"Virtualization is a mechanism to abstract the operating system, hardware and system resources, hiding from the application the complexity of the underlying resources."

Le fils de l'homme (The Son of Man)
René Magritte (1898-1967)
1964. Oil on canvas. 116 cm x 89 cm
"We desire to see what's hidden behind the visible."

Improving Resource Efficiency in Virtualized Datacenters

By Marcelo Amaral

Advisors:
David Carrera
Jordà Polo

A dissertation submitted in partial fulfilment of the requirements for the degree of:

Doctor of Philosophy at Universitat Politècnica de Catalunya

Barcelona (Spain)
2019

Technical University of Catalunya – BarcelonaTech (UPC)

"Everything we see hides another thing, we always want to see what is hidden by what we see. There is an interest in that which is hidden and which the visible does not show us. This interest can take the form of a quite intense feeling, a sort of conflict, one might say, between the visible that is hidden and the visible that is present."

— **Rene Magritte, 1965**

Marcelo Carneiro do Amaral: *Improving Resource Efficiency in Virtualized Datacenters,* Topology-Aware Resource Provision Techniques, © January 2019

Dedicated to my loving wife.

Dedicated also in memory of my mother.

1948 – 2010

ABSTRACT

Modern applications demand resources at an unprecedented level and, therefore, datacenters are required to scale efficiently when more resources are added to the infrastructure, increasing their efficiency and flexibility to manage workloads. A technology that confers advantages towards resource-efficiency is virtualization. A virtualized data center offers higher management flexibility and at the same time increases resource utilization by allowing workload collocation and isolation. Therefore, server virtualization has now fully penetrated the market and become widely accepted on both the scientific and industrial community.

At the same time, infrastructures have been shifting away from traditional datacenters toward a software-based architecture with the focus on flexibility and customization, allowing data center technologies to transition to the resource disaggregation paradigm, in an attempt to expose expensive resources such as accelerators and flash memories as pooled network resources that can be accessed across all the datacenter nodes. As a consequence of this paradigm, operators can increase resource utilization by allocating spare fragmented resources to remote applications.

Virtualization and resource disaggregation mechanisms simplify the complexity and significantly enhances the flexibility of datacenter management. However, they pose new challenges on how to extract the best performance of an unknown underlying platform layer which is not fully exposed to the applications. More specifically, in such environments, applications have limited accesses and view to the resources, since these technologies abstract the view of the hardware topology and their characteristics, and limit the access to resources by performing fine-grain resource partitioning. Therefore, the potential to fully exploit the resource-efficiency in virtualized and disaggregated datacenters is conditional on an intelligent system making informed decisions to orchestrate the resources with a high-level view of the system.

Additionally, as organizations demand management solutions to optimize how workloads efficiently run on virtualized datacenters, workload orchestration is still facing the following three research challenges: i) understanding the performance impact caused by both the virtualization and the underlying hardware characteristics; ii) efficiently managing the application scheduling observing completion time goals and hardware characteristics; and iii) adequately allocating resources to each application in a flexible and automated manner.

In this context, this thesis contributes to the datacenter management to improve the resource-efficiency on virtualized environments by C1) evaluating and characterizing the performance of applications running on *virtual environment* (i.e., containers or virtual machines) over complex hardware architectures (e.g. NUMA topologies), C2) providing new topology-aware multi-GPU workload scheduling techniques that maximize the overall performance while minimizing the quality-of-service violations and C3) proposing and evaluating a novel automatic workload orchestration for pooled resources and disaggregated architectures capable of improving resource utilization across servers. The combination of the three methods proposed in this thesis creates a new range of options to enhance the resource-efficiency of a datacenter in regards to performance (C1 and C2) and system utilization (C3).

Keywords: topology-aware scheduling, performance analysis, virtualization, NUMA, resource disaggregation, software-defined architecture, workload optimized systems, and datacenters.

ACKNOWLEDGEMENTS

When I was about to finish my master's degree in 2013, the desire to start a new journey in a doctoral program began to grow. At that moment, I was not sure what to do, I just knew that getting a Ph.D. was definitely a good option. Not only to improve my technical skills but also to take advantage of the opportunity to experience experiences abroad, to grow and to improve myself professionally.

The motivation to start looking for doctoral opportunities was a combination of many factors. The excellent influences of doctoral students that I admired, and still admire, are one of them — especially Charles Miers and Carlos Costa. Charles once advised me to start a doctoral program at a "different" university where I could meet not only new people but also open doors for more opportunities. Even today I find it a piece of great advice. Carlos gave me the will to always seek the best opportunities and showed me that, with dedication, it is possible to get far. He also helped to improve my critical thinking and gave lots of advice during my Ph.D.

At the end of 2013, after submitting my project to many doctoral programs, I applied for a very attractive program in Barcelona, Spain. The prospect of being able to live in Europe, the fact that the project was involved with two exciting companies - the Barcelona Supercomputing Center and IBM - allowing me to do important projects, and also the fact that the program included a stay part of the time in Barcelona and part in New York were some of the main reasons that made me very excited with this opportunity. And of course, it is essential to say that Barcelona has very good weather!

Now, towards the end of 2018, I am very proud and pleased to have started this challenging and exciting life. Over the years, I have met many people who have helped me with technical discussions and a good relationship with friendship. That is why, first of all, I want to thank my advisors, David Carrera Perez, and Jorda Polo, for their advice not only in research but in career and life in general. Working on a topic, which you are passionate about, is one of the most important aspects for doctoral students. Their support and the people I've met on IBM projects have been invaluable. In addition, I would like to mention Gosia Steinder, Alessandro Morari, Bruce D'Amora, Alaa Youssef, Nelson Gonzalez, Chih-Chieh Yang, Seetharami Seelam, Iqbal Mohomed, Merve Unuvar, Ricardo Koller, as well as the other IBM Watson people I worked with and shared lunches and coffees.

I was very lucky to interact with brilliant people at the Barcelona Supercomputing Center. Thanks to all members of the HiEST group, BSC and UPC, past and present, for everything I learned from them. Nicola Cadenelli, Josep Lluís Berral Garcia, Cesare Cugnasco, Shuja-ur-Rehman Baig, David Buchaca, Alberto Gutierrez, Aaron Call, among others: thank you! My year in Barcelona was improved by the many wonderful friends I met here and with the support of my friends in Brazil. Thank you, Nicola, Thais, Giovana, Jeff, Danilo, Mauro, Tom, Rocio, Hector, Marti, Felipe, Vinicius, Hugo, Guilherme, Paulo, Douglas, Randerson and many others for the fun memories.

Especially, I want to thank my wife, Cleide Sousa, for all her advice, support, patience, and encouragement, and for being always present - even when I was on another continent, I was in Barcelona, and she was in Brazil. Last but not least, I want to thank my parents and my family, especially my mother Maria Heloisa, my sister Camila Amaral and my aunt, Nilza Helena. With them, I learned values and had every encouragement to continue, fight for my happiness and achieve dreams. Although my mother and aunt are not here, physically, they still live deep in my heart and still help with all of their teachings. They deserve this public recognition, and more!

This thesis was supported by the Barcelona Super Computing Center (BSC) and IBM Watson Research Center for Supercomputing collaboration agreement. It had 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).

AGRADECIMENTOS

Quando eu estava prestes a terminar o meu mestrado, em 2013, a vontade de iniciar uma nova jornada em um programa de doutorado começou a crescer. Naquele momento, eu não tinha certeza sobre o que fazer, sabia apenas que começar um doutorado era definitivamente uma boa opção. Não somente para melhorar as minhas habilidades técnicas, mas também para aproveitar a oportunidade de vivenciar experiências no exterior, crescer e me aperfeiçoar profissionalmente.

A motivação para começar a procurar oportunidades de doutorado foi uma combinação de muitos fatores. As excelentes influências de estudantes de doutorado que eu admirava – e ainda admiro – é um deles. Especialmente o Charles Miers e o Carlos Costa. Uma vez, Charles me aconselhou a começar um programa de doutorado em uma universidade "diferente", onde eu poderia não apenas conhecer novas pessoas, mas também abrir portas para mais oportunidades. Ainda hoje acho um ótimo conselho. Carlos, igualmente, passou-me a vontade de buscar sempre as melhores oportunidades e me mostrou que, com dedicação, é possível chegar longe. Ele também ajudou a melhorar meu pensamento crítico e deu muitos conselhos durante o meu doutorado.

No final de 2013, após submeter o meu projeto para muitos programas de doutorado, candidatei-me a um programa bastante atraente em Barcelona, na Espanha. A chance de poder viver na Europa, o fato de que o projeto estava envolvido com duas empresas empolgantes – o Centro de Supercomputação de Barcelona e a IBM, permitindo que eu realizasse projetos importantes –, e também o fato de o programa incluir uma parte do tempo em Barcelona e outra em Nova York foram alguns dos principais motivos que me deixaram muito animado. E, claro, é essencial dizer que Barcelona tem um clima muito bom!

Agora, ao final de 2018, estou muito orgulhoso e satisfeito por ter iniciado essa vida desafiadora e emocionante. Ao longo desses anos, conheci muitas pessoas que me ajudaram com discussões técnicas e um bom relacionamento de amizade. Por isso, em primeiro lugar, quero agradecer aos meus orientadores, David Carrera Perez e Jorda Polo, por seus conselhos não apenas em pesquisa, mas na carreira e na vida, de maneira geral. Trabalhar em um tópico, o qual você é apaixonado, é um dos aspectos mais importantes para os estudantes de doutorado. O apoio deles, e de pessoas que conheci em projetos na IBM, tem sido inestimável. Além disso, quero mencionar Gosia Steinder, Alessandro Morari, Bruce D'Amora, Alaa Youssef, Nelson Gonzalez, Chih-Chieh Yang, Seetharami Seelam, Iqbal Mohomed, Merve Unuvar, Ricardo Koller, bem como as outras pessoas do IBM Watson com quem trabalhei e compartilhei almoços e cafés.

Tive muita sorte em interagir com pessoas brilhantes no Centro de Supercomputação de Barcelona. Obrigado a todos os membros do grupo HiEST, BSC e UPC, passado e atual, por tudo o que aprendi com eles. Nicola Cadenelli, Josep Lluís Berral Garcia, Cesare Cugnasco, Shuja-ur-Rehman Baig, David Buchaca, Alberto Gutierrez, Aaron Call, entre outros: obrigado!

Meu ano em Barcelona foi melhorado pelos muitos amigos maravilhosos que conheci aqui e pelo apoio de meus amigos no Brasil. Obrigado Nicola, Thaís, Giovana, Jeff, Danilo, Mauro, Tom, Rocio, Hector, Marti, Felipe, Vinicius, Hugo, Guilherme, Paulo, Douglas, Randerson e muitos outros pelas lembranças divertidas.

Especialmente, quero agradecer a minha esposa, Cleide Sousa, por todos os seus conselhos, o apoio, a paciência e o encorajamento, e por estar sempre presente – até mesmo quando estava em outro continente, eu aqui e ela no Brasil. Por último, mas não menos importante, quero agradecer aos meus pais e minha família, principalmente a minha mãe Maria Heloisa, a minha irmã Camila Amaral e a minha tia, Nilza Helena. Com eles aprendi valores e tive todo o apoio encorajador para continuar, lutar pela minha felicidade e alcançar sonhos. Apesar de minha mãe e minha tia não estarem aqui, fisicamente, elas ainda vivem no fundo do meu coração e ainda ajudam com todos os seus ensinamentos. Elas merecem esse reconhecimento público, e muito mais!

Esta tese foi apoiada pelo Barcelona Super Computing Center (BSC) e pela IBM Watson Research Center com o acordo de colaboração de Supercomputação. Essa tese também recebeu financiamento do Conselho Europeu de Investigação (CEI) no âmbito do programa de investigação e inovação Horizonte 2020 da União Europeia (convenção de subvenção No. 639595). Essa tese também foi parcialmente apoiada pelo Ministério da Economia da Espanha sob contrato TIN2015-65316-P e Generalitat de Catalunya sob contrato 2014SGR1051, pelo programa ICREA Academia, e pelo programa BSC-CNS Severo Ochoa (SEV-2015-0493).

Contents

List of Figures

List of Tables

ACRONYMS

DL	Deep Learning
DNS	Domain Name System
FPGA	Field-Programmable Gate Array
GPU	Graphics Processing Unit
HPC	High-Performance Computing
HTC	High-Throughput Computing
MPS	Multi-Process Service
NN	Neural Network
NUMA	Non-Uniform Memory Architecture
NVMe	Non-Volatile Memory Express
OS	Operating System
P2P	Peer-to-Peer
QoS	Quality-of-Services
SDE	Software Defined Environment
SLA	Service-Level Agreement
VM	Virtual Machine

1 INTRODUCTION

With the technology evolution plus the growth experienced on the available information over the last years, the resource demand has been increasing in both traditional science and engineering marketplaces (to solve complex computational problems for aerospace, finance, life sciences, etc.) and the enormous commercial marketplaces of commerce and industry (to provide highly available services such as web servers, big data and artificial intelligence applications). Not so long ago, the access to many computing resources on large clusters was relatively difficult and expensive. But today, it is much easier to access the popular off-premise cloud computing resources, and even the on-premise high-performance computing (HPC) clusters are more accessible. Consequently, these trends have been massively contributing to an increase in resource demand. To satisfy this growing resource demand, datacenter operators have been attempting to enhance the resource-efficiency, i.e., the effective utility extracted from the system resources, before investing in additional compute resources, in an attempt to maximize their business competitiveness and return on investment of their infrastructures.

A key technology to improve resource-efficiency of datacenters is virtualization: a systematic abstraction of hardware, system resources, and operating systems, to convert dedicated physical resources into virtual shared resources and simplify the complexity of management and deployment of applications. Virtualization can enable efficient collocation of workloads due to its isolation capabilities, and besides that, provides environments for applications that have specific software requirements including OS version dependencies and libraries.

In parallel, the resource disaggregation paradigm has emerged in the last years in an attempt to decouple datacenter resources (accelerators, flash memories) from the host machines where they are connected. Under this paradigm, operators can increase resource utilization by allocating spare fragmented resources to remote applications. For example, if an application is using all the compute capabilities of one server, its other resources cannot be allocated to another application. However, the extra resources can be remotely exposed to applications in different servers with available compute resources to prevent resource wastage.

For all these reasons, virtualization and disaggregation have been shifting away from traditional datacenters toward a software-based architecture with the focus on flexibility and customization.

1.1 MOTIVATION & CHALLENGES

Virtualization and resource disaggregation mechanisms simplify the complexity and significantly enhances the flexibility of datacenter management. However, they pose new challenges on how to extract the best performance of an unknown underlying platform layer which is not fully exposed to the applications. More specifically, in such environments, applications have limited accesses and view to the resources, since these technologies abstract the view of the hardware topology and their characteristics, and limit the access to resources by performing fine-grain resource partitioning (i.e., enforcing resource isolation between applications).

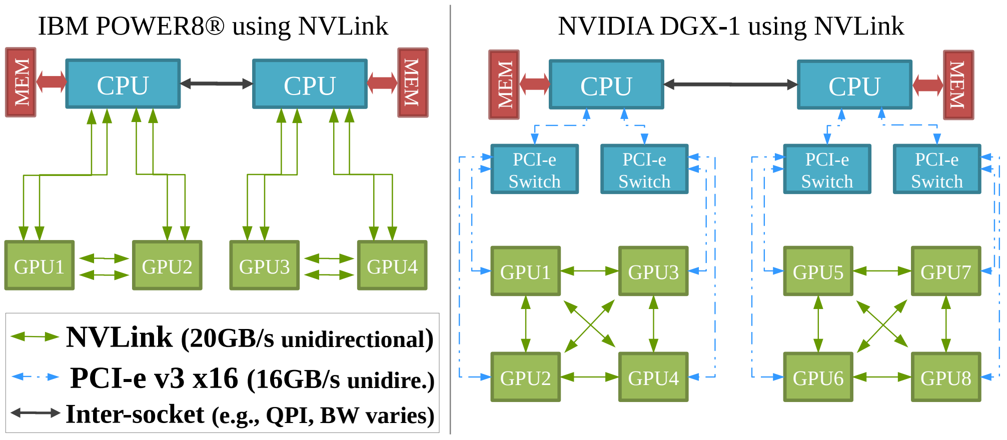

IBM POWER8® using NVLink
NVIDIA DGX-1 using NVLink
MEM
CPU
PCI-e Switch
GPU1
GPU2
GPU3
GPU4
GPU5
GPU6
GPU7
GPU8
NVLink (20GB/s unidirectional)
PCI-e v3 x16 (16GB/s unidire.)
Inter-socket (e.g., QPI, BW varies)

Figure 1: Examples of GPU physical topology over two representative NUMA systems.

Nonetheless, the underlying hardware topology is paramount for determining the application performance. To illustrate this issue, consider Figure 1 that shows the connectivity topology between the GPUs and CPUs for two representatives systems with Non-Uniform Memory Architecture (NUMA) topology. In these systems, communications can take place in the same CPU domain, or across domains with a penalty in the latency and bandwidth. As a result of these complex connectivity topologies, the application performance depends on which resources are allocated for computations and how they are connected.

Therefore, the potential to fully exploit the resource-efficiency in virtualized and disaggregated datacenters is conditional on an intelligent system making informed decisions to orchestrate the resources with a high-level view of the system. Among these decisions, those made by the scheduler and the resource allocator to select which application to run next among those in the wait queue and decides which resources to allocate for running them, are particularly important for ensuring high levels of system performance. Poor decisions can lead to poor resource usage and in consequence poor performance of critical workloads. Therefore, organizations demand management solutions able to take the communication requirements of the workloads, consider the topology of the system, to provision resources for the new workload meeting the workload requirements. An intelligent scheduler also enables users to get access to the resources necessary without worrying about the detailed topology of the underlying hardware.

In this thesis, we want to demonstrate that ***it is possible to develop resource management strategies that optimize the performance of both high-performance and cloud-native workloads for virtualized and disaggregated datacenters***.

To achieve this goal, three incremental research challenges have been addressed: i) understand the performance impact caused by both the virtualization and the underlying hardware characteristics; ii) efficiently managing the application scheduling observing completion time goals and hardware characteristics; and iii) adequately allocating resources to each application in a flexible way. Given the scale and complexity of modern datacenters, all of those items should be implemented in an automated manner.

1.2 CONTRIBUTIONS

The focus of this thesis is to improve datacenter scalability, by increasing resource-efficiency on virtualized environments via smart management. To archive that we:

C1 Evaluate and characterize the performance of applications running on *virtual environment* (i.e., containers or virtual machines) over complex hardware architectures;

C2 Propose and evaluate new topology-aware multi-GPU high performance artificial intelligent (AI) workload scheduling techniques that maximize the overall performance while minimizing the quality-of-service violations; and

C3 Propose and evaluate a novel automatic workload orchestration for pooled resources and disaggregated architectures capable of improving resource utilization across servers.

To this end, the combination of our proposed methods creates a new range of options to increase datacenter-wide utilization (C3), by improving cluster management, while performing best-efforts on guaranteeing that each scheduled application has its performance requirements satisfies (C1 and C2). Figure 2 summarizes the main contributions

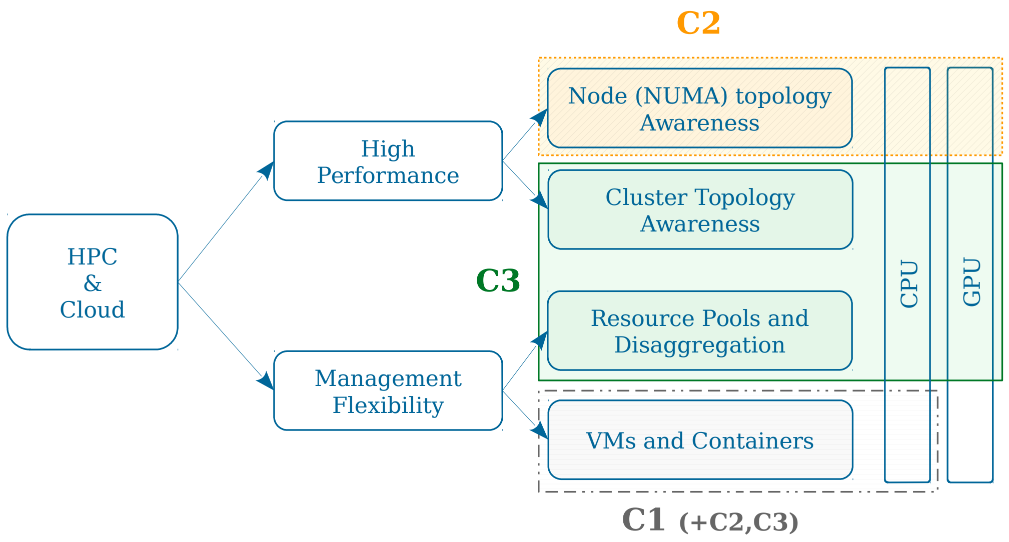

C2
Node (NUMA) topology Awareness
High Performance
Cluster Topology Awareness
HPC & Cloud
C3
Resource Pools and Disaggregation
Management Flexibility
VMs and Containers
CPU
GPU
C1 (+C2,C3)

Figure 2: Summary of contributions. The first contribution (C1) is an in-depth performance evaluation of *virtual environment* and NUMA architecture. The second contribution (C2) is new topology-aware multi-GPU high performance AI workload scheduling techniques. The third contribution (C3) is a novel automatic workload orchestration for pooled resources and disaggregated architectures. Contribution C1 is incremental to C2 and C3.

of this thesis, targeting the scope of virtualized datacenters within HPC and cloud environments. Note that, both the second contribution (C2) and the third contribution (C3) encompass the scope of the first contribution (C1) by using the *virtual environment* in the evaluation.

Next, we further detail the three contributions of this thesis.

1.2.1 *[C1] Performance Characterization of Containerized and Accelerated Workloads*

The execution stack of a virtualized application involves many components and middleware that directly impact on the overall performance. Giving that in combination with the hidden hardware complexity because of virtualization, determine the application performance is not an easy task. Moreover, in a shared environment, the problem becomes even more acute, since resource sharing can introduce undesirable interferences. Thence, in-depth knowledge of the virtualization layers that compose the execution stack and the underlying hardware characteristics is paramount to characterize the performance of an application in such an environment.

The **first contribution** of this thesis is the performance characterization of workloads running workloads running over virtualized environments and NUMA topologies over different configurations. Its novelty resides in the fact that we developed tools and methods to measure the performance of applications running on top of virtualization technologies over servers composed by NUMA architectures. We start the study evaluating the performance of applications running on different *virtual environments*. The evaluation exposes the performance of applications running on top of OS container or VM technologies; it targets the most fundamental components of the analysis. The monitoring is done collecting the status from the processor hardware counters in addition to the application completion time. Later, we execute the containerized applications on different scenario varying the placement and resource allocation.

As shown by experiments, the performance of virtualization via OS-level containers is almost as efficient as running directly in a bare-metal machine. Additionally, the results also show that a sub-optimal resource allocation on NUMA topologies can introduce a performance penalty and that not only the performance of threads running on the processors are impacted, but also the performance of tasks running on GPUs. In further analysis, the experiments also show that a smarter resource allocation on a multi-GPU system can improve the performance in ≈30%. These performance monitoring models and methods developed in the first contribution were extensively applied throughout this thesis, and in that sense, the second and third contributions are incremental to the first.

The work performed in this area has directly resulted in the following publications:

[3] Marcelo Amaral, Jordà Polo, David Carrera, Iqbal I. Mohomed, Merve Unuvar, and Malgorzata Steinder. "Performance Evaluation of Microservices Architectures Using Containers." In: 2015 IEEE 14th International Symposium on Network Computing and Applications. 2015, pp. 27–34. Cambridge, MA.

[122] Shuja-ur-Rehman Baig, Marcelo Amaral, Jordà Polo and David Carrera, "Performance Characterization of Spark Workloads on Shared NUMA Systems," 2018 IEEE Fourth International Conference on Big Data Computing Service and Applications (BigDataService), Bamberg, 2018, pp. 41-48.

1.2.2 *[C2] Topology-Aware Multi-GPU High-Performance AI Workload Scheduling*

Recent advances in the theory of neural networks (NNs), new computer hardware such as GPUs, availability of training data, and the ease of access of resources through cloud have allowed deep learning (DL) to be increasingly adopted as a part of business-critical processes in healthcare, autonomous vehicles, natural language processing, and internet of things. Consequently, many online platforms that offer image-processing and speech-recognition systems leveraged by trained DL NNs are emerging to deliver various business critical services, such as IBM Watson [60], Microsoft Project Oxford [97], Amazon Machine Learning [7], and Google Prediction API [45].

Although training on multiple GPUs can deliver many advantages, it presents new challenges in workload management and scheduling for obtaining optimal performance. The performance depends on both the GPUs and CPUs connectivity on the physical topology, as discussed in section 1.1 and illustrated in Figure 1. Jobs in this environment have varied GPU requirements: some need a single GPU, some need GPUs with NVLink, others require multiple GPUs, but communication requirements are minimal, etc. In such environments, the scheduler should be able to take the communication requirements of the workloads, consider the topology of the system, consider existing applications and their GPU and link utilization and provision the GPUs for the new workload that meet the workload requirements. This enables users to get access to the resources necessary without worrying about the detailed topology of the underlying hardware. Furthermore, both cloud and HPC systems can benefit from GPU topology-aware scheduling techniques.

The **second contribution** of this thesis is an algorithm with two new scheduling policies for placing GPU-based workloads in modern multi-GPU systems. The foundation of the algorithm is based on the use of a new graph mapping algorithm that considers the job's performance objectives and the system topology. Applications can express their performance objectives as Service Level Objectives (SLOs) that are later translated into abstract Utility Functions. As shown by experiments, the proposed algorithm presents the performance improvements that topology-aware scheduling confers for DL workloads using multiple GPUs. The results show a speedup of up to ≈1.30x in the cumulative execution time and no SLO violations compared to greedy approaches.

The work performed in this area has resulted in the following publications:

[5] Marcelo Amaral, Jordà Polo, David Carrera, Seetharami Seelam, and Malgorzata Steinder. “Topology-aware GPU Scheduling for Learning Workloads in Cloud Environments.” In: Proceedings of the IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis. SC ’17. 17:1–17:12. Denver, Colorado: ACM, 2017.

[4] Marcelo Amaral, Yurdaer N Doganata, Iqbal I Mohomed, Asser N Tantawi, Merve Unuvar, "Selecting Resource Allocation Policies and Resolving Resource Conflicts", Patent US9697045B2. Publication date: Mar 24, 2015. Granted data: Jul 04, 2017.

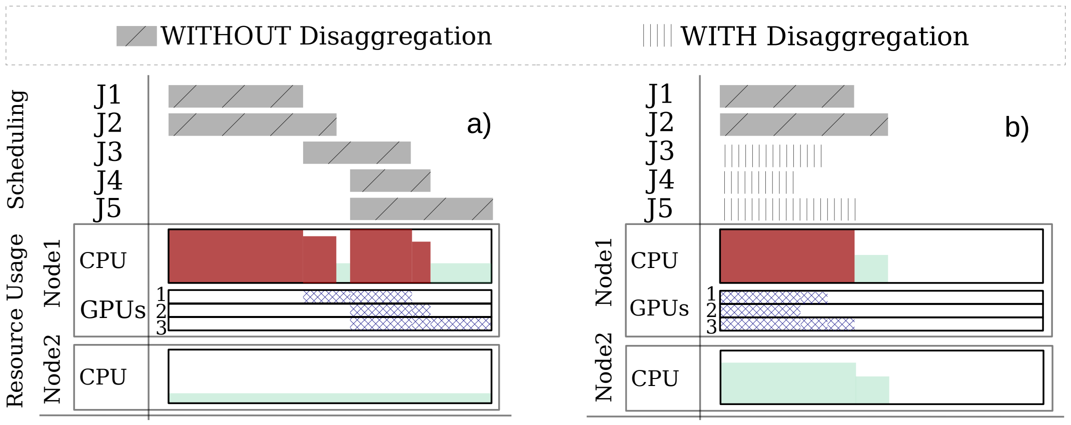


Figure 3: Scheduling examples: WITH versus WITHOUT support of resource disaggregation. When resource disaggregation is enabled, the waiting jobs can have their placement advanced, increasing the resource utilization and decreasing the total job makespan.

1.2.3 *[C3] Workload Orchestration for Pooled Resources and Disaggregated Architectures*

Traditional datacenters consist of monolithic building blocks that tightly integrate a small number of resources (i.e., CPU, memory, storage, and accelerators) for computing tasks of the system software and applications. 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 spare resource fragmentation [39, 71, 112].

For instance, if a CPU-bound application saturating 100% of processor cores uses little or no GPU in the system, the available GPUs cannot be allocated to other workloads due to lack of processing resources. In such a scenario, even though a clever orchestrator could assign a workload to a proper computer node, resource fragmentation still exists because one workload cannot allocate resources from different nodes. More specifically, Figure 3 illustrates this issue showing a scenario of two jobs consuming all the memory of the node #1, which has spare GPUs. This placement is preventing the scheduling of queued jobs that require GPUs. However, in the second scenario, GPU disaggregation is enabled, then, the waiting jobs can get CPU and Memory from node #2 and remotely access the GPUs from node #1. This process not only increases the system resource utilization but also decreases the job make-span.

Consequently, both industry and research communities have been concentrating efforts on enabling the shift from a mainboard-as-a-unit only paradigm to a more flexible software-defined block-as-a-unit approach [52, 62, 63, 112, 121]. In such a Software Defined Environment (SDE) [83], the control and management planes are decoupled from their data planes so that they can be deployed anywhere within the data center. More specifically, resources can be disaggregated and treated as a pool of resources that can be remotely accessed from other servers. This increases the orchestration flexibility since allocating disaggregated resources enables fine-grained sharing and efficient provisioning of the cluster resources across multiple applications. This flexibility also improves the operation efficiency in a datacenter, which is of the topmost concern to cloud providers because the overall economics ultimately determines their business competitiveness and the return on investment [70].

To that end, the **third contribution** of this thesis is a flow-network-based framework that orchestrates disaggregated resources on cloud systems employing best-efforts on preventing SLO violations while maximizing the system utilization. The framework is called disaggregated resource maestro (*DRMaestro*), and its main idea is to automatically discover and allocate disaggregated resources in the cluster for a job as if the resources are attached to the local machine that the job is placed. For the job standpoint, it is only using local resources. For that reason, building *DRMaestro* poses several interesting and challenging system problems, such as, how to: 1) enable transparent resource disaggregation, 2) automatically control and determine the optimal placement, and 3) cope with sharing-induced performance interference.

The work performed in this area has resulted in the following publication:

[submitted and under review] Marcelo Amaral, Jordà Polo, David Carrera, Nelson Gonzalez, Chih-Chieh Yang, Alessandro Morari, Bruce D'Amora, Alaa Youssef, Malgorzata Steinder, "DRMaestro: Orchestrating Disaggregated Resources on Heterogeneous Cloud Systems."

1.3 THESIS ORGANIZATION

The rest of this thesis is organized as follows: Chapter 2 introduces some basic concepts about virtualization and their associated technologies; about the concepts of how the in-node underlying topology interconnects CPUs, memory and other devices; and about different architectures to enable resource disaggregation. Chapter 3 presents a detailed analysis of virtualized environment via combining 1) a rigorous performance analysis of virtualization technologies, with 2) a detailed evaluation of what extent the underlying hardware topology and the network load can impact on the performance of different applications. Chapter 4 introduces and evaluates a new topology-aware multi-GPU scheduling algorithm, executing experiments with learning workload over different scenarios and configurations. Chapter 5 presents and evaluates a new workload orchestration for pooled resources and disaggregated architectures, using a flow-network-based scheduling algorithm to decide when and how to allocate disaggregated resources. Finally, Chapter 6 presents the conclusions and the future work of this thesis.

