Amaral *et al. Journal of Cloud Computing: Advances, Systems and Applications* (2021) 10:22
https://doi.org/10.1186/s13677-021-00238-6

Journal of Cloud Computing:
Advances, Systems and Applications

RESEARCH Open Access

DRMaestro: orchestrating disaggregated resources on virtualized data-centers

Marcelo Amaral[1,2*], Jordà Polo[3], David Carrera[2,3], Nelson Gonzalez[1], Chih-Chieh Yang[1], Alessandro Morari[1], Bruce D'Amora[1], Alaa Youssef[1] and Malgorzata Steinder[1]

Abstract

Modern applications demand resources at an unprecedented level. In this sense, data-centers are required to scale efficiently to cope with such demand. Resource disaggregation has the potential to improve resource-efficiency by allowing the deployment of workloads in more flexible ways. Therefore, the industry is shifting towards disaggregated architectures, which enables new ways to structure hardware resources in data centers. However, determining the best performing resource provisioning is a complicated task. The optimality of resource allocation in a disaggregated data center depends on its topology and the workload collocation. This paper presents *DRMaestro*, a framework to orchestrate disaggregated resources transparently from the applications. *DRMaestro* uses a novel flow-network model to determine the optimal placement in multiple phases while employing best-efforts on preventing workload performance interference. We first evaluate the impact of disaggregation regarding the additional network requirements under higher network load. The results show that for some applications the impact is minimal, but other ones can suffer up to 80% slowdown in the data transfer part. After that, we evaluate *DRMaestro* via a real prototype on Kubernetes and a trace-driven simulation. The results show that *DRMaestro* can reduce the total job makespan with a speedup of up to ≈1.20x and decrease the QoS violation up to ≈2.64x comparing with another orchestrator that does not support resource disaggregation.

Keywords: Orchestration, Resource allocation, Cloud computing, Resources disaggregation, Composable architecture, GPU

Introduction

Traditional data centers consist of monolithic building blocks that tightly integrate a small number of resources (i.e., CPU, memory, storage, and accelerators) for computing tasks. The main flaws of such server-centric architecture are the dearth of resource provisioning flexibility and agility. In particular, the resource allocation within the boundary of the mainboard leads to resource fragmentation and inefficiencies [1–3]. Therefore, industry and research communities have been concentrating efforts on enabling the shift from mainboard-as-a-unit server architecture to a more flexible software-defined block-as-a-unit approach [4–7]. Systems within a disaggregated (or composable) architecture are re-factored so that the subsystems can communicate via a network as a single system; resources are pooled together and provisioned independently [8]. It increases the orchestration flexibility since allocating disaggregated resources enables fine-grained provisioning decisions and efficient sharing of the cluster resources across multiple applications.

The most commonly used resource provisioning and scheduling methods, such as the ones implemented in Mesos [9], Kubernetes [10], YARN [11], Borg [12], and

*Correspondence: marcelo.amaral1@ibm.com
[1]IBM Watson Research Center, Yorktown Heights, US
[2]Department of Computer Architecture, Universitat Politècnica de Catalunya (UPC) - BarcelonaTECH., Barcelona, Spain
Full list of author information is available at the end of the article

Springer Open

© The Author(s). 2021 **Open Access** This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

Firmament [13] resource managers, aims to consolidate as many jobs into servers as possible or load-balance the resource usage across the cluster. However, even though an intelligent orchestrator could assign a workload to a proper compute node, resource fragmentation still exists. This leads to the problem that some valuable resources, such as GPUs or NVMes, cannot be allocated because of the lack of computing power or memory availability within a node.

As for the application placement problem, three aspects need to be considered. First, the current approaches do not consider the possibility to allocate remote resources transparently to the application, the access to resources across multiple nodes must be explicitly managed by the application, which can lead to resource fragmentation in the cluster. Second, a disaggregate resource relies on network communication to transfer data which is often overlooked. More specifically, the applications that do not require network communication, will transparently start to transfer data via network connections to the remote resources. Then, the usage of disaggregated resources may introduce orders of magnitude higher networking bandwidth, additional latency, and memory usage not present when the application is using a directly attached resource within a node. Third, apart from the additional resource usage, an application can transparently become sensitive to the network load, presenting performance variation, as we verify in our experiments, which is often overlooked.

In this work, we conduct a comprehensive analysis of the application placement problem considering the possibility to allocate disaggregated resources. Nonetheless, determining the best-performing resource provisioning and job scheduling, in most of its relevant forms, is known to be NP-hard. In addition to that, the problem is even more acute considering the possibility to allocate disaggregated resources. Then, we present *DRMaestro*, a flow-network-based approach that solves the problem in multiple phases to minimize the complexity and allow the introduction of placement preferences. The system orchestrates disaggregated resources on cloud environments preventing applications' Service Level Objectives (SLOs) violations when using disaggregated resources while maximizing the overall system utilization. We consider the measured network communication dependencies from an application using disaggregate resources, the network load, and server-side resource constraints. The main contributions are summarized as follows:

- The **first contribution** is the investigation of to what extent the introduced network sensitivity and requirements by using disaggregated resources can affect the application's performance.
- The **second contribution** is the proposal of a novel framework to orchestrate and transparently allocate disaggregated resources, which we call *DRMaestro*.
- The **third contribution** is a novel flow-network model to determine the placement of jobs in a set of machines considering the possibility to allocate disaggregated resources.
- The **fourth contribution** is the creation of a network-aware policy to prevent co-scheduled jobs that become network sensitive by using disaggregated resources with other network-intensive jobs.

The rest of the paper is structured as follows. "Problem formulation" section further describes the resource provisioning problem when considering disaggregated resources. "Related work" section discusses the key differences between our work and the related works. "DRMaestro architecture and implementation" section presents the architecture and implementation of *DRMaestro*. "Experimental evaluation" section provides the evaluation of our framework in a real prototype and "Trace-driven simulation evaluation" section in a trace-driven simulation. Finally, "Conclusion" section presents the conclusions.

Problem formulation

To tackle the placement problem (i.e., the knapsack problem) of mapping a set of jobs with a set of machines, a promising approach that has been widely investigated and efficiently solved is to use graph isomorphism [14, 15]. By modeling the placement problem in a bipartite graph, the problem can be modeled as a flow-network problem which can be efficiently solved with one of the de facto flow-network-based existing algorithms to find the optimal match by minimizing the costs [13, 16, 17].

The placement problem

The main challenge is how to model the resource disaggregation as a flow-network problem. . To illustrate that, let's consider the scenario where the GPU is disaggregated. In this case, the model should consider that the traffic flow rests on the behavioral assumption that the job's performance depends on any prevailing system flow, which might introduce an inherent delay. Note that, this problem should attain an equilibrium because the delay that one job incurs depends on the flow of other jobs, and all jobs are simultaneously choosing their best path. Additionally, the model also should consider that a machine has limited capacity, and to satisfy the job's demand; it might have resources allocated beyond the available spares at a larger cost, which is allocating disaggregated resources. Moreover, for performance reasons, it is also desired to have mechanisms to define placement preference (i.e., affinity to local resources).

A formal statement

Let $G = (N, A)$ be a directed bipartite network graph whose arcs carry *flow* from the source to a sink node, in which each arc (i, j) and $(i, j) \in A$ has a nonnegative capacity x_{ij} and a cost c_{ij} associated with every arc $(i, j) \in A$. Each node $i \in$ N has a number b_i that indicates its supply or demand depending the node type. If it is a job's task, then $b_i > 0$, else if the node is a machine, $b_i < 0$. Additionally, it is always assumed that there are no loops, the flow-network is finite, and the solution is feasible, that is, there must be enough resources to place all jobs.

The goal is to find a flow f that minimizes Eq. 1 while respecting the *feasibility constraints* in Eq. 2 and the capacity in Eqs. 3 and 4, as follows:

$$Minimize \sum_{(i,j)\in A} c_{ij} f_{ij} \tag{1}$$

subject to

$$\sum_{j:(i,j)\in A} f_{ij} - \sum_{j:(j,i)\in A} f_{ji} = b_i, \quad \forall i \in N \tag{2}$$

$$0 \leq f_{ij} \leq x_{ij}, \quad \forall (i,j) \in A \tag{3}$$

$$\sum_{i\in N} b_{ir} \leq 0 \quad \forall r \in R \tag{4}$$

$$and \quad b_i = \sum_{r\in R} \frac{b_{ir}}{max_b_{ir}}, \quad \forall i \in A \tag{5}$$

A *"feasible flow"* assigns a non-negative integer flow f_{ij} to each edge $\in A$ up to the maximum capacity x_{ij}. Additionally, when more than one resource is represented, Eq. 4 will be repeated for each resource $r \in$ R. Also, each flow, demand, or capacity will have the values normalized. For example, let's consider the supply b_{ir}, each resource $r \in$ R will be normalized with the maximum existing supplier in the cluster of the given resource, as shown in Eq. 5.

In this work, we define the cost c_{ij} associated with every arc as the complement of the utility Ut_i multiplied by the placement preference P_i. The Utility Function is inspired in the work of [18] and is presented in Eq. 6. Where τ_c represents an average completion time goal and the $\varphi_{r,l}$ represents the expected completion time of the job with the current resource r and load l.

$$Ut(\varphi_{r,l}) = \begin{cases} \frac{\tau_c - \varphi_{r,l}}{\tau_c} & \text{if} \varphi_{r,l} \leq \tau_c \\ \left(\frac{\tau_c - \varphi_{r,l}}{\tau_c}\right) . \left(\frac{10 - z_c}{9}\right) & \text{if} \varphi_{r,l} \geq \tau_c \end{cases} \tag{6}$$

The utility value is bounded by one and is always greater or equal to zero as long as the placement meets the goal. That is, $Ut(\varphi_{r,l}) \in [1, 0]$. If the goal is violated, the utility function yields negative values scaling with the magnitude of z_c loss, according to the importance of the sharing-induced performance interference, where the $z_c = 1$ has the highest importance.

The preference P_i is determined by the list of preferred machines from each job's task i, which can be determined by affinity, resource constraints (i.e., a specific type of resource), or by different phases in our scheduling process (since our scheduling approach define preference in different phases).

Apart from the communication cost in the data center, the cost of external network load interference in the application performance is a significant factor to be considered, as we detail in our experiments in "Experiment 1 - performance analysis of network-interference from using GPU disaggregation" section.

To be specific, we assume that the expected completion time $\varphi_{r,l}$ of a job might suffer a slowdown when using resource disaggregation accordingly to the network load l. Therefore, the expected completion time $\varphi_{r,l}$ must be multiplied by a sensitivity factor $sf * l$ concerning the network load l. The sensitivity factor $sf * l$ might be experimentally or statistically calculated.

Therefore, we define two policies: the first policy (i) do not consider the network interference in the evaluation of the expected completion time $\varphi_{r,l}$ of the job; and the second policy (ii) perform a different calculation for $\varphi_{r,l} * sf$ by considering the level of interference from the network load that the application can suffer when accessing disaggregated resources.

Flow-network-based min-cost algorithms

A flow-network-based resource provisioning system can use any min-cost algorithm, but some algorithms are better suited for this problem as further described by Gog et al. [13]. The simplest min-cost algorithm is the *cycle canceling* [19] that computes a max-flow solution and then performs multiple iterations augmenting flow along with negative-cost directed cycles in the residual capacity of the network. The algorithm finishes with an optimal solution once no negative-cost cycles exist. On the other hand, different from *cycle canceling*, the *successive shortest path* [16] algorithm attempts to keep the costs reduced in all the steps to try to achieve feasibility, repeatedly selecting the shortest paths. The algorithm *cost scaling* [17, 20, 21] iterates multiple times attempting to reduce the cost while maintaining feasibility and relies on a relaxed complementary slackness condition called ϵ-optimality.

