] Cheng Zhang, Jun Bi, Yu Zhou, Abdul Basit Dogar, and Jianping Wu. Hyperv: A high performance hypervisor for virtualization of the programmable data plane. In *IEEE International Conference on Computer Communication and Networks (ICCCN)*, pages 1–9, 2017.

[87] Cheng Zhang, Jun Bi, Yu Zhou, and Jianping Wu. Hypervdp: High-performance virtualization of the programmable data plane. *IEEE Journal on Selected Areas in Communications*, 37(3):556–569, 2019.

[88] Haoyu Zhang, Logan Stafman, Andrew Or, and Michael J. Freedman. SLAQ: quality-driven scheduling for distributed machine learning. In *ACM Symposium on Cloud Computing (SoCC)*, pages 390–404. ACM, 2017.

[89] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman P. Amarasinghe. Graphit: a high-performance graph DSL. *Proceedings of the ACM on Programming Languages*, 2:121:1–121:30, 2018.

[90] Peng Zheng, Theophilus Benson, and Chengchen Hu. P4visor: lightweight virtualization and composition primitives for building and testing modular programs. In *ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, pages 98–111, 2018.

[91] Wei Zhou, K Preston White, and Hongfeng Yu. Improving short job latency performance in hybrid job schedulers with dice. In *International Conference on Parallel Processing (ICPP)*, page 56, 2019.

[92] Hang Zhu, Zhihao Bai, Jialin Li, Ellis Michael, Dan RK Ports, Ion Stoica, and Xin Jin. Harmonia: Near-linear scalability for replicated storage with in-network conflict detection. *Proceedings of the VLDB Endowment*, 13(3):376–389, 2019.