2 BACKGROUND

The fundamental concepts used during the elaboration of the other chapters are briefly introduced in this chapter and further described and discussed in the other chapters along this thesis. We first describe the concept regarding virtualization, since the thesis focus on virtualized datacenters. Then, we detail the Non-Uniform Memory Architecture, which plays the key role in all methods developed in this thesis to improve workload's performance. Last, but not least, we introduce the basic concepts about resource disaggregation, which is at the end, in our case, an extended virtualization technique.

2.1 DATACENTER VIRTUALIZATION

The emergence of commodity-off-the-shelf computers with high processing power, fast network connections, advanced accelerator technologies (e. g., Graphics Processing Units (GPUs) and Field-Programmable Gate Array (FPGAs)), Linux OS and virtualization were fundamental for the rising of cloud computing, enabling the access to a shared pool of configurable and often virtualized resources typically billed on a pay-as-you-use. The key technology that compounds the cloud build-blocks is the virtualization, which plays an essential role in improving the datacenter's resource-efficiency. Virtualization has been shifting away from traditional datacenters toward a software-based architecture with the focus on flexibility and customization. While in the past, virtualization technologies were mostly exclusive to off-premise cloud computing environments, currently, the HPC community has recently started to offer virtualization into their on-premise datacenters (*HPC as a service - HPCaaS*[1]), even though they have been apprehensive about the security and performance in the beginning. Although not all kind of HPC applications can benefit from a public cloud yet, there still exist various applications that can utilize the cloud facilities [48, 51, 68, 147], most especially the loosely coupled ones that are less latency-sensitive. However, with the increase of microservices, and technologies for DevOps, transforming existing HPC applications into Software-as-a-Service (SaaS) can become a trend to make virtualization in HPC more popular [104]. A concrete example of that is the Amazon EC2 HPC cloud cluster becoming part of the HPC TOP500

1 IBM supports HPCaaS from Rescale to deploy HPC jobs on the IBM Cloud

ranking; in 2011 it was in 42nd position [6]. In this direction, virtualized datacenters are becoming an attractive option for various kind of applications including cloud and HPC, by providing cost-effective and resource-efficiency solutions. Next, we detail the two leading technologies used to create *virtual environments*: i) virtual machines (VMs) and ii) OS containers.

VMs are a widely used building block of workload management and deployment. They are heavily used in both traditional data center environments and clouds (private, public and hybrid clouds). The commonly used term VM refers to server virtualization, which can be accomplished via full virtualization or paravirtualization. In paravirtualization, differently from full virtualization, the guest OS is aware that it has been virtualized and can provide directly communicate with the host (hypervisor) using the drivers. While in the past, VM was the default technology used to create *virtual environments*, in recent years, there has been a resurgence of interest on OS container technology, which provides a more lightweight mechanism. OS containers are operating-system-level virtualization under Linux kernel that can isolate and control resources for a set of processes. Because a container does not emulate all the physical hardware component and the guest OS as the VMs do, it is lightweight (consuming fewer resources) and presents fewer performance overheads. A single server can reasonably have 100s of containers running and memory usually ends up being the scarce resource; moreover, containers start up very quickly - under 1 to 2 seconds in most cases.

The core of container technology relies on Linux namespace [73] and cGroups [74]. The former is an abstraction that wraps a set of processes appearing that they are an isolated instance. Linux namespace isolates the set of filesystem mount points seen by the group of processes. cGroups organize the processes in a hierarchy tree; they also limit, police and account the resource usage of the process group. One can run a single application within a container whose namespaces are isolated from other processes on the system. Notwithstanding, the main capability of a container is to allow to run a complete copy of the Linux OS within it without the overhead of a running hypervisor. Although the kernel is shared, containers have limited access to the modules and drivers that it has inherited. Some examples of the currently available container technology are: OpenVZ [108], Rocket [124], Docker [29], Singularity [130], Resource containers [9] among others.

While the concept of operating system level virtualization is not new (e.g., chroot and jails in BSD), there has been a great deal of industry interest in Linux containers and Docker [29] implementation in particular. There are many reasons for this resurgence but from a technical perspective, two of the biggest reasons are (i) the improvements in namespace support in the Linux kernel available in popular distributions, and (ii) a specific implementation of containers - Docker - that has successfully created an attractive packaging format, useful tools, and diverse ecosystem.

In recent years, there have been significant advances in virtual networking features in Linux. Some notable mechanisms include network namespaces, veth pairs, tap devices as well as virtual switches such as OpenvSwitch and Linux Bridges. The mechanisms can be used to provide a high degree of flexibility and control of networking and form the basis of Software Defined Networking (SDN) technologies such as OpenStack Neutron [18, 109], Calico [17], Romana [125], Flannel [23], Weave Net [142], etc. For instance, a physical host might have just one physical network interface, while a number of guest containers can run in isolation by having their own network namespaces, with tap devices wired into an OpenvSwitch. Moreover, one could set up tunnels between the OpenvSwitch instances on different machines and can enable communication between instances. In practice, there are many ways of setting up virtual networking, with varying effects on throughput, latency and CPU utilization. Currently, HPC networks using RDMA can be performed within containers, but with limitations, only the containers configured with host networking can use RDMA [136].

For facilitating the management of container in a large-scale cluster, many resource management frameworks have been proposed during the last years. Kubernetes provides lightweight, simple and self-healing services management, and we describe it further in Section 2.1.1. Mesos [94] is another open source project that is intended to be an operating system for a datacenter. It is designed to manage the different type of workloads via a hierarchical resource management solution that increases the scalability and reduces the latency due to resource scheduling [53]. Docker Swarm [28] is a built-in framework created by the Docker community, that natively manage a cluster of Docker engines. However, by the time of this thesis, Kubernetes and Docker containers were the most popular, widespread and mature approaches. Additionally, many big companies (such as IBM, Amazon, Google, Microsoft, Huawei, among others) were endeavoring a lot of efforts to contribute to the Kubernetes' development. Then, because of those reasons, this thesis focused on using Docker and Kubernetes in the experimental evaluations.

2.1.1 *Kubernetes*

Kubernetes is an open source container orchestration platform targeting large-scale cloud-enabled workloads. It was initially proposed by Google engineers[2] to deploy, scale, and manage containerized applications and heavily influenced by Google's Borg system [16, 139]. Kubernetes uses a set of primitives to deploy, scale and manage containerized ap-

2 In 2015 Kubernetes was donated to the Cloud Native Computing Foundation.

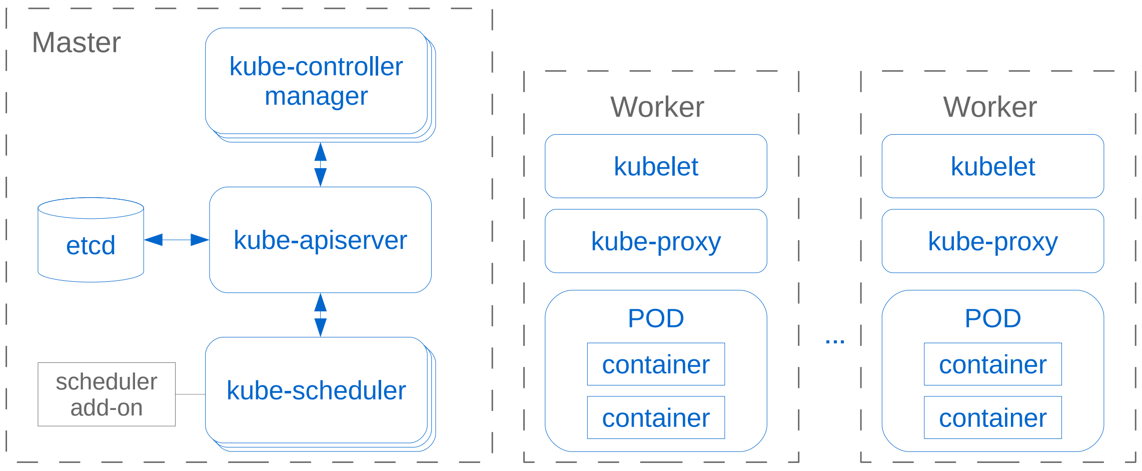


Figure 4: Kubernetes architecture. The master node can be composed by a built-in scheduler or an add-on one. The workers start pods that can contain one or more containers.

plications. These primitives are extensible to accommodate a wide variety of workloads. The basic unit of scheduling is called a "pod" which can be composed of one or more containers. A pod is considered a single unit and cannot span over multiple server nodes. Therefore, an application running over multiple nodes will be composed of multiple pods. Kubernetes also provides the capability to use key-value pairs to label pods and other system resources, which can later be used in the scheduler. For instance, a node composed of specific resources such as NMVe or GPUs can contain labels to identify the resources, and the scheduler can filter the available nodes based on the labels.

The Kubernetes architecture is a master-slave configuration as depicted in Figure 4. The Master uses an etcd key-value store to maintain information about the state of the entire cluster. An API server provides an HTTP interface for both internal and external access to the Kubernetes master. It processes REST or Protocol Buffers (Protobuf) requests and updates the etcd data store. The Scheduler selects the node that an unscheduled pod should run. The built-in scheduler knows about resource requirements, availability, affinity, etc. to make scheduling decisions. However, another specialized scheduler can be provided as an add-on plugin. In the worker, the Kubelet is responsible for discovering the resources and keeps track of running state on each node and relays that information to the Kubernetes master. Kubelet is also responsible for managing the pods and starting the containers.

Kubernetes has a controller that manages the state of the cluster and provides mechanisms to replicate and scale multiple copies of a pod across the cluster of server nodes. A set of pods that work together is called a Service in Kubernetes. Services are exposed within a cluster and are assigned an IP address and the Domain Name System (DNS) name so it can be discovered by other Kubernetes applications and services. Kubernetes also has the concept of Job, which differs from the server by running up to its completion, while a service never finishes. More specifically, a job creates one or more pods and ensures that a specified number of them successfully terminate [22].

For monitoring the resources, Kubeletes typically uses Heapster, which discovers all nodes in the cluster and queries usage information from the kubelets agents. The kubelet itself fetches the data from cAdvisor (container Advisor), which is a running daemon that keeps historical resource usage and network statistics for each container.

2.2 IN-NODE UNDERLYING NUMA TOPOLOGY

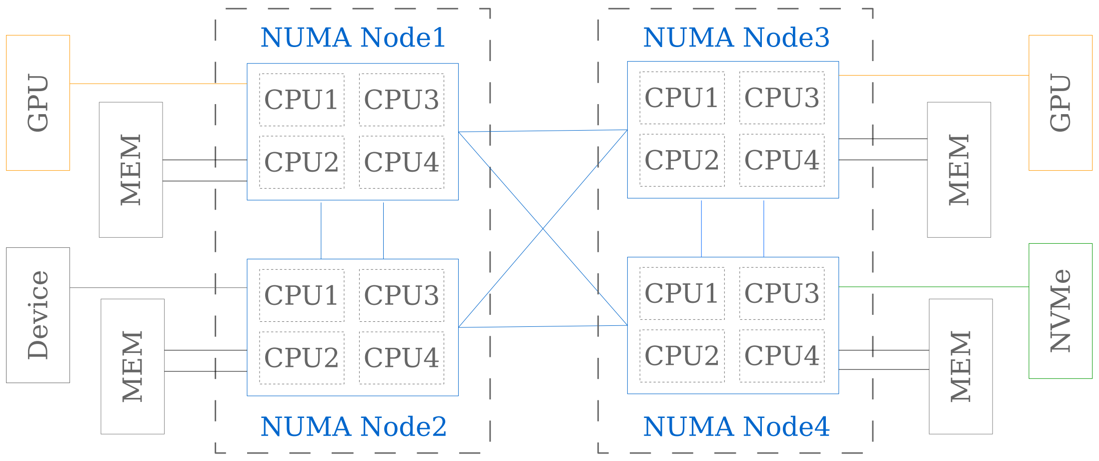


Figure 5: Non-Uniform Memory Architecture (NUMA) topology example: memory, CPU, GPUs and other devices can be interconnected via different paths and technologies.

Due to the growth in the number of cores in modern processors, parallel systems are built using NUMA, which has gained wide acceptance in the industry, setting the new standard for building new generation enterprise servers. These processors can be connected to large amounts of physical memory, in the range of up to a couple of terabytes for the time being. This opens an enormous range of opportunities for runtimes and applications that aim to improve their performance by leveraging low latencies and high bandwidth provided by RAM. The result is that today there are several examples of applications that have started pushing the *in-memory computing* paradigm to accelerate tasks.

To deliver such a large physical memory capacity, sockets in NUMA systems are connected through high-performance connections, and each socket can have multiple processors with its memory, as illustrated in Figure 5. A process running in a NUMA system can access the memory of its node as well as the remote node where the latency of memory accesses on remote nodes is significantly high compared to local memory accesses [11]. Ideally, memory accesses are kept local to avoid this additional latency and contention on interconnect links. Moreover, the bandwidth of memory accesses to last-level caches, and DRAM memory also depends on the access type that is local or remote. Later on, we also discuss that NUMA not only impacts that intercommunication performance of tasks running on CPUs from different NUMA domains but also on other devices such as GPUs. As we can see in Figure 5, a process running on NUMA node #1 can access the GPU on Node #3 passing through intra and inter-node connections. But note that the communication between devices on different NUMA domains depends on how they are connected, which can be subject to additional overheads.

2.3 RESOURCE DISAGGREGATION

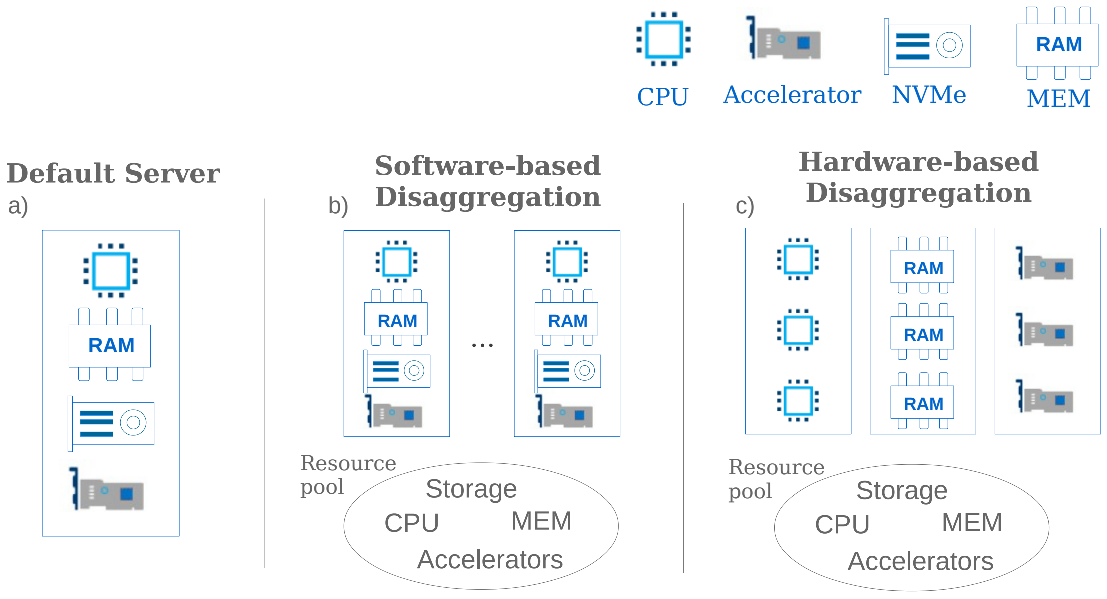


Figure 6: Server architectures examples. a) Default architecture without resource disaggregation. b) Software-based resource disaggregation using an API to expose remote resources. c) Hardware-based resource disaggregation with a rack composed by modules with several homogeneous resources which are specialized in being rich in one type of resource.

By default, today's datacenters consist of servers built as monolithic building blocks tightly integrates with a small number of resources (i.e., CPU, memory, storage, and accelerators) for computing tasks of the system software and applications, such as in Figure 6 a). However, the main flaws of such server-centric architecture are the shortage of resource provisioning flexibility and agility. In particular, the resource allocation within the boundary of the mainboard leads to spare resource fragmentation [39, 71, 112]. To illustrate this issue, consider Figure 3 (a), where two CPU-bound jobs are saturating 100% of processor cores from node #1, and the GPUs available in this node cannot be allocated to other jobs due to lack of processing resources. In such a scenario, even though a clever orchestrator could assign a workload to a proper computing node, resource fragmentation still exists because a waiting job cannot allocate the GPU resources from node #1. However, if the idle GPUs from node #1 could be remotely exposed to node #2, then the waiting jobs could start running.

Industry and research communities have been concentrating efforts on enabling the shift from a mainboard-as-a-unit only paradigm to a more flexible software-defined block-as-a-unit approach [52, 62, 63, 112, 121]. In such a software-defined environment [83], the control and management planes are decoupled from their data planes so that they can be deployed anywhere within the data center. More specifically, resources can be disaggregated and treated as a pool of resources that can be remotely accessed from other servers. This increases the orchestration flexibility since allocating disaggregated resources enables fine-grained provisioning decisions and efficient sharing of the cluster resources across multiple applications. Disaggregated resources can simplify the data-center management complexity by allowing a high-level of customization.

Systems within a disaggregated (or composable) data center are re-factored so that the subsystems can communicate via a network as a single system; resources are pooled together and provisioned independently [86]. A composable architecture at the cluster level might contain both server-centric and rack scale architecture, where the later can be a software-based or hardware-based architecture. For example, Figure 6 shows the three different configurations: the server-centric architecture as the default one composed by a small amount of each resource in a single box; b) the software-based resource disaggregation architecture is composed by several server-centric servers, but their resources can be remotely accessed from other servers as the resources are locally available using a software API; and c) a rack-scale is a box composed of modules specialized in being rich in one type of resource that can be remotely exposed and combined to create virtual servers. Both architectures in Figure 6 b) and c) introduce the concept of resource virtualization offering disaggregated resources, and at a high-level view, they show a pool of resources that can be accessed as their resources have no boundaries.

The disaggregated resources can be managed through a centralized or a distributed model and accessed by application programming models through hardware based, hypervisor/operating system based, or middleware based approaches as discussed in [84]. In this study, we focus on the software-based approach. In Chapter 5, we present our proposed orchestrator that can allocate disaggregated resources establishing the channel between the host and the resource (e.g., GPU) through RDMA/PCIe and exposes the access to jobs via a library, such as HFCUDA that is detailed in Section 5.4.

3 PERFORMANCE CHARACTERIZATION OF VIRTUALIZED WORKLOADS

The execution stack of a virtualized application involves many components and middleware that directly impact the overall performance. Giving that in combination with the limited view of the topology complexity because of virtualization and resource disaggregation, determining the application performance becomes a complex task. Therefore, in this chapter, we execute a performance analysis of the virtualization layers and the underlying hardware characteristics to characterize the performance of applications in such environments. This evaluation will also help to illustrate some of the key motivating factors behind the proposed scheduling techniques that we will present, discuss and evaluate over this thesis. Additionally, the performance evaluation in this first contribution provides the necessary information for defining the performance model of applications in the experiment conducted on the other chapters. More specifically, the experiments performed in this chapter quantifying the performance impact of applications running in their best-performing scenarios, and also in other situations with different configurations, resource allocation, and application collocation.

3.1 PERFORMANCE OF OS CONTAINERS

In this experiment, to widely evaluate the virtualization environments, we analyze (i) the computing performance, (ii) the overhead in the startup/creation process and (iii) the network performance of a *virtual environment* (i.e., containers or virtual machines), in three different configurations. The **first configuration** is a simple OS container running directly on the host, and we call this configuration as *regular-container*. The **second configuration** is a container running inside another container, which we call *nested-container*. This configuration illustrates the scenario of a cloud provider offering containers as an infrastructure-as-a-service (such as discussed in [16]), and then, a given user starts its own containers into this given *virtual environment*. The primer container offered by the cloud provider, we will call parent-container and the other containers created inside the parent-container, we will call child-containers. Finally, the **third configuration** is using a *virtual machine*. For these experiments, the VMs and containers are created using KVM and Docker technologies, respectively, and we use the default configuration for all the configurable parameters.

3.1.1 *Premises, limitations and other discussions*

The goal of these experiments is to show the overhead and performance impact when comparing the default configurations of *virtual environment*, which are more commonly used by the end users in Cloud scenarios. Therefore, we do not aim to show in these experiments the performance impact of each setting that can fine-tune the virtualization performance of Virtual Machines or Containers, such as configuring the parameters for processor pinning, driver by-passing, NUMA-awareness, etc.

Moreover, by the time that these experiments were conducted in this thesis - 5 years ago -, OS container was a relatively new technology and measuring the performance impact, and overhead of containers was an import research contribution. Nowadays, the understanding of the container's performance and its advantages are much more mature and widespread. Therefore, in these experiments, we summarized only the key advantage that we measured via experimentation. It is worth noting that the experiments conducted in this chapter are still relevant since they compare the basic configuration of OS containers and KVM virtual machines. The experiments show the overhead and performance impact when comparing the default configurations, giving the baseline to understand better the performance implications.

3.1.2 *Evaluation infrastructure*

The machines that are used to run the experiments are two 2-way Intel Xeon E5-2630L, each one composed by 2 sockets, 6 cores per socket, 2 hyper-threads per core at 2GHz, 64GB of RAM and 1Gbps Network Interface Card (NIC). They are Linux box running Ubuntu 14.04 (Trusty Tahr) with Linux Kernel 3.13 Those machines are in the same rack and are connected with 2 stacked Gigabit Cisco Switch model 3750X with 48port each, connected through StackWise+ connector. For containers, we used Docker 1.0.1, while virtual machines were provided by KVM 2.0.0 configured with Intel Virtualization Technology (VT-x) and network Gigabit mode (virtio). Experiments focused on CPU are based on the Sysbench benchmark [78, 134] version 0.4.12. Experiments focused on network use Netperf version 2.6.0, and the machines are configured with OpenvSwitch version 2.0.2, and the Linux Bridge version natively available in Ubuntu 14.04. All the programs were compiled using gcc version 4.8.2 and Python 2.7.6. The configuration of the testbed is summarized in Table 1.

Table 1: Testbed configuration

	Version/Model
Processor	Intel Xeon E5-2630L
Sockets	2
Cores per socket	6
Hyper-threads	2
Frequency	2GHz
Ram	64GB
Network bandwidth	1Gbps
Docker version	1.0.1
KVM version	2.0.0
Sysbench version	0.4.12
Netperf version	2.6.0
OpenvSwitch version	2.0.2

3.1.3 *CPU Performance Evaluation*

In order to evaluate the computing performance, we used Sysbench [134] running its CPU benchmark, where each request calculates prime numbers up to a certain value specified by the `cpu-max-primes` option, in this experiment, this is set as 40,000; all calculations are performed using 64-bit integers. In the experiments, we measured the completion time and ranged the number of concurrent Sysbench instances from 1 to 256. We run Sysbench in all *virtual environment* configurations described before plus in a *host* configuration in which we execute it directly on the bare-metal machine to be the performance baseline. In the *regular-container* configuration, we execute multiple containers, each one running a single Sysbench instance. When using the *nested-container* configuration, we execute multiple child-containers running a Sysbench instance each. Lastly, Sysbench is also executed within *virtual machines*.

3.1.4 *Overhead on the Computing Resource*

In this experiment, no resource constraints are set for containers or virtual machines, so the scalability is expected to grow linearly with the number of available CPU cores, and each configuration was executed ten times.

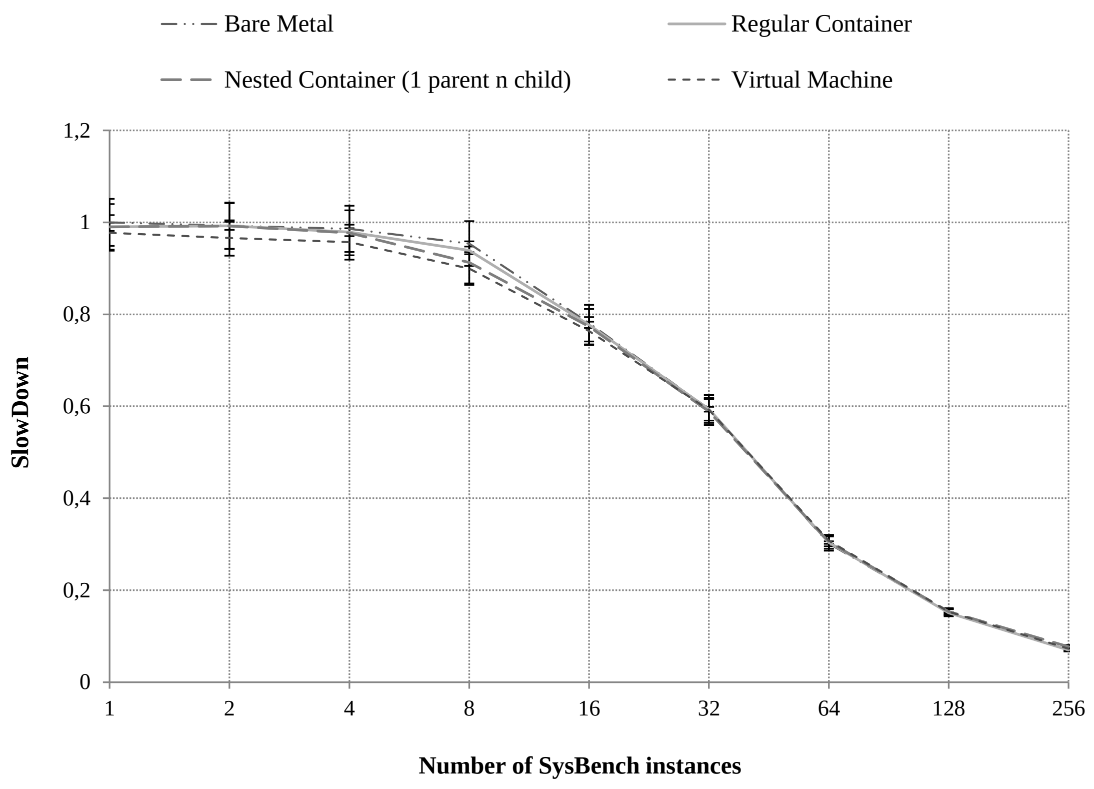


Figure 7: Observed slowdown of Sysbench with increasing number of instances relative to running a single Sysbench instance in bare-metal.

The results of this experiment are illustrated in Figure 7, which shows the observed slowdown of running Sysbench when increasing the number of instances. In particular, the Figure shows the average of 10 executions for each one of the tested environments: bare-metal, regular containers, nested-containers, and virtual machines. The baseline to measure the slowdown is the execution time of one single Sysbench instance running in bare-metal. The results put in evidence the CPU performance when comparing containers versus bare-metal. In the tested configuration, the performance of the application regarding the CPU had no significant difference, and when comparing bare-metal versus VM, the performance had a very low impact of only ≈2%.

The low CPU overhead is thanks to the improved virtualization support for virtual machines in modern processors for efficiently enabling virtualization technologies and no additional software layers (i.e., overlays) for containers accessing the CPUs and OS drivers.

3.1.5 *Overhead of Virtual Environment Creation*

To evaluate the overhead in the *virtual environment* start-up/creation process, we measure the time to create 1 up to 256 concurrent *virtual environments*. Each *virtual environment* simply launches a dummy application that takes a negligible amount of time (in particular, we used Sysbench as in the CPU experiment, but configured with `cpu-max-primes` set to 1), effectively allowing us to compare creation times under different environments. For the case of *regular-containers*, we measure the elapsed time between starting the container up and exiting from it. For *nested-containers*, we measure the elapsed time between starting and exiting from the parent container, which includes loading a locally-stored child image as well as starting-up and exiting from the child container. Finally, for a virtual machine, we measure the time to create/start and delete a VM.

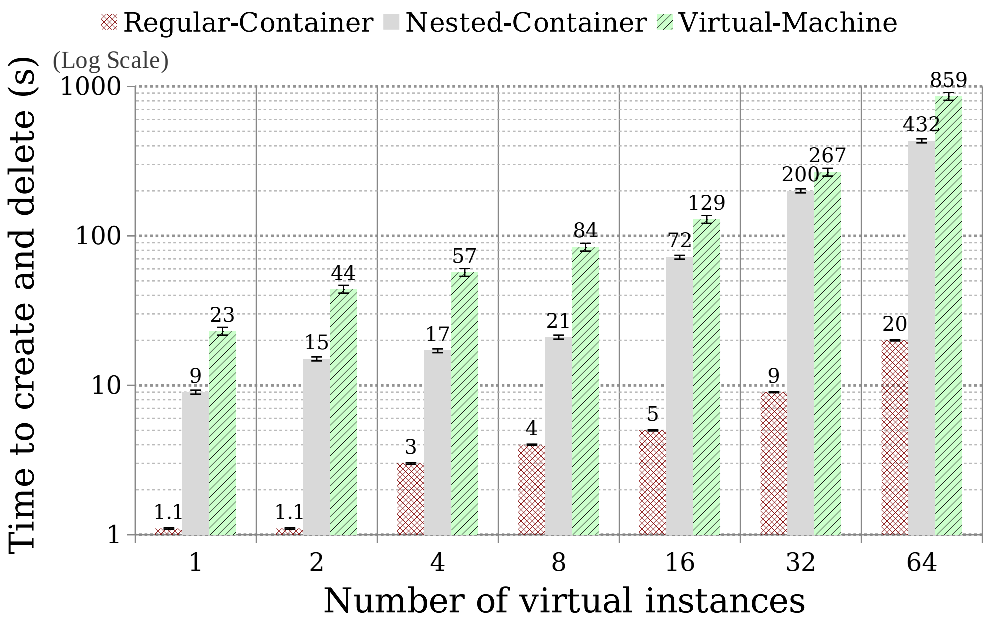


Figure 8: Time to create and exit from *virtual environment* of an increasing number from 1 up to 256 *virtual environment* instances.

The results of our experiments are illustrated in Figure 8, which shows the measured time to create a different number of instances under each configuration. As expected *regular-containers* are always the fastest approach, followed by *nested-containers* and virtual machines. While the creation of a single *nested-container* has almost eight times more overhead than the creation of one *regular-container*, the creation of *nested-containers* is still more than twice as fast as virtual machines. This additional overhead for *nested-containers* is related to the initialization of Docker in the parent container, which also involves loading an image stored locally in the host and the creation of the child container itself. Additionally, when creating more than eight *nested-containers*, the overall creation time seems to increase more than linearly and becomes a lot closer to virtual machines than *regular-containers*. The host machine only has 12 cores, so the behavior when overloading the cores is significantly different. However, *nested-containers* are still twice as fast as virtual machines in most scenarios.

3.1.6 *Network Performance Analysis*

The goal of this experiment is to measure the network performance, with a focus on studying the communication overhead of different technologies when running in the same host.