The worst-case complexity of the *cycle canceling*, *successive shortest path* and *cost scaling* algorithms are compared in Table 1. Where, in the context of the problem in this work, K is the number of disaggregated resources plus one, N is the number of nodes, M the number of arcs, C is the largest arc cost, and U the largest arc capacity. In our problem, $M > N > C > U$. It is worth mentioning that although the worst-case complexities suggest that *successive shortest path* performs better, Gog et al. [13]

Table 1 Worst-case time complexity for min-cost algorithms

Algorithm	Worst-case complexity
Cycle canceling	$O(KNM^2CU)$
Successive shortest path	$O(KN^2Mlog(NC))$
Cost Scaling	$O(KN^2Ulog(N))$

showed that *cost scaling* scales better since in practice the algorithms rarely reach the worst-case.

Related work

In this section, we review related work and the state of the art for resource disaggregation middleware to clarify our contributions.

Orchestration

Orchestration based on flow network

Isard et al. [22] introduce a powerful and flexible new framework for scheduling concurrent distributed jobs, enforcing data locality, fairness, and being a starvation-free method. The scheduling problem is mapped to a graph data structure, where the edge weights and capacities encode the competing demands. They called their framework Quincy and evaluated it with a cluster containing a few hundred computers. They showed an example of how efficient scheduling problems on flow-based-network models can be, which has inspired our work. The experimental results showed that Quincy provides better fairness, while substantially improving the data locality when compared with default greedy approaches.

Gog et al. [13] proposed Firmament by extending the work of [22] via implementing different flow-network-based algorithms to solve their proposed model, improving the scheduling latency. The Gog et al. results showed that they improved the placement latency by 20x over Quincy for an experiment with 12k machines. Additionally, they showed that Firmament's 99th percentile response time is 3.4x better than the SwarmKit [23] and Kubernetes [10] ones, and 6.2x better than Sparrow [24] response time.

While these previous works have done a great job showing that flow-network based orchestration outperforms greedy approaches, their proposed flow-network models have a rigid topology of the resources, preventing the allocations of resource disaggregation as they are. More specifically, a job can only be placed within a node and a node cannot access resources of other nodes, as it happens with resource disaggregation. Additionally, their model only considers CPU and memory, they do not define hot-plugged resources such as GPUs and NVMes. Quincy does not include network load in its model. Finally, although Firmament includes network load as a resource constraint in its model, it does not include either network cost regarding resource disaggregation or the application sensitivity to the external network load. The model that our work uses includes the possible network interference that disaggregated resources can suffer due to introduce additional network requirements.

Orchestration considering the network traffic

Meng et al. [25] studied the effect of network traffic patterns between VMs to optimize VM migration on host machines. The study was performed in operational data centers, and the results showed the distribution of VMs in data centers was very uneven. They found that VMs with a relatively high traffic rate tend to have high network usage, and VMs with a low rate constantly exhibit a low rate. Then, they formulated a traffic-aware VM mapping problem and proposed an approximation algorithm to solve this problem. The experiments compare the impact of the traffic patterns and the network architectures and the results showed that the proposed algorithm increases the performance and greatly decreased the communication costs of a network.

Zhang et. al. [26] conducted a study of the network-aware VM migration problem, considering not only the underlying network topology of a data center, the server-side resource, the communication dependencies among VMs in the application tier, but also the data traffic of VM migration from one host to another. They applied two heuristic algorithms such as genetic algorithm (GA) and artificial bee colony (ABC) to solve the optimization problem of minimizing the communication and the VM migration cost. Where the communication cost is defined regarding the distance between the VM and the destination physical machine, i.e., latency, delay, or the number of hops between servers, and the VM migration cost is the distance times the VM size. The results showed that the ABC algorithm scales better than GA regarding the problem size.

These works share the similarity with our work regarding the consideration of the network load and the communication cost in the optimization problem. However, they do not consider sensitivity (i.e, the slowdown factor) respecting the network load as we do in our optimization problem. Additionally, their optimization problem cannot be directly applied to solve the placement of disaggregated resources since their model only allocates VMs within the boundary of a server, and a server cannot access resources on other servers.

Orchestration considering disaggregation

Iserte et al. [27] provide an extension in the cluster resource manager SLURM [28] by including a new type of resource called "rgpu", to obtain access from any application to any GPU in the cluster using the library rCUDA [7] to access the remote GPU. They extended the SLURM

job submission with new parameters so that the user can request to allocate remote GPUs. They have also provided a naïve scheduler that first attempts to allocate local GPUs, but if it is not possible it randomly selects other available GPUs in the cluster. Iserte et al. [29] have done another work that extends OpenStack [30] to support remote GPUs using rCUDA. They extended OpenStack to allow the user to allocate local or disaggregated GPUs from a pool of GPUs. The main limitations of these works are that the user must explicitly define the number of remote GPUs, i.e., rgpus, which is not transparent to the user. Additionally, in this work, the scheduler randomly selects GPUs without considering the network and other affinity preferences.

Lama et al. [31] proposes a power-aware virtual OpenCL (pVOCL) framework that controls the peak power consumption of the underlying server systems and dynamic scheduling of GPU resources for online power management in virtualized GPU environments. pVOCL uses the VOCL [32] middleware to virtualize GPUs for applications using OpenCL. pVOCL framework uses a power-aware dynamic placement and migration approach. The results showed an improvement of up to 29% in energy efficiency by turning off compute nodes when they are idling. The main goal of this work however is to use GPU virtualization for power management in GPU-enabled servers, and they do not leverage the benefit of the possibility to remotely access virtualized GPUs.

Resource disaggregation middleware

Previously, similar works to the HFCUDA library (detailed in "HFCUDA" section) have been proposed, implemented, and evaluated to intercepts CUDA calls as a software middleware. Oikawa et al. [33] proposed, DS-CUDA, implementing a redundant mechanism to replicate the job execution in other GPUs to minimize the impact of errors, which they claimed to be frequent in cloud environments. For experimental evaluation, they implemented DS-CUDA to intercepts calls from the CUDA toolkit 4.1. Unfortunately we cannot use DS-CUDA for this work because the CUDA toolkit is too old.

Liang and Chang [34] proposed GridCuda: a software development toolkit to develop CUDA programs and to aggregate GPU resources in computational grids for the execution of CUDA programs. The runtime system of GridCuda can automatically cooperate with the resource brokers of computational grids to discover and allocate GPUs available for jobs according to the user's resource requirements. The implementation supports the CUDA toolkit 4.0 and demands modifications in the applications' code to be implemented using their toolkit. Similarly to DS-CUDA, we cannot use GridCuda in this work because the CUDA toolkit is too old.

Merritt et al. [35] present Shadowfax to virtualize GPUs over Xen hypervisor 3.2.1, supporting CUDA toolkit 1.1.

The rCUDA [7] is a middleware that traps the CUDA call APIs from an application and forwards the calls to a remote server to process the request in the remote GPU. rCUDA implements most of the CUDA API and also provides support for other libraries such as cuFF, cuBLAS, and cuSPARSE. rCUDA also supports network transmission with RDMA over InfiniBand or Ethernet networks. The results showed that the middleware overhead impact varies by GPU architecture due to computing power. Additionally, the results showed that for some specific scenarios, the middleware achieves better performance than the native CUDA calls dues to the usage of pinned memory. At the time of this writing, rCUDA is the most popular middleware for GPU virtualization. It supports the CUDA toolkit up to version 9.1, can be used by both VMs and containers, and has low communication overhead when using InfiniBand networks. Unlike other works rCuda requires users to allocate available GPUs by themselves for the execution of their CUDA programs and the code is proprietary, making it hard to run on top of POWER8 architectures, and preventing us from understanding the underlying behavior and as well as implementing missing features.

Finally, Xiao et al. [32] presented the VOLC, a middleware to virtualize GPU for OpenCL calls. The main limitation of this work is that although it provides GPU virtualization it does not provides support for remote access.

Therefore, in this work, we used the HFCUDA library, our in-house middleware for GPU virtualization.

DRMaestro architecture and implementation

Our proposed framework is a dynamic, loop-based controller that can manage resources from both server-centric and/or disaggregated architectures. *DRMaestro's* architecture is depicted in Fig. 1, and the key components are as follows. (i) A *Resource Manager* (*RM*) module that interacts with the end-user receiving the submission of jobs and enforcing placement decisions. (ii) A *Placement Decision Maker* (*PDM*) module responsible for updating the flow-network models and making placement decisions. Finally, (iii) a *Flow-Network Solver* (*FNS*) module that executes a min-cost algorithm to solve the flow-network models.

Placement decision maker (PDM) module

DRMaestro's PDM module determines the optimal placement for a given set of jobs in an online scheduling approach. As detailed in the Alg. 1 and illustrated in Fig. 1, it first receives the cluster status (the snapshot of the current resource usage), and then, based on this status it gathers a list of jobs from the waiting job queue. Only

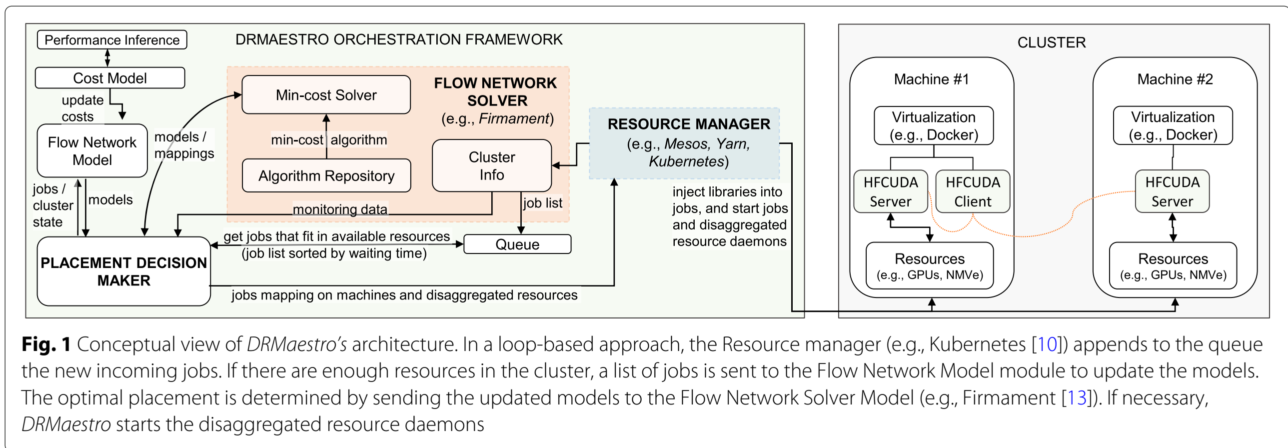


Fig. 1 Conceptual view of *DRMaestro's* architecture. In a loop-based approach, the Resource manager (e.g., Kubernetes [10]) appends to the queue the new incoming jobs. If there are enough resources in the cluster, a list of jobs is sent to the Flow Network Model module to update the models. The optimal placement is determined by sending the updated models to the Flow Network Solver Model (e.g., Firmament [13]). If necessary, *DRMaestro* starts the disaggregated resource daemons

the jobs that can fit in the currently available resources are gathered from the queue, and to prevent starvation, the queue is sorted by the total wait time of each waiting job. This data is then used to update the models within the *Flow Network Model* (*FNM*) module and forward the updated models to the *FNS* module, which will return the mapping results of each model.

Each requested resource that is possible to be disaggregated is treated individually, i.e., if different resources are disaggregated, they are allocated in different phases. This is because carefully separating the optimization problem into independent sub-problems is effective at reducing the complexity of finding the optimal placement. We further detail our flow-network models in "Flow-network model (FNM) module" section.

Therefore, *FNM* maintains multiple flow-network models that are solved sequentially, as illustrated in Fig. 2. The first flow-network model will represent the placement of server-centric resources. The other following models will represent the placement of the disaggregated resources. After the first model is solved, the other models (if any) update the machine preferences based on the decisions made with the previous models, as shown in Algo. 2, when calling the function `UpdateModelPreferences()`.