To evaluate the network performance, we select the Netperf benchmark [69]. In particular, the Netperf tests used to evaluate performance are TCP Stream (`TCP_STREAM`) and TCP Request/Response (`TCP_RR`). `TCP_STREAM` is a simple test that transfers a certain amount of data from a client running `netperf` to a server running `netserver`. This test calculates the throughput and does not include the time to establish the connection. On the other hand, `TCP_RR` is a synchronous test that consists of exchanging requests and responses (transactions), and which can be used to infer one-way and round-trip latencies. In particular, `TCP_RR` executions were configured to run in burst mode to have more than one transaction at the same time, and the socket buffer size for connection data set to 256K. Throughput and round-trip latency per transaction were measured with Netperf under different *virtual environment* configurations and network virtualization technologies, such as Host-Network (directly using the native host-network stack), OpenvSwitch [110], and Linux Bridge. OpenvSwitch is configured only for routing packets, and there is no additional encapsulation, while Linux Bridge is combined with forwarding NAT iptable rules. By the time of this experiment, to create Docker containers with OpenvSwitch, an additional configuration was required, involving the creation of virtual interface *veth pairs*, binding one pair in the OpenvSwitch bridge and the other in the already running container.

The results are presented in 2 and Tables 3. All executions were repeated five times, and the table includes the average times and the standard deviations. In this experiment, it should also be noted that, even though the host is composed of a single 1 Gigabit network interface card, the throughput is higher than 1Gbps since client and server are running in the same host and the network packets are only going to the loopback interface. We do not focus on measuring the performance impact of overlay networks, that are typically used by containers and virtual machines to communicate across servers with virtual IPs and additional package encapsulations. We focus on showing the overhead within a single host. This allows us to focus the experiment on only analyze the virtualization overhead regarding the *virtual environment*.

Table 2: Network latency evaluation for different configurations of client/server under bare-metal, container and virtual machine on a single host machine, using the loopback interface. The first column shows where the Iperf client or server were executing.

	Latency		
(Client - Server)	Host-Network	Linux Bridge	Open vSwitch
Host - Host	102.77 µs σ=0.95	-	-
Container - Host	104.48µs σ=1.45	231.97µs σ=5.3	229.37µs σ=6.38
Host - Container	105.0µs σ=1.94	230.17µs σ=7.35	217.76µs σ=4.63
Virtual machine - Host	-	424.92µs σ=14.09	465.53µs σ=43.57
Host - Virtual machine	-	397.53µs σ=12.08	420.14µs σ=27.09

Table 3: Network throughput evaluation for different configurations of client/server under bare-metal, container and virtual machine on a single host machine, using the loopback interface. The first column shows where the Iperf client or server were executing.

	Throughput		
(Client - Server)	Host-Network	Linux Bridge	Open vSwitch
Host - Host	35.71 Gbps σ=0.32	-	-
Container - Host	35.13 Gbps σ=0.48	15.82 Gbps σ=0.36	16.01 Gbps σ=0.47
Host - Container	34.96 Gbps σ=0.63	15.96 Gbps σ=0.51	16.86 Gbps σ=0.35
Virtual machine - Host	-	8.64 Gbps σ=0.28	7.94 Gbps σ=0.69
Host - Virtual machine	-	9.24 Gbps σ=0.27	8.77 Gbps σ=0.55

As it can be observed in Tables 2 and 3, the network performance when using containers is generally higher than using virtual machines, and it can be as fast as bare-metal under certain configurations. For instance, when containers are configured with Host-Network, they basically present the same performance as bare-metal in terms of throughput and latency. On the other hand, when containers or virtual machines are configured with Linux Bridge or OpenvSwitch, there is a significant performance impact. Even though OpenvSwitch is supposed to achieve higher performance than Linux Bridge, as stated in [18], our results show that their behavior in terms of throughput and latency under these configurations is similar, and their performance is not significantly different, only achieving approximately half of the throughput, and almost twice as much latency. Finally, it should also be noted that even if virtual machines are accelerated to provide high computing performance, the network is still behind regarding performance when compared to containers, and it's approximately twice as slow, even when compared against the same network virtualization technologies. This is mostly because a VM emulates all the network stack in its guest OS and hypervisor, while containers have direct access to the host kernel.

Table 4: Network throughput and latency evaluation of nested-containers

Parent	Child	Throughput	Latency
Host-network	Host-network	33.74 Gbps σ=2.3	109.31μs σ=7.66
Host-network	Linux Bridge	16.06 Gbps σ=0.73	228.90μs σ=10.52
Linux Bridge	Host-network	15.56 Gbps σ=0.97	236.72μs σ=14.72
Linux Bridge	Linux Bridge	12.53 Gbps σ=0.75	293.84μs σ=18.45
Open vSwitch	Host-network	16.9 Gbps σ=0.44	217.25μs σ=5.63
Open vSwitch	Linux Bridge	12.19 Gbps σ=0.57	301.54μs σ=14.63

In addition to studying the performance of different virtualization environments, in Table 4 we also show the result of evaluating network throughput and latency of nested-containers under different combinations of network configurations for parent and child-containers. Unfortunately, in this experiment OpenvSwitch could not be used in the child-container due to privilege-related issues, so OpenvSwitch parent-containers are only compared against Host-Network and Linux bridge child-containers. However, as expected, when a parent or a child-container is configured with Host-Network, the performance is significantly better than with Linux Bridge or OpenvSwitch, which are once again approximately twice as slow concerning both, throughput and latency. While Host-Network provides higher performance, it also has certain security tradeoffs and scalability limitations. For example, when using Host-Network all containers will be in the same network namespace with no isolation; they will have the same IPs (only the host IPs); and they will also share the same socket (or verbs) ports and will not have any virtual-network isolation, such as the one provided by vxlan.

3.1.7 *Related Works of Performance Evaluation of Virtual Environment*

At the time that this thesis researched and evaluated the overhead impact of containers, microservices management had started to gain popularity. Google Cloud Platform implemented and opened sourced to a pre-production beta cluster management project

-Kubernetes- that can be used for better management of large scale microservices [44]. Kubernetes promotes container technology via microservices architecture. Therefore, our goal in this experimental evaluation was to execute a performance analysis for container technology in microservices architecture. At this time, to the best of our knowledge, our work was the first one to analyze performance and management overhead of microservices architecture using containers and nested containers.

At this time, the most closer related works were the evaluations comparing virtual machines and Linux containers. In the work [36], they showed that containers result in equal or better performance than VM in almost all cases, but both VMs and containers require additional tuning to support I/O-intensive applications. Other works have evaluated only the performance of virtual machines [58, 59] without considering Linux containers, they basically compared only the performance of applications running over hypervisors versus running directly in the bare-metal machine. In addition to that, there were also some works comparing virtualization and containerization in the standpoint of isolation with the cost of latency and overhead. Those works did a good job to show that containers bring less overhead along with similar but less isolation than virtual machines. Concerning the isolation capabilities of containers, [30] studied the performance of different container implementations such as Linux, Docker, Warden Container, Imctfy, and OpenVZ along with virtual machines. As for the trade-off between performance and isolation between containers and virtual machines, [145] examined some experiments for High-Performance Computing environments. Performance of network virtualization for containers was also well studied in the Linux literature [20]. The work in [16] has detail how Google has been managing their datacenters. The discussed that resource isolation provided by containers is not perfect, it still needs to be improved. They concluded that containers could not prevent interference in resources that the OS kernel does not manage (e.g., level 3 processor caches and memory bandwidth), and containers, by this time, need to be supported by an additional security layer (such as virtual machines). They also concluded that the benefit from containerization goes beyond merely enabling higher levels of utilization, but it transforms the data center from being machine-oriented to being application-oriented. That is since well-designed containers are scoped to a single application, managing containers means managing applications rather than machines, which has the potential to improve application deployment and introspection dramatically.

However, these works commented before did not evaluate the management overhead by comparing the virtual environment initialization time as we showed in our analysis. Additionally, these works did not evaluate the impact of multiple virtualization layers as we have evaluated in this work. More specifically, as stated before, we have also evaluated the scenario of a cloud provider offering containers as an infrastructure-as-a-service, where a user can start its own containers (i.e., the child one) into the container acquired from the Cloud provider (i.e., the parent one). Thus, the user's container would have multiple levels of OS containers virtualization. Therefore, in our experiments, we showed the impact of creating containers, and the CPU and network overhead in such environments.

3.2 NUMA IMPACT ON GPU-BASED LEARNING WORKLOADS

The main sources of performance perturbation on multi-GPU applications are how the allocated GPUs are connected, i.e., the NUMA topology, and how much of the shared bus bandwidth other applications are utilizing.

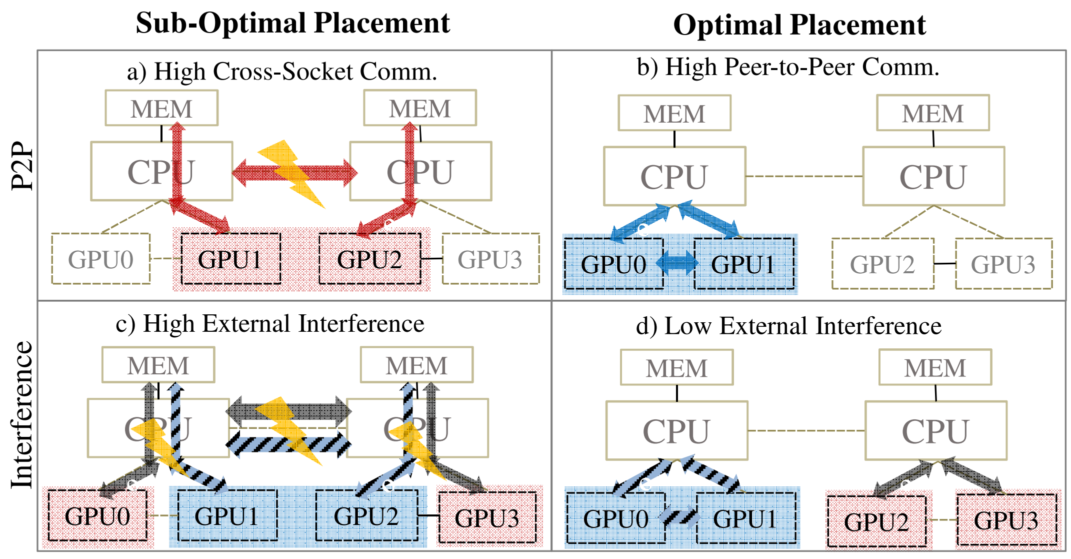

Sub-Optimal Placement
Optimal Placement
P2P
Interference
a) High Cross-Socket Comm.
b) High Peer-to-Peer Comm.
c) High External Interference
d) Low External Interference
MEM
CPU
GPU0
GPU1
GPU2
GPU3

Figure 9: *Pack* vesus *Spread* scheduling policies, and job collocation versus running solo.

To illustrate these issues, Figure 9 illustrates two different possible problems that might occur on top a single machine with hardware topology composed of two sockets and two GPUs per socket (e.g., the same topology shown in Figure 1 for the POWER8® system). For example, in the case an application has the communication more intensive in the CPU-to-GPU part, it benefits more from the scenario a). Otherwise, if it is more intensive in the GPU-to-GPU communication part, it will benefit better from being in scenario b). This behavior is because the GPUs within the same socket are located at a "shorter" distance (from a topology perspective) than the GPUs located across sockets. Besides, GPUs on the same socket can utilize the higher bandwidth and lower latency network (e.g., NVLink) to communicate instead of going over the PCI-e and the QPI links to communicate across CPU sockets. However, even though, two applications perform better when allocating GPUs from different socket domains, if they are co-collated, that is, sharing the inter-socket connection, they might perform better if they are "isolated" on different socket domains, as shown in Figure 9 c) and d).

Therefore, in this section, we evaluate two general purpose workload placement strategies: *pack* and *spread*. We define these policies as follows:

THE PACK POLICY systematically favors minimizing the distance between GPUs, to prioritize the performance of GPU-to-GPU communication.

THE SPREAD POLICY attempts to allocate GPUs from different sockets and prioritizes the performance of CPU-to-GPU communication.

Note that, the *spread* policy promotes better resource utilization by minimizing fragmentation where fragmentation means that the resource cannot be allocated consecutively in the same machine. To illustrate that, consider two different scenarios, the first we have four machines with four GPUs available on each, and another scenario with four machines with only one GPU available on each. Giving that the maximum number of available GPUs per machine is 4, we define here that only the second scenario presents fragmentation; in the first scenario, the GPUs are fully available.

Another factor that impacts the performance of either *pack* or *spread* placement schemes is the interference introduced by other applications sharing the system resources. For this reason, the placement algorithms should take not only the static topology of the system but also the runtime utilization metrics from currently executing applications for scheduling decisions. Later, in Section 4.1, we combine these policies into a utility function that is used by our proposed placement algorithm that allocates GPUs to jobs considering the underlying topology characteristics.

Next, we detail the Deep Learning workload used in the conducted experiments in this section and their characteristics that are relevant for topology-aware scheduling of applications using multiple GPUs; we also describe the testing platform used during the experiments, and we evaluate the impact of the placement strategies to allocate GPUs for the DL applications. All the experiments are performed with the applications running over Docker containers.

3.2.1 *Deep Learning Workload Description and Limitations*

With the increasing popularity of the DL methods, several deep learning software frameworks have been proposed to enable efficient development and implementation of DL applications. By the time that these experiments were conducted in this thesis, the list of available frameworks included, Caffe, Theano, Torch, TensorFlow, DeepLearning4J, deepmat, Eblearn, Neon, PyLearn, among others [8].

It is worth to mention that, by the time of this thesis research, the most known DL frameworks were Caffe, Theano, Torch, TensorFlow, DeepLearning4J, deepmat, Eblearn, Neon, and PyLearn [8], being Caffe the most popular one. Additionally, by this time, the most used neural network implementations were AlexNet, CaffeRef (based on AlexNet) and GoogLeNet, but our results are equally applicable to other frameworks and NN implementations. While each framework and NN develop different algorithms and try to optimize various aspects of training, they share similar GPU communication algorithms [141], which is the most significant part of the application in our analysis.

Even though, by the time of this thesis already existed different approaches to implement NNs in regards to divide the workload when using multiple GPUs: data-parallelization and model-parallelization, we have only focused on data-parallelization approach. This is because the model-parallelization approaches were uncommon for cloud deployments at the time of this research. The main difference between these two approaches is: in data-parallelization, the data is partitioned and spread to different GPUs, and in model-parallelization, the NN model is partitioned, and different GPUs work on different parts of the model, for example, each GPU will have different NN layers of a multi-layer NN.

Note that, model-based parallelism presents more communication than the data-based parallelism approach since it requires more steps to share the weights and perform synchronizations. Nowadays, it is more common to find examples of model-based parallelism, and we believe that our experiments and our proposed algorithm evaluated later in this thesis, in Section 4.3, will be even more critical for model-parallelization workloads because of the higher communication requirements and provide higher resource-efficiency than the experiments presented in this thesis.

In spite of these applications represent a small set of applications, we have varied them on many different configurations to represent a more significant range of different applications.

During our experiments, we found that the key parameter that plays a significant role in the GPU-to-GPU communication for our tested NNs is the *batch size*. This parameter determines how many samples will be loaded in each GPU. Thus, the NN will analyze in each training step the number of samples defined by the *batch size*, which directly impacts the amount of communication and computation in each training step. The lower the batch size is, the noisier the training signal is going to be; the higher it is, the longer it will take to compute the stochastic gradient descent. Noise is an important component for solving nonlinear problems, then apart from all the other parameters to determine the NN accuracy, the *batch size* is also an important one. Additionally, the *batch size* also determines the level of parallelism the NN can reach since the batch size partitions the dataset [12]. Therefore, in this thesis we have tested the NN configured with different *batch size* over different scenarios.

However, note that, although many other workloads could be used in our experiments, especially in the ones in Chapter 4.3, we have only used these DL ones because of their significant GPU-to-GPU communication. But, by the time of this thesis, applications with high GPU-to-GPU communication were uncommon in Cloud environments. But, we believe that with the advent of newer technologies providing higher performance for GPU-to-GPU communication (e.g., NVLink and PCIe 4 technologies) will encourage the creation of a new set of new applications that can further benefit from a system composed by multiple inter-connected GPUs. In this scenario, we believe that our proposed algorithm can significantly improve the performance of a system formed by heterogeneous workloads; we leave such investigation for future work when more applications with high GPU-to-GPU communication become available.

3.2.2 *Testing Platform and Configuration*

All experiments are conducted on an IBM POWER8® System S822LC release, codenamed as "Minsky" shown in Figure 1. The server has two sockets and eight cores per socket that run at 3.32 GHz and two NVIDIA GPU P100's per socket. Each GPU has 3584 processor cores at boot clocks from 1328 MHz to 1480 MHz, and 16 GB of memory. Each socket is connected with 256 GB of DRAM. Where the intra-socket CPU-to-GPU and GPU-to-GPU are linked via dual NVLinks that uses NVIDIA's new High-Speed Signaling interconnects (NVHS). A single link supports up to 20GB/s of unidirectional bandwidth between endpoints. A high-level illustration of the hardware topology is pictured in Figure 1 and Figure 9. The configuration of the testbed is summarized in Table 5.

For the software stack, this machine is configured with Red Hat Enterprise Linux Server release 7.3 (Maipo), kernel version 3.10.0-514.el7.ppc64le, Caffe version v0.15.14-nv-ppc compiled with NCCL 1.2.3, CUDA 8.0 and CUDA driver 375.39. All Caffe workloads are configured with a set of images from the dataset used in the 2014 ImageNet Large Scale Visual Recognition Challenge [126] (which is one of the most well-known datasets for image classification and publicly available on the ImageNet competition website [61], at the time that this thesis performed this experiments).

Table 5: Testbed configuration

	Version/Model
Processor	IBM Power8 S822LC
Sockets	2
Cores per socket	8
Hyper-threads	8
Frequency	3.32GHz
Ram	512GB
Network bandwidth	1Gbps
GPU models	NVIDIA P100
GPU interconnection	NVLinks
Docker version	1.9.1
Caffe Framework	v0.15.14-nv-ppc
NCCL	1.2.3
CUDA	8.0
CUDA driver	375.39

All experiments were repeated five times. For each experiment, the maximum number of iterations is 4000, except when generating the GPU profile where the iterations are only 40. The iterations are decreased because, at the time these experiments were performed in this thesis, profiling was consuming a lot of memory, and a large profile did not fit in the GPU memory. The tool used to profile the application was the NVIDIA `nvprofile`. For all workloads, the NN training batch sizes range from 1 up to 128. We defined some labels for the sizes to improve the readability, with "tiny" representing 1, "small" 4, "medium" 64 and "large" 128.

3.2.3 *Pack versus Spread Scheduling Policy*

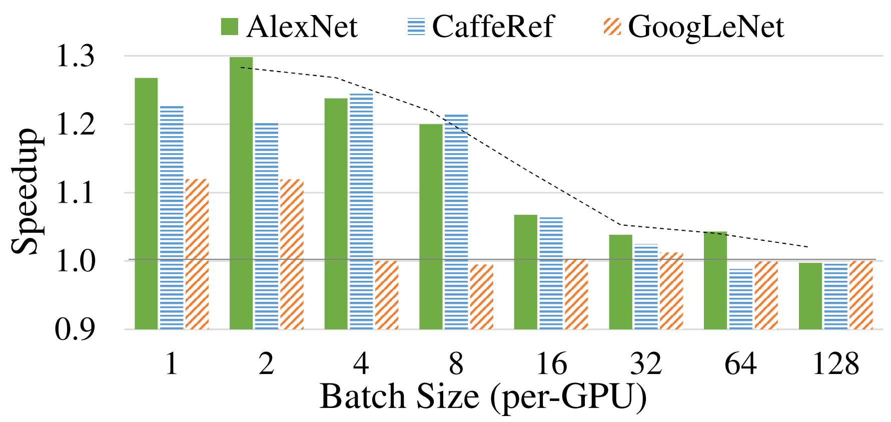


Figure 10: The *Pack* policy (P2P-enabled) vs. the *Spread* policy (non-P2P-enabled). A speedup higher than 1 represents that the *pack* policy performs better than *spread* policy.

Figure 10 shows the relative speedup achieved when allocating GPUs within the same socket (pack) or over cross-socket (spread). When the speedup is higher than 1, the application performs better with the *pack* strategy. The performance depends on both the workload type and the batch size. When AlexNet is configured with batch size 1 or 2, it has a speedup of up to ≈1.30x, but for batch sizes larger than 16 both *pack* or *spread* have even performance. GoogLeNet has different behavior than the other NNs with less or no impact, which will be better detailed later.

To better explain the cause of the performance delivered by the strategies, the application breakdown is presented in Figure 11. The analysis shows the percentage of computation and communication represented in the whole execution time. The results indicate that larger batch sizes significantly increase computation time, while communication time becomes less significant overall.

Taking AlexNet, for instance, when configured with tiny batch sizes, the computation time is ≈1s for 40 iterations; with big batch sizes, this time increases to ≈66s. The communication time instead remains ≈2s for all batch sizes. While NNs with a bigger batch size increases the amount of data exchanged between the GPUs, it starts to spend much longer time performing computation in the GPU for each batch step. Hence, the communication starts to be less frequent with bigger batch sizes. On the other hand, smaller batch sizes require many more steps to process the whole dataset and then require more frequent communication. This behavior can be verified with the NVLink bandwidth usage in Figure 12.

The communication frequency directly impacts the usage of the NVLink bandwidth. The NN configured with a small batch size reaches higher NVLink bandwidth usage

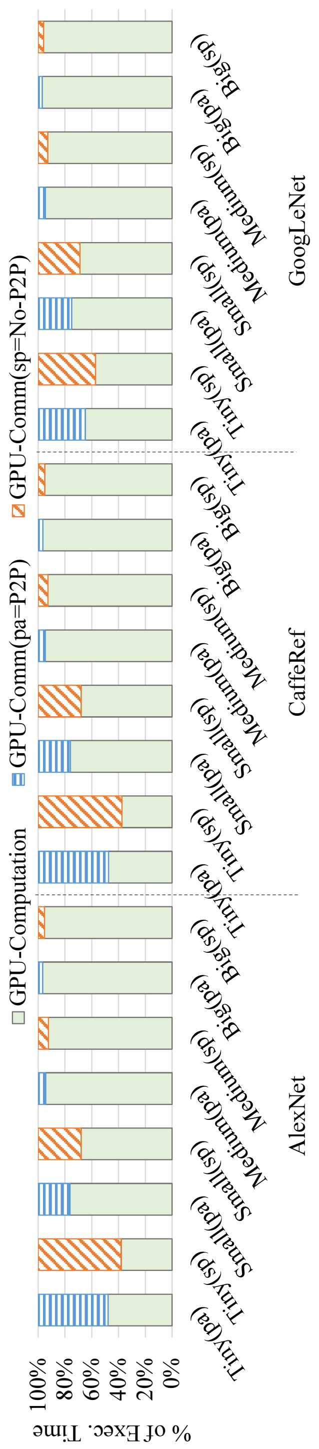


Figure 11: Application breakdown showing the percentage GPU computation and communication in relation to the whole execution time. All workloads were tested on different scenarios allocating the GPUs using the *pack* policy (pa) or the *spread* policy (sp), and ranging the batch size between tiny (1), small (4), medium (64) and large (128).

≈40GB/s, while the NN with a bigger batch size barely reaches ≈6GB/s, as in Figure 12 (the NVLink bandwidth calculation is described later in Section 4.3.1).

GoogLeNet is the less intuitive case. Since this NN contains sizable neural network layers, and typically the intensity of communications depends on the amount of infor-

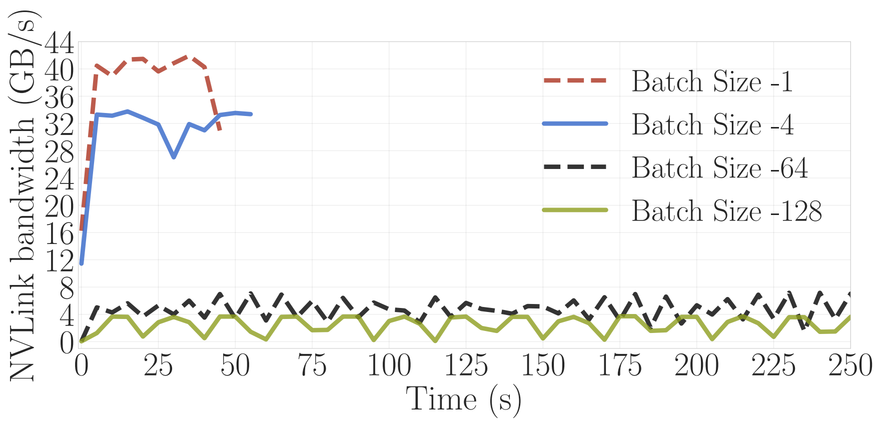


Figure 12: NVlink bandwidth usage for AlexNet.

mation exchanged between the layer, it is expected that GoogLeNet performs more communication than the other NNs. Nonetheless, GoogLeNet performs less communication because of its Inception Modules, which in consequence reduces the NN layers output by applying filtering and clustering techniques.

We have also executed the same experiments on a POWER8® machine equipped with a PCI-e Gen3 bus instead of the NVLink, as well as NVIDIA K80 GPUs instead of P100. We do not include additional figures in this thesis but summarize the results as follows. The impact of *pack* strategy is similar between NVLink-based and PCI-e-based machines except for larger batch sizes, where the difference starts to be evident. For instance, AlexNet with a batch equals one the speedup is ≈1.27x with NVLink and ≈1.24x with PCI-e. For a batch size equals two, the speedup drops from ≈1.30x with NVLink to ≈1.21x with PCI-e. For a batch size equals eight, the speedup decreases from ≈1.20x to only ≈1.1x. In conclusion, while the topology impact in the GPU communication performance is still significant in the PCI-e-based machine, improvements on the placement decision of DL workloads are even more necessary in NVLink-based machines.

3.2.4 *Jobs in a Co-Scheduled Environment*

A typical approach to increase resource utilization in a data center is co-scheduling workloads on the same machine. While it confers cost benefits, it comes with an inherent performance impact. Although the GPUs are not shared in this work (jobs have private access to GPUs), collocated applications share the bus interconnections among other resources. Therefore, the goal of this experiment is to evaluate the performance impact of the *pack* and *spread* strategies in a co-scheduled environment. Differently, from the previous test, this experiment shows application interference.

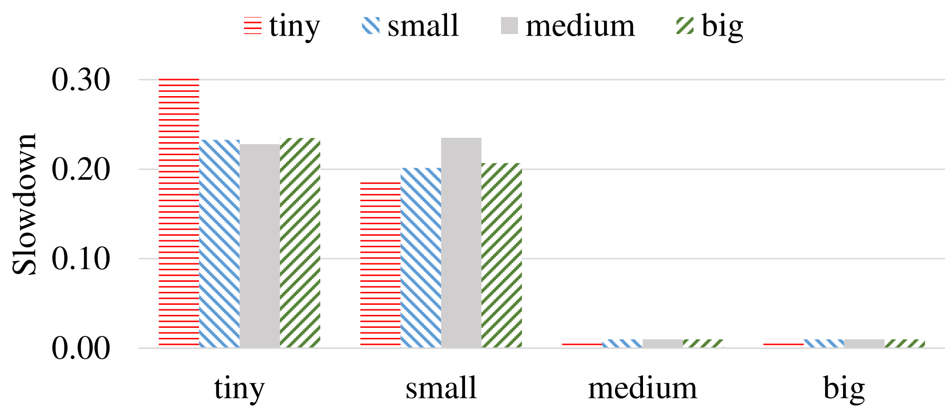


Figure 13: Normalized performance of Job #0 co-located with Job #1 versus Job #0 running solo in a machine. Both Job #0 and Job #1 are training an AlexNet NN and being configured with different batch sizes, creating different placement scenarios. When the slowdown is #0, it represents no performance interference in Job #0, else, it represents Job #0 with a slowdown in its execution time because of being co-located with Job #1.

We have performed an experiment that collocated two jobs in the same machine, starting concurrently. Each job is an AlexNet NN requesting two GPUs and varying the batch size. The results are shown in Figure 13, where zero represents no slowdown of co-scheduling two jobs in the same machine and a value higher than zero accounts for the slowdown percentage. Note that, a job with high GPU communication is more sensitive to interference than a job with lower communication.

As analyzed in the previous experiment (Section 3.2.3) and showed in Figure 12, the batch size plays the main role in defining the amount of communication and the job's performance sensitiveness. For that reason, when co-scheduling two jobs with a tiny batch, the suffered slowdown is higher, which is up to ≈30%. But when collocating two jobs with a big batch, the performance interference is very small or nonexistent. This is because a job with a big batch is not sensitive to perturbations in the bandwidth since it requires low bandwidth. Nevertheless, a job composed by a big batch can cause performance interference since it still consumes bandwidth. For instance, in Figure 13, if the first job has a big batch and the second a tiny batch, the slowdown is ≈24%, or ≈21% if the second has a small batch.

3.2.5 *Related Works of Performance Evaluation of NUMA Topology*

Although the NUMA concept is not new, and have been widely used on high-performance computers for many years, at the time of this analysis in this thesis, this architecture has started to be also popular in the Cloud environments. Most especially with the advent of machines for DL workloads running over several interconnected GPUs.

The NUMA impact of CPU-based workloads is well-known and was deeply investigated in previous works. Basic support for a NUMA-aware scheduler first appeared for Linux in kernel 2.5 and evolved over time. Zaytsev et al. [54] executed benchmarks to measure the impact of NUMA tuning on CPU/memory performance. They showed that the performance impact depends on the application characteristics and that for some situations manual binding is better than only rely on the OS's resource allocation. Psaroudakis et al. [117] have further evaluated the performance impact of NUMA topology on column-store database management systems (DBMS). They proposed also a smart technique to partition the data across processor sockets to load balance memory access and CPU utilization. Their results showed that their approach adaptively tracks a utilization imbalance across sockets, and can move or re-partition tables at run-time to fix the imbalance, improving the performance up to 4x compared to the default OS strategies. The work in [81] has also further investigated the performance impact of placement of threads and memory on NUMA systems. In their work, they do not only consider whether the threads and data are placed on the same or different nodes, but how these nodes are connected; some links might have different bandwidth. In their work, they propose a dynamic thread and memory placement algorithm for the OS task scheduling process with up to 218% better performance than when the placement is chosen randomly. They used a micro-benchmark that simulate several different applications in their evaluation. However, these previous works are CPU-based only, considering only the effect of CPU task placement and memory allocation on NUMA systems. They do not consider the allocation of other devices such as GPUs. Next, we describe the works that have also considered GPUs in the analysis.

Faraji et al. [35] has evaluated the performance difference of intranode GPU communication channels and proposed a topology-aware GPU selection scheme to assign GPU devices to MPI processes based on the GPU-to-GPU communication pattern and the physical characteristics of a multi-GPU machine. They have evaluated the performance impact of three different intranode GPU PCIe pair levels. The level #0 defines the communication path between GPUs connected in the same PCIe internal switch. The level #1 represents the path between GPU pairs traversing multiple internal switches. The level #2 is composed by the path traversing a PCIe host bridge, and level #3, the path traversing a socket-level link (e.g., QuickPath Interconnect - QPI). A simple MPI memory copy is used to measure the latency and bandwidth over all the levels. Their results show that only for sizable messages (e.g., higher than 64K) the topology starts to impact the performance. This work presents an interesting analysis to expose how the PCIe topology impacts GPU-to-GPU communication. In addition to this work, there exist other studies that have researched various GPU-aware point-to-point and collective operations to improve the GPU communication performance that has also quantified the performance impact of communication over asymmetric communication paths [96], [116], [34]. NVIDIA has also created the Collective Communication Library (NCCL) to facilitate the programming over such complex typologies [91]. By the time of this thesis, there was available only the NCCL version 1, but recently they also have released version 2 that also support efficient collective communication primitives on multi-nodes [100].

Our work, on the other hand, differently from these previous works, used a macro-benchmark that emulated real DP applications using real data, instead of only use synthetic micro-benchmarks forcing uncommon application behavior (e.g., with high load and contiguous communication). Moreover, our work differs from them, by also evaluating the performance impact on the GPU-to-GPU communication in a co-scheduled environment, quantifying the external interference that collocated applications can introduce into each other. Finally, by the time of this thesis, NVLink was relatively new technology, and to our best knowledge, we were the first work to evaluate the performance of heterogeneous paths in an intranode topology composed by NVLink technologies, experimenting with emulated real DP applications using real data and also considering the scenarios with collocated applications.

3.3 NETWORK IMPACT OVER RESOURCE DISAGGREGATION

Although resource disaggregation confers many advantages, it introduces the challenge to deal with additional network requirements. That is, an application that does not require network communication will transparently start to rely on transferring data via network connections. Additionally, the usage of disaggregated resources may introduce orders of magnitude higher networking bandwidth, additional latency and memory usage not present in a directly attached system. Thus, depending on the application resource usage pattern, resource disaggregation can introduce high or minimal network sensitiveness.

Therefore, to investigate of how extent the introduced network sensitiveness - by using disaggregation resources - can impact the application's performance, we conducted an experiment that collocates an application using remote GPUs with another network intensive application introducing artificial network load. Hence, in this experiment, we collocate Rodinia applications accessing remote GPUs using HFCUDA library with the application Iperf that creates network pressure in the cluster. Note that, our goal in this experiment is not to show the impact of disaggregation versus non-disaggregation since it depends exclusively on the techniques used to enable disaggregation. That is, it depends on how efficient is the software, or hardware-based disaggregation is implemented. Our goal here is to show how sensitive to the network an application can become by using a disaggregated resource. We rely upon and have the premise that the disaggregation technique implemented is efficient enough with minimal overhead.

3.3.1 *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 physical state of 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 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. The configuration of the testbed is summarized in Table 6. In all experiments, the applications were executed into Docker containers.

3.3.2 *GPU-based applications*

The workload is composed of applications from Rodinia Benchmark Suite for Heterogeneous Computing [21]. We selected only the applications that most differs from each other in regards to the resource usage pattern. To determine that behavior, we executed the applications over remote GPUs using the HFCUDA library; a detailed description of this library is provided later in Section 5.4.3. Next, we briefly describe the selected applications and their configurations used in the experiments; their performance breakdown is shown in Figure 14. All the applications parameters were carefully defined to make the applications as intensive as possible and with similar completion time as much as possible.

B+TREE performs queries on large n-ary trees and is configured with an input of 10k elements.

Table 6: Testbed configuration

	Version/Model
Processor	IBM Power8
Sockets	2
Cores per socket	10
Hyper-threads	8
Frequency	3.4GHz
Ram	512GB
Network bandwidth	10Gbps
GPU models	NVIDIA K80
GPU interconnection	PCIe
Docker version	1.10
CUDA	9.1
CUDA driver	387.36
Kubernetes	1.10

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 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 dynamics 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 that uses anisotropic diffusion to reduce noise in the image, and we configured it with 10k iterations, a saturation coefficient of 0.5, 5k rows and 4k columns.

3.3.3 *Network Performance Characterization of Jobs Using Disaggregated Resources*

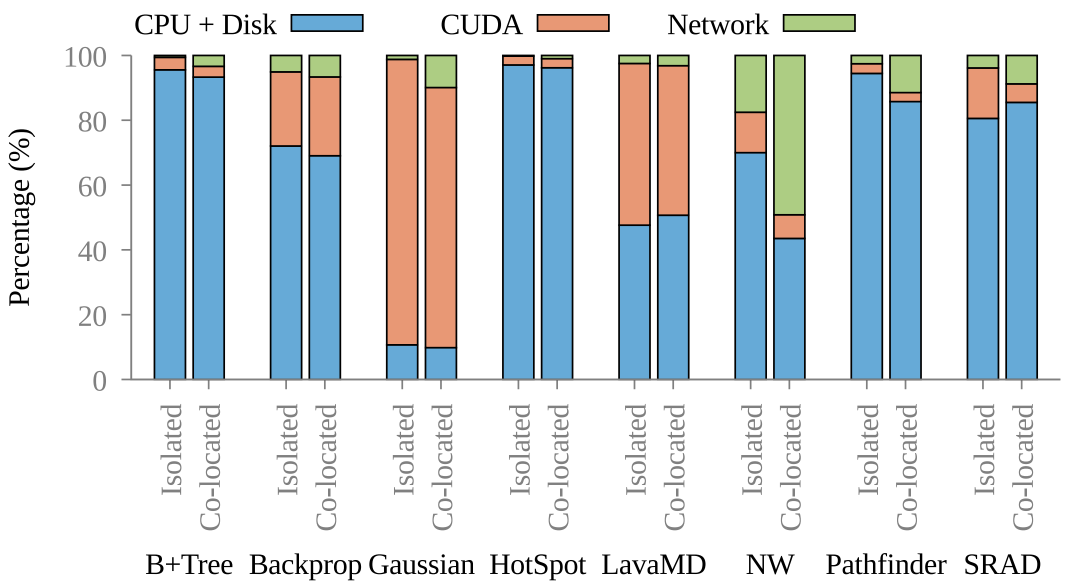


Figure 14: Usage breakdown for different Rodinia applications running with HFCUDA, with and without collocation with network workloads

Figure 14 shows the characterization of Rodinia applications, breaking down how much time is spent on CPU and disk, network, and GPU (CUDA), and comparing how each application behaves running in isolation and running it co-located with other CPU-only network-intensive workloads. As expected, all applications increase the network time when running with other workloads that use the network, as discussed as follows.

For the application b+tree, the percentage of time spent in the network communication in comparison with the whole execution time has increased from 0.2% to 1%. This shows that although the network part in this application is less significant, b+tree still suffer from performance variation when co-scheduled with a network-intensive workload. On the other hand, NW has an expressive impact on its performance, increasing the percentage of time spent in network communication from 17% to 49%. Backpropagation increased from 5% to 6%. Gaussian increased from 1% to 9%. LavaMD increased from 2% to 3%. Pathfinder increased from 2% to 11%. And finally, Srad increased from 3% to 8%.

Therefore, this experiment gives us a global view of the network impact of network-shared-induced interference of the collocation of an application using a disaggregated GPU with a network-intensive application. These results show that the effect is different depending on the Rodinia application, where each application has different characteristics, some transfer more data to the GPUs, spend more time in computation in CPU, or perform more computation in the GPU.

3.3.4 *Related Works of Performance Evaluation of Resource Disaggregation*

While resource disaggregation confers greater modularity to a datacenter infrastructure, permitting operators to optimize their deployments for improving efficiency and performance, it introduces new challenges regarding the network and sharing-induced performance interference. In this Section, we describe the related works in regards, the network requirements for resource disaggregation, GPU virtualization and performance interference as follows.

By the time of these experiments, the most related work was executed in the work of [39]. This work extensively evaluated the network latency and bandwidth requirements to determine the feasibility and limitations to enable disaggregation of different type of resources, i.e., CPU, Memory, Disk, etc. In their experiments, they concluded, as expected, that the key requirement to enable disaggregation lays on low latency and high throughput, but not all type of resources can already be disaggregated with current available off-the-shelf technologies. That is, they claimed that it is still tough to fully embrace CPU-to-CPU (cache coherence) and CPU-to-memory traffic into the external network. Their experiments also show that, in some specific cases, when the working set size of an application is bigger than physical memory, memory disaggregation improves the application's performance. In their conclusions, they pointed out that some of the possible issues that a datacenter might face in the future are that the network must be able to scale up to millions of disaggregated resources. This means that the network will be even more critical to be improved in datacenters. However, they did not perform experiments to quantify how extent the network congestion may cause performance interference in the applications using a disaggregated resource; they left it for future works. Therefore, as a complement to this work, our work performed experiments to measure the performance impact that an additional network load can cause on different applications using disaggregated resources, only for GPUs in our case.

Because of technology limitations and the difficult to access a high variate of expense resources, our work has focused only on GPU disaggregation. In our experiments, we have enabled software-based GPU disaggregation via an external library working as a middleware between the application and the remote resource. The middleware intercepts application's calls to resources/drivers/run-times and transparently forwards the calls to the remote node that hosts the corresponding disaggregated resource. Previously, some works have proposed, implemented and evaluated middleware to intercepts CUDA calls, such as follows. Oikawa et al. [107] proposed DS-CUDA, a middleware to use many GPUs in a cloud environment with lower cost and higher security. They propose 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 the experimental evaluating, they implemented DS-CUDA to intercepts calls from CUDA toolkit 4.1. Liang and Chang [85] 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 co-operate 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 CUDA toolkit 4.0 and demands modifications in the applications' code to be implemented using their toolkit. Merritt et al. [93] presents Shadowfax that assemblies a subset of all components from potentially different physical nodes in a cluster, allowing applications to access these resources as if they were local, but with increased access latencies whenever application code executes on a remote GPGPU. There is flexibility in how GPU assemblies may be built. If end users set low-latency as a priority, assemblies may be limited to those that only use locally accessible GPUs, avoiding the use of cluster-level interconnect. The prototype is implemented over Xen hypervisor 3.2.1 and supporting CUDA toolkit 1.1. The rCUDA [121] work, at the time of this writing, is the most popular middleware for GPU virtualization. It supports CUDA toolkit up to 9.1, can be used by both VMs and containers, and has low communication overhead when using InfiniBand networks. rCUDA differently from the other works requires users to allocate available GPUs by themselves for the execution of their CUDA programs and the code is proprietary. Finally, Xiao et al. [146] presented the VOLC, a middleware to virtualized GPU for OpenCL calls.

It is worth noting that, these above mentioned middlewares can be understood that they virtualize GPUs by enabling accesses to GPUs from virtual environment, e.g., from virtual machines. Hence, we describe next the related works regarding GPU virtualization but that does not support disaggregation. Gupta et al. [50] proposed GViM, a mechanism for improving GPU-accelerated VMs for HPC applications. GViM offers low overheads by carefully managing the memory, and device resources used the applications and guest OSs. They implemented a prototype over Xen hypervisor 3.2.1 and evaluated it with benchmarks using CUDA toolkit 1.1. Shi et al. [128] proposed vCUDA that intercepts and redirects CUDA commands and data in VMs to a CUDA enabled graphics device, performing real graphics computations into the virtual machine monitor. They have evaluated the performance of HPC applications running on VM using vCUDA, demonstrating that, in their scenarios, hardware accelerated high-performance computing jobs can run as efficiently in a virtualized environment as in a native host. The implemented a prototype over Xen hypervisor version 3.0.3 and supporting CUDA toolkit 1.1. Montella et al. [98, 99] have proposed GVirtus that enables a completely transparent layer among GPUs and VMs. Their approach supports CUDA toolkit up to

9.1, OpenCL up to 2.1, memory management and scheduling that multiplexes back-end processes spawned by the GVirtuS Backend driver.

Although, by the time of this thesis there were many libraries to enable GPU disaggregation we decided to implement our own library[3], the HFCUDA, which is detailed in Section 5.4.3. This is due to most of these works are not up to date, providing support of only to old CUDA versions (e.g., older than 8.0 and we used HFCUDA with support to CUDA 9.1), or only offer support for VMs. Additionally, even though, rCUDA supports the newer versions of CUDA, by the time of this thesis writing, their code was proprietary, preventing us from extending it to support and incorporate new features, if necessary, and cross-compiled it for different architectures. E.g., at this time that we performed our experiments, rCUDA was only compiled to x86 architectures, and we needed it compiled to PowerPC architectures. Also, we believed that creating a similar library that could be open-source could benefit the research community and engage researches and companies to further develop and improve the library in the future. Therefore, we supported the implementation of the HFCUDA library that has been developed by IBM.

Last but not least, there are also some related works that have evaluated the performance variability of applications running on co-scheduled cloud environments. For instance, Iosup et al. [64] assessed the dependability of cloud computing services by analyzing long-term performance traces from two Amazon Web Services and Google App Engine. Through a trace-based simulation, they evaluated the impact of the variability observed for the studied cloud services on three large-scale applications, in scientific computing, social networks, and social gaming. As expected, they concluded that the impact of performance variability depends on the application and that several cloud services exhibit high variation in the monthly median values, which indicates large performance changes throughout the year. Leitner et al. [80] investigated predictability of performance in public Infrastructure-as-a-Service (IaaS) clouds. They formulated hypotheses relating to the nature of performance variations in IaaS systems. They found out that there are relevant differences between providers (Amazon Elastic Compute Cloud, Google Compute Engine, Microsoft Azure, and IBM Softlayer). They also showed that hardware heterogeneity, as often seen as a core property of public clouds, only exists in Azure and a small number of Amazon's EC2 instance types at the time of their evaluation. Similarly, they also measured that the effect of multi-tenancy on performance predictability is not equally pronounced in all cloud providers. El-Khamra et al. [32]

3 Note that, the proposal and implementation of HFCUDA library is more a software engineering challenge and less a research challenge, the concept and architecture was already defined in the current existing related libraries, e.g., rCUDA. Therefore, in this thesis, our real research contribution relies on how to efficiently use such type of library to improve resource-efficiency on datacenters via intelligent scheduling techniques.

evaluated and characterized runtime fluctuations for a given application kernel, representing MPI/parallel workloads on two different cloud offerings, Amazon's EC2 and Eucalyptus. Their results show that, the variability depends both on the application and that different cloud providers have different performance variation, such as variation in performance on a single core is more pronounced on Eucalyptus than on EC2. Kayıran et al. [72] showed that GPU-based applications tend to monopolize the shared hardware resources, such as memory and network, because of their high thread-level parallelism (TLP), which makes much more complex the task to evaluate the performance impact of co-located applications. They proposed via simulation, a mechanism that considers both GPU core state and system-wide memory and network congestion information to dynamically decide on the level of GPU concurrency to maximize system performance. They showed that there is always a trade-off, and their mechanism performs a fair approach minimizing the impacts in both CPU and GPU-based applications. In their results, if the CPU-based application has their performance prioritized, the system can improve their performance in 24%, while penalizing on the GPU-based applications in 11%. They also presented the results of applying a mechanism that fairly managed the performance for both CPU and GPU-based application, improving the performance of all application in 7%. Wu and Hong [144] have also evaluated and proposed a mechanism to assist the co-location of CPU-only jobs with CPU-based jobs. In their studies, they discovered that the lack of overlap between CPU/GPU computation is a significant obstacle in the efficient utilization of the heterogeneous system. They analyzed the factors that impact the job's performance regarding resource isolation, CPU load, and GPU memory demands, using MPI/CUDA benchmarks. The results indicate that, in some specific scenarios, when those factors are appropriately managed, the collocated CPU-only job can efficiently scavenge the underutilized CPU resource without affecting the performance of both collocated jobs. The experimental results showed that the system demonstrates 15% gain in throughput and 10% gain in both CPU and GPU utilization. Finally, Delimitrou and Kozyrakis [25] evaluated and characterized the performance interference between co-located workloads in cloud environments. They proposed a mechanism based on collaborative filtering techniques to make online predictions of the performance impact in reasonable time, along with a scheduling technique to improve the datacenter resource-efficiency for CPU-based workloads.

Although these works have done a good job on defining the source of the performance evaluation on cloud environments, they focus only on CPU-based workloads running only on local resources. Our work instead analyzes the performance variation of workloads using disaggregated resources, GPUs and also in collocation with other applications, as shown in this chapter and also shown in analyzes in other chapters.

3.4 FINAL CONSIDERATIONS

In this chapter, we first evaluated and characterized the performance impact that applications can suffer from *virtual environments*. We showed that while OS containers offer clear advantages concerning lightness and performance under several circumstances, they still show some specific challenges from the infrastructure management perspective in regards to overheads, performance, isolation, and scalability. After that, we evaluated the impact of the underlying NUMA topology on intra-GPU communication that is typically fully exposed to the applications because of *virtual environments* abstraction. We illustrated that multi-GPU applications running over modern-NUMA architectures present new challenges as they usually require inter-GPU communications. Our experiments show an optimal resource allocation can reflect in a speed-up of up to ≈1.30x. Finally, we further evaluated the topology impact, but in the context of resource disaggregation. We showed that because resource disaggregation introduces additional network requirements into the application, because of accessing remote resources, the application's performance depends on both the topology and the current network load. In conclusion, these experiments put in evidence the necessity of a scheduling algorithm that is aware of both the underlying topology and possible performance interference to provide Quality of Service (QoS) for jobs. In the next chapters, we will introduce the proposed scheduling algorithms that we created to address the discussed problems.

4

TOPOLOGY-AWARE MULTI-GPU HIGH PERFORMANCE AI WORKLOAD SCHEDULING

Recent advances in the theory of Neural Networks (NNs), new computer hardware such as Graphics Processing Units (GPUs), availability of training data, and the ease of access through cloud have allowed Deep Learning (DL) to be increasingly adopted as a part of business-critical processes in health care, autonomous vehicles, natural language processing, and Internet of Things. Consequently, many online platforms that offer image-processing and speech-recognition systems leveraged by trained DL NNs are emerging to deliver various business critical services, such as IBM Watson [60], Microsoft Project Oxford [97], Amazon Machine Learning [7], and Google Prediction API [45].

Training DL NNs is a computationally intensive process. An image-processing application, for instance, might demand the analysis of millions of pixels in one of many layers of the NN that takes several hours to days of computations [31]. A promising approach to increase the levels of efficiency in processing time and power consumption of the training process is using one or more GPUs. Computing the NNs on multiple GPUs further reduces training times, enables to handle larger amounts of data, and increases the accuracy of the trained models. Hence, multiple GPUs has become a common practice for DL applications [31, 46]. Although training on multiple GPUs can deliver many advantages, it presents new challenges in workload management and scheduling for obtaining optimal performance. The performance depends on both the GPUs and CPUs connectivity on the physical topology, and the application's tasks communication pattern.

To illustrate this issue, consider Figure 1 which shows the connectivity topology between the GPUs and CPUs for two representatives DL cognitive systems. In these systems, multiple link technologies such as PCI-e and NVLink connect GPUs to each other and GPUs to host CPUs. NVLink offers better bandwidth and lower power consumption over PCI-e. In the figure, the IBM POWER8® system consists of four GPUs and two CPUs with two GPUs per CPU socket. The two GPUs on each socket are connected with dual lane NVLink to achieve up to 40GB/s unidirectional bandwidth, and each of the GPUs is also linked to the socket with two lanes of NVLink. The two CPUs are connected via the system bus. NVIDIA DGX-1 has 8 GPUs connected to two CPU sockets.

The GPUs are connected over a hybrid cube-mesh topology: the 12 edges of the cube are connected via single lane NVLink, and the diagonals of two of six faces are also connected via NVLink. Each of the GPUs is also connected to a PCI-e switch so it can communicate to a GPU that is not connected to it via the NVLink and communicate to the CPU as well.

In these systems, communications can take place directly between devices, in the so-called Peer-to-Peer model (P2P), or it should be routed through the main memory of the processors containing the bus controllers. For example, in the case of DGX-1, the communication between GPU1 and GPU5 will go over the PCI-e switches and the system bus (such as quick path interconnect – QPI). As a result of these complex connectivity topologies between different GPUs, the application performance depends on which GPUs are allocated for computations and how the GPUs are connected to each other (via PCI-e or NVLink).

Additionally, this challenge becomes acute in shared systems, like cloud computing, where multiple applications from different users share the GPUs on the system. At the time of this thesis, it was uncommon to share a single GPU between two applications so sharing here means different applications get different sets of GPUs in the same machine. Jobs in this environment have varied GPU requirements: some need a single GPU, some need GPUs with NVLink, others require multiple GPUs, but communication requirements are minimal, etc. In such environments, cloud scheduler should be able to take the communication requirements of the workloads, consider the topology of the system, consider existing applications and their GPU and link utilization and provision the GPUs for the new workload that meet the workload requirements. This enables users to get access to the resources necessary without worrying about the detailed topology of the underlying hardware. Major cloud providers such as IBM, Amazon, Google, Microsoft, and others provide multi-GPU systems as a service today via virtual machines, and most of them have systems with similar GPU topology described in Figure 1; so that job scheduling and resource management becomes critical at the time of running multi-GPU based applications on a shared system. Thus, those systems require the same placement functionality proposed in this work to exploit the capabilities of modern cognitive systems fully. Furthermore, both cloud and HPC systems can benefit from a GPU topology-aware schedule.

In this thesis, we present an algorithm with two new scheduling policies for placing GPU workloads in modern multi-GPU systems. The foundation of the algorithm is based on the use of a new graph mapping algorithm that considers the job's performance objectives and the system topology. Applications can express their performance objectives as Service Level Objectives (SLOs) that are later translated into abstract Utility Functions.

The result of using the proposed algorithm is a minimization of the communication cost, reduction of system resource contention and an increase in the system utilization.

4.1 TOPOLOGY-AWARE SCHEDULING ALGORITHM

To overcome the problems discussed in the earlier section, we propose a topology-aware scheduling algorithm that makes decisions based on the workload's communication, the possible interference from currently running workloads, and the overall resource allocation of the system. The algorithm's core is a graph mapping mechanism: one graph represents the job's tasks and their communication requirements, and the other graph represents the physical GPU topology. The mapping algorithm produces the GPU allocation that satisfies communication requirements of jobs while minimizing resource interference and fragmentation.

4.1.1 *Topology Representation*

4.1.1.1 *Job graph*

This graph represents the communication requirements of tasks (i.e., GPUs). Vertexes represent GPUs and edges represent communication, as illustrated in Figure 16. Each edge has an associated weight denoting the communication volume, given by the average GPU-to-GPU bandwidth usage. During the mapping process, this weight is normalized by the total available bandwidth in the physical machine, where a value equal to #0 represents no communication and higher than #0 accounts for the communication level.

4.1.1.2 *Physical system topology graph*

This graph represents the GPU topology based on the underlying hardware of a machine or a set of machines connected by a network. An example of how different physical GPU topologies are modeled is illustrated in Figure 15, which shows the graph of Figure 1's topology. The physical graph can be understood as composed of multiple levels, where the first level is the network. Just after this level, there is the machine level, as represented by the vertexes M{X}, where X is the machine ID. The next level is the socket level and is represented as S{Y}, where Y accounts for the socket ID. Other levels can exist between the socket and the GPU, such as levels representing multiple PCI-e or NVLink switches. The last level represents the GPUs.

A GPU vertex can be directly connected to the socket vertex, to an intermediate vertex, and/or directly connected to other GPUs, which represents a direct NVLink connection between the GPUs. Consequently, some GPUs will have multiple paths to communicate. The path distance is given by the sum of the weight of the edges of the path.

In our implementation, the weights are defined based on the distances defined by the Linux OS, which represents qualitative distances. In the prototype, we created the graphs based on the weights determined by the Operating System and collected by using the `hwloc` library [14]. In this model, a higher level has a more significant weight to represent longer distances. However, note that those weights can also be given by quantitative measurements, such as executing memory copies and measuring the bandwidth or latency on different configurations. Both approaches, using qualitative or quantitative weights, can provide a good overview of the system, but the second can confers higher accuracy. An example of a graph representing the physical topology of a machine is shown in Figure 15, where each level right after the GPU level has weight 1, whilst at higher levels, such as the socket level, the edges have weight 20.

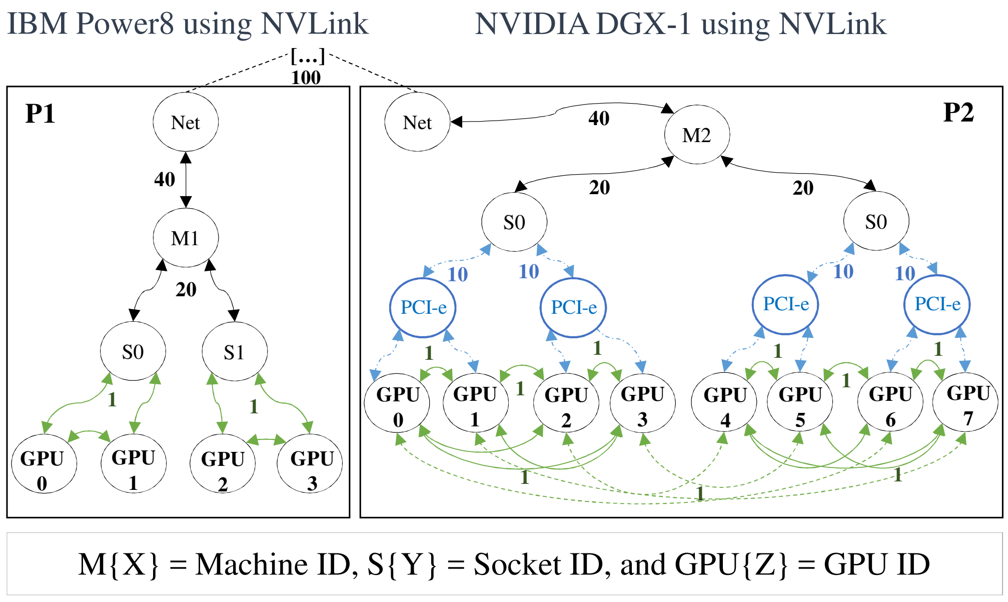


Figure 15: GPU physical topology graph.

4.1.2 *Job Profile*

The profile includes not only the job's communication graph but also a performance model defining the level of interference the collocated jobs will suffer and cause. This model is created from experimentation using historical data. Although any the performance model for the job profile can be generated in several different ways, in this thesis, we performed experiments injecting artificial load, using micro-benchmarks, onto the shared resources and measuring the interference, i.e., the impact on run-time of other collocated jobs. As the experiments that we performed in Section 3.2. Therefore, to generate the offline job profile, we first execute the job in the best know performing scenario to serve as the baseline, and then, we execute the job in other different configurations and job collocation. With the measurements, we quantitatively define the performance impact of the application running on a different scenario. If a not know scenario occurs, we then use a default performance model to represent the scenario.

4.1.3 *Objective Function and Constraints*

Our objective function focuses on minimizing the tasks communication cost (t^{cc}), external resource interference (I^b), and resource fragmentation (ω^d). Formally, it can be defined as follows:

$$\text{MIN} \quad \alpha^{cc}\frac{t^{cc}}{t^w} + \alpha^b\frac{I^b}{I^w} + \alpha^d\frac{\omega^d}{\omega^w} \tag{1}$$

where $\alpha^{cc} + \alpha^b + \alpha^d = 1$. All parameters t^{cc}, I^b and ω^d are normalized against the corresponding worst case t^w, I^w and ω^w (i.e., the scenario with the lowest bandwidth, the highest interference, and the highest fragmentation). For the minimum t^{cc}, we allocate GPUs as close as possible once all constraints are met. For the minimum I^b, we allocate GPUs with the lowest possible amount of bus sharing. For the minimum ω^d, we map GPUs from the most fragmented domains to increase the cluster utilization.

The constraints that we define in this thesis are the resource capacity as the number of GPUs and the memory bandwidth. Formally, all possible solutions must meet the inequality constraints defined as $t^{gpu} \leqslant p^{gpu}$ and $t^{bw} \leqslant p^{bw}$, where t^x and p^x denote the resource requirement of a given application and the available capacity of a given node for the resource type x, respectively. Other constraints can be added for different scenarios than the ones we show in our experiments.

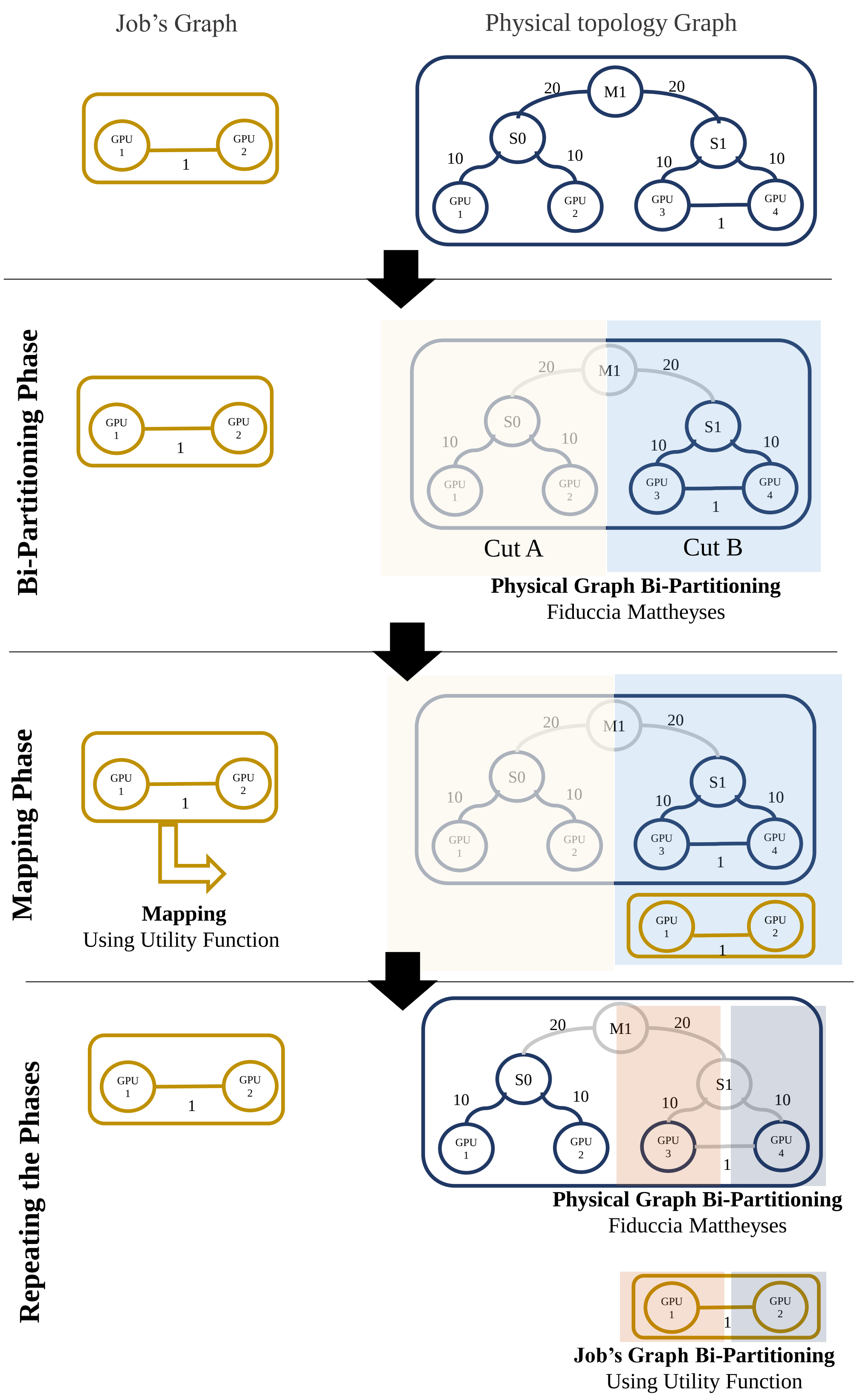


Figure 16: Step by step illustration of our proposed algorithm. The algorithm takes two graphs, where one can represent the requested job's GPUs and their level of communications as weights and the other graphs the system physical NUMA topology. The algorithm recursively performs several partitioning and mapping steps until there is a one to one match between the graphs partitions.