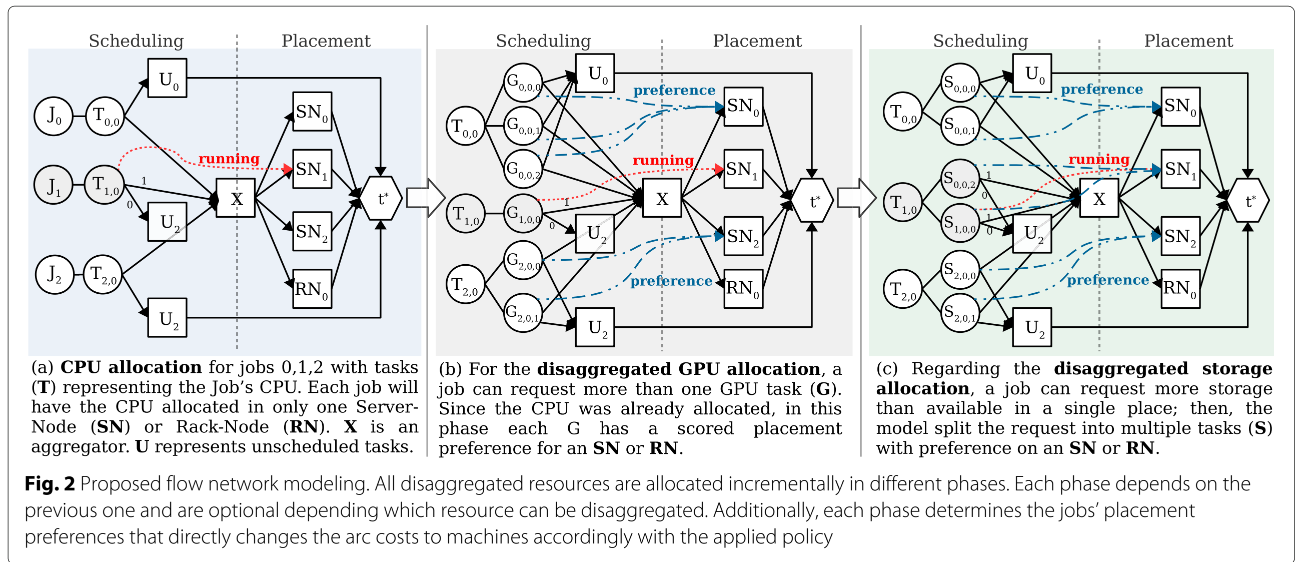
After the FNS module returns all the solved models, the final placement decision of each job is extracted based on the analysis of the optimal flow from all solved models, as shown in Algo. 2 when calling the `GetMapping()` function. By default, we gang schedule all resources (but our method can be easily extended to allocate resources incrementally). Hence, in our proposal, if any requested resource (i.e., job's "sub-task" in our model) is left unscheduled in any of the solved models, the job will not be scheduled. If the job is not scheduled it will be added back to the queue with an increment to its total wait time counter. After that, *PDM* sends the placement decisions to *RM* to configure and start the jobs.

Algorithm 1 DRMaestro Placement Process

```
function PLACEMENT()
    cs = get_cluster_state()
    jobs = queue.pop_jobs(cs)  //only jobs that fit in the resources
    update_flow_network_models(jobs, cs)
    mp = MAPPING(jobs)         //Algorithm 2
    for job in m do
        if job.state == scheduled then
            for r in disaggregated_resources do
                for {task, m} in m[job][r] do
                    bind_to_node(job, m)
                end for
            end for
            wait_all_hfcuda_severs_start()
            m = mp[job][not_disaggr_group][0]
            inject_info(job.manifest, mp)
            bind_to_node(job.manifest, m)
        else
            job.waiting_counter++
            queue.push(job)
        end if
    end for
end function
```

Flow-network model (FNM) module

Although, the placement problem can be modeled as a flow-network model in many different ways, for elegance and clarity we have attempted to create the simplest possible model. We created our models inspired by the models applied in [13, 22], but, unlike the previous ones, we encompass the system with disaggregated architectures. In our flow-network model, the allocation of both server-centric and disaggregated resources is incrementally solved in different phases, using a divide and conquer approach. The approach is incremental as it decides to place each resource at a time without modifying the allocation plan from previous decisions in the other phases.



(a) **CPU allocation** for jobs 0,1,2 with tasks (**T**) representing the Job's CPU. Each job will have the CPU allocated in only one Server-Node (**SN**) or Rack-Node (**RN**). **X** is an aggregator. **U** represents unscheduled tasks.

(b) For the **disaggregated GPU allocation**, a job can request more than one GPU task (**G**). Since the CPU was already allocated, in this phase each G has a scored placement preference for an **SN** or **RN**.

(c) Regarding the **disaggregated storage allocation**, a job can request more storage than available in a single place; then, the model split the request into multiple tasks (**S**) with preference on an **SN** or **RN**.

Fig. 2 Proposed flow network modeling. All disaggregated resources are allocated incrementally in different phases. Each phase depends on the previous one and are optional depending which resource can be disaggregated. Additionally, each phase determines the jobs' placement preferences that directly changes the arc costs to machines accordingly with the applied policy

We have carefully defined the problem in multiple phases to enable the possibility to express placement preferences (i.e., define affinity). For instance, when a job is placed in machine #1 (having the server-centric resources allocated in machine #1), in our model, it will be possible to define preferences to allocate the resources that can be disaggregated in the local machine or a remote machine, depending on the allocation policy.

This multi-phase approach requires the definition of multiple flow-network models, as illustrated in Fig. 2. In this figure, in each model, the node U_* determines if the job will be activated or left unscheduled. Note that, the cost to leave a task unscheduled must be initially higher than placing it. Also, in this figure, the node X stands for an aggregator to reduce the number of possible arcs from jobs to machines, and the t^* is the sink node. The SN represents a server-node with software-defined resource disaggregation, i.e., using a software-based middleware to disaggregate the resource. The RN represents a rack-node with hardware-defined resource disaggregation. To apply the scheduling policy, the arc weights and capacity are properly determined.

Algorithm 2 DRMaestro Flow-Network Mapping

```
function MAPPING(incoming_jobs)
    prev_model = nil;
    map_m = [[]]
    for (group in resource_groups) do
        model = flow_models[group]
        if (prev_model not empty) then
            UpdatePref(model, prev_model)
        end if
        map_m[group] = GetMappings(model)
        prev_model = map_m[group]
    end for
    for (job in incoming_jobs) do
        job.state = scheduled
        for (t in job[group].tasks()) do
            m = map_m[group][t]
            if (t not in map_m[group]) then
                job.state = unscheduled
            else
                map[job][group] = {t, m}
            end if
        end for
    end for
end function
```

Fig. 2a illustrates the first phase, where the job will be placed and allocated server-centric resources. In this figure, the $T_{k,m}$ represents each task of job J_k with the group of the server-centric resource, for instance, CPU and memory. In a more intricate description, this phase is a snapshot of the cluster, showing the available machines (SN and RN), and also the running and incoming jobs (i.e., the jobs picked from the waiting list). In this phase, the placement of each task $T_{k,m}$ will determine the placement preferences in the other phases.

Fig. 2b illustrates the second phase and will exist only if a disaggregated resource is required to be allocated. This phase is incremental to the previous phase and will update the graph of the flow-network model to represent the allocation of a group of a type of resource. For example, Fig. 2b creates new "sub-tasks" $G_{k,m,z}$ for each z GPUs that each task $T_{k,m}$ is requesting. It allows the allocation of each GPU independently and possibly going to different machines, i.e., allocating disaggregated GPUs. Additionally, the mapping result from the first phase is translated here to preference arcs for each "sub-tasks", which will dictate the costs. To illustrate that, consider the case where Job J_0 best performs when its CPU and GPU are on the

same machine, and the applied scheduling policy prioritizes locality. If in the first phase, task $T_{0,0}$ is placed in SN_0, in the second phase, the sub-tasks $G_{0,0,*}$ will have a preference for SN_0.

Figure 2c illustrates the third phase and stands for the allocation of another disaggregated resource. Note that, for each type of resource that can be disaggregated a new phase will be created. In this example, we are showing a phase to allocate disaggregated GPU and a phase to allocate Non-Volatile Memory Express (NVMe) storage. However, in the case that there are more resource groups to be provisioned, the method keeps transforming the graph and translating the placements into preference arcs. Thence, in Fig. 2c, the "sub-tasks" $S_{k,q}$ will allocate the disaggregated NVMes. Note that the level of parallelism is defined by the number of sub-tasks. For example, in the case that a task has only two "sub-tasks" $S_{k,q}$, this task can access up to two machines at most, but this process is transparent for the applications.

Cost model (CM) module

The *CM* module is responsible to determine the cost of the edges in the flow-network model according to a given policy. As detailed in "A formal statement" section, in this work, the cost c_{ij} is defined as the complement of the utility Ut_i, in Eq. 6, times the preference P_i. The preference P_i is determined by the list of preferred machines from each job's task i, which can be determined by affinity, resource constraints (i.e., a specific type of resource), or by different phases in our scheduling process (since our scheduling approach defines preference in different phases). The cost c_{ij} for all edges, except the one between the job i and the unscheduled state U_i, is determined by Eq. 7. On the other hand, cost c_{ij} of the edges that point to an unscheduled state U_i is determined by Eq. 8, which, to avoid starvation, has an incremental counter W_i that increases every time that the job i is left unscheduled after each scheduler iteration.

Therefore, in our model, the cost c_{ij} for all edges, except the one between the job i and the unscheduled state U_i, is determined by Eq. 7. On the other hand, cost c_{ij} of the edges that point to an unscheduled state U_i is determined by Eq. 8, which, to avoid starvation, has an incremental counter W_i that increases every time that the job i is left unscheduled after each scheduler iteration.

$$c_{ij} = (1 - Ut(\varphi_{r,l}))P_i \tag{7}$$

$$c_{ij} = (1 - Ut(\varphi_{r,l}))P_i + W_i \tag{8}$$

The network-aware (NA) policy

The *NA* policy updates the costs to place a task or leaves it unscheduled based on the network load and the sensitivity of the task to such a load. Where the sensitivity is given by a job profile created from experimentation using historical data (i.e., offline profiling), as shown in "Experiment 1 - performance analysis of network-interference from using GPU disaggregation" section. The model will contain the execution time of the best-performing scenarios for the application accessing disaggregated resources, along with the execution time in another scenario where an artificial load was introduced in the network. Then, with this policy, the cost is calculated considering the expected slowdown that this application might suffer given the current load in the network. Additionally, in this policy, for each scheduling decision, if the placement does not satisfy the task's SLO, the placement will be postponed. Where the SLO is defined as the task's expected completion time.

The inference engine

This module is implemented with a simple approach that keeps the information (i.e., the offline profiling) of the average job's completion time in each cluster state. While we keep it simple since in practice a complicated inference engine is not typically applied because of the introducing delay in the placement, we plan to extend *DRMaestro* to also use a more sophisticated engine such as using decision tree [36, 37] or statistical clustering [38–40]. But for now, our classification is based on the job's Docker image metadata and an off-line model created with both historical data from experimentation. In the absence of a model, the application is classified as unknown using a neutral cost value (e.g., as 1).

Flow-network solver (FNS)

The *FNS* module continuously consumes telemetry data to update the flow-network models and the list of incoming jobs to add to the queue of waiting jobs. Additionally, this module is responsible to solve the flow-network models using a min-cost algorithm and replies to the *PDM* module the solved the models. Most of the telemetry cluster data is given by Kubernetes Heapster.

Resource manager (RM) module

The *RM* module is responsible for monitoring the cluster, enforce the placement decisions, and trigger the orchestration with a set of jobs L when $L > 0$. Each job, on its arrival, is put into a queue (sorted by the waiting time), and in a loop-based approach, the resource manager sends a set of jobs to the orchestrator to find the optimal placement for them. After that, it maps and runs each job's task within its target machine. When it is necessary, before starting any job, the resource disaggregation daemons will be started.