4.1.4 *Placement and Scheduling Topology-Aware Algorithm*

Our placement process is formally defined as a function $\psi()$ taking the job's graph A and the physical topology P as $\psi(A, P)$ and transforming them into the GPU list g. Where $|A|$ is the number of requested GPUs, $|P|$ is the number of available GPUs, and $|g|$ is the number of allocated GPUs to the job, being $|g| \leqslant |P|$.

Our proposed algorithm is a loop-based approach that each iteration attempts to place jobs while there are jobs in the waiting queue Q and available resources. Otherwise, the scheduler sleeps until a job has finished or a time interval has expired. During each iteration, the scheduler takes a job from Q and filters the available nodes, eliminating the ones that do not satisfy the constraints (e.g., resources types, anti-affinity, etc.), creating the graph P'.

More specifically, as described in the algorithm 1 and further illustrated in Figure 16, our proposed algorithm takes two graphs, where one represents the requested job's GPUs and their level of communications as weights and the other graph the system physical topology that connects the GPUs. The algorithm recursively performs several partitioning and mapping steps until there is a one to one match between the graphs partitions where the mapping steps are based on the utility function, as later defined in Equation 2. In each recursive iteration, the algorithm firstly bipartite the physical graph, and then, it attempts to map the job's graph into one of the two partitions created in the previous step. However, if the second graph does not fit into a partition, it is also bipartite the second graph. Note that, before starting the process, the solution must be feasible, it must exist enough resources to place the jobs.

We define two scheduling policies for the proposed algorithm. One policy is referred to TOPO-AWARE-P which allows out-of-order execution of jobs and postpone the placement that the job's utility is lower than a threshold defined in the job's profile. The other policy is the TOPO-AWARE, where the jobs are placed as soon as they arrive without consideration for future jobs.

Next, we further detail the algorithm. In the algorithm 1, the function `DRB()` is called to traverse the physical graph P' and define the GPU allocation. After that, if the utility of the solution s does not satisfy the job's requirements and the policy allows postponement, the job is added back to the waiting queue at the end of the iteration; otherwise, the placement is enforced.

The function `DRB`(), outlined in Algorithm 2, is based on the Hierarchical Static Mapping Dual Recursive Bi-partitioning algorithm proposed by [33] and implemented by [113]. Its asymptotic complexity is defined as $\Theta(|E_A| * \log_2(|V_P|))$ [114], where in our

Algorithm 1 Topology-aware job placement algorithm

```
A; //application's job communication pattern graph
P; //physical topology graph
C; //communication cost array
Q; //jobs waiting queue sorted by their arrival time (oldest to newest)
function scheduler(P)
    while True do
        while availableResources(P) and $Q \neq \emptyset$ do
            $A \leftarrow$ Q.pop()
            $P' \leftarrow$ filterHostsByConstraints(A, P)
            s = DRB(A, P', C)
            if U(s) < A.minimal_utility and postpone = True then
                postponed_list.add(A)
            else
                place(A, s)
            end if
        end while
        Q.add(postponed_list)
        sleep(interval) //wakeup after an event (e.g a job has finished)
    end while
end function
```

Algorithm 2 Recursive Bi-Partitioning Mapping based in [33]

```
1:  function DRB(A, P, C)
2:      if (|A| == 0) then
3:          return nil //This partition is not a candidate
4:      end if
5:      if (|P| == 1) then
6:          return $g \leftarrow (P, A)$ //Map job's task to physical GPU
7:      end if
8:      $(P^0, P^1)$ = physicalGraphBiPartition(P)
9:      $(A^0, C^0, A^1, C^1)$ = jobGraphBiPartition(A, $P^0$, $P^1$, C)
10:     $g^0$ = DRB($A^0, P^0, C^0$)
11:     $g^1$ = DRB($A^1, P^1, C^1$)
12:     return ($g^0$+$g^1$)
13: end function
```

case $|E_A|$ is the number of edges from the job's graph and $|V_P|$ is the number of a vertex from the physical graph.

More specifically, during each recursive iteration of DRB() two other functions are called, the first is physicalGraphBiPartition() to bi-partition the physical graph P, and the second is the jobGraphBiPartition() to bi-partition the job's graph A. The recursion stops when A = $\emptyset$, returning $\emptyset$, or when P^y only has one element, returning the mapping pair (A^y, P^y), where $y \in \{0,1\}$ partitions. The C parameter is an array that contains the

Algorithm 3 Utility-based job graph bi-partitioning

```
1: function jobGraphBiPartition($A$, $P^0$, $P^1$, $C$)
2:     while $A \neq \emptyset$ do
3:         $task \leftarrow A$.pop()
4:         $(P^0.t^{cc}, P^1.t^{cc}) \leftarrow$ getCommCost($task$, $P^0$, $P^1$, $C$)
5:         $(P^0.I^b, P^1.I^b) \leftarrow$ getInter($task$, $P^0$, $P^1$, $A.profile$)
6:         $(P^0.\omega^d, P^1.\omega^d) \leftarrow$ getFragmentation($P^0$, $P^1$, $A$)
7:         if (U($task$, $P^0$) $\geqslant$ U($task$, $P^1$)) and ($constraints$) then
8:             $A^0$.add($task$)
9:         else
10:            $A^1$.add($task$)
11:        end if
12:    end while
13:    return $A^0$, $P^0.t^{cc}$, $A^1$, $P^1.t^{cc}$
14: end function
```

communication cost of all GPUs, even the ones not into the sub-partition P^y. C is used to calculate the communication cost between sub-partitions.

Similarly to the implementation of `DRB()` in [113], the physical graph bi-partition is performed with the well-known Fiduccia Mattheyses algorithm [38] that minimizes the cut-sets in linear time. However, differently, from [113], we do not only account the communication cost, but also the job's preference using a utility function to bi-partition the job's graph, as shown in the function `jobGraphBiPartition()` outlined in Algorithm 3.

Algorithm 3 creates two sub-partitions A^0 and A^1, where each partition can have part or all the job's tasks. Since the tasks in A^0 will be placed in P^0 and A^1 in P^1, the function evaluates for each task which sub-partition P^y provides higher utility. Then, if P^1 gives better utility and has enough available resources, the task is added to A^0. Otherwise, the task is added to A^1.

For each task, Algorithm 3 evaluates each sub-partition via calculating the communication cost t, the workload performance interference I of co-scheduled jobs and the resource fragmentation ω, using the functions `getCommCost()`, `getInter()` and `getFragmentation()`, respectively. Then, with those parameters, the job's utility is calculated using the utility function U, which can be defined as the convex function in Equation 2.

$$U = (\alpha^{cc}\frac{1}{t} + \alpha^{b}\frac{1}{I} + \alpha^{d}\frac{1}{\omega}) \tag{2}$$

Next, we describe how the U parameters are calculated. The communication cost (t) is defined as the sum of the combinatorial shortest paths p between all GPUs within the solution as:

$$t = \sum_{i=1}^{|P|} \sum_{j=1}^{|P|-i} p_{i,j}, \text{where } i \neq j \tag{3}$$

The level of interference (I) is measured using the job's profile. As described in section 4.1.2, the profile is composed by the completion time of the job running solo and running with other jobs (or with artificial loads). Therefore, the algorithm measures the average slowdown that the job suffers and causes in the currently running jobs. Thus, the average interference is calculated as follows:

$$I = \frac{\sum_{j=1}^{running\_jobs+1} (solo\_time(j)/collocation\_time(j))}{running\_jobs + 1} \tag{4}$$

System fragmentation (ω) is the average fragmentation of all sockets, which is calculated as follows:

$$\omega = \frac{\sum_{i=1}^{sockets} (freeGPUs(socket_i)/totalGPUs(socket_i))}{sockets} \tag{5}$$

4.2 PREMISES, LIMITATIONS AND OTHER DISCUSSIONS

The algorithm behaves as a greedy algorithm since the assignment of a task to a physical GPU is never reconsidered. Hence, we perform a best-effort approach to find the optimal solution. The algorithm preferentially places as many tasks as possible for a job in the same node. If a job wants to get all its tasks spread across different nodes instead, it needs to define anti-collocation policies for its tasks, and in response, they will be placed on different nodes. The easiest way to define that is by creating multiple disconnected applications graph, where each graph will represent a set of tasks that will be dis-jointly placed on different nodes. Also, if a job does not support multi-node, it must be defined with a single-node constraint in the profile. Moreover, if a job cannot be placed, its placement is postponed to the next iteration of the scheduler. To avoid starvation and enforce fairness as much as possible, the job waiting queue is sorted by the job's arrival time. Thus, the oldest jobs have priority to be placed.

The approach that we used to quantify the performance impact of collocated applications to generate the performance impact is very computationally costly. It requires a combinatorial collocation of a set of known applications over many different scenarios.

While this approach is highly accurate, it might not be realistic to perform it in a large scale cloud scenario. Therefore, a promising approach to overcome that limitation is by using advanced prediction models, such as using decision tree [37, 118] or statistical clustering [26, 56, 87]. These approaches can be used to predict the performance of unknown jobs using the models from known applications, which can enlarge the range of the analysis and improve the accuracy of the system. We believe that by using such a prediction model, our approach can be easily adopted in current cloud environments. But, note that, the main goal of our approach is to minimize the communication cost between GPUs, and the consideration of the sharing interference is to enable efficiency in the system further. That is, in the case that the interference cannot be quantified, the algorithm can still work without it, but with slightly lower quality decisions. However, the system will still delivery high-quality decisions to improve the resource-efficiency than comparing with a topology agnostic schedule.

Additionally, by the time of this thesis, it was uncommon to share a single GPU between multiple applications, even though, there were some libraries to help to enable that, such as Multi-Process Service (MPS) [101]. At this time, MPS had harsh limitations preventing its usage on 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. In the experiments within this thesis, we had only access to GPUs with pre-Volta architectures. Therefore, we did not enable GPU sharing with multiple applications. Moreover, even though, the new NVIDIA Volta architectures implement now fully isolated GPU address spaces, there are still limitations to turn the sharing GPU a reality in a cloud environment. That is, 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 [101]. Additionally, the CUDA run-time still does not expose in its API mechanisms to enable the control to process preemption, which can efficiently allow time-sharing of processes and prevent starvation of big kernels. Thus, when these limitations are exceeded, the model can be extended to enable further resource-efficiency by time-sharing GPUs between multiple applications.

4.3 PROTOTYPE EVALUATION

In this section, we present a prototype implementation to evaluate the proposed topology-aware scheduler algorithm. This experimental evaluation was performed on a single machine with characteristics described in section 3.2.2 and summarized in Table 5.

While the focus of this work is on learning workloads, any workload can be submitted in the prototype. Also, there is no need to change how applications are implemented to use the scheduler. In the future, we plan to test the proposed algorithm in a cluster manager framework like Kubernetes [44] or Mesos [53], similar to the enhancements described in the related work [148].

4.3.1 *Prototype Implementation*

We implemented the prototype for the scheduler using C and Python. The program continuously loads JSON files containing the necessary information about the submitted jobs. To place a job, the system creates the job's manifest, filling it with the information received from the JSON file, and uses that information to determine the placement of the job. If the algorithm decides to place the job, it enforces the decision of running the job on the given machine. Until the job finishes, the system keeps track of the execution of the job while collecting statistics including the ending time.

For the placement, the system captures various performance metrics. The DRAM memory bandwidth is calculated using the POWER8® performance counters described in [1], which are accessed using the library Perfmon2 [115]. To calculate the NVLink bandwidth (which is shown in most of the experiments), we access the NVIDIA CUDA driver API using the command `nvidia-smi nvlink -i $gpu_id` that returns the transmitted bytes from each link. Then, the algorithm calculates the NVLink bandwidth usage of CPU-to-GPU or GPU-to-GPU communication based on their link connections. In all experiments, the applications were executed into Docker containers.

For discovering the topology during the system startup, it executes the `nvidia-smi topo -matrix` command[4] to create a matrix of GPUs, and the to include socket distance and CPU locality in the model, the command `numactl --hardware` is executed. For enforcing the decisions, before running an application, the system first defines the order of the GPU ID's by exporting the parameter `CUDA_DEVICE_ORDER=PCI_BUS_ID`, and then, for each application, it exposes only the specified GPU list from the scheduler decisions using the parameter `CUDA_VISIBLE_DEVICES=$gpu_list`. For preventing performance variability related to NUMA remote memory access, the applications with only GPUs in the same socket are bound to the socket using the command `numactl`.

To feed the performance prediction model, the application profiles are experimentally generated, defining the optimal resource allocation (best-performing) and some possible sub-optimal resource allocation (worst-performing) for both solo (when the job runs

4 The system targets only NVIDIA GPUs. But, for detecting GPUs from other vendors another library can be used, such as the `hwloc` library [14].

alone with no other jobs) and co-scheduled modes, as previously shown in Section 3.2. The profile then contains the 95th percentile of the execution time from five executions of each workload within different scenarios. A simple, but effective performance prediction approach is then performed using the profiles, characterizing the workload slowdowns for various configurations; we plan to extend it with more robust statistical techniques in the future. Since Caffe framework is based on data-parallelism model, all GPUs perform similar work, and then, they have a similar amount of communication between each other. Therefore, we define in the workload graph all GPUs communicating with each other with the same weight. However, for different batch sizes, different weights are used, ranging from #4 to #1, where #4 represents the smallest and #1 the largest batch size.

4.3.2 *Prototype Evaluation*

We implement two well-known greedy approaches: First Come First Served (FCFS) with a FIFO queue, and Best Fit (BF) performing bin packing (i.e., allocating first the GPUs from highly used domains) and compare them to our proposed placement algorithm with the two scheduling policies: TOPO-AWARE and TOPO-AWARE-P. Finally, we evaluate the prototype in a cloud environment, where jobs have varied GPU requirements: some needing a single GPU, some needing more than two GPUs, some requiring P2P to be fully satisfied, others needing multiple GPUs, but communication requirements are minimal. Additionally, as in a cloud environment, the jobs concurrently share any machine's resources.

In all experiments, we used the same workload as used and further detailed in Section 3.2.

Config.	Job0	Job1	Job2	Job3	Job4	Job5
DL NN	A	G	A	A	A	C
Batch Size	1	4	1	4	1	1
Num. GPUs	1	1	1	2	2	2
Min. Utility	0.3	0.3	0.3	0.5	0.5	0.5
Arrival Time	0.51s	15.03s	24.36s	25.33s	29.33s	29.89s

Table 7: A=AlexNet, C=CaffeRef, G=GoogLeNet

4.3.2.1 *Description of the experiment*

Our first experiment is a simple, easy-to-verify scenario, with five jobs dynamically sharing the machine described in Section 3.2.2. The workload configurations are summarized

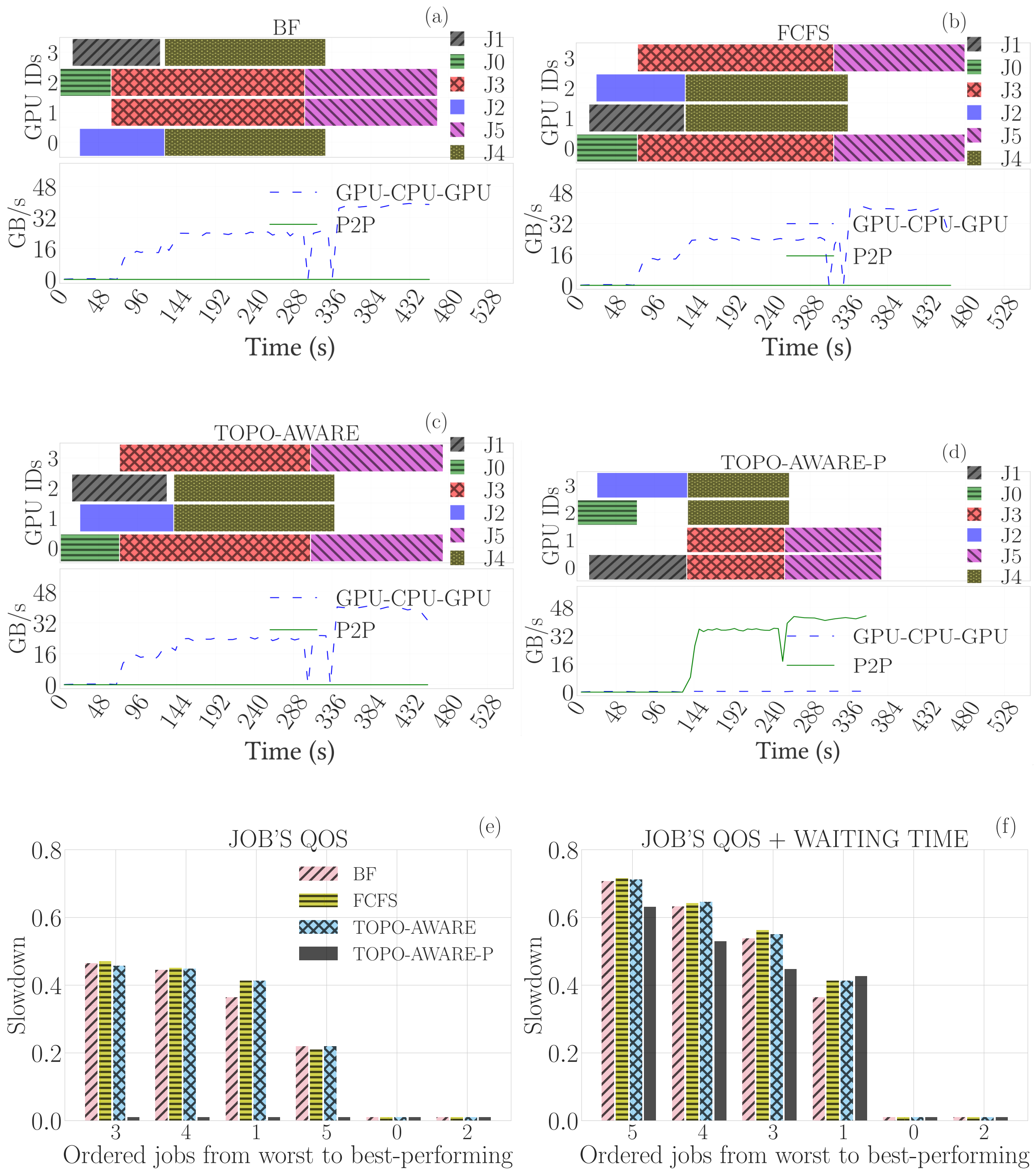


Figure 17: [Prototype] Figures (a) to (d) present the timeline of the placement decisions of each evaluated algorithm. A colored box can be on one or more GPU IDs, which represents the GPU allocation for a job. It also presents for each configuration the average NVLink bandwidth. Figures (e) and (f) present the slowdown in comparison with the ideal scenario and the jobs are ordered from worst to best-performing.

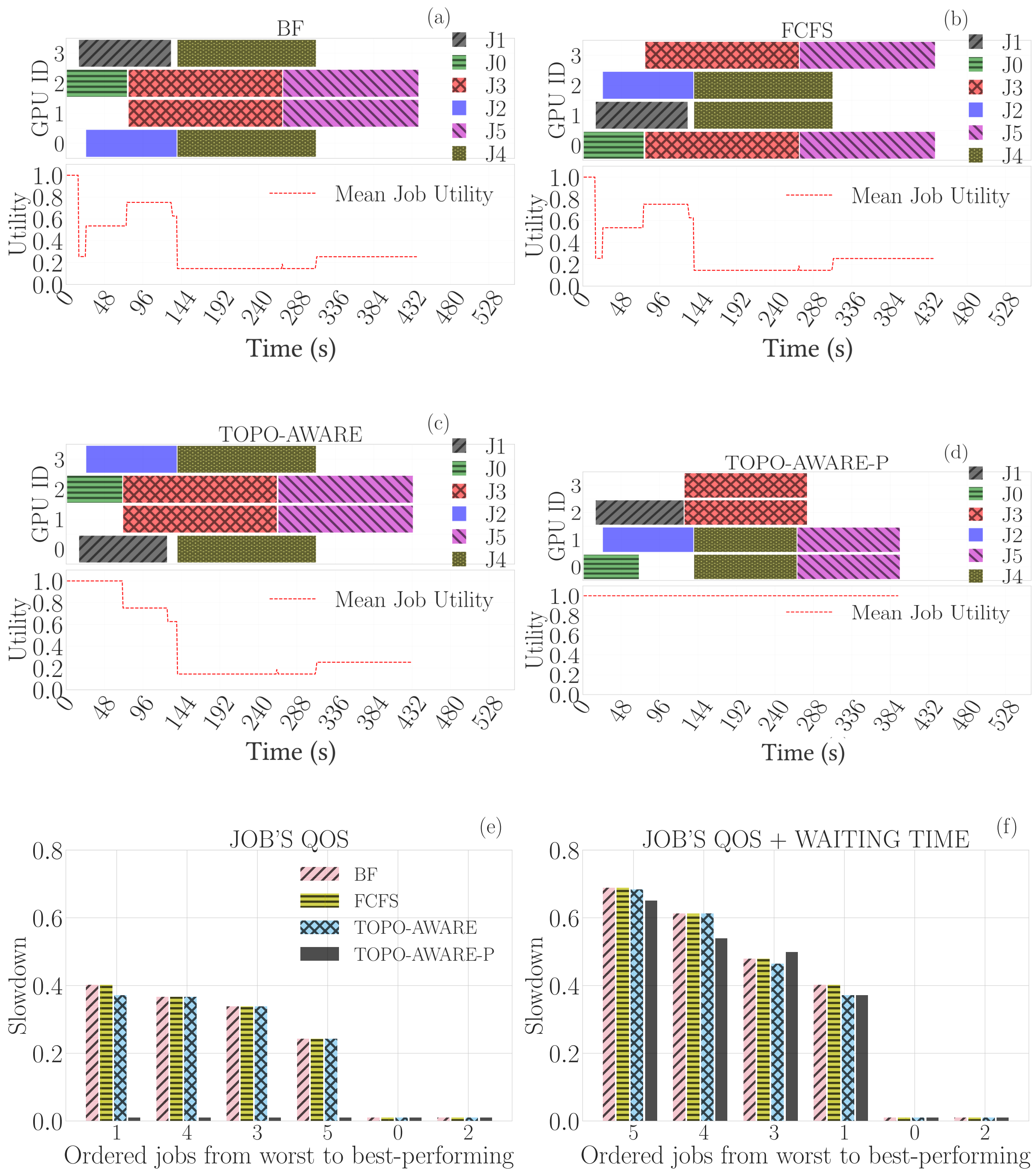


Figure 18: [Simulation] Behavioral description of the simulation performing a similar experiment to that shown for the prototype in Figure 17. Figures (a) to (d) present the time line of the placement decisions of each evaluated algorithm. A colored box can be on one or more GPU IDs, which represents the GPU allocation for a job. It also presents for each configuration the average job utility. Figures (e) and (f) present the slowdown in comparison with the ideal scenario and the jobs are ordered from worst to best-performing.

in Table 7. Jobs' arrival time follows a Poisson distribution configured with $\lambda = 10$ (i.e., the arrival of ten jobs per minute), except the Job #0 which arrives at time $t = 0.51s$ to introduce the initial load in the system. We set equal weights (0.33) to the parameters of the utility function in Equation 2 to provide equal consideration for communication cost and resource interference and fragmentation. Small batch sizes represent a reliable example of NNs that requires high GPU communication (especially for NNs using model-parallelism). Hence, we conduct this experiment using small batch sizes.

4.3.2.2 *Prototype experimental results*

The results are shown in Figure 17. In the beginning, only Job #0 is being placed. And since it requires only one GPU and there is no other job to cause interference, any placement decision fully satisfies its requirements. At the 15th second, Job #1 arrives, and the profile indicates that it suffers interference from Job #0. Thus, the overall system utility will be lower if Job #0 and Job #1 are collocated in the same CPU socket. On the other hand, TOPO-AWARE-P prevents undesirable collocation; it places Job #1 on a different socket than Job #0. When Job #3 arrives, it cannot be placed since it requires more GPUs than available. So Job #3 is only placed after Job #0 has finished, ≈70th second. However, at this point resource availability is non-uniform: the available GPUs are in different sockets.

Here is where the TOPO-AWARE-P differs from the other approaches. If Job #3 receives the two free GPUs, one from each of the sockets, this will result in cross-socket communication over the CPU bus and results in lower performance. For this reason, the TOPO-AWARE-P delays the job placement to until it can allocate co-located GPUs, that is, when these GPUs become available. Any job with the utility lower than a threshold defined in the job's profile will have the placement postponed to the next scheduler iteration. As a result, the TOPO-AWARE-P policy performs better in regards to job's execution time than the other policies, as shown in Figure 17 (d) vs Figures 17 (a)-(c). For example, Job #3 had the completion time as ≈120s for the scenario with the TOPO-AWARE-P (Figure 17 (d)), and ≈240s with the other algorithms. Note that the performance improvement is mainly related to enabling P2P over the NVLink interface to Job #3. Only the TOPO-AWARE-P provides P2P for jobs as shown in Figure 17 (d), in all the other scenarios the GPU communication is routed through the processor's memory, which leads to higher latency, and lower bandwidth because of additional memory copies and potential contention of the shared bus.

The quality of the placement is highlighted in Figure 17 (e) and (f). Both figures show the job's slowdown compared to the ideal scenario, where the job has the fastest execution time. Also, both figures sort jobs from worst to best-performing. While Figure

17 (e) focuses on showing the job slowdown strictly related to the placement decision, Figure 17 (f) shows the slowdown also considering the waiting time in the scheduler's queue. The results indicate that TOPO-AWARE-P is the most efficient algorithm. For instance, with TOPO-AWARE-P, jobs #1, #3, and #4 have no slowdown compared to the best-performing scenario, while these same jobs suffer ≈50% slowdown when the other algorithms are making placement decisions, as shown in Figure 17 (e).

Intuitively, delaying jobs gives the impression that the queue waiting time might end up being longer. However, the results surprisingly show that TOPO-AWARE-P has a lower waiting time for some jobs than other algorithms, as shown in Figure 17 (f). This happens because having better knowledge of the requirements enables the scheduler to prevent performance interference, and then some jobs will execute faster, opening space to place other jobs sooner. This can also be seen in the job makespan (cumulative execution time) of the algorithms. BF finishes in ≈461.7s, FCFS in ≈456.2s, TOPO-AWARE in ≈454.2s, and TOPO-AWARE-P ≈356.9s. Hence, TOPO-AWARE-P affords a speedup of ≈1.30x, ≈1.28x, and ≈1.27x, respectively.

4.4 TRACE-DRIVE SIMULATION EVALUATION

Based on the logs from the prototype described in Section 4.3, we developed a trace-driven simulation to evaluate the scheduling algorithm in large shared clusters. In this section, we first describe the main characteristics and configuration of the simulation. And second, we validate the simulation and perform experiments with a larger number of jobs and machines.

To evaluate the scalability, the proposed algorithm was executed to handle trace-driven simulated data at different scales of the system. The traces are generated by performing multiple experiments on the previously described prototype. Afterward, the trace files are parsed and transformed into a format compatible with the simulator, creating application and resource usage profiles. For generating the workloads, a Poisson distribution with arrival rate $\lambda = 10$ is used. To create the job's configuration, we used a Binomial distribution generating integer values between 0 and 3 to define the batch size, where 0=tiny, 1=small, 2=medium, and 3=big. And also a Binomial distribution generating integer values between 0 and 2 to determine the NN type, where 0=AlexNet, 1=CaffeRef, and 2=GoogLeNet. Additionally, all simulated machines are homogeneous and follow the hardware topology described in Section 3.2.2. All the jobs can run in the machines when there are enough resources.

4.4.1 *Validation of The Simulation*

We validate the reliability of the simulation system by comparing it with the same scenario as in the prototype experiments in Section 4.3. The simulation results are shown in Figure 18. The algorithms behave very similarly in both the prototype and the simulation, despite some expected small differences, which are acceptable when considering the standard deviations.

4.4.2 *Large-Scale Cluster Simulation and Results*

To verify the behavior of the proposed algorithm in a large-scale environment, we use the trace-driven simulation in two different scenarios as follows.

4.4.2.1 *Scenario 1: 100 jobs and 5 machines*

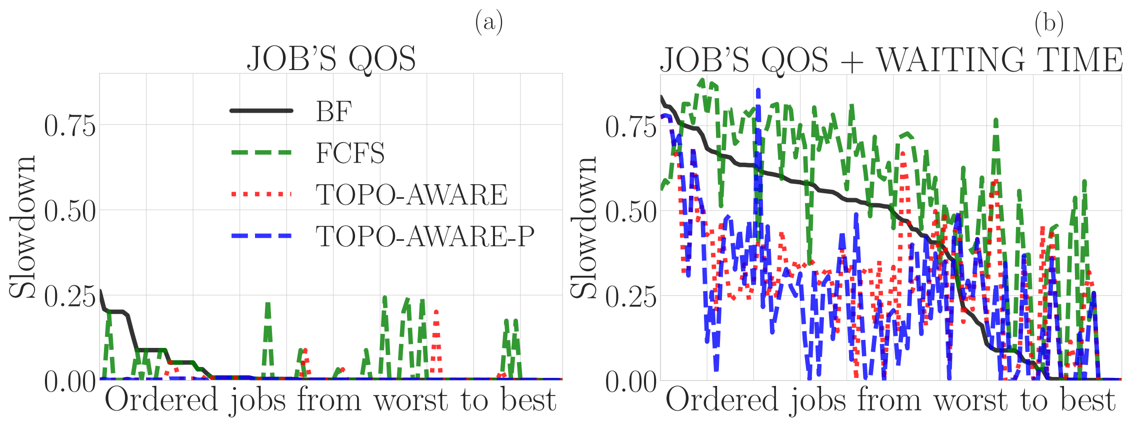


Figure 19: Scenario 1: 100 jobs and 5 machines. Job's slowdown relative to the best performing configuration.

We start the first experiment with a few machines and jobs. The results in Figure 19 (a) show that the TOPO-AWARE-P policy performs slightly better than the other; it does not violate the job's SLO. The other strategies introduce similar slowdowns in general, except FCFS that adds slowdown in more jobs.

The performance difference between the placement strategies is more evident when analyzing the waiting time of jobs in the scheduling queue, as illustrated in Figure 19 (b). Both TOPO-AWARE and TOPO-AWARE-P clearly outperform the greedy algorithms.

The lower performance of the greedy algorithms is explained by the fact that a sub-optimal placement decision can also limit the possible placements of other jobs. If a fragmented machine is left with only one GPU and the waiting jobs require more GPUs, the jobs must wait to be placed until enough resource becomes available. While less expressive, TOPO-AWARE-P performs better than only TOPO-AWARE. The second still presents slowdown in some jobs, and the former does not, since it allows out-of-order execution of jobs. TOPO-AWARE-P results in better performance because it does not schedule jobs to resources that do not fully satisfy its QoS.