In our implementation, the *RM* module is built on Kubernetes [10] since it one of the most widely applied resources managers in today's cloud data centers. For

connecting the *FNS* module with the *RM* module, we use the Kubernetes support for an add-on scheduler. The interface between these modules was based on the Kubernetes's scheduler add-on Poseidon [41]. We extended this add-on scheduler to interpret the new format of the placement decisions that *DRMaestro* supplies and enforce the setup of daemons responsible to enable the resource disaggregation. Additionally, it also injects into the job the required libraries and information to access the disaggregated resource before running the job.

Enabling GPU disaggregation

The disaggregated resources can be managed through a centralized or a distributed model and accessed by application programming models through hardware, hypervisor/operating system, or middleware based approaches as discussed in [42]. In this study, we focus on the middleware-based approach.

GPU disaggregation into DRMaestro

Disaggregation is implemented via a simple approach, with no change in the Kubernetes architecture. *DRMaestro* dynamically creates new disaggregated daemon servers as Kubernetes jobs and injects into the job the required libraries and environment variables to set up the channel between the job and the disaggregated GPU. Then, for each application submitted into Kubernetes that will need remotely access the disaggregated GPUs, we create HFCUDA servers in the nodes that have access to the GPUs. Where HFCUDA is our in-house GPU Disaggregation middleware. Therefore, as a Kubernetes job, each HFCUDA server is created on the node as a Pod with a container accessing the local (server-centric) GPUs. Consequently, the application is created with the HFCUDA client, which will access the servers to access the GPUs remotely. Note that the process to create the HFCUDA client and server is transparent from the user, the framework receives a regular Kubernetes manifest and then injects the necessary libraries, environment variable, affinity constraints and creates all the Kubernetes pods necessary to run the job with HFCUDA.

HFCUDA

HFCUDA is used since most of the similar libraries target only virtual machines, support only old CUDA versions, do not support POWER8 architectures, or have a proprietary source code (which prevents us from understanding the underlying behavior as well as implementing missing features), as discussed in "Resource disaggregation middleware" section. The HFCUDA runtime is a cross-platform library written in C for portability and performance. The architecture is composed of a "client" library hooking all CUDA calls and offloads API-related data via a network (either Socket/PCIe or RDMA/PCIe) to a "server" that executes the calls into activated GPUs installed at the server node and returns the results. The application does not need to be changed, except that it must be compiled with the cudart library set as shared and be started with `LD_PRELOAD` to load the HFCUDA wrapper. Additionally, the application's shared library must be provided to the server. The server uses the application's shared library to execute the CUDA kernels. That is, it receives the CUDA Kernel name and its parameters from the client and loads and configures the function from the shared library into the GPU. Therefore, the server depends on the application, then, we dynamically create it. This confers security benefits and reduces the exchanged information between the server and the client. Additionally, when the server starts, it initializes CUDA and creates a context in each GPU that it has access to. Thence, since a different server is created for a new application, different applications will never share the same context.

Premises, limitations, and other discussions

Although it is possible to represent in our model the sharing of GPU or job preemption, we did not enable it during our experiments because, at the time of this work, there were still some limitations to enable them. Even though, there were some libraries to help to enable that, such as Multi-Process Service (MPS) [43]. At this writing time, MPS has harsh limitations preventing its usage in Cloud environments. For instance, in pre-Volta NVIDIA GPU architectures (e.g., Pascal and Kepler), the process sharing the GPU did not have isolated address spaces, that is an out-of-range write in a CUDA Kernel could modify the memory state of another process without triggering an error. Moreover, even though, the new NVIDIA Volta architectures implement now fully isolated GPU address spaces, there are still limitations regarding sharing GPU in a cloud environment. For example, a GPU exception generated by any client will be reported to all clients, and a fatal GPU exception triggered by one client will terminate the GPU activity of all clients [43].

Additionally, the network latency and bandwidth can be a limitation when using disaggregated resources. Then, for good performance, the InfiniBand network is desirable. In this work, we used software-based disaggregation technology, but more advanced ones, such as implemented directly with hardware support, will benefit the performance.

Finally, the underlying non-uniform memory topology interconnecting GPUs impacts the GPU communication performance as demonstrated in [44]. Although this is not the focus of this work, the topology cost can be easily added to the flow-network costs; we leave it for future work.

Experimental evaluation

We evaluate *DRMaestro* by comparing it to, *Firmament* [13], the most relevant related flow-network-based scheduling system. It is noteworthy that *Firmament* was extensively evaluated showing that it always performs better than greedy approaches. The fundamental difference between *Firmament* and *DRMaestro* is that *Firmament* is built to manage only server-centric systems; it does do not allocate disaggregated resources as *DRMaestro* does.

Next, we present the main goals of the experiments in this section.

The first goal (I) *is to investigate to what extent the introduced network sensitivity/requirements - by using disaggregated resources - can affect the application's performance*, as will be elaborated in "Experiment 1 - performance analysis of network-interference from using GPU disaggregation" section. The motivation is although resource disaggregation confers many advantages, it introduces the challenge to deal with added network requirements. An application that does not require network communication will transparently start to rely on transferring data via network connections.

The second goal (II) *is to investigate to what extent the resource-efficiency of the cluster can be leveraged by using GPU disaggregation*, as will be presented in "Experiment 2 - resource-efficiency leveraged by GPU disaggregation" section. The motivation is that one of the main advantages of resource disaggregation is to enable the allocation of spare resources that could not be allocated in normal situations. That is, a machine that does not have enough available computing resources cannot assign its idle resource for waiting jobs. Therefore, we create all the experiments within a scenario where some machines have part of their computing resources already allocated. More specifically, we warm up the cluster by allocating some long-running jobs before starting the experiments.

The third goal (III) *is to investigate to what extent the QoS violations of the jobs are mitigated by using our Network-Aware (NA) policy*, as will be described in "Experiment 3 - resource-efficiency leveraged by mitigating QoS violations" section. The motivation is that since disaggregated resources may introduce network sensitivity into the job, we hope to mitigate QoS violations by preventing collocation with network-intensive jobs.

Testing environment

All experiments are conducted on 4 IBM® Firestone POWER8® servers. Each Firestone is configured with 512GB DRAM, 2 x POWER8® 3.4Ghz CPU (10 cores per CPU and 8 threads per core), 4 x NVIDIA® K80 GPUs, 1 x dual-ported Mellanox® EDR IB, and 4 x 1/10GbE Broadcom Ethernet. The InfiniBand network provides connectivity between distributed applications tasks and access to the IBM® Spectrum Scale storage servers running GPFS file system. The 1/10 GbE networks are used for three purposes: (1) connectivity via a private network to a management controller node that can be used to provision software via the xCAT cluster management software (2) connectivity to the BMC (Baseboard Management Controller) that monitors the physical state of the computer and can be accessed to remotely power on/off a server (3) connectivity to the internet. The Spectrum Scale storage servers provide 1.2 PB of storage accessed via the General Parallel File System (GPFS). Each Firestone server is provisioned with Ubuntu 16.04 with the 4.4.0-31-generic kernel. A development stack including GNU toolchain, IBM® XL compilers, CUDA 9.1 with driver version 387.36, and Kubernetes version 1.10 is deployed across the cluster. In all experiments, the applications were executed in Docker containers.

The benchmark

The workload is composed of applications from the Rodinia Benchmark Suite for Heterogeneous Computing [45]. We selected only the applications that most differ from each other concerning the resource usage pattern, and we configured them as follows:

- **B+tree** performs queries on large n-ary trees and is configured with an input of 10k elements.
- **Back Propagation** is a pattern recognition application that implements a single training step of a neural network. But we extended it to have multiple iterations, and we configured it with 100k input elements and 10k iterations.
- **Gaussian** implements the Gaussian elimination solver for a system of linear equations and is configured with a generated matrix of 6k elements.
- **HotSpot** is a transient thermal differential equation solver and is configured with both the thermal and the temperature data with a matrix of 8k elements.
- **LavaMD** performs a step in a larger molecular dynamic simulation; we configured it with 114 cluster nodes (called boxes).
- **Needleman-Wunsch (NW)** is a bioinformatics application that runs a global optimization method for DNA sequence alignment, and we configured it with 40k rows/columns and a penalty of 10.
- **Pathfinder** computes the path on a 2D grid with the smallest total cost and is configured with 10 million columns, 200 rows, and with the pyramid height as 100.
- **Speckle Reducing Anisotropic Diffusion (SRAD)** is an image processing algorithm using anisotropic diffusion to reduce noise in images and configured with 10k iterations, 0.5 of saturation coefficient, 5k rows, and 4k columns.

Experiment 1 - performance analysis of network-interference from using GPU disaggregation

To quantify the performance impact of the introduced network sensitivity/requirements by using disaggregated resources, we collocate Rodinia applications accessing remote GPUs using the HFCUDA library with Iperf [46] on the same machines. However, it is important to highlight that the goal of this experiment is not to show the impact of disaggregation versus non-disaggregation since it depends exclusively on the techniques used to enable disaggregation. More specifically, it depends on how efficiently the software or hardware-based disaggregation is implemented. Instead, the goal of this experiment is to show how sensitive to the network the application can become by using a disaggregated resource.

The performance breakdown of how much time is spent on the Network, GPU (CUDA), and CPU and disk of the Rodinia applications accessing disaggregated GPUs, with and without additional network load, is presented in Fig. 3. The figure illustrates the comparison of how each application behaves running in isolation and running collocated with another CPU-only network-intensive workload. As expected, all jobs increase the time relative to the network transfer of data to a remote GPU when running collocated with another job. For instance, in the case of HotSpot, the increase is from 0.2% to 1% (i.e., 80% higher), while NW goes from 17% to 49% (i.e., 32% higher). Thence, in conclusion, the added network load may affect differently the performance of the applications. Some applications can become more network sensitive because of their communication pattern with the GPU.

Experiment 2 - resource-efficiency leveraged by GPU disaggregation

To quantify the improvement realized by leveraging the resource-efficiency of the cluster using GPU disaggregation, we first evaluate the implemented prototype of *DRMaestro* and *Firmament* within a small cluster composed of four machines. During this experimentation, we generate jobs following a normal distribution to define the job type (i.e., the Rodinia application). A Poisson process with an exponential distribution and an arrival rate of 16 jobs per second is used. Two scenarios are evaluated. The first scenario (i) submits 128 jobs to the Resource Manager (*RM*) module, and the second scenario (ii), to increase the pressure in the system, 512 jobs are submitted. To configure this scenario, we warm up the cluster allocating some CPU-only long-running jobs before starting the experiments. The first jobs only request CPU and Memory and execute for ≈2000s. The warm-up enables us to show the benefit of resource disaggregation when there are spare fragmented resources in the cluster.

The results are shown in Figs. 4 and 5 and summarized in Tables 2 and 3. As expected, in these scenarios, where part of the cluster CPUs have already been allocated, some GPUs cannot be locally allocated because of the shortage of computing resources.

Figure 4 shows that *Firmament* performs worse than *DRMaestro* regarding resource-efficiency when submitting 128 jobs in a system with 4 machines. During the execution of this experiment, *Firmament* could only allocate 8 GPUs at most. On the contrary, when GPU disaggregation is enabled, by using *DRMaestro*, all the GPUs are allocated. The small interval where some GPUs are not being

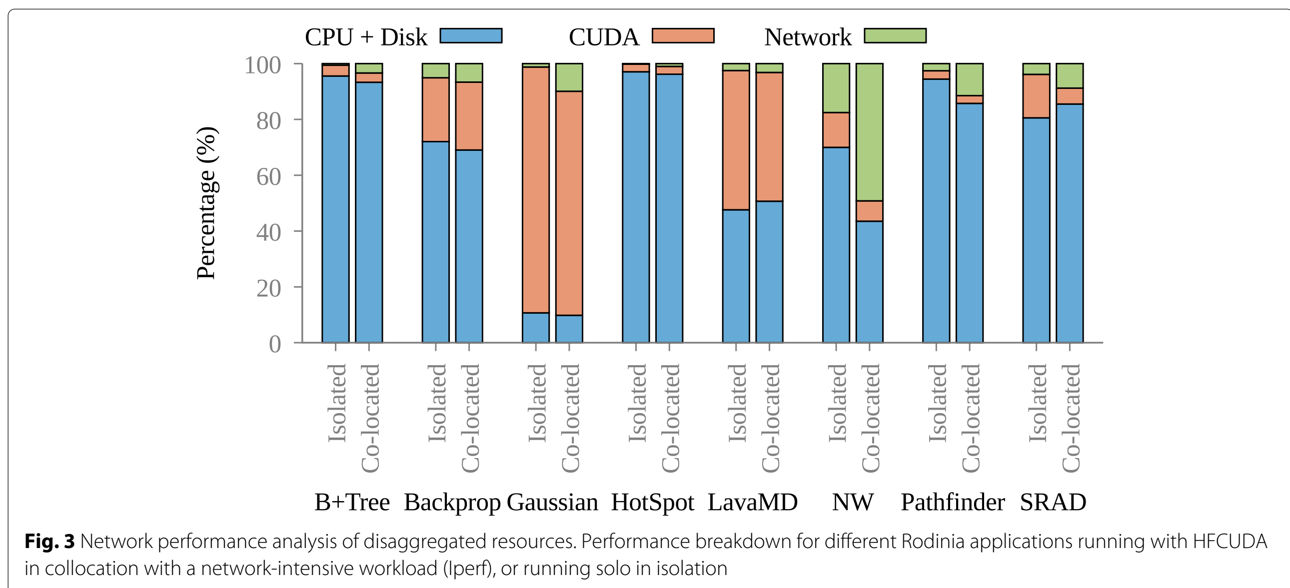


Fig. 3 Network performance analysis of disaggregated resources. Performance breakdown for different Rodinia applications running with HFCUDA in collocation with a network-intensive workload (Iperf), or running solo in isolation

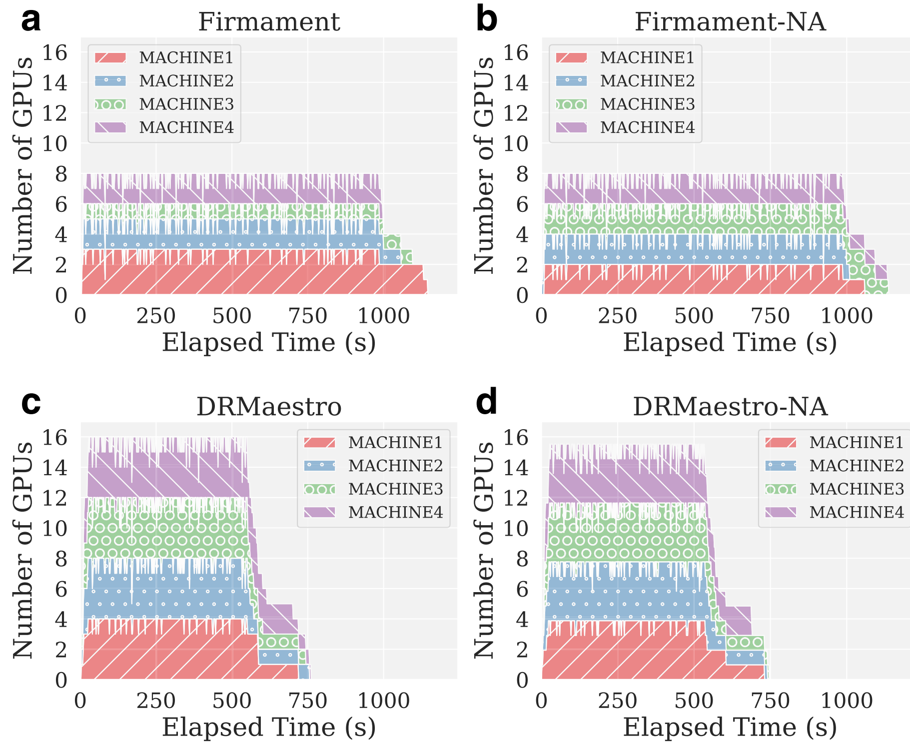


Fig. 4 Experiment results of orchestration with *DRMaestro*. Orchestration performed with *DRMaestro* with the implemented prototype in a cluster with 4 machines and 128 jobs

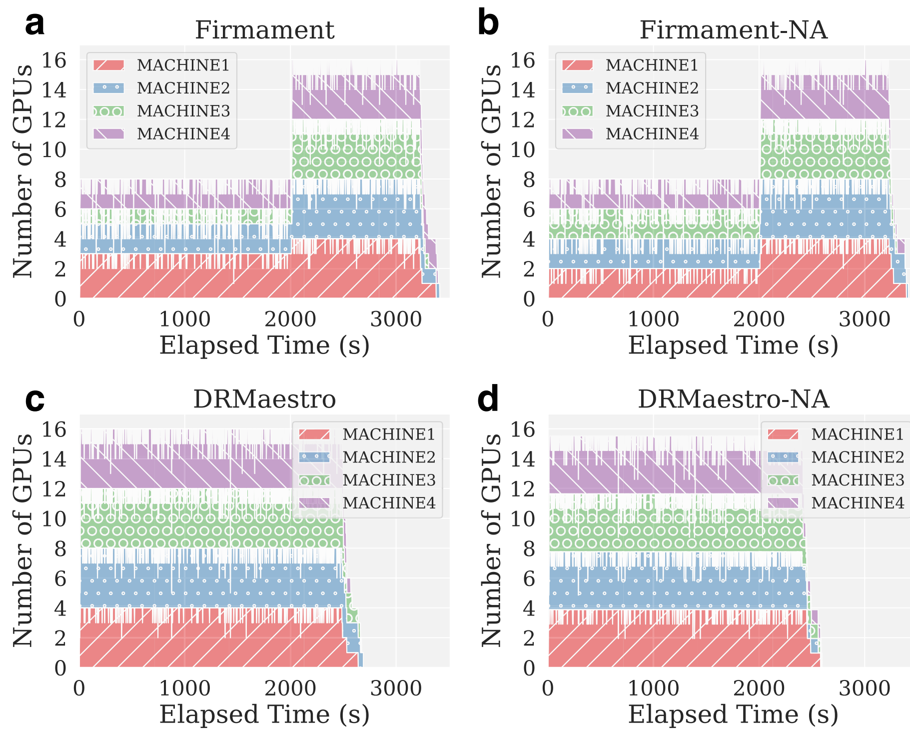


Fig. 5 Experiment results of orchestration with *DRMaestro*. Orchestration performed with *DRMaestro* with the implemented prototype in a cluster with 4 machines and 512 jobs

Table 2 Total makespan of a set of jobs

			Execution time		
# Job	# Node	Net Policy	Fir.	DR.	Ratio Fir.\DR.
128	4	False	1145.16s	760.53s	→ ≈1.50X
128	4	True	-	744.61s	
				↓	↘
Network Policy Improvement				≈1.02X	≈1.54
512	4	False	3413.27s	2690.08s	→ ≈1.27X
512	4	True	-	2593.50s	
				↓	↘
Network Policy Improvement				≈1.04X	≈1.32

Where Fir. is Firmament and DR. is DRMaestro

used happens because of the interval between finishing and starting the jobs. Besides, to improve the resource utilization of the cluster, *DRMaestro* also decreases the total makespan with a speedup of up to ≈1.50X, as shown in Table 2. This is explained by the fact that *DRMaestro* can place more waiting jobs by allocating spare idle GPUs. On the other hand, since resource disaggregation introduces additional network requirements/sensitivity into some jobs, the number of jobs with violated QoS (the number of jobs that suffered slowdown) is ≈9% higher when using *DRMaestro* without the *NA* policy, as detailed in Table 3. The results with the *NA* policy enabled is discussed in "Experiment 3 - resource-efficiency leveraged by mitigating QoS violations" section.

When evaluating a more intensive scenario, submitting 512 jobs instead of only 128, *DRMaestro* still performs better than *Firmament* regarding maximizing the resource utilization and decreasing the makespan. The results are illustrated in Fig. 5. In this scenario, submitting 512 jobs, after the ≈2000s, *Firmament* starts to allocate almost all the GPUs in the cluster. This behavior is explained because the long-running jobs that started during the warm-up phase have finished. Therefore, all the machines start to have enough computing power to allocate the waiting jobs requiring GPU.

Since the scenario that submits 512 jobs introduces more pressure in the system, *DRMaestro* has less room for improvements. This leads to a smaller speedup in the makespan when compared with the previous scenario that had 128 jobs, as shown in Table 2. Additionally, the percentage of jobs with violated QoS is ≈4% higher in *DRMaestro* without using the *NA* policy.

The scheduling overhead is illustrated in Fig. 6, which shows the Cumulative Distribution Function (CDF) of the average time that the algorithm and all the other scheduling mechanism spend on making and enforcing decisions. The *DRMaestro* scheduler that allocates disaggregated resources is ≈2 times slower than the *Firmament* scheduler that does not allocate disaggregated resources. This behavior is expected since the *DRMaestro* scheduler runs the model twice, in two consecutive phases. Nonetheless, the *DRMaestro* scheduler overhead is still minimal; it is in the order of sub-seconds.

Table 3 Number of jobs that violate QoS

			Number of jobs with violated QoS		
# Job	# Node	Net Policy	Fir.	DR.	Ratio Fir.\DR.
128	4	False	106	117	→ ≈0.91x
128	4	True	-	112	
				↓	↘
Network Policy Improvement				≈1.04x	≈0.95x
512	4	False	445	463	→ ≈0.96x
512	4	True	-	431	
				↓	↘
Network Policy Improvement				≈1.07x	≈1.03x

Where Fir. is Firmament and DR. is DRMaestro

Experiment 3 - resource-efficiency leveraged by mitigating QoS violations

As presented in the earlier experiments, the number of jobs with violated QoS is higher in *DRMaestro* then in *Firmament*. This is because the jobs that are using disaggregated resources might have performance perturbation according to the network load intensity. Therefore, to mitigate the QoS violation from network interference in jobs using disaggregated resources, we implemented a Network-Aware (*NA*) policy. This policy determines the placement cost based on the current network load and the network sensitivity of the job, and postpones the placement of jobs in suboptimal placement conditions, as described in "The network-aware (NA) policy" section.

In both scenarios, submitting 128 and 512 jobs on 4 machines, the *NA* policy is effective at improving the QoS violations for *DRMaestro*. There was an improvement of ≈4% for the scenario with 128 jobs and ≈7% for the other one. It is worth noting that for the scenario with 512 jobs, *DRMaestro* with *NA* policy has effectively protected the QoS, showing fewer violations than *Firmament*. Before, without the policy, *DRMaestro* was introducing ≈9% more QoS violation than *Firmament*, but, with the *NA* policy, *DRMaestro* started to have ≈3% less QoS violations. The policy also confers improvement in the total makespan, ≈4%, as detailed in Table 2. Furthermore, if there are more available resources, *DRMaestro* with *NA* policy will have more room for improvements, as we

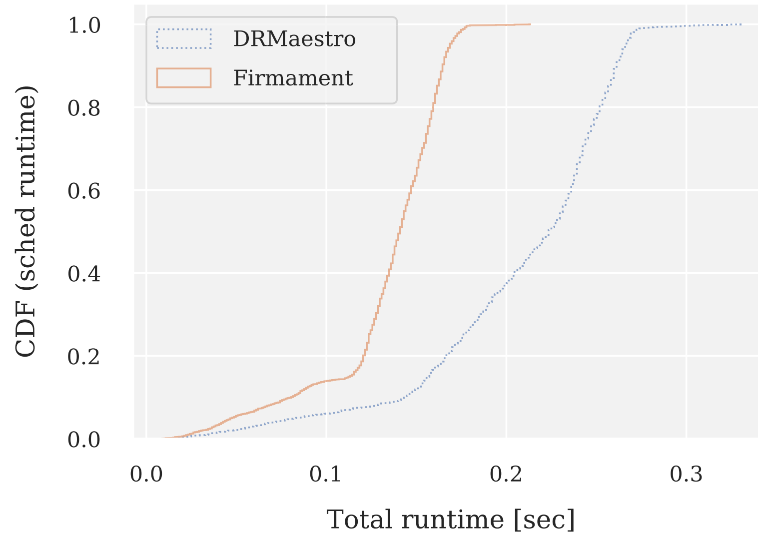


Fig. 6 Scheduler execution time. The *DRMaestro* scheduler runs 2 times slower than *Firmament* scheduler, since *DRMaestro* executes the model twice in two phases

further detail in a large-scale scenario in "Trace-driven simulation evaluation" section.