4.4.2.2 *Scenario 2: 10k jobs and 1k machines*

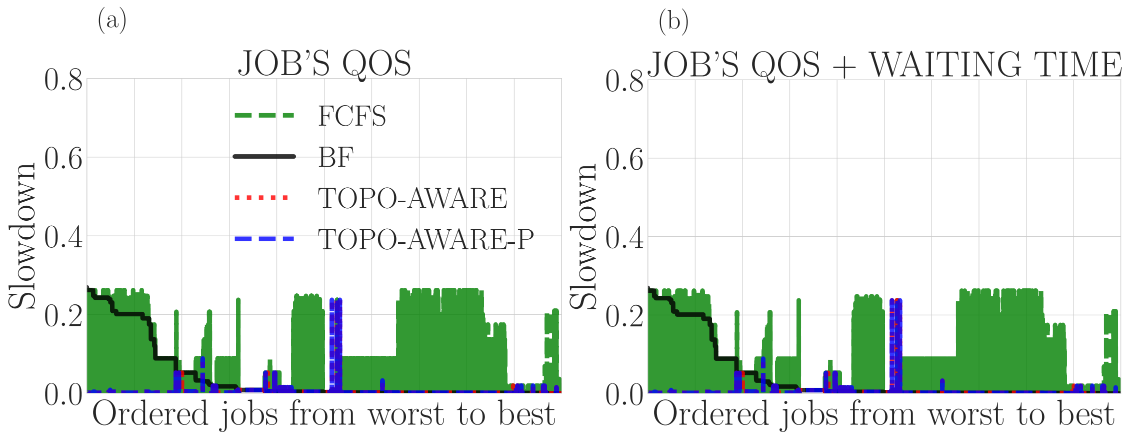


Figure 20: Scenario 2: 10k jobs and 1k machines. Job's slowdown relative to the best performing configuration.

The results in Figure 20 show that the FCFS algorithm has the worst performance, followed by BF. In summary, the new algorithm significantly and consistently outperforms the greedy algorithms in achieving the least slowdown and in minimizing the waiting time. The new algorithm's ability to achieve this is mainly due to its utility-based heuristics and the strategy that does not place jobs when the placement is not efficient from a communication perspective.

4.4.2.3 *Overhead*

The average time that the algorithms spend when evaluating the placement decision in scenario 2 is ≈3s for TOPO-AWARE and TOPO-AWARE-P, while for FCFS and BF it is ≈0.45s and ≈0.44s respectively. Although the proposed algorithm has higher overhead, that is, only 3 seconds on average, it is still fast enough for scheduling learning workload on a cluster with high demands.

The proposed algorithm has a higher execution time than the greedy ones mainly because it requires more computation to provide a better decision. Note that in the worst case, our proposed algorithm will evaluate $\Theta(|V_P|) * \Theta(|E_A| * \log_2(|V_P|))$, where the first Θ represents the host filtering phase and the second represents the phase to make the placement decision. Where the $|E_A|$ is the number of edges from the job's graph and $|V_P|$ is the number of a vertex from the physical graph. The other greedy algorithms have the asymptotic complexity as $\Theta(|E_A| + |V_P|)$ since every machine will be explored in the worst case.

4.5 RELATED WORKS

As we discussed in Section 3.2.5, over the last years, there have been various research efforts to optimize the performance of task placement on NUMA architectures. In Section 3.2.5, we have only focused on the description of the works regarding the performance evaluation of applications running on such environments. In Section 3.3.4, we have described the related works in regards to the performance evaluation of co-located applications. In this section, we detail the works that have proposed scheduling technique for tuning the performance of applications considering the underlying node topology.

Many previous researchers have been proposing heuristics for graph mapping such as graph contraction [10], and graph embedding [135], [143], [89], [137] and recursive bi-partitioning algorithm [33] that has been implemented in the software package SCOTCH [113]. Note that, the SCOTCH library implements many mapping algorithms, and one of them is the DRB, the algorithm that we have extended in our approach. [49] evaluated the performance of many parallel graph partitioning algorithms; they compared ParMETIS and PT-SCOTCH (parallel SCOTCH) frameworks. Their results show that for the majority of the experiments, PT-SCOTCH had better performance. While those methods have been proved to be an effective approach, most of them are contiguous with static allocation approaches leading to resource fragmentation and focus only minimizing the communication cost, not considering the other characteristics, such as the resource sharing-induced performance interference.

Topology-aware mapping has also been extensively studied in the context of CPU-to-CPU communications. In [120], the authors propose a topology-aware mapping mechanism for two of the MPI topology functions, but they do not consider the GPU-to-GPU communication topology. The CPU communications are extracted via profiling the application, and an undirected graph structure is generated to represent it with weighted vertices and edges. In addition, a weight is assigned to each vertex of a process, and edges between processes to represent the computation and communication requirements, respectively. After that, the actual mapping is performed by the SCOTCH library. Mércier and Jeannot [92] modify the implementation of functions in MPICH2 to be topology-aware with reordering of processes. In this implementation, one process on each multi-core node extracts the architecture of that node and sends it to a global root process. This work, similarly to our work, represents the whole hardware architecture of the system in a tree structure considering the hierarchical architecture of the NUMA nodes. The authors implemented an algorithm called TreeMatch that performs the mapping of a graph describing the jobs communication pattern and the graph that represents the physical topology. With a bottom-up approach, the algorithm works recursively on each level of the memory hierarchy in such a way that the cost of remaining communications is minimized. The algorithm performs the graph mapping algorithm using the SCOTCH library. Kindratenko et al. [75] proposed a CUDA wrapper that works in sync with Torque batch system. The wrapper overrides some CUDA device management API calls to expose GPUs to users, taking into account the heterogeneous clusters with machines with different CPU-GPU bandwidth. The work that is more related to our work is the work in Faraji et al. [35] that evaluated the performance of difference intranode GPU communication channels and proposed a topology-aware GPU selection scheme to assign GPU devices to MPI processes based on the GPU-to-GPU communication pattern and the physical characteristics of a multi-GPU machine. With profile information from the MPI application, the algorithm allocates GPUs performing a graph mapping algorithm using the SCOTCH library.

However, while the approaches of these previous works effectively minimize the communication cost between GPUs and CPUs, they do not consider the possible performance interference from co-scheduled jobs as we do in our work. Moreover, they only implemented topology-aware GPU selection in the task-level scheduling (e.g., extending the MPI runtime), in contrast, we proposed and implemented a cluster-level scheduling algorithm. Our approach has the advantage of having the global view of the system, making decisions based on all the jobs running in the clusters and not only for the tasks of a specific application. In addition to that, these works only conducted experiments using synthetic micro-benchmarks, and we performed our work using a macro-benchmark that emulated real DP applications using real data.

In addition, there exist various previous works that evaluated and proposed scheduling approaches to mitigate performance interference for co-scheduled jobs. Verma et al. [140] propose a power-aware placement algorithm that considers the performance impact of co-location of heterogeneous applications with small and large memory footprint. For example, some applications that the total working set size is smaller than the physical machine's CPU cache size will degrade in performance if they are packed with large applications because of thrashing in the CPU cache. Also, some applications, whose working set does not entirely fit into the cache, will be impacted by other applications on the same machine. The work investigated the aspects of modeling the power consumption of applications and the impact of platform virtualization, and then, used the generated models to make decisions with the proposed algorithm.

Other works proposed scheduling algorithms to avoid problematic collocation within the same machine. The work [105] proposed a system called DeepDive that transparently identifying and managing performance interference between virtual machines co-located on the same physical machine in Infrastructure-as-a-Service cloud environments. DeepDive transparently inspects low-level metrics from hardware performance counters and hypervisor statistics about each VM. It has an analyzer that clones the VM on-demand and executes it in a sandboxed environment. A proxy duplicate the client request to also send it to the cloned VM. The analyzer then uses the low-level measurements to estimate the performance of the original and cloned VMs. In the absence of interference, the analyzer updates the repository with the new information, otherwise, it calls a placement-manager to determine a preferable change in the VM placement that mitigates the interference. That is, it migrates the VM to eliminate or reduce performance interference. Delimitrou and Kozyrakis [27] proposed a cluster management system called Quasar that use collaborative filtering techniques to predict the possible performance impacts of co-located jobs to determine optimal scheduling and placement. Quasar monitors workload performance and adjusts resource allocation and assignment when needed. The results show that the proposed system improves resource utilization by 47% in an experiment with 200 servers in the Amazon EC2 cluster, while meeting workload's performance constraints. Nathuji et al. [103] proposed a cluster management system Q-Cloud that tunes resource allocations to mitigate performance interference effects on cloud systems. Q-Cloud allows applications to specify multiple levels of QoS as application Q-states. With such information, Q-Clouds dynamically provisions underutilized resources to enable elevated QoS levels, thereby improving system efficiency. Their experimental results show that in their analysis Q-Clouds could improve the system utilization up to 35%.

Additionally, other works proposed scheduling algorithms with best-efforts to minimize the resources interference on co-scheduled CPU-based applications via performing low-level resource partitioning. For example, Qureshi and Patt [119] proposed a utility-based cache partitioning (UCP) mechanism partitions a shared cache between multiple applications depending on the reduction in cache misses that each application is likely to obtain for a given amount of cache resources. Their proposed approach has a lightweight monitoring mechanism that requires less than 2kB of storage. The information collected by the monitoring circuits is used by a partitioning algorithm to decide the number of cache resources allocated to each application. Their experimental evaluation shows that UCP can improve the performance of a dual-core system by up to 23% and on average 11% over LRU-based cache partitioning. Gundu at al. [47] proposed a memory bandwidth reservation technique in the cloud to avoid information leakage in the memory controller. This work, differently from prior works that implemented temporal partitioning, proposes and evaluates bandwidth reservation. Via simulations, they show that while temporal memory partitioning can degrade performance by 61% in an 8-core platform; their bandwidth reservation only degrades performance by under 1% on average. Finally, Lo et al. [88] a feedback-based controller that dynamically manages multiple hardware and software isolation mechanisms, such as CPU, memory, and network isolation (using containers, cache partitioning and network traffic control mechanisms), to ensure that the latency-sensitive job meets latency targets while maximizing the resources by co-locating best-effort tasks (i.e., Batch analytics frameworks). In their experimental evaluation, using production latency-critical and batch workloads from Google, they demonstrated average server utilization of 90% without latency violations across all the load and colocation scenarios that we evaluated, where the typical utilization was between 10% to 50% without they proposed system.

These previous works describe the performance bottlenecks for CPU-only application and/or providing best-efforts on mitigating workload performance interference of co-schedule jobs. However, they neither directly show the performance constraints of mixing multiple GPU-based learning workloads, nor do they propose a GPU-topology-aware scheduling algorithm as we do. Therefore, in this thesis, differently from the above-related works, we further analyzed and mitigated some possible performance problems, and leverage P2P communication for multi-GPU based learning workloads in a co-scheduled environment.

4.6 FINAL CONSIDERATIONS

Multi-GPU applications are becoming popular because they can deliver performance improvements and increased energy efficiency. But at the same time, they present new challenges as they usually require inter-GPU communications. Such communications can take place directly between devices (with P2P) or may need to be routed through the processors' main memory, depending on the system topology and the resource allocations for the existing jobs.

In this thesis, we presented a new topology-aware placement algorithm for scheduling workloads in modern multi-GPU systems. The foundation of this approach is based on the use of a new graph mapping algorithm built from application objectives and the system topology. Applications can express their performance objectives as SLOs that are later translated into abstract utility functions to drive the placement decisions. The algorithm has been validated through the construction of a real prototype on top of an

IBM POWER8® system enabled with 4 NVIDIA Tesla P100 cards, as well as through large-scale simulations.

Our experiments show that our algorithm effectively reduces the communication cost while preventing interference related to resource contention, mainly for the scheduling policy that allows postponing the placement of unsatisfied jobs. In particular, with this policy, the performance impact of minimizing the GPU communication cost and avoiding interference reflects in a speedup of up to ≈1.30x in the cumulative execution time, and no SLO violations. Finally, a trace-driven simulation of a large-scale cluster reveals that compared with greedyapproaches our algorithm produces solutions that satisfy more jobs, minimizes the SLO violations and improves the job's execution time even in a heavily loaded scenario.

Although in the simulation we could experiment with heterogeneous machines to show the impact of different NUMA typologies, this was not the goal of this experiment; we only aimed to validate the system based on the physical machines used in the prototype experiment. In addition, by the time of this experiments, heterogeneity in cloud providers was still not common as investigated in [80] (e.g., only found in Microsoft Azure and a small number of Amazon EC2 instance). Therefore, we leave the evaluation with heterogeneous machines (composed of different NUMA topology and number of GPUs) for future work.

5 WORKLOAD ORCHESTRATION FOR POOLED RESOURCES AND DISAGGREGATED ARCHITECTURES

Traditional datacenters consist of monolithic building blocks that tightly integrate a small number of resources (i.e., CPU, memory, storage, and accelerators) for computing tasks of the system software and applications. 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 spare resource fragmentation [39, 71, 112]. To illustrate this issue, consider Figure 3 (a), where two CPU-bound jobs are saturating 100% of processor cores from node #1, and the GPUs available in this node cannot be allocated to other jobs due to lack of processing resources. In such a scenario, even though a clever orchestrator could assign a workload to a proper compute node, resource fragmentation still exists because a waiting job cannot allocate the GPU resources from node #1. However, if the idle GPUs from node #1 could be remotely exposed to node #2, then the waiting jobs could start running.

Determining the best-performing resource provisioning and job scheduling, in most of its relevant forms, is known to be NP-hard and considering the possibility to allocate disaggregated resources, further complicates this task. Furthermore, the optimality of the placement depends on the performance variability related to the topology (i.e., the topology is composed by both the resources that are local in the machine and the disaggregated ones that are accessible over a heterogeneous network topology) and the likelihood of sharing-induced performance interference because of co-scheduled jobs. Therefore, an ideal orchestrator would efficiently and transparently allocate disaggregated resources to maximize the cluster utilization while employing best-effort approaches to prevent violations of the applications' Service Level Objectives (SLOs) (e.g., prevent the completion time of an application overpass a given threshold).

To that end, in this thesis, we present *DRMaestro*, a flow-network-based framework that orchestrates disaggregated resources on cloud systems to help meet SLOs while maximizing the system utilization. The main idea of *DRMaestro* is to automatically discover and allocate disaggregated resources in the cluster for a job as if the resources are attached to the local machine where the job is placed. From the job standpoint, it is only using local resources. For that reason, building *DRMaestro* poses interesting research challenges, such as, how to: 1) enable transparent resource disaggregation, 2) automatically control and determine the optimal placement with an online scheduling approach, and 3) cope with sharing-induced performance interference because of co-scheduled jobs. To the extent of our knowledge, our work is the first one to apply a flow-network model to solve the scheduling and placement problem considering resource disaggregation.

Enabling transparent resource disaggregation is our first challenge because of the complexity of configuration and technology constraints. Disaggregation can be provided either by software or hardware-based approaches and in both cases, the orchestration system must to start new services and configure middlewares and/or drivers for each application that will have the disaggregated resource allocated. The application must not be aware that it is accessing a remote resource and all the steps to configure it must be automated. Additionally, not all type of resource can be disaggregated since disaggregation poses additional overheads. When the access latency and bandwidth of the remote resources become noticeable, there might be a significant performance penalty. Disaggregated GPU, FPGA, and SSD are much less performance demanding than CPU and DRAM as they are likely to have their local memory, and will often enlist in computations that last many milliseconds. Thence, in this work, we focus on GPU disaggregation, but the proposed method can be applied to any resource.

Optimal placement is challenging because the requirements of performance, fairness, and cloud provider often conflict. The optimality conditions for the problem are the equilibrium of the conditions. Hence, the goal of our orchestrator is to minimize the placement and scheduling cost for a set of jobs while maximizing the cluster utilization. For doing so, our proposal generalizes the algorithm in [41, 65]. *DRMaestro* first translates the job's SLO (e.g., completion time) into abstract utility functions and cost models that will dictate the job's level of satisfaction with the placement decisions. Second, both the job and cluster resources are expressed as graphs to model the problem conveniently. Third, the algorithm employs a graph mapping technique based on flow-network disciplines to determine optimal placement, as detailed in Section 5.3.

Mitigating sharing-induced performance interference is important to be considered. While resource disaggregation is a promising approach to increase resource efficiency, it comes with the price of possibly introducing an inherent performance interference. Remote network, CPU and memory usage can cause interference in a set of sensible applications. Thus, our last challenge is dealing with the inherent performance interference of collocated applications in a shared environment. We do this by off-line and dynamically collecting historical runs to derive the job interference when collocated with other jobs within the same node and over network overloaded conditions. Then, based on a simple yet effective classification, we mitigate the interference by preventing the collocation of critical applications when possible. We detail this in Section 5.3.

We validate our design implementing *DRMaestro* over Kubernetes [123] as shown in Section 5.4, and running representative applications in a cluster with 4 POWER8® machines with 4 GPUs each, as further discussed in Section 5.6. The applications used are based on the Rodinia Benchmark Suite [21]. These applications stress both computational throughput on CPUs and GPUs, and we use them to demonstrate the extensibility of *DRMaestro* and to evaluate the efficiency and scalability of our proposal. We then perform several faithful trace-driven simulations with traces from the testbed and show that our solution provides higher cluster utilization and lower SLO violations.

5.1 MOTIVATION AND BACKGROUND ON DISAGGREGATED RESOURCES

Although there is a vast range of available cluster resource managers, even the widely deployed ones such as Mesos [53], Kubernetes [123], YARN [138], and Borg [139], to the best of our knowledge, they do not provide native mechanisms for orchestrating disaggregated resources transparently to applications. More specifically, they lack an intelligent scheduling algorithm that considers the possibility to allocate disaggregated resources when there are not enough resources in a server. And, they also have the absence of mechanisms to transparently enable resource disaggregation like launching software-based middlewares and injecting the necessary information into the application to work with the middlewares or the necessary drivers to control the racks with hardware-based resource disaggregation. Additionally, most of the available resource managers perform only a task-by-task placement instead of a batching placement, restricting the decisions due to not having further consideration of waiting jobs. For each job, they typically first verify the feasibility to identify a suitable machine (i.e., if it has enough resources), then scores them according to a preference order, and finally enforces the placement of the job on the best-scoring machine.

Therefore, we believe that an efficient orchestration that manages GPU disaggregation will be a must feature to improve the resource efficiency of next-generation of data centers. A large set of applications will benefit from that, ranging from HPC, Deep Learning to Cloud Gaming. Examples are the data centers composed by server-centric nodes (e.g., the Amazon P2, the IBM® POWER8® boxes, the NVIDIA® DGX-2, etc.) and rack-centric nodes (e.g., the Facebook Disaggregated Rack [62], dReDBox [71], etc.) with multiple GPUs to run applications, as well as the NVIDIA® Cloud Gaming platform [102]. A comprehensive introduction defining resource disaggregation and detailing both the software and the hardware-based architectures is given in Section 2.3.

5.2 DRMAESTRO OVERVIEW

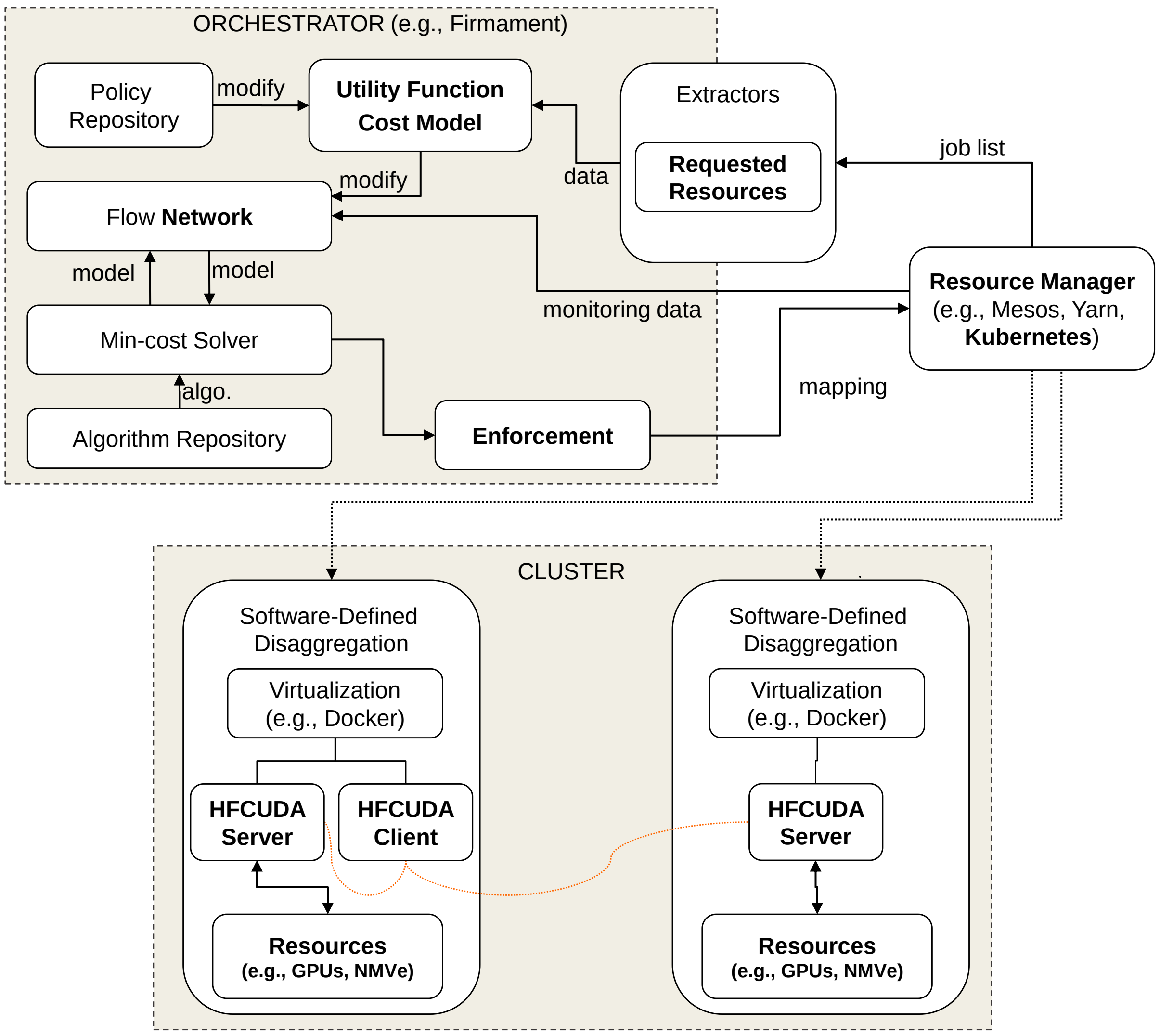

ORCHESTRATOR (e.g., Firmament)
Policy Repository
modify
Utility Function Cost Model
Extractors
Requested Resources
data
modify
Flow Network
job list
model
model
Min-cost Solver
monitoring data
Resource Manager (e.g., Mesos, Yarn, Kubernetes)
algo.
mapping
Algorithm Repository
Enforcement
CLUSTER
Software-Defined Disaggregation
Virtualization (e.g., Docker)
HFCUDA Server
HFCUDA Client
Resources (e.g., GPUs, NMVe)
Software-Defined Disaggregation
Virtualization (e.g., Docker)
HFCUDA Server
Resources (e.g., GPUs, NMVe)

Figure 21: Conceptual view of *DRMaestro's* architecture. The cluster can be composed of both server-centric (i.e., implementing a software-based resource disaggregation) and/or rack-centric (i.e., implementing a hardware-based resource disaggregation) nodes. The orchestrator receives a job list along with the job's profile when available to determine the optimal placement on the available nodes. The orchestrator might also start the disaggregated resource daemons, when necessary, to transparently allocate disaggregated resources to the jobs.

We have established the need for a dynamic orchestration framework that efficiently manages disaggregated resources to improve the cluster utilization while fairly maximize the job's utility. In this section, we present our proposed framework *DRMaestro* that assigns a set jobs to a set machine considering the possibility to disaggregated the resources, where a given machine can remotely access the resources of other machines in a way that the applications think that all the resources are local. **First**, *DRMaestro* implements a flow-network model to determine both the optimal scheduling and placement for a given set of jobs in an online scheduling approach. Each requested resource that is possible to be disaggregated is treated individually (e.g., if both GPUs and NVMes are disaggregated, they are allocated on different phases). This is due to carefully separating the optimization problem into independent sub-problems is effective at reducing the complexity of finding the optimal placement. That is, we use a divide and conquer approach. **Second**, our approach is incremental as it decides to place each resource at a time without modifying the allocation plan from both previous decisions and running jobs. **Third**, *DRMaestro* accomplish fairness and prevent starvation by equalizing the achieved relative performance between jobs. **Forth**, security is lower for GPU on the cloud [107]; there is no further isolation mechanism during the execution. Therefore, *DRMaestro* dynamically allocates GPUs attempting to enforce isolation as much as possible using the *virtual environment* mechanisms, such as containers cgroups, namespaces, sysmted, among other tools.

5.2.1 *The Key Components of the DRMaestro's architecture*

The *DRMaestro's* architecture is depicted in Figure 21, showing that the framework is composed of different modules. It has a resource manager that interacts with the end-user receiving the submission of jobs and forwards the job list along with their information to the orchestrator that is responsible for making the scheduling and placement decisions. After the orchestrator has made the decisions, it forwards them to the resource manager enforce them. Next, we detail the key components of system architecture.

THE RESOURCE MANAGER 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, 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 initiated.

THE EXTRACTION OF SLOS AND REQUESTED RESOURCES occurs just after the orchestrator has received the job list; it then extracts the performance goals from the manifests, and then, when available, an offline job profile is also used to determine the know scenarios that the application might have its performance impacted. Note that, the user either defines the application SLO (i.e., performance goals) or explicitly approves it. In this work, we focus on non-interactive workloads, where the performance goal is typically relative to the completion time. Nonetheless, the performance goals can be easily extended to include support for interactive workloads (e.g., web-services) in which the performance goals are relative to average or percentile response time or throughput over a short time interval.

THE COST MODEL depends on the scheduling policy, the job's waiting time, and the extracted job's information that we describe later on.

THE FLOW-NETWORK MODEL incrementally determines the optimal resource allocation considering all jobs at once. For each iteration (i.e., when the orchestrator is trigged), the model is updated with information of the cluster state and the cost models of the incoming jobs. More details are given in Section §5.3.2.

THE MIN-COST SOLVER is applied for each flow-network model to determine which jobs should be activated (i.e., scheduled or unscheduled), match the jobs' resources to the available nodes. It is noteworthy that the solver first places a task representing the CPU with all the other resources that cannot be disaggregated, which will guide later the placement of disaggregated resources via defining placement preferences.

THE MIN-COST ALGORITHM, in our model, any min-cost algorithm for flow-network model to solve online scheduling problems can be used in the Solver.

5.3 NETWORK-FLOW-BASED SCHEDULING AND PLACEMENT ALGORITHM

In this section, we first describe the problem and formally define it, and after, we detail the proposed framework to address the problem describing its architecture and algorithm.

5.3.1 *The Scheduling Problem*

The scheduling problem, in most of its relevant forms, is known to be NP-hard, most specifically when considering multiple mutually dependent scheduling goals. To tackle that problem, a promising approach that has been widely investigated and efficiently solved is by using graph isomorphism (aka, Graph Matching Theory) [2, 15, 90]. By modeling the scheduling and resource provisioning 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 cost [2, 41, 42]. A bipartite graph consists of two sets with one representing the n jobs and other the m machines, and arcs (i, j) weights representing the placement cost c_{ij}.

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 job's utility depends on any prevailing system flow, which might introduce an inherent delay. Note that, this problem should attain an equilibrium because of the delay that one job incurs depends on the flow of other jobs, and all jobs are simultaneously choosing their best path. Moreover, 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.

Therefore, our framework divides the problem into multiple sub-problems and performs an incremental approach by allocating each disaggregated resource in a different phase. It is incremental since, in each phase, we do not change the previous ones, but compose the next phase with information from the previous one. More specifically, for example, lets assume that the first phase will place the "main" set of resources, in this case, the CPU and memory, and other phases will place the disaggregated resource (the GPU), where each phase relies on the placement preferences defined in the previous phase (if any). In such a model, we define that the assignment of jobs is shepherd via both the jobs and machines preferences expressed as costs and capacity limits to satisfy a global goal. The solver then determines the optimal flow, exhibiting the best trade-offs between (i) activate a job or keep it unscheduled, (ii) place the job in one or another machine, and (iii) determine when to allocate a disaggregated resource or allocate it locally remotely.

5.3.2 *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, the $b_i > 0$, 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 the Equation 6, while respecting the *feasibility constraints* in Equation 7 and the capacity in Equation 8 and 9, as follows:

$$\text{Minimize} \sum_{(i,j)\in A} c_{ij} f_{ij} \tag{6}$$

subject to

$$\sum_{j:(i,j)\in A} f_{ij} - \sum_{j:(j,i)\in A} f_{ji} = b_i, \quad \forall i \in N \tag{7}$$

$$0 \leqslant f_{ij} \leqslant x_{ij}, \quad \forall (i,j) \in A \tag{8}$$

$$\text{and} \quad \sum_{i\in N} b_i \leqslant 0 \tag{9}$$

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 being represented, each flow, demand or capacity will have the sum of the normalized values of each type of resource $r \in R$. For example, let's consider the supply b_i, each resource will be normalized with the maximum existing supplier in the cluster, as shown in Equation 10.

$$b_i = \sum_{r\in R} \frac{b_{ir}}{max\_b_{ir}}, \quad \forall i \in A \tag{10}$$

The demands, capacities, costs and flows follow the same normalization pattern as done in Equation 10.

Despite that fact any utility function can be used, we have defined the following utility showed in Equation 11, which is inspired in the work of [19]. If τ_c represents an average completion time goal, the instantaneous utility of a job with the resources r and load l expecting the completion time $\varphi_{r,l}$ is given by:

$$Ut(\varphi_{r,l}) = \begin{cases} \frac{\tau_c - \varphi_{r,l}}{\tau_c} & \text{if } \varphi_{r,l} \leqslant \tau_c \\ (\frac{\tau_c - \varphi_{r,l}}{\tau_c}).(\frac{10 - z_c}{9}) & \text{if } \varphi_{r,l} \geqslant \tau_c \end{cases} \tag{11}$$

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.

$$c_{ij} = (1 - Ut(\varphi_{r,l}))P_i \tag{12}$$

$$c_{ij} = (1 - Ut(\varphi_{r,l}))P_i + W_i \tag{13}$$

The cost c_{ij}, as shown in Equation 12, is defined as the complement of the utility Ut_i times the preference P_i. The preference P_i is determined by the list of preferred machines from each job 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). The cost c_{ij} for all edges, except the one between the jobs i and the unscheduled state U_i, is determined by the Equation 12. On the other hand, cost c_{ij} of the edges that point to an unscheduled state U_i is determined by the Equation 13, which, in order to avoid starvation, has an incremental counter W_i that increases every time that the job i is left unscheduled after each scheduler iteration.

5.3.3 *Scheduling and Placement as Flow-Network Model*

Our proposed framework, *DRMaestro*, is a dynamic, loop-based controller that can manage resources from both server-centric and/or disaggregated architectures. The core of the proposed scheduling algorithm is illustrated in the Alg. 4. This algorithm receives a list of jobs from a waiting list sorted by the waiting time and filtered by the available resources in the clusters. The algorithm then calls the `mapping()` function with the sum of requested resources from the set of jobs lower or equal to the total of currently available resources in the clusters. The `mapping()` function updates the flow-network-model (which is detailed later in this section) with the new incoming jobs and returns the mapping of the jobs to machines. If a job is not mapped, it is added back to the waiting list, incrementing its waiting-time counter. On the other hand, if a job is scheduled, and disaggregated resources were allocated, to enforce the placement decisions, containers with the middleware to enable resource disaggregation is started. Note that, in the case of hardware-based resource disaggregation no middleware is needed to be launched.