Trace-driven simulation evaluation

To evaluate the potential of *DRMaestro* over a large-scale scenario - with thousands of jobs and hundreds of machines - we implemented an event-driven simulation based on traces collected from the prototype. This experiment shares the evaluation goals II (resource-efficiency leveraged by disaggregation) and III (QoS violations mitigated by *NA* policy) like the previous experiment.

Trace-driven simulation implementation

The simulation simulates machines that are possible to be configured with different NUMA topologies with different number CPUs, GPUs, and memory regions. The job description defines the job type since each job requests a different amount of resources, and has different completion time and performance interference models, as detailed in the previous experiment.

The simulation has a generator for synthetic workloads, creating events that will represent the arrival of a new job (following a Poisson process with an exponential distribution), taking as a parameter the time interval between job arrivals. The event that stands for the arrival of a job is handled by creating a job in the system and submitting it to the scheduler module. There is also an event to call the scheduler: this event is periodically generated by a given interval. This scheduling event updates the flow-network model with the new jobs and calls the flow-network algorithm to solve the model. With the results from the solver, it generates events to place the tasks in the simulated machines, and also events to complete the tasks.

The task completion time is given by a parameter that is re-evaluated each time that a new task starts or a running task finishes. Note that, the module that defines the task execution time does not adjust only the completion time of the new incoming task but also the completion time of all currently running tasks in the machine. The execution time is then determined based on a given histogram of the jobs' completion time, for each job type, that is generated from the traces from the prototype. The execution time also changes based on the currently running jobs in the machine, which uses a pre-defined profile model loaded from the traces, slow down or speed up the completion time based on the task's resource allocation and collocation.

Trace-driven simulation configuration

To generate the traces, we executed the prototype over different configurations, warm-ups, and job arrival rates. To be statistically representative, each scenario was executed ten times. Afterward, the trace files are parsed and transformed into a format compatible with the simulator, creating application execution time and resource usage profiles.

For generating workloads, a Poisson process with the same arrival rate (16 jobs per second) as the first experiment is used. To create the job's configuration, we use a uniform distribution generating the job's type (Rodinia applications). All simulated machines are configured with the same architecture as the machines used in the testbed.

Therefore, all jobs can run when there are enough resources.

Additionally, as in the previous experiment, we warm up the cluster with initial jobs requesting 50% of the available CPUs with no network requirements.

Experiment 4: resource-efficiency leveraged by GPU disaggregation over a large-scale scenario in a trace-driven simulation

In this experiment, we quantify the improvements to maximize the resource utilization of the cluster by using GPU disaggregation. We first evaluate both *DRMaestro* and *Firmament* using the trace-driven simulation configured to submit 512 jobs to 16 machines. After that, we simulate a more intensive scenario submitting 4096 jobs to 128 machines.

The results show that *DRMaestro*, even with the more intensive scenario, provides higher resource utilization, scheduling flexibility, minimizing the makespan, and QoS violations. For example, Fig. 7 and Table 4 show a speedup of up to ≈1.21X in the accumulative execution time of the experiment when using *DRMaestro* instead of *Firmament*. Like in the previous experiment, in "Experimental evaluation" section, *Firmament* could only allocate a few GPUs because of the lack of available computing power in some machines. However, just after the warm-up jobs finished, *Firmament* can allocate all the GPUs in the cluster. Conversely, unlike the earlier experiment, with this experiment, *DRMaestro* does not allocate all the GPUs in the system during all the experiment execution. This is due to the arrival rate of the jobs, if the system is not fully stressed and there are no waiting jobs, *DRMaestro* will not allocate idle GPUs. This explains why the GPUs are not fully utilized since the beginning. We have confirmed this with other experiments that vary the arrival rate and the number of jobs and machines.

Regarding the QoS, unlike the previous experiment, in this experiment, there are more machines (16 instead of only 4), and hence, there are more possibilities for improvement. Thence, *DRMaestro* did not introduce more QoS violations than *Firmament*. However, by using the *NA* policy, *DRMaestro* could perform much better as will be discussed in "Experiment 5 - resource-efficiency leveraged by mitigating QoS violations over a large-scale scenario in a trace-driven simulation" section.

In the second experiment performed with the simulator, we evaluated a larger scenario submitting 4096 jobs to be placed on 128 machines. This scenario is interesting since it shows that although *DRMaestro* could confer less speedup in the makespan like the other experiment, *DRMaestro* performed much better regarding QoS violation. Improving the number of QoS violations by ≈65%.

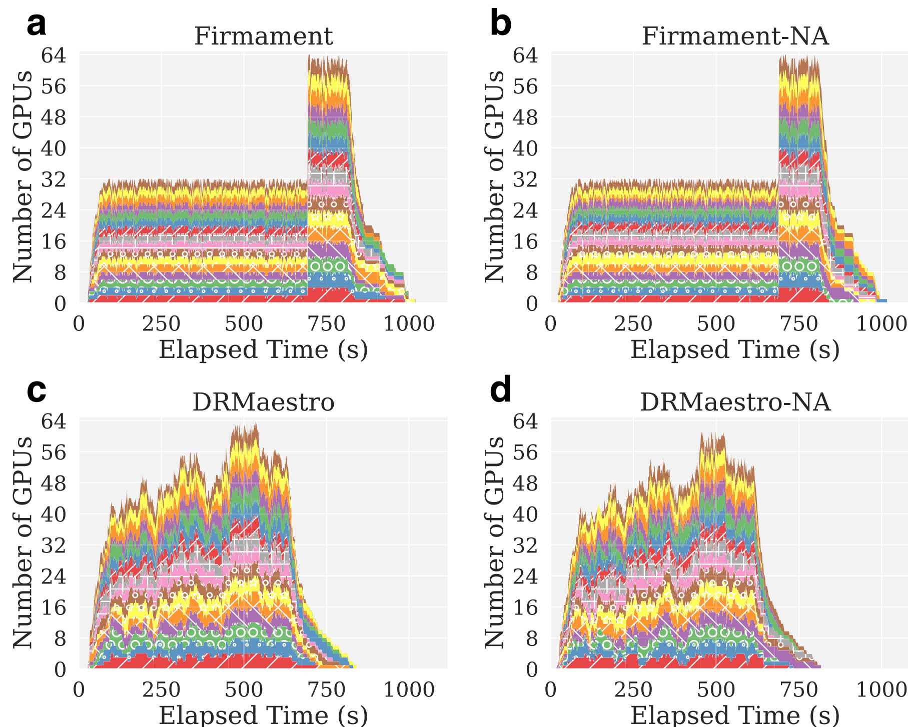


Fig. 7 Simulation results of orchestration with *DRMaestro*. Orchestration performed with *DRMaestro* with the implemented prototype in a cluster with 16 machines and 512 jobs

Table 4 Total makespan. Where Fir. is Firmament and DR. is DRMaestro

			Execution time		
# Job	# Node	Net Policy	Fir.	DR.	Ratio Fir.\DR.
512	16	False	1019.33s	842.58s	→ ≈1.21x
512	16	True	-	815.56s	
				↓	↘
Network Policy Improvement				≈1.03x	≈1.25x
4096	128	False	1087.73s	972.20s	→ ≈1.12x
4096	128	True	-	907.18s	
				↓	↘
Network Policy Improvement				≈1.07x	≈1.20x

Although the overhead of the *DRMaestro* scheduler is closer to *Firmament* in this scenario, there are some cases that the *DRMaestro* scheduler runs slower. This comportment is shown in the CDF in Fig. 8. That the *DRMaestro* scheduler runs slower is expected since the allocation of disaggregated GPUs requires the model to run twice. On the other hand, the behavior of both schedulers' overhead looks similar and is explained by the fact that the *DRMaestro* scheduler can place more jobs by enabling GPU disaggregation. Thus, the number of waiting jobs in the *DRMaestro*'s queue decreases faster with *Firmament*, minimizing the pressure on the *DRMaestro* scheduler.

Experiment 5 - resource-efficiency leveraged by mitigating QoS violations over a large-scale scenario in a trace-driven simulation

As discussed before, to mitigate the QoS violations introduced in the application due to disaggregated resources, we developed the *NA* policy. In this experiment, we evaluate the effects of this policy in two large-scale scenarios by using our trace-driven simulation. The first scenario submits 512 jobs to be placed on 16 machines, and the second one submits 4096 jobs to be placed on 128 machines.

The results are shown in Fig. 9, and Tables 4 and 5. The *NA* policy improved both the total makespan and the QoS violation count for both scenarios. In the first scenario, the improvement in the makespan is minimal, i.e. only 3%, since the amount of resources is limited. However, the policy decreases the *DRMaestro* QoS violation count by ≈40%.

On the other hand, in the second scenario, the *DRMaestro* scheduler with the *NA* policy decreased the total makespan by ≈7%. The *DRMaestro* scheduler with the *NA* policy decreased the total makespan by ≈20% compared with *Firmament*. Moreover, the *NA* policy reduces the number of *DRMaestro* QoS violations by ≈60% compared with *DRMaestro* without the policy and improved the number of QoS violations ≈2.64x compared with *Firmament*. The results of the QoS violation are much different in this scenario than the previous one. This is explained by the fact that in this scenario there are many more available resources, i.e., from only 16 machines to 128, then, there is much more room to perform better placements, providing more efficient job collocation. Hence, by improving

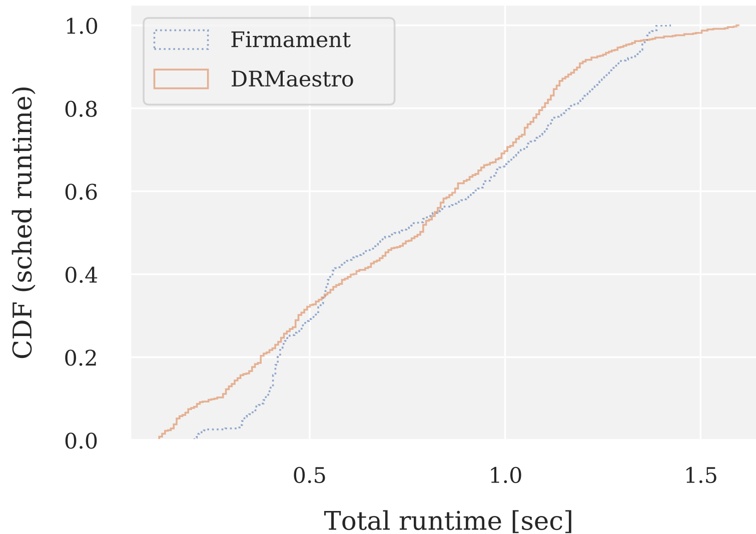


Fig. 8 Scheduler execution time. Both *DRMaestro* and *Firmament* scheduler appear to have similar overhead, expect that *DRMaestro* runs slower in some cases for the scenario with 4096 jobs and 128 machines

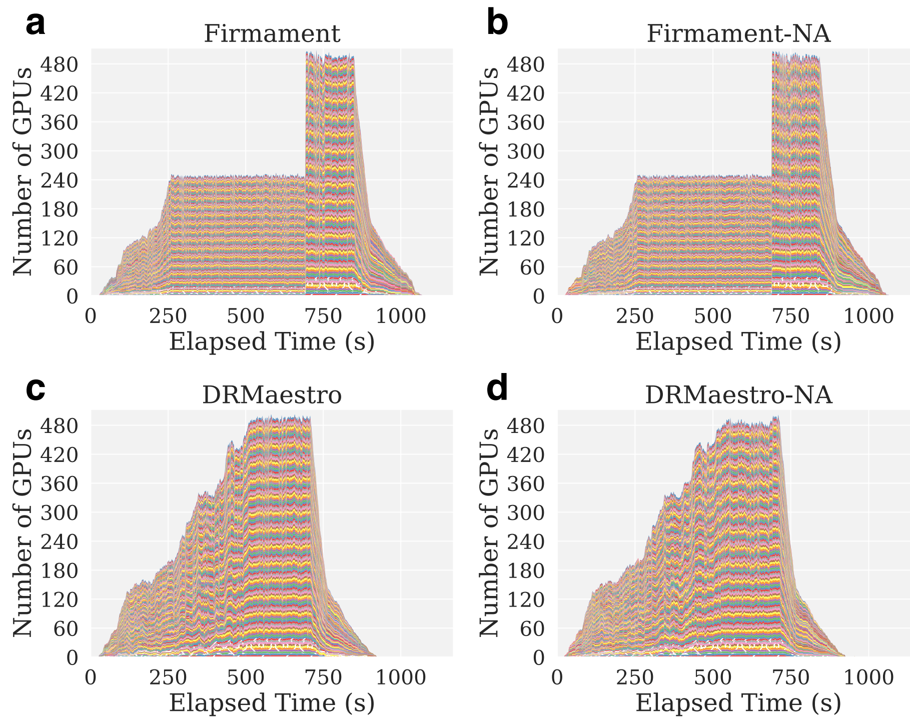


Fig. 9 Simulation results of orchestration with *DRMaestro*. Orchestration performed with *DRMaestro* with the implemented prototype in a cluster with 128 machines and 4096 jobs

the quality of the job collocation, more jobs will suffer less slowdown which will also affect the total makespan.

The comparison of the performance of the *Firmament* and the *DRMaestro* schedulers regarding QoS for the second scenario is presented in Fig. 10. This figure illustrates the improvements that the *NA* policy confers to *DRMaestro* by showing the normalized job execution time sorted from worst to best.

Table 5 Number of jobs with violated QoS

			Number of Jobs with violated QoS		
# Job	# Node	Net Policy	Fir.	DR.	Ratio Fir.\DR.
512	16	False	377	382	→ ≈0.99x
512	16	True	-	272	
				↓	↘
Network Policy Improvement				≈1.40x	≈1.39x
4096	128	False	2160	1310	→ ≈1.65x
4096	128	True	-	818	
				↓	↘
Network Policy Improvement				≈1.60x	≈2.64x

Conclusion

In this paper we have presented DRMaestro, a novel framework to orchestrate disaggregated resources on shared cloud systems. DRMaestro addresses some of the main challenges found in data-centers with disaggregated architectures, providing a mechanism to enable transparent disaggregation of resources, as well as an optimized placement of workloads that improves resource efficiency while avoiding interference.

This work first evaluated the impact of disaggregation regarding the additional network requirements under higher network load. The results show that for some applications the impact is minimal, but other ones can suffer up to 80% slowdown in the data transfer part. Then, the framework is validated through the implementation of a prototype on Kubernetes and evaluated by performing several trace-driven simulations with traces from the testbed that show that our solution provides higher cluster utilization and fewer SLO violations.

Our experiments driven by representative workloads demonstrate the effectiveness of our proposal in different scenarios. To the best of our knowledge, this is the first scheduling framework to take into account resource disaggregation that optimizes placement while mitigating sharing-induced interferences. The experiments show the trade-off of enabling resource disaggregation in a shared

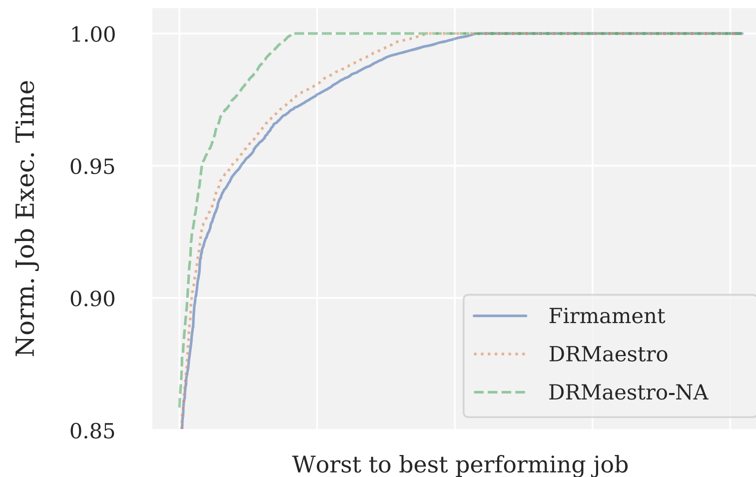


Fig. 10 Job's QoS. Normalized job execution time for the scheduling policies with and without network awareness, ordered from worst to best. For a scenario with ≈4k jobs and 128 machines

cluster. *DRMaestro* reduces the total job makespan with a speedup of up to ≈1.20x and decreases the number of QoS violations up to ≈2.64x compared with similar orchestrator based on flow-network discipline that does not support resource disaggregation.

To the best of our knowledge, our work is the first to apply a flow-network model to solve the scheduling and placement problem for a cluster composed of disaggregated resources.

For future work, we plan to introduce into the scheduler the notion of minimizing GPU communication cost, along with the possibility to share GPUs among different jobs.

Acknowledgements
We thank our anonymous reviewers for their feedback and suggestions, most specially Scott Trent for the valuable comments.

Author's contributions
Marcelo Amaral conceived of the initial idea, wrote the algorithm, and carried out the experiments as a Ph.D. student under the supervision of David Carrera and Jordà Polo. Marcelo Amaral wrote the manuscript. Nelson Gonzalez conceived of the initial idea and wrote the HFCUDA library. Chih-Chieh Yang performed the analysis of the possible performance interference introduced by additional network load in disaggregated resources. Alessandro Morari, Bruce D'Amora, Alaa Youssef, and Malgorzata Steinder provided a valuable discussion about the framework architecture and the results, along with the testbed environment to execute the experiments. The author(s) read and approved the final manuscript.

Authors' information
Marcelo Amaral received his Bachelor degree in Computer Science (2009) and M.Sc. in Computer Engineer (2014) at University of São Paulo (USP), Brazil, and his Ph.D. on Computer Architecture (2019) at Universitat Politècnica de Catalunya (UPC), Spain. His main research interest is focused on the performance management of datacenter workloads.
Jordà Polo received an M.S. from the Technical University of Catalonia (UPC) in 2009, and a Ph.D. from the same university in 2014. He currently leads a team of Ph.D. students and postdocs at the Data-Centric Computing group at Barcelona Supercomputing Center (BSC), doing research in software-defined infrastructures and data-centric architectures, proposing new models and algorithms for placement of jobs and data in future data-centers.
David Carrera received the MS degree at the Technical University of Catalonia (UPC) in 2002 and his Ph.D. from the same university in 2008. He is an associate professor at the Computer Architecture Department of the UPC. He is also the Head of the "DataCentric Computing" research group at the Barcelona Supercomputing Centre (BSC). He has participated in several EU-funded projects and been PI for several industrial projects and collaborations with IBM, Microsoft, and Cisco among others. In 2015 he was awarded an ERC Starting Grant, and in 2010 he received an IBM Faculty Award.
Nelson Mimura Gonzalez is research at IBM T.J. Watson Research Center in Yorktown Heights. He received both his B.S. M.S. And Ph.D. in computer engineering from the University of São Paulo (USP). His past research experiences include cloud and edge computing resource management for the efficient execution of large-scale scientific applications. His current focus is on scaling distributed applications to both current state-of-the-art supercomputers, and disaggregated architectures.
Chih-Chieh Yang is a research IBM T.J. Watson Research Center in Yorktown Heights. He received both his B.S. and M.S. from National Tsing Hua University, and accumulated industry experiences working for the top Taiwanese IC design house, Mediatek, for several years before starting and eventually earning his Ph.D. from the University of Illinois at Urbana-Champaign. His past research experiences include designing software components that facilitate the development of distributed applications and high-level parallel programming abstractions. His current focus is on scaling distributed machine learning applications to both current state-of-the-art supercomputers and future extreme-scale HPC systems.
Bruce D'Amora is a Senior Technical Staff Member in the Data-Centric Solutions department at IBM T.J. Watson Research Center in Yorktown Heights, NY. He manages the Cognitive and Cloud solutions department focusing on the enablement of HPC and Cognitive workflows using cloud-native technologies. Other interests include computational steering and visualization for high-performance computing applications. He led the performance analysis effort for CORAL systems and the application to bring for BlueGene/Q. Prior to his arrival at IBM Research, he led IBM's development of the OpenGL graphics API and served as their representative on the OpenGL Architecture Review Board.
Alaa Youssef leads the Container Cloud Platform team at IBM T.J. Watson Research Center. His research focus is on cloud-native computing, secure distributed systems, and continuous delivery of large software systems. He has authored and co-authored many technical publications and holds over a dozen issued patents. He is a senior architect who has held multiple technical and management positions in IBM Research, and in Services in multiple geographies. He received his Ph.D. in Computer Science from Old Dominion University, Virginia, and his MSc in Computer Engineering from Alexandria University, Egypt.

Alessandro Morari leads the Cloud Security Analytics team in the Hybrid Cloud department at IBM Research. His main research interests are System Software, Cloud computing, Machine Learning, Performance, and Statistical Modeling. He has been involved in the development of data-intensive system software for the Summit and Sierra supercomputers, commissioned by the US Department of Energy. He holds a Ph.D. in Computer Architecture from the Polytechnic University of Catalonia, an M.Sc. in Computer Engineering for the University of Rome Tor Vergata, and a B.Sc. in Computer Engineering from the Roma Tre University.
Malgorzata (Gosia) Steinder researches on container-based cloud platforms. Her team has delivered technologies behind IBM Cloud Kubernetes Service and its Vulnerability Advisor. Earlier, she led a research team that invented and developed virtual machine management technology in IBM Pure Application System and IBM WebSphere CloudBurst Appliance and developed a dynamic VM placement algorithm for IBM Platform Resource Scheduler. She was also a key member of developing workload management technology in WebSphere Virtual enterprise. She was a Ph.D. student and a Graduate Fellow at the University of Delaware. She completed her master degree at AGH University of Science and Technology, Krakow, Poland.

Funding
This project is supported by the IBM/BSC Technology Center for Supercomputing collaboration agreement. It has also received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 639595). It is also partially supported by the Ministry of Economy of Spain under contract TIN2015-65316-P and Generalitat de Catalunya under contract 2014SGR1051, by the ICREA Academia program, and by the BSC-CNS Severo Ochoa program (SEV-2015-0493).

Availability of data and materials
The test cases are generated by an algorithm, which will be made available under an appropriate open source license upon acceptance of the article.

Competing interests
The authors declare that they have no competing interests.

Author details
[1]IBM Watson Research Center, Yorktown Heights, US. [2]Department of Computer Architecture, Universitat Politècnica de Catalunya (UPC) - BarcelonaTECH., Barcelona, Spain. [3]Barcelona Supercomputing Center (BSC), Barcelona, Spain.