Although, the 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, inspired in the models applied in [41, 65]. Our proposed model is illustrated in Figure 22 (a), (b) and (c). All resource groups (aggregated and disaggregated) are incrementally solved on different flow-network models to define the optimal placement. In each model, the node U_* determines if the job will be activated or left unscheduled. The node X represents an aggregator to reduce the number of possible arcs from jobs to machines

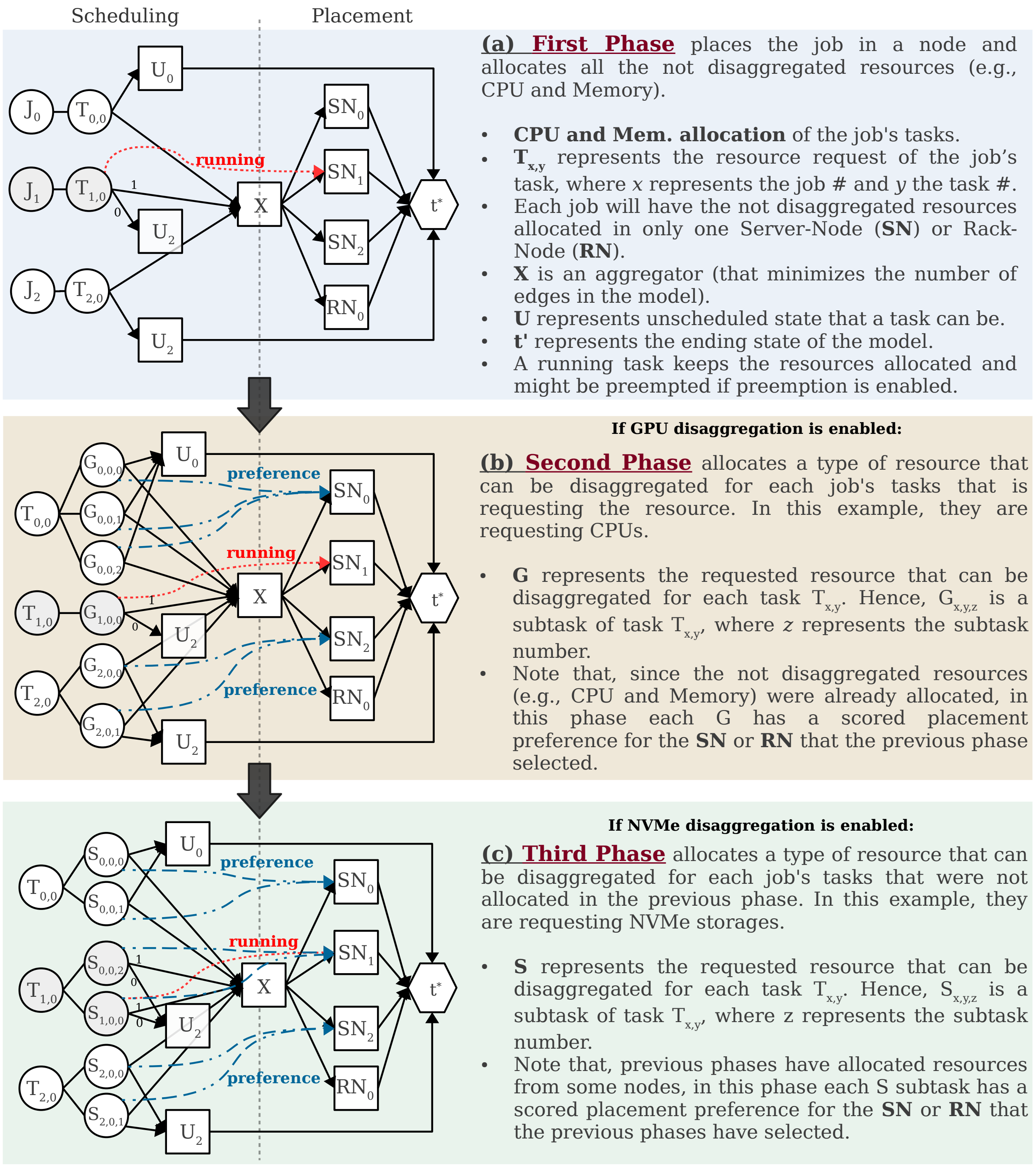
Algorithm 4 *DRMaestro*: scheduling and placing jobs

```
function scheduler( )
    jobs = Q.pop_jobs()
    m = mapping(jobs)        //Algorithm 5
    for job in m do
        if job.state == scheduled then
            for resource in disaggregated_resources do
                for {task, machine} in m[job][resource] do
                    bind_hfcuda_server_to_node(job, machine))
                end for
            end for
            wait_all_hfcuda_severs_start()
            machine = m[job][not_disaggr_group][0]
            inject_hfcuda_client_and_server_info(job.manifest)
            bind_job_to_node(job.manifest, machine)
        else
            job.waiting_counter++
            Q.push(job)
        end if
    end for
end function
```

(which might reduce the complexity, as shown in [41, 65]), the t^* is the sink node. The SN represent a server-node with software-defined resource disaggregation as in Figure 6 b). The RN represents a rack-node with hardware-defined resource disaggregation as in Figure 6 c). To apply the scheduling policy, the arc weights and capacity are properly determined.

In the first phase, as illustrated in Figure 22 (a), the $T_{k,m}$ represents each task of job J_k with the group of the aggregated resource (e.g., 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). This first phase determines the placement of the resources that cannot be disaggregated. The placement of each task $T_{k,m}$ will determine the placement preferences in the other phases.

After solving the first phase, the graph will be transformed to represent the next resource group. For example, Figure 22 (b) creates new "sub-tasks" $G_{k,m,z}$ for each z GPUs that each task $T_{k,m}$ is requesting. It allows to allocate 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, the task $T_{0,0}$ is placed in SN_0,



(a) First Phase places the job in a node and allocates all the not disaggregated resources (e.g., CPU and Memory).

- **CPU and Mem. allocation** of the job's tasks.
- $T_{x,y}$ represents the resource request of the job's task, where x represents the job # and y the task #.
- Each job will have the not disaggregated resources allocated in only one Server-Node (**SN**) or Rack-Node (**RN**).
- **X** is an aggregator (that minimizes the number of edges in the model).
- **U** represents unscheduled state that a task can be.
- **t'** represents the ending state of the model.
- A running task keeps the resources allocated and might be preempted if preemption is enabled.

If GPU disaggregation is enabled:

(b) Second Phase allocates a type of resource that can be disaggregated for each job's tasks that is requesting the resource. In this example, they are requesting CPUs.

- **G** represents the requested resource that can be disaggregated for each task $T_{x,y}$. Hence, $G_{x,y,z}$ is a subtask of task $T_{x,y}$, where z represents the subtask number.
- Note that, since the not disaggregated resources (e.g., CPU and Memory) were already allocated, in this phase each G has a scored placement preference for the **SN** or **RN** that the previous phase selected.

If NVMe disaggregation is enabled:

(c) Third Phase allocates a type of resource that can be disaggregated for each job's tasks that were not allocated in the previous phase. In this example, they are requesting NVMe storages.

- **S** represents the requested resource that can be disaggregated for each task $T_{x,y}$. Hence, $S_{x,y,z}$ is a subtask of task $T_{x,y}$, where z represents the subtask number.
- Note that, previous phases have allocated resources from some nodes, in this phase each S subtask has a scored placement preference for the **SN** or **RN** that the previous phases have selected.

Figure 22: All disaggregated resources are allocated incrementally in different phases. Each phase depends on the previous one and are optional depending which research can be disaggregated. Additionally, Each phase determines the jobs' placement preferences that directly changes the arcs costs to machines accordingly with the applied policy.

in the second phase, the sub-tasks $G_{0,*}$ will have a preference on SN_0, as illustrated in Figure 22 (b).

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. For instance, in Figure 22 (c), it will start a third phase to schedule and place the "sub-tasks" $S_{k,q}$ to allocate the storage, such as Non-Volatile Memory Express (NVMe). 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 it is transparent for the application.

Algorithm 5 *DRMaestro*: mapping jobs to machines

```
function mapping(incoming_jobs)
    previous_solved_model = nil;
    mapping_models = [[]]
    for (group in resource_groups)) do
        model = flow_models[group]
        if (previous_model not empty) then
            UpdateModelPreferences(model, previous_solved_model)
        end if
        mapping_models[group] = GetMappings(model)
        previous_solved_model = mapping_models[group]
    end for
    for (job in incoming_jobs) do
        job.state = scheduled
        for (task in job[group].tasks()) do
            machine = mapping_models[group][task]
            if (task not in mapping_models[group]) then
                job.state = unscheduled
            else
                mapping[job][group] = {task, machine}
            end if
        end for
    end for
end function
```

The final placement and resource provisioning results are extracted based on the analysis of the optimal flow from all solved models, as shown in Algo. 5. The Algorithm calls the solver for each model that represents a resource group by calling the function `GetMapping()`. This function calls the flow-network solver for the given model, runs the flow-network algorithm, and returns the mapping of tasks to machines. If a task is not mapped to a machine, it means that this task will not be scheduled. Since we do not remove any running task from each solved model, the feasibility is always maintained. Only the tasks and jobs that have completed are removed from the model. After the first phase, the other phases (if any) update the machine preferences based on the decisions made on the previous phase, then, the algorithm first calls the function `UpdateModelPreferences()` before calling `GetMapping()`. After the algorithm has solved the models of all group of resources, it parses all the solved models to identify which jobs will be scheduled or will be left unscheduled. By default, we gang schedule all resources (but the method can be easily extended to allocate resources incrementally). Therefore, in our proposal, if any "sub-task" is left unscheduled, the job will not be scheduled. It is worthy to mention that if a "sub-task" is left unscheduled, it is because the solver has optimally found that it is better to unscheduled it instead of allocating a bad performing resource.

5.4 THE DRMAESTRO IMPLEMENTATION

We implement the design of §5.2 as extensions to Firmament [41], which is a flow-network-based scheduler, Kubernetes [123] which is a cluster resource manager, and the Poseidon [133], which is the Firmament add-on Kubernetes scheduler. Referring back to the architecture of Figure 21, we implement the main components of *DRMaestro* as follows. First, the automatic orchestration engine operates as a standalone service. It continuously consumes telemetry data, and keep waiting for jobs. Second, the orchestrator is triggered when a list of jobs arrives. Third, the orchestrator Firmament's mapping engine is extended to create flow-network models for both the aggregated and the disaggregated resources. We further detail it as follows:

THE SCHEDULER ADD-ON onto Kubernetes is extended both to understand and interpret the new format of the placement decisions, as well as, it also generates and starts the HFCUDA client and server daemons, as shown in Algorithm 4 and further explained in Section 5.4.3.

THE INFERENCE ENGINE is a simple approach that maintains the information of the average job's completion time in a given 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 use a more sophisticated engine such as [26]. 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. As the experiments that we performed in Section 3.3. In the absence of a model, the application is classified as unknown using a neutral cost value (e.g., as 1). Most of the telemetry cluster data is given by Kubernetes Heapster.

ANY MIN-COST ALGORITHM can be used. Albeit there exist lots of suitable algorithms that can solve the problem in reasonable time, for this thesis we use an implementation of the *successive approximation push-relabel* algorithm (*aka* Cost Scaling) described in [42], successfully used in [41, 57, 65] and detailed in Section 5.4.1. The asymptotic complexity of the algorithm in our context is $O(K * N^2 M \log(NC))$, where K is the number of disaggregated resources plus one, N is the number of nodes, M the number of arcs, and C is the largest arc cost.

THE FINAL MAPPING is the aggregation of all individual mappings from each solved model. We use the Firmament's GetMappings function [41] to extract the mapping results from each solved model, and then, we combine their results, as shown in Algorithm 5.

THE SECURITY problem is mitigated with a simple yet effective approach. First, we use the available cloud virtualization mechanisms to isolate both the CPU, memory and storage, such as the container `cgroup` and namespaces, by creating OS containers. Second, the middleware that offloads the CUDA calls is also encapsulated into virtualization environments so that a user cannot access resources from others. Third, only the data related to CUDA API are used via a network, no additional files are used (which could be a problem as detailed in [107]).

5.4.1 *Flow-network solving algorithms*

The simplest min-cost max-flow algorithm is the **cycle canceling** [76]. This algorithm computes a max-flow solution and then performs multiple iterations augmenting flow along negative-cost directed cycles in the residual network. It guarantees that the overall solution cost decreases by pushing flow along with the cycle. The algorithm finishes with an optimal solution once no negative-cost cycles remain, that is, the negative cycle optimality condition is met.

Differently, from the cycle-canceling algorithm, the **successive shortest path algorithm** [41] attempts to keep costs reduced in all the step to try to achieve feasibility. This algorithm repeatedly selects a source node and sends flow from it to the sink along the shortest path.

Cost scaling [42, 43] iterates multiple times attempting to reduce the cost while maintaining feasibility, and relies on a relaxed complementary slackness condition called ϵ-optimality. The definitions is that, a flow is ϵ-optimal if the flow on arcs with $c_{ij} < \epsilon$ is zero and there are no arcs with $c_{ij} < -\epsilon$ on which flow can be sent. In the beginning, ϵ is equal to the maximum arc cost, but after each iteration the value of ϵ quickly decreases because it is divided by a constant factor in order to achieve the ϵ-optimality The cost scaling algorithm finishes when $\frac{1}{n}$-optimality is achieved since this is it will be similar to the complementary slackness optimality condition.

Table 8 compares the worst-case complexity of these three discussed algorithms. Where K is the number of disaggregated resources plus one, N is the number of nodes, M the number of arcs, and C is the most substantial arc cost. Although the complexities suggest that successive shortest path performs better, the work in [41] showed that cost scaling scale better than the cycle canceling and successive shortest path algorithms. Therefore, in this thesis, we performed our experiment using the cost scaling algorithm.

Table 8: Worst-case time complexities for min-cost max-flow algorithms. 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, $M > N > C > U$.

Algorithm	Worst-case complexity
Cycle canceling	$O(KNM^2CU)$
Succesive shortest path	$O(KN^2M\log(NC))$
Cost Scaling	$O(KN^2U\log(N))$

5.4.2 *Job Profile*

The profiles of the jobs include not only the requested resources but also a performance model defining the level of interference from the network load that the application can suffer when accessing disaggregated resources. This model is created from experimentation using historical data, as the experiments showed in Chapter 3.3. Where the evaluated applications are firstly executed in their best-performing scenarios, that is, without any additional network load interference. After that, we measure the performance impact of the application running on different configurations when accessing a disaggregated resource; such as collocated with a network intensive application. These experiments are then used to generate an offline job profile that will be later used in the tests that evaluate the implementation of our proposed framework, in a testbed environment.

Moreover, as stated before, in the second contribution, we believe that both proposed framework can be further improved by extending then to instead of using offline models, to use an online approach with more advanced prediction models, such as using decision tree [37, 118] or statistical clustering [26, 56, 87]. Because of the cloud's high variability, our model does not need to be optimal; high-quality decisions will be accurate enough. We leave this extension for future work.

5.4.3 *Enabling GPU Disaggregation*

While some approaches to disaggregated GPU have been proposed before, such as rCuda [121], GViM [121], DS-CUDA [107] and GridCuda [85], most of them target only virtual machines, support only old CUDA versions, or have a closed source code which prevents us from implementing missing features (we further detail it in Section 5.7). Therefore, we implemented our own in-house middleware, so-called HFCUDA.

The HFCUDA runtime is a cross-platform library written in C for portability and performance. The architecture is composed by a "client" library hooking all CUDA calls and offloads API-related data via a network (either Ethernet or Infiniband) to a "server" that executes the calls into activated GPUs installed at the server node, and return 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 client. 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 as stated before 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. Thence, the job's threads can access the GPUs within the same context. But, note that since a different server is created for a new application, different applications will never share the same context. The client is composed of a host and a GPU address manager. Thus, each address is stored in a hash table along with its information. Additionally, the addresses of GPUs from different machines can collide, therefore, since the client intercepts all CUDA calls, it creates a *GPU virtual address* of 64bits where the first 12 most significant bits (left to right) are used to represent the *Virtual GPU Id* (note that more bits can be used to address more GPUs). Then, in each call, from the client to the server, the *virtual address* is translated to the *physical* one and vice-versa.

Into *DRMaestro*, we need to manage the GPU allocation in a way that the Kubelet from the remote node will be aware the resource is allocated, that is, it must be aware when its local GPU are being remotely used. However, we do not intend to excessively modify the Kubernetes architecture, since it is not typically easily accepted by the developing community. To solve that problem, we propose a simple approach for dynamically creating new HFCUDA servers as Kubernetes jobs. Then, for each application submitted into Kubernetes that will need access to remote GPUs, we create HFCUDA servers in the nodes that have the GPUs. Therefore, as a Kubernetes job, each HFCUDA server is created on the node as a Pod with a container (e.g., using Docker or Singularity for instance) accessing the local 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 the job with HFCUDA.

5.4.4 *Trace-driven simulation*

As we stated before, we have extended the framework Firmament [41], which has support for an event-driven simulation. The framework simulates machines that are possible to be configured with different NUMA topology with different number CPU and memory regions. The framework also has a generator for synthetic workloads, creating events that will represent the arrival of a new job, taking as a parameter the time interval between job arrivals. The event that represents 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.

Therefore, in addition to all extensions that we made to implement the support for *DRMaestro* into Firmament, we had also extended the simulation part by adding support of GPUs in the emulated machines (before it was only accounting for CPU and Memory). The job arrival process was also modified to support a Poisson process with an exponential distribution receiving a job arrival rate. We have also extended the job description to define the job type since each one request a different amount of resources, and have different completion time and performance interference. The module that defines the task execution time was also extended to define the execution time based on the job type. 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 via pre-defined profile loaded from the traces, slowdown or speedup the completion time, when a new task is placed or after a task completes. Note that, it does not adjust only the completion time of the new incoming task but also the completion time of the currently running tasks in the machine. Finally, we have also extended the metrics generated by the simulation to collect information about resource utilization.

5.5 PREMISES, LIMITATIONS AND OTHER DISCUSSIONS

The approach we used to quantify the performance impact of collocated applications to generate the performance impact is very computationally costly. It requires a combinatorial collocation of a set of known applications over many different scenarios. While this approach is highly accurate, it might not be realistic to perform it in a large scale cloud scenario. Therefore, a promising approach to overcome that limitation is by using advanced prediction models, such as using decision tree [37, 118] or statistical clustering [26, 56, 87]. These approaches can be used to predict the performance of unknown jobs using the models from known applications, which can enlarge the range of the analysis and improve the accuracy of the system. We believe that by using such a prediction model, our approach can be more easily adopted in current cloud environments. But, note that, the main goal of our approach is to minimize the communication cost between GPUs, and the consideration of the sharing interference is to enable efficiency in the system. That is, in the case that the interference cannot be quantified, the algorithm can still work without it, but with a slightly lower quality decision. However, because of the cloud's high variability, it will still delivery high-quality decisions to improve the resource-efficiency then comparing with a topology agnostic scheduling.

Additionally, by the time of this thesis, it was uncommon to share a single GPU between multiple applications, even though, there were some libraries to help to enable that, such as Multi-Process Service (MPS) [101]. At this time, MPS had harsh limitations preventing its usage on 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. For the experiments of this thesis, we had only access to experiment with pre-Volta architectures. Therefore, we did not enable GPU sharing with multiple applications. Moreover, even though, the new NVIDIA Volta architectures implement now fully isolated GPU address spaces, there are still limitations to turn the sharing GPU a reality in a cloud environment. That is, 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 [101]. Additionally, the CUDA run-time still does not expose in its API mechanisms to enable the control to process preemption to efficiently perform time-sharing of processes and prevent starvation of big kernels. Thus, when these limitations are exceeded, the model can be extended to enable further resource-efficiency by time-sharing GPUs between multiple applications.

5.6 EXPERIMENTAL EVALUATION

In this section, we demonstrate the effectiveness and scalability of *DRMaestro* through cluster runs and simulations. We run a set of applications under different conditions, and with the scheduler considering and not considering the possibility to allocate disaggregated resources.

5.6.1 *Experiments.*

Our experiments combine experiments on a local testbed cluster and scale-up trace-driven simulations. The prototype evaluation was performed on a single machine with characteristics described in section 3.3.1 and summarized in Table 6.

THE FIRST EXPERIMENT consists of a small scenario composed of four machines and 512 jobs to execute in a testbed with our implemented prototype.

THE SECOND EXPERIMENT we use a trace-driven simulation to test a large-scale scenario with ≈ 4k jobs and 128 machines.

THE THIRD EXPERIMENT consists of using the implemented trace-driven simulation to evaluate the measured network performance impact on a large-scale scenario, using the traces from the previous experiment plus the traces from the experiments form Section 3.3.

As stated before, 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 with a scenario where some machines have part of their resources already allocated. Note that, this configuration can also represent the scenario where the cluster has a pool of remote resources as discussed in Section 2.3 and illustrated if Figure 6 b) and c). To configure this scenario, we warm up the cluster allocating some long-running jobs before starting the experiments. More specifically, the initial jobs only request CPU and Memory and last ≈800s. Additionally, each experiment is configured with a Poison process using an exponential distribution with an arrival rate of 16 jobs per second. Although we have tested other rates, we only show this one that presents enough pressure to illustrate the scheduler behavior. Finally, in all experiments, we evaluate the schedule with two different configurations: (i) not enabling and (ii) enabling resource disaggregation, and the applications were executed into Docker containers.

5.6.2 *Experiment 1: Comparison between scheduling with and without resource disaggregation in a local testbed cluster*

In this experiment, we execute the implemented prototype of the scheduler in a physical cluster. We can see the total GPU utilization on those scenarios in Figure 23. This figure shows that, as expected, when we enable GPU disaggregation in a scenario that some GPUs cannot be locally allocated because of the shortage of resources, the cluster

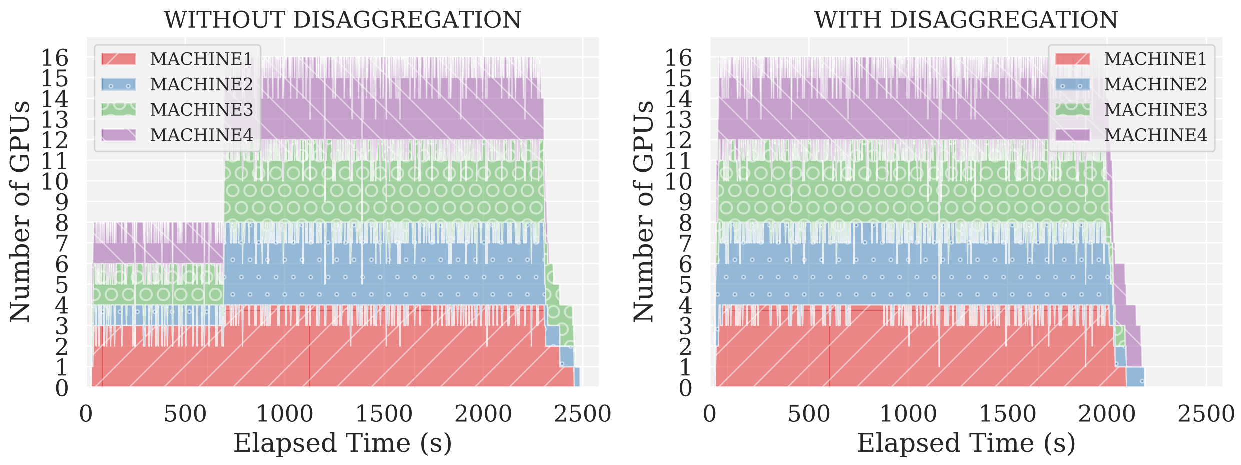


Figure 23: [Prototype] Scheduling with and without GPU disaggregation, in a cluster with 4 machines and 512 jobs.

utilization increases. We can see that when disaggregation is not enabled, most of the time, only 6 GPUs are being used. Just after the long-running jobs finished the 16 GPUs can be allocated. On the contrary, when GPU disaggregation is enabled, all the GPUs are allocated. The small interval that some GPUs are not being used happens because of the interval between finishing and starting a job. Moreover, disaggregation confers the benefit to advance the placement of waiting jobs that could not be placed before, and then, the total makespan is reduced from ≈2486 to ≈2191s. Therefore, this experiment presents a speedup of ≈1.13x when enabling GPU disaggregation.

The scheduling overhead is illustrated in Figure 24, which shows the cumulative distribution function of the average time that the algorithm and all the other scheduling mechanism spend on making the decision and enforcing the decisions. The scheduler allocating disaggregated resources is ≈2x slower than the case that does not allocate remote resources. This behavior is expected since the scheduler runs the model twice on two consecutive phases. However, the scheduler overhead itself is still minimal; it is in the order of sub-seconds in both cases.

5.6.3 *Experiment 2: Validation of The Simulation*

As we performed in the second contribution, we also validate the reliability of the simulation system, in this third contribution, by comparing it with the same scenario as in the prototype experiments in Section 5.6.2. The simulation results are shown in Figure 25. The algorithms behave very similarly in both the prototype and the simulation, despite some expected small differences, which are not very noticeable because of standard deviations are very small in this experiments.

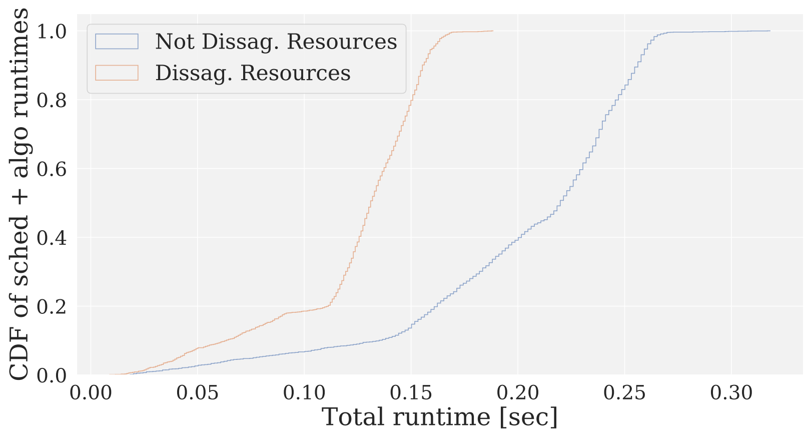


Figure 24: The scheduling considering resource disaggregation runs 2x SLOWER than the one not considering.

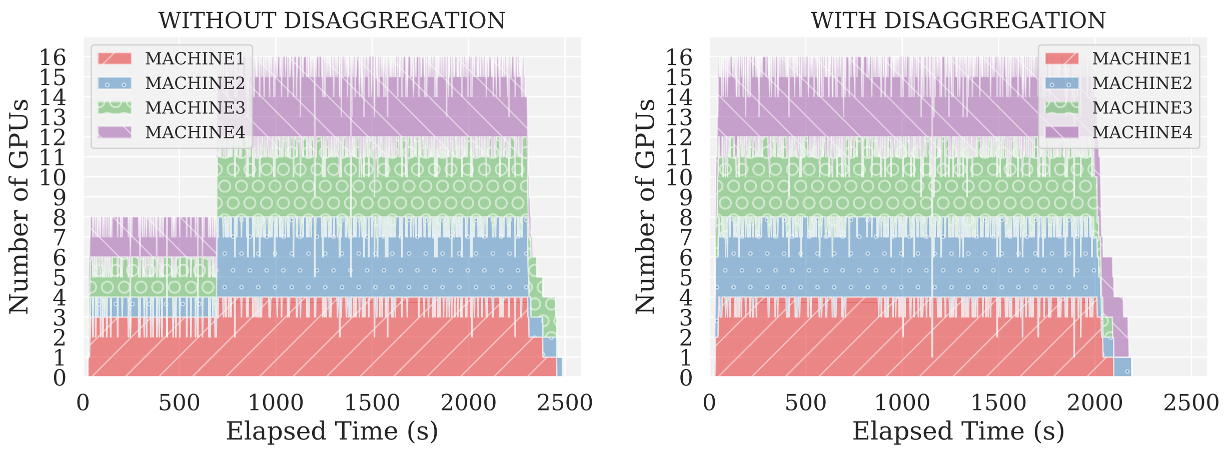


Figure 25: [Simulation] Scheduling with and without GPU disaggregation, in a cluster with 4 machines and 512 jobs.

5.6.4 *Experiment 3: Comparison between scheduling with and without resource disaggregation without considering the network*

We developed a trace-driven simulation to evaluate the scheduling algorithm in a large-scale cluster. To generate the traces, we executed the prototype described in Section 5.4 ten times with different configurations, warm-ups, and job arrival rates. Afterward, the trace files are parsed and transformed into a format compatible with the simulator, creating application and resource usage profiles.

In this experiment, we warm up the cluster with initial jobs requesting resources similarly to the previous experiment, that is only requesting CPU and memory without using the network. We submit then ≈4k jobs and simulate 128 machines. For generating workloads, a Poisson process with the same arrival rate as the first experiment is used. To create the job's configuration, we use a Uniform distribution generating the job's

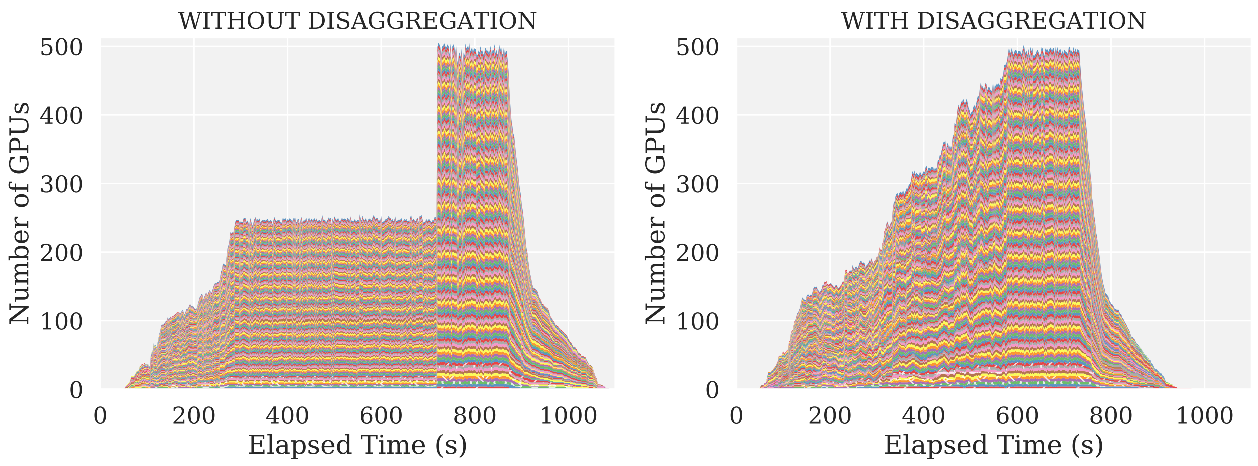


Figure 26: Scheduling with and without GPU disaggregation, in a cluster with 128 machines and 4096 jobs. Enabling GPU disaggregation gives a speedup of ≈1.12X than without disaggregation.

type (Rodinia application). All simulated machines are similar to the machines used in the testbed. Therefore, all the jobs can run in the machines when there are enough resources.

The results show that even and most especially in a more intensive scenario, resource disaggregation provides higher resource utilization, scheduling flexibility and minimize the makespan. For example, Figure 26 shows a speedup of up to ≈1.12X; the scenario without enabling resource disaggregation has a makespan of ≈1087, and when enabling disaggregation it drops to ≈972. Note that, in the beginning, between time 0 to ≈300, the job arrival rate and the available resource have a good match, and the system is not stressed, which explain why the GPUs are not fully utilized since the beginning. We have confirmed that with other experiments varying the arrival rate and the number of jobs and machines.