Received: 26 September 2020 Accepted: 9 February 2021
Published online: 06 March 2021

References

1. Gao PX, Narayan A, Karandikar S, Carreira J, Han S, Agarwal R, Ratnasamy S, Shenker S (2016) Network requirements for resource disaggregation. In: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation OSDI'16. USENIX Association, Berkeley, CA, USA. pp 249–264. http://dl.acm.org/citation.cfm?id=3026877.3026897
2. Katrinis K, Syrivelis D, Pnevmatikatos D, Zervas G, Theodoropoulos D, Koutsopoulos I, Hasharoni K, Raho D, Pinto C, Espina F, Lopez-Buedo S, Chen Q, Nemirovsky M, Roca D, Klos H, Berends T (2016) Rack-scale disaggregated cloud data centers: The dReDBox project vision. In: 2016 Design, Automation Test in Europe Conference Exhibition (DATE). pp 690–695
3. Pagés A, Serrano R, Perell J, Spadaro S (2017) On the benefits of resource disaggregation for virtual data centre provisioning in optical data centres. Comput Commun 107:60–74. https://doi.org/10.1016/j.comcom.2017.03.009
4. Taylor J (2015) Facebook's data center infrastructure: Open compute, disaggregated rack, and beyond. In: 2015 Optical Fiber Communications Conference and Exhibition (OFC). p 1
5. Keeton K (2015) The machine: An architecture for memory-centric computing. In: Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers ROSS '15. Association for Computing Machinery, New York, NY, USA. p 1. https://doi.org/10.1145/2768405.2768406
6. Krishnapura S, Achuthan S, Lal V, Tang T (2017) Disaggregated Servers Drive Data Center Efficiency and Innovation. Intel, USA. https://www.intel.com/content/dam/www/public/us/en/documents/best-practices/disaggregated-server-architecture-drives-data-center-efficiency-paper.pdf
7. Reano C, Silla F (2017) A comparative performance analysis of remote GPU virtualization over three generations of GPUs. In: 46th International Conference on Parallel Processing WGorkshops (ICPPW). pp 121–128. https://doi.org/10.1109/ICPPW.2017.29
8. Lin AD, Li CS, Liao W, Franke H (2018) Capacity optimization for resource pooling in virtualized data centers with composable systems. IEEE Trans Parallel Distrib Syst 29(2):324–337. https://doi.org/10.1109/TPDS.2017.2757479
9. Hindman B, Konwinski A, Zaharia M, Ghodsi A, Joseph AD, Katz R, Shenker S, Stoica I (2011) Mesos: A platform for fine-grained resource sharing in the data center. In: Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation NSDI'11. USENIX Association, Berkeley, CA, USA. pp 295–308. http://dl.acm.org/citation.cfm?id=1972457.1972488
10. Rensin DK (2015) Kubernetes - Scheduling the Future at Cloud Scale. Oreilly, 1005 Gravenstein Highway North Sebastopol, CA 95472. http://www.oreilly.com/webops-perf/free/kubernetes.csp
11. Vavilapalli VK, Murthy AC, Douglas C, Agarwal S, Konar M, Evans R, Graves T, Lowe J, Shah H, Seth S, Saha B, Curino C, O'Malley O, Radia S, Reed B, Baldeschwieler E (2013) Apache hadoop yarn: Yet another resource negotiator. In: Proceedings of the Symposium on Cloud Computing SOCC '13. ACM, New York. pp 1–16. https://doi.org/10.1145/2523616.2523633
12. Verma A, Pedrosa L, Korupolu M, Oppenheimer D, Tune E, Wilkes J (2015) Large-scale cluster management at Google with Borg. In: Proceedings of the Tenth European Conference on Computer Systems EuroSys '15. ACM, New York, NY, USA. pp 1–17. https://doi.org/10.1145/2741948.2741964
13. Gog I, Schwarzkopf M, Gleave A, Watson RNM, Hand S (2016) Firmament: Fast, centralized cluster scheduling at scale. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16). USENIX Association, Savannah, GA. pp 99–115
14. Lovász L, Plummer MD (1986) Matching theory. North-Holland Mathematics Studies, Vol. 121. North-Holland, Amsterdam, Netherlands. https://doi.org/10.1016/S0304-0208(08)73637-5. http://www.sciencedirect.com/science/article/pii/S0304020808736375
15. Bunke H, Jiang X (2000) Graph Matching and Similarity. Springer, Boston, MA
16. Ahuja RK, Magnanti TL, Orlin JB (1993) Network Flows: Theory, Algorithms, and Applications. Prentice-Hall, Inc., Upper Saddle River, NJ, USA
17. Goldberg AV (1997) An efficient implementation of a scaling minimum-cost flow algorithm. J Algorithm 22(1):1–29. https://doi.org/10.1006/jagm.1995.0805
18. Carrera D, Steinder M, Whalley I, Torres J, Ayguadé E (2008) Enabling resource sharing between transactional and batch workloads using dynamic application placement. In: Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware '08. Springer, New York. pp 203–222. http://dl.acm.org/citation.cfm?id=1496950.1496964
19. Klein M (1967) A primal method for minimal cost flows with applications to the assignment and transportation problems. Manag Sci 14(3):205–220. https://doi.org/10.1287/mnsc.14.3.205
20. Goldberg AV, Tarjan RE (1990) Finding minimum-cost circulations by successive approximation. Math Oper Res 15(3):430–466. https://doi.org/10.1287/moor.15.3.430
21. Hu Y, Song M, Li T (2017) Towards "full containerization" in containerized network function virtualization. In: Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems. ACM, New York, NY, USA. pp 467–481. https://doi.org/10.1145/3037697.3037713
22. Isard M, Prabhakaran V, Currey J, Wieder U, Talwar K, Goldberg A (2009) Quincy: Fair scheduling for distributed computing clusters. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles. ACM, New York. pp 261–276. https://doi.org/10.1145/1629575.1629601
23. Soppelsa F, Kaewkasi C (2017) Native Docker Clustering with Swarm. Packt Publishing, USA. https://dl.acm.org/doi/book/10.5555/3153103

24. Ousterhout K, Wendell P, Zaharia M, Stoica I (2013) Sparrow: Distributed, low latency scheduling. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. ACM, New York. pp 69–84. https://doi.org/10.1145/2517349.2522716
25. Meng X, Pappas V, Zhang L (2010) Improving the scalability of data center networks with traffic-aware virtual machine placement. In: Proceedings of the 29th Conference on Information Communications. IEEE Press, San Diego. pp 1154–1162
26. Zhang W, Han S, He H, Chen H (2017) Network-aware virtual machine migration in an overcommitted cloud. Futur Gener Comput Syst 76:428–442. https://doi.org/10.1016/j.future.2016.03.009
27. Iserte S, Castelló A, Mayo R, Quintana-Ortí ES, Silla F, Duato J, Reaño C, Prades J (2014) Slurm support for remote GPU virtualization: Implementation and performance study. In: Proceedings of the 2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing. IEEE Computer Society, Washington, DC, USA. pp 318–325. https://doi.org/10.1109/SBAC-PAD.2014.49
28. Jette MA, Yoo AB, Grondona M (2003) Slurm: Simple linux utility for resource management. In: Workshop on Job Scheduling Strategies for Parallel Processing. Springer, Berlin, Heidelberg. pp 44–60. https://doi.org/10.1007/10968987_3. http://citeseerx.ist.psu.edu/viewdoc/summary?cid=184264
29. Iserte S, Clemente-Castelló FJ, Castelló A, Mayo R, Quintana-Ortí ES (2016) Enabling GPU virtualization in cloud environments. In: Proceedings of the 6th International Conference on Cloud Computing and Services Science. SCITEPRESS - Science and Technology Publications, Lda, Portugal. pp 249–256. https://doi.org/10.5220/0005780502490256
30. Sefraoui O, Aissaoui M, Eleuldj M (2012) Openstack: Toward an open-source solution for cloud computing. Int J Comput Appl 55(3):38–42. https://doi.org/10.5120/8738-2991
31. Lama P, Li Y, Aji AM, Balaji P, Dinan J, Xiao S, Zhang Y, Feng W, Thakur R, Zhou X (2013) pVOCL: Power-aware dynamic placement and migration in virtualized GPU environments. In: 2013 IEEE 33rd International Conference on Distributed Computing Systems. pp 145–154. https://doi.org/10.1109/ICDCS.2013.51
32. Xiao S, Balaji P, Zhu Q, Thakur R, Coghlan S, Lin H, Wen G, Hong J, Feng W (2012) VOCL: An optimized environment for transparent virtualization of graphics processing units. In: 2012 Innovative Parallel Computing (InPar). pp 1–12. https://doi.org/10.1109/InPar.2012.6339609
33. Oikawa M, Kawai A, Nomura K, Yasuoka K, Yoshikawa K, Narumi T (2012) DS-CUDA: a middleware to use many GPUs in the cloud environment. In: 2012 SC Companion: High Performance Computing, Networking Storage and Analysis. pp 1207–1214. https://doi.org/10.1109/SC.Companion.2012.146
34. Liang TY, Chang YW (2011) GridCuda: a grid-enabled CUDA programming toolkit. In: 2011 IEEE Workshops of International Conference on Advanced Information Networking and Applications. pp 141–146. https://doi.org/10.1109/WAINA.2011.82
35. Merritt AM, Gupta V, Verma A, Gavrilovska A, Schwan K (2011) Shadowfax: scaling in heterogeneous cluster systems via GPGPU assemblies. In: Proceedings of the 5th International Workshop on Virtualization Technologies in Distributed Computing. ACM, New York, NY, USA. pp 3–10. https://doi.org/10.1145/1996121.1996124
36. Quinlan JR (1986) Induction of decision trees. Mach Learn 1(1):81–106. https://doi.org/10.1023/A:1022643204877
37. Fenacci D, Franke B, Thomson J (2010) Workload characterization supporting the development of domain-specific compiler optimizations using decision trees for data mining. In: Proceedings of the 13th International Workshop on Software; Compilers for Embedded Systems. ACM, New York, NY, USA. pp 1–10. http://doi.acm.org/10.1145/1811212.1811219
38. Delimitrou C, Kozyrakis C (2014) Quasar: Resource-efficient and qos-aware cluster management. In: Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems. ACM, New York, NY, USA. pp 127–144. https://doi.org/10.1145/2541940.2541941
39. Lo D, Cheng L, Govindaraju R, Ranganathan P, Kozyrakis C (2015) Heracles: Improving resource efficiency at scale. In: Proceedings of the 42Nd Annual International Symposium on Computer Architecture. ACM, New York, NY, USA. pp 450–462. https://doi.org/10.1145/2749469.2749475
40. Hoste K, Phansalkar A, Eeckhout L, Georges A, John LK, De Bosschere K (2006) Performance prediction based on inherent program similarity. In: Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques. ACM, New York, NY, USA. pp 114–122. https://doi.org/10.1145/1152154.1152174
41. Srivastava S (2018) The Poseidon an add-on Kubernetes scheduler for Firmament scheduler framework. https://github.com/kubernetes-sigs/poseidon
42. Li C-S, Franke H, Parris C, Abali B, Kesavan M, Chang V (2017) Composable architecture for rack scale big data computing. Futur Gener Comput Syst 67:180–193. https://doi.org/10.1016/j.future.2016.07.014
43. NVIDIA (2019) Multi-process service (MPS). In: Multi-Process Service (MPS). p 1. https://docs.nvidia.com/deploy/mps/index.html. Accessed in 29 January 2019
44. Amaral M, Polo J, Carrera D, Seelam S, Steinder M (2017) Topology-aware GPU scheduling for learning workloads in cloud environments. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. ACM, Denver, CO, USA. pp 1–12. https://doi.org/10.1145/3126908.3126933
45. Che S, Boyer M, Meng J, Tarjan D, Sheaffer JW, Lee S-H, Skadron K (2009) Rodinia: A benchmark suite for heterogeneous computing. In: Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC). IEEE Computer Society, Washington. pp 44–54. https://doi.org/10.1109/IISWC.2009.5306797
46. Dugan J, Elliott S, Mah BA, Poskanzer J, Prabhu K (2014) iperf3: A TCP, UDP, and SCTP network bandwidth measurement tool. University of California Lawrence Berkeley National Laboratory and U.S. Department of Energy. https://github.com/esnet/iperf. Accessed in 21 Jan 2015

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Submit your manuscript to a SpringerOpen journal and benefit from:

- Convenient online submission
- Rigorous peer review
- Open access: articles freely available online
- High visibility within the field
- Retaining the copyright to your article

Submit your next manuscript at ▶ springeropen.com