Figure 27 shows the schedule overhead. Surprisingly, in this experiment, the overhead of the scheduler allocating disaggregated resources is lower. This is because the overall waiting jobs get scheduled and terminate faster, therefore reducing the number of waiting jobs in the queue, and then reducing the pressure on the schedule. It is possible to see in the figure that in very few cases the scheduler with disaggregation runs slower, which is at the beginning of the experiment. This scenario illustrates the flexibility and performance that resource disaggregation confers to the scheduler.

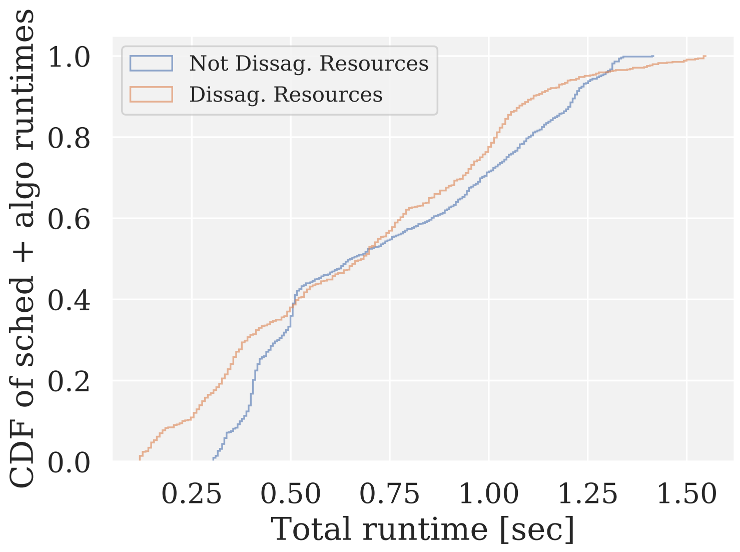


Figure 27: The scheduling considering resource disaggregation runs FASTER than the one not considering, but in overall the difference is very small. For a scenario with 4096 jobs and 128 machines

5.6.5 *Constrained Network Experiments*

In this experiment, using the collected traces from the previous experiment plus the traces from the experiments done in Section 3.3, we use the simulation to evaluate the scheduler configured with two different policies: (i) with and (ii) without network-awareness to define the costs.

5.6.5.1 *Network Awareness*

The network-aware policy updates the costs to place a task and leaves it unscheduled based on the network load, and the sensitivity of the task to such a load. More specifically, the cost is defined as the normalized current network load by the maximum machine load times the network-interference factor defined from the traces of the previous experiments. Additionally, in this policy, for each scheduling decision, if the placement does not satisfy the task's SLO, its placement is postponed. The SLO is defined as the task's expected completion time. But, to avoid starvation, we established a threshold to limit the number of time that the task's placement can be postponed, in this simulation we defined it as 10.

In these experiments, we evaluate two different scenarios. The first scenario is a scenario composed of only 64 jobs to be placed over four machines. The second scenario is a large-scale scenario consisting of ≈4k jobs to be placed over 128 machines. We evaluate these scenarios with different configurations of enabling or not enabling resource disaggregation and/or enabling or not enabling the network-aware policy, as follows:

Scenario with 64 jobs and four machines. For all configurations in this scenario, when we enable the network-aware policy, both the total makespan increases, as summarized in Table 9, and the resource utilization decreases, as illustrated in Figure 28 that the total GPUs allocated are never the maximum. This behavior is expected because with this policy some jobs can have their placement postponed if their QoS is violated. Note that, in the case that GPU disaggregation is enabled, the network-aware policy has a bigger impact. It is explained by the fact that resource disaggregation increases the network load by both introducing additional network load and increasing the overall cluster utilization by placing more jobs. Thus, because the cluster has more network load, more jobs might suffer from network performance interference, and the policy will then postpone much more times the placement of jobs.

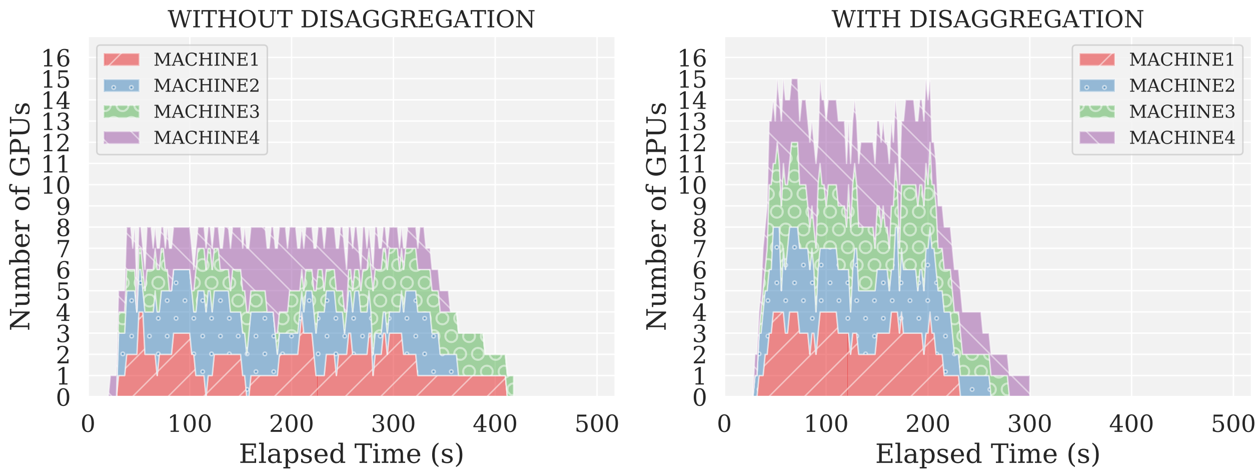


Figure 28: Scheduling with and without GPU disaggregation, with the scheduler configured with a network-aware policy with 64 jobs and 4 machines. Enabling GPU disaggregation gives a speedup of ≈1.4X than without disaggregation.

Table 9: Total makespan of running 64 jobs over 4 machines

		Disaggregation		
Number of Jobs	Nework-Awareness	False	True	Disaggregation Speedup
64	False	413.08s	261.77s	≈1.58X
64	True	418.83s	300.07s	≈1.40X
Network Policy Slowdown		≈0.99X	≈0.87X	

We can see in Table 9 that when the scenario is configured with resource disaggregation, enabling the network-aware policy slowdown the total makespan from ≈261s to ≈300s, i.e., it gives a makespan ≈1.15X slower. However, even though the network-aware policy is enabled, resource disaggregation is still more efficient than without it. For example, the makespan when the network-aware policy is enabled is ≈418s without disaggregation, and it decreases to ≈300s when enabling disaggregation. Hence, resource disaggregation presents a speedup of ≈1.40x in the total makespan. Next, we will show that although the network-aware policy slowdown and decrease resource usage, it confers advantages in regards to QoS violations.

Figure 29 illustrates the jobs' QoS impact whether the network-aware policy is enabled or not. This figure presents the job's execution time normalized by its execution time in the best performing scenario, i.e., when it is running solo without network interference. While the scenario with resource disaggregation introduces more slowdown in the jobs and possibly more QoS violations, the cluster resource usage is maximized, and the total makespan is minimized. The additional slowdowns are a natural effect from increasing the overall cluster resource utilization, which introduces more pressure. Finally, it is possible to see in Figure 29 that the network-aware policy minimizes the network performance impact.

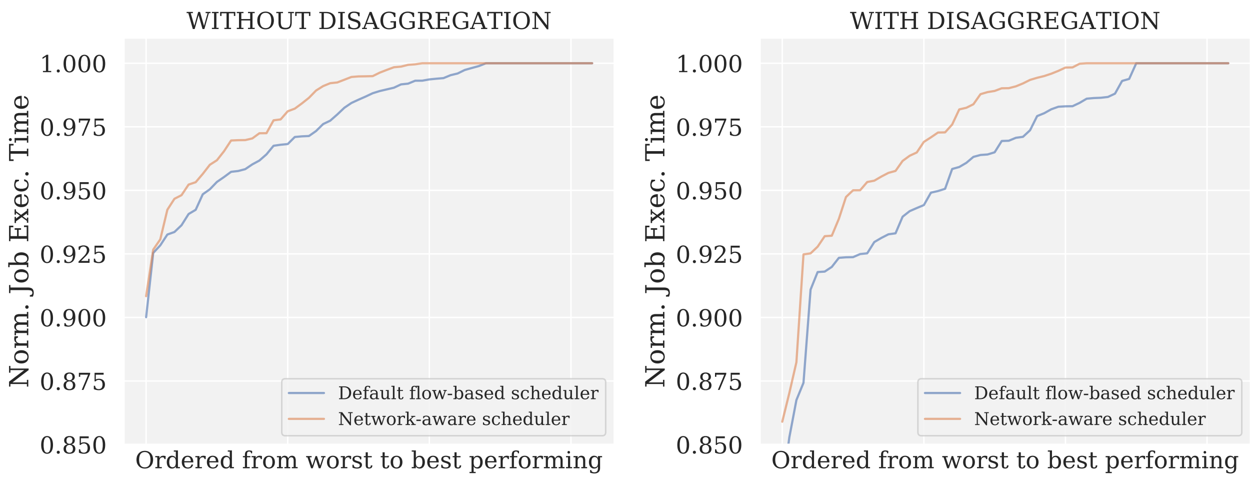


Figure 29: Normalized job execution time for the scheduling policies with and without network awareness, ordered from worst to best performing job. For a scenario with 64 jobs and 4 machines

Scenario with 4096 jobs and 128 machines. The results of this experiment are surprisingly different from the previous scenario with only 64 jobs. The previous scenario, the network-aware policy increased the total makespan for all configurations. This scenario instead, the total makespan has actually a speedup when enabling such a policy. This is explained by the fact that in this scenario there are much more available resources, i.e., from only four machines to 128 ones, then, there is much more room to perform better placements, providing more efficient job collocation. Hence, by improving the quality of the job collocation, more jobs will suffer less slowdown which will also impact the total

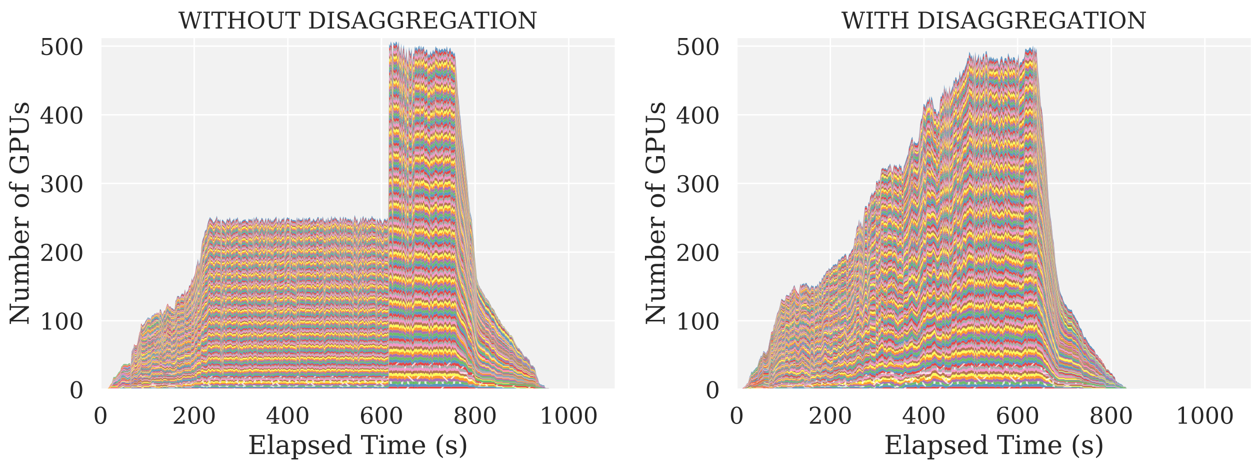


Figure 30: Scheduling with and without GPU disaggregation, with the scheduler configured with a network-aware policy with ≈4k jobs and 128 machines. Enabling GPU disaggregation gives a speedup of ≈1.12X than without disaggregation.

Table 10: Total makespan of running 4096 jobs over 128 machines

		Disaggregation		
Number of Jobs	Nework-Awareness	False	True	Disaggregation Speedup
4096	False	1087.73s	972.20s	≈1.12X
4096	True	959.21s	862.15s	≈1.11X
Network Policy Speedup		≈1.13X	≈1.13X	

makespan. These results can be seen in Table 10. When using resource disaggregation, enabling the policy speedups the total makespan from ≈972s to ≈862s, i.e., it gives a makespan ≈1.13X faster. When comparing the scenario with and without resource disaggregation using the network-aware policy, the total makespan decreases from ≈959s to ≈862s, granting a speedup of ≈1.11X in the total makespan. Finally, when comparing the scenario without both disaggregation and the network-aware policy (≈1087s) versus enabling both (≈862s) gives a speedup of ≈1.26X.

As occurred in the previous scenario, the network-aware policy decreases the resource usage. We can see that by comparing the results from the scenario without the network-aware policy in Figure 26, which has higher GPU utilization than the configuration in this experiment, as shown in Figure 30.

As in the previous experiment, we also show here, in Figure 31, the jobs' QoS impact from using the network-aware policy. One can see that even in a large-scale scenario, the network-aware policy still minimizes the network performance impact on the overall jobs performance, in both configurations of enabling and not enabling resource disaggregation. With this experiment, we further illustrate the trade-off between enabling and

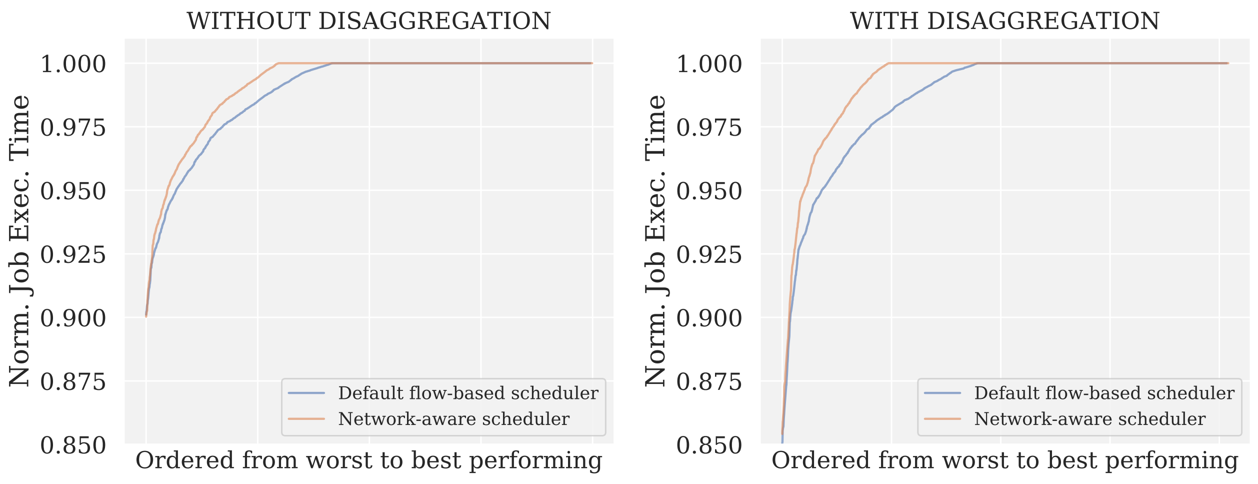


Figure 31: Normalized job execution time for the scheduling policies with and without network awareness, ordered from worst to best performing job. For a scenario with ≈4k jobs and 128 machines

not enabling the network-aware policy, which is between increasing the makespan and decreasing the resource utilization versus minimizing the job's QoS violations.

5.7 RELATED WORKS

We introduced in Section 2.3 the key components that define resource disaggregation and the main differences between software-based and hardware-based architectures. In this section, we first describe the related works that have proposed and evaluated hardware-based disaggregated architectures, and then, we show the works related to the software-based ones, followed by the description of the related works related to scheduling techniques.

The work in [63] introduces and details the Intel rack scale design architecture with disaggregated and composable resources that can be pooled as needed. They describe that the key concept is to break down the well-known servers that are in today's datacenters. Where those servers formed by a fixed ratio of computing storage and networking resources can be broken down into separate resources pools that can be interconnected, or "composed", on demand into logical systems or "nodes", which can be optimized for specific applications. This concept means that many different resources, e.g., compute, hard disk (HD) and NVMe storage, non-volatile memory modules, GPU, FPGA, and networking modules can be installed individually within a rack. These modules can be packaged as blades. And, throughout switches, a rack can be connected to other racks creating a management domain. Several other works have evaluated the feasibility of such architecture and proposed enhancements. Li et al. [84] have qualitatively assessed the opportunities and challenges for leverage disaggregated datacenter architecture. They compared some programming models that can be used to access the disaggregated resources and commented on the implications for the network, resource provisioning, and management. They argued that cloud computing could highly benefit from resource disaggregation because they often have to be configured differently in response to different workloads requirements, and disaggregation provides such required flexibility. In addition, they also mentioned that disaggregation brings efficiency to the system lifecycle. That is, traditional systems impose identical lifecycle for every component (resources) within a system. Hence, all of the components are replaced or upgraded at the same time, preventing the earlier adoption of newer technologies at the component level. In their experimental results, they showed that, as expected, a high percentage of local data always introduces fewer penalties. However, the difference starts to decrease with different ratio of local vs. remote data with the data block size is more substantial, which reduces the overhead in the data transfer. They also showed that in their experiments, that for a given configuration, accessing data from across multiple disks connected via Ethernet poses less of a bandwidth restriction than SATA (local), improving throughput and latency of data access, preventing the need for data locality. Costa et al. [24] described and evaluated the possible implications for network protocols and focused on network routing and rate control mechanism to efficiently share the pool of resources. For the network routing protocol, they used the Valiant Load Balancing (VLB), where the packets are not always routed along the shortest path, is composed by excellent load balancing properties and agnostic to the input traffic matrix. The main idea behind their rate control protocol is that given the knowledge of the network topology and all active flows, each node can independently determine the load on each network link, and then, the fair sending rate for its flows. Via simulation, the evaluated they proposed protocols. They showed that while VLB achieves good load balance, most bottlenecked links is the bottleneck for almost all active flows since each flow uses nearly all links in the network. Novakovic [106] proposed and evaluated a rack-scale memory pooling (RSMP) technique that can reduce the networking overhead of disaggregated memory. He showed that by using his proposed Scale-Out NUMA technique, the RSMP could improve the throughput of a key-value store application up to 8.2X over a traditional scale-out deployment.

The work in [71] introduce and detail the IBM rack scale design architecture for cloud datacenters, so-called dReDBox project. The design is based on microservers based on System on a Chip (SoC) architecture. In such architecture, the memory modules and accelerators will be placed in separate modular servers interconnect via a high-speed, low-latency optoelectronic system fabric, and be allocated in arbitrary set accordingly to the decisions made by a resource/power management software. The defined that the dReD-

Box disaggregated platform requires orchestration support that is not currently available by state-of-the-art datacenter resource management tools. The key requirements are, (i) allocate components and set appropriate forwarding information to interconnect them based on hardware physical environment and software performance requirement, (ii) maintain a consistent distribution and memory isolation to run applications, and (iii) efficiently manage power taking advantage of the component level usage information. Note that, our work in this thesis, is a first step in the direction to achieve such orchestration system. The work in [95] evaluated the feasibility to enable memory disaggregation in the dReDBox project. They proposed a model to represent and analyze it and statistics-based queuing-based simulator to analyze applications performance in disaggregated systems. Their results show that the network layers may introduce overheads that degrade the performance of their examined applications (a video streaming application, a network monitoring application and an application using collaborative encryption functions) up to 66%, and that low memory access bandwidth may degrade the performance up to 20%. As described before, the work in [39] deeply evaluated the network requirements for enabling resource disaggregation for many different types of resources. Pagés et al. [112] has also investigated the minimal network requirement to support disaggregated resources on virtual datacenters. They also propose a resource provisioning mechanism to minimize the necessary amount of computing resources (CPU cores, storage, memory) and the number of different wavelength channels per link in an optical datacenter network. The results show that disaggregated datacenter architecture allows a substantial reduction regarding needed computing resources. In their evaluated simulated scenario, where applications in server-centric architecture need to over-provision resources, the experiments show a decrease circa 46% for computing resources since with resource disaggregation they claim that resources can be tightly allocated to match the exact needs of the virtual machines. Finally, Klimovic et al. [77] have successfully disaggregated flash storage is a promising way to handle flash overprovisioning. By tuning the remote access to flash storages, they showed a trade-off in their experiments. While remote flash storages access can introduce a penalty up to 20% in the throughout, disaggregation allows improving the cluster resource efficiency by promoting resource-efficient scale-out for applications.

As we also detailed in Section 2.3 that resource disaggregation can be enabled by a software-based middleware that intercepts the call to the local resource, forwards it to the remote resource and forwards back to the application the results. As stated before, since thesis focus on GPU disaggregation, we have detailed the related works in Section 3.3.4 that also use or implement a middleware for GPU disaggregation. In Section 3.3.4, we also showed the related works regarding performance interference of co-located applications in cloud environments.

In this section, we also detail the related works that have proposed scheduling technique to allocate disaggregated resources. Hong et al. [55] performed an extensive survey for GPU virtualization techniques and scheduling methods. Although there exist several scheduling methods to schedule job's task into GPUs, varying from priority-based to load-balancing-based approaches, those scheduling works perform fine-grained scheduling, being implemented in hypervisor or OS. Additionally, these scheduling methods do not just focus on low-level scheduling instead of cluster scheduling; they do not consider the possibility to allocate disaggregated GPUs. Which implies no additional overheads depending on the topology and possible challenges related to sharing-induced performance interference, as we showed in the experiments throughout this thesis. Moreover, by the time of this writing, to the best of our knowledge, few works are proposing cluster-level scheduling and placement technique to orchestrate disaggregated resource efficiently. Also, note that, even though, the works related to distributed file systems [82, 129] and network attached storage [13, 40], apparently, seems similar to the concept of disaggregated resources, these works have a fundamental difference in the way that they are accessed by the applications. These works provide a unified file system that is shared between all the applications. The cluster scheduler itself does not need to take into considering the allocation of the file system, the application will always have access to the file system, but it might have some quota limitation of storage usage. The disaggregated resource, on the other hand, can be understood as a single unit can need to be scheduled and allocated to the application.

Next, we detail some of the existing works regarding orchestrating disaggregated GPUs. Iserte et al. [66] provides an extension in the cluster resource manager Slurm [131] by including a new type of resource the "rgpu", to obtain access from any application to any GPU in the cluster, using rCUDA [121] to access the remote GPU. Therefore, when a user submits a job to Slurm, the user must describe the number of rgpus that the job must have allocated. It is also possible to specify the amount of GPU memory the job requires to be reserved. The scheduler first attempts to allocate local GPUs, but if it is not possible it randomly selects other available GPUs in the cluster. The work shows a basic experiment showing that, because of disaggregated GPUs, the application can scale to more GPUs and achieve better performance then only accessing the local GPUs. Additionally, Iserte et al. [67] have done another work that extend OpenStack [127] to support remote GPUs using rCUDA. They extended the OpenStack to allow the user to allocate local or disaggregated GPUs from a pool of GPUs. Lama et al. [79] propose the pVOCL that use the VOCL [146] middleware to virtualize GPUs for applications using OpenCL. The pVOCL enable dynamic scheduling of GPU resource for online power management in virtualized GPU environments, using a power-aware dynamic placement and migration approach.

While our work is inspired in the works [65] and [41] that provide a flow-network-based scheduler, these works do not support resource disaggregation. Isard et al. [65] introduce a powerful and flexible new framework for scheduling concurrent distributed jobs, enforcing data locality, fairness and being a starvation-free method. As we presented in our work, the scheduling problem is mapped to a graph data structure, where the edge weights and capacities encode the competing demands. They called their framework as Quincy and evaluated over a cluster with a few hundred computers. They show an example of how efficiently model scheduling problem on flow-based-network models, which has inspired our work. The experimental results show that Quincy provides better fairness when requested, while substantially improving the data locality. The scenario that they have evaluated presented a throughput increase of up to 40%. Gog et al. [41] extended the work of [65] by implementing different flow-network-based algorithms to solve the model, improving the scheduling latency. They call their framework as Firmament. Their results show that they improved the placement latency by 20x over Quincy for an experiment with 12k machines. Additionally, in their experiments, they show that Firmament's 99th percentile response time is 3.4X better than the SwarmKit [132] and Kubernetes [123] ones, and 6.2X better than Sparrow [111] response time.

However, while these previous works have done an excellent job using remote GPU with a resource manager, their approaches are limited to the job explicitly requesting remote GPUs. Additionally, these works provide very simple scheduling policies like randomly selecting the GPUs. In our work, instead, we use a more advanced schedule algorithm based on flow-network to define the optimal placement and allocate remote GPUs transparently to the user. Additionally, another main difference from these approaches to our approach is to consider the job's preference for resource and also the possible additional network interference that the jobs might have because of accessing remote resources. Therefore, to the best of our knowledge, our work is the first one to apply flow-network model to solve the scheduling and placement problem considering resource disaggregation.

5.8 FINAL CONSIDERATIONS

In this thesis we have presented *DRMaestro*, a novel framework to orchestrate disaggregated resources on cloud systems. *DRMaestro* addresses some of the main challenges found in datacenters 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.

The framework is first validated through the implementation of a prototype over Kubernetes and then evaluated by performing several trace-driven simulations with traces from the testbed that show that our solution provides higher cluster utilization and lower SLO violations.

Our experiments in a simulated environment 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 performance interference. The experiments show the trade-off of enabling resource disaggregation in a shared cluster. While disaggregation can further reduce the job makespan, i.e., ≈1.26x, it also introduces slowdown due to increased resource utilization. Finally, we also showed the effectiveness of a network-aware policy to mitigate the additional slowdowns in the job's execution time.

For future work, we plan to introduce into the framework the notion of the underlying machine topology that interconnects the GPUs similarly the work performed in Section 4.

6 CONCLUSIONS AND FUTURE WORK

6.1 CONCLUSIONS

In this thesis, we presented three complementary steps toward the creation of practical systems to achieve higher efficiency for datacenters. First, we presented a highly-detailed evaluation to characterize the performance of applications running on *virtual environment* (i.e., containers or virtual machines) over complex hardware architectures (e.g., non-uniform memory architectures). This study makes possible a deep understanding of the behavior of the complex execution stack of applications running on *virtual environments* over hardware-resources connected in a non-uniform fashion. Secondly, we use this performance characterization to uncover the potential to improve the resource-efficiency of virtualized datacenters. And then, we propose an intelligent system making informed decisions to orchestrate the datacenter's resources. The system implements a new topology-aware placement algorithm for scheduling workloads in multi-GPU systems with the devices interconnected in a non-uniform manner. Finally, we further exploit the resource-efficiency potential via increase the datacenter's resource usage through enabling resource disaggregation. In this sense, we proposed and evaluated a novel automatic workload orchestration for pooled resources and disaggregated architectures capable of improving resource utilization across servers. We have proven that all the presented techniques achieve their performance objectives whereas they fairly satisfy all applications, by employing best-efforts to mitigate the job's SLO violations. Next, we summarize in more detail the work presented and achievements obtained in this thesis.

6.1.1 *Performance Characterization of Containerized and Accelerated Workloads*

The first contribution of this thesis is the performance characterization of workloads running over virtualized environments and NUMA topologies over different configurations. In more details, we correlated detailed system information with high-level performance data to characterize the performance of applications running on top of virtualization technologies over servers composed by NUMA architectures.

The evaluation exposed the performance of applications running on top of OS container or VM technologies on different scenario varying the placement and resource allocation. The data collected was later used to uncover the potential to improve the resource-efficiency of virtualized datacenters.

We showed by experiments running on top of machines composed by IBM POWER8® processors, that the performance of virtualization via OS-level containers is almost as efficient as running directly in a bare-metal machine. Additionally, we showed that a sub-optimal resource allocation on NUMA topologies can introduce a performance penalty and that not only the performance of threads running on the processors are impacted, but also the performance of tasks running on GPUs. In a further analysis, we tried different manual resource allocation, showing that a smarter resource allocation on a multi-GPU system can improve the workloads completion time in ≈30%.

Finally, these performance monitoring models and methods developed in the first contribution were extensively applied during the development of the orchestration systems.

6.1.2 *Topology-Aware Multi-GPU High-Performance AI Workload Scheduling*

The second contribution of this thesis consists of an algorithm with two new scheduling policies for placing GPU-based workloads in multi-GPU systems with the devices interconnected in a non-uniform way. The foundation of the algorithm is based on the use of a new graph mapping algorithm that considers the job's performance objectives and the system topology. Applications can express their performance objectives as SLOs that are later translated into abstract Utility Functions. The result of using the proposed algorithm is a minimization of the communication cost, reduction of system resource contention and an increase in the system utilization.

We showed by experiments using a prototype implementation and a trace-driven simulation that the proposed algorithm presents the performance improvements that topology-aware scheduling can confer for DL workloads using multiple GPUs. The analysis revealed that optimal resource allocation can reflect in a speed-up of up to ≈1.30x in the cumulative execution time, and no SLO violations. The trace-driven simulation of a large-scale cluster showed that compared with greedyapproaches our algorithm produces solutions that satisfy more jobs, minimizes the SLO violations and improves the job's execution time even in a heavily loaded scenario.

Finally, these results evidence the necessity of a scheduling algorithm that is aware of the performance interference to provide QoS for jobs.

6.1.3 *Workload Orchestration for Pooled Resources and Disaggregated Architectures*

The third contribution of this thesis consists of a technique that allows maximizing the cluster resource utilization even in a situation where a machine does not have enough computing capabilities to be allocated. We used a flow-network-based framework to orchestrate disaggregated resources on cloud systems employing best-efforts on preventing SLO violations while maximizing the system utilization.

We called the proposed framework as disaggregated resource maestro (*DRMaestro*), and its main idea is to automatically discover and allocate disaggregated resources in the cluster for a job as if the resources are attached to the local machine that the job is placed. For the job standpoint, it is only using local resources. Our system is driven by high-level applications goals that determine the placement cost considering the application satisfaction with how well the goals are met. With that, the framework enables transparent resource disaggregation while automatically controlling and determining the optimal placement.

The framework was validated through both the implementation of a prototype over Kubernetes and via a trace-driven simulation using traces from the prototype.

Our experiments demonstrated the effectiveness of our proposal in different scenarios. The experiments revealed the trade-off of enabling resource disaggregation in a shared and virtualized cluster. While disaggregation can further reduce the job makespan, i.e., $\approx$1.26x, it also introduces slowdown in the job's completion time due to increased overall resource utilization. We also showed the effectiveness of a network-aware policy to mitigate the additional slowdowns in the job's execution time via trace-driven simulations.

Finally, we believe that enabling resource disaggregation to improve the cluster resource utilization and make the scheduling process more flexible is an exciting research problem. It requires a novel resource allocation algorithm capable of transparently remote expose resources while reasoning which is the best placement for applications with different characteristics. It must also consider different resources that might be virtualized and disaggregated.

6.2 FUTURE WORKS

We believe that the contributions described above open many interesting paths for future research. Therefore, in this section, we present some promising future directions for the work done in this thesis.

APPLICATION CHARACTERIZATION: In Section 3, we showed the performance impact from virtualization and the underlying topology in a set of applications. In the evaluation, we considered some proof-of-concept applications. However, to cover the full complexity of real-world Cloud and HPC scenarios, additional techniques, and assessments for complex applications need to be further explored. This would require further research on applications that are intensive on multiple resources, e.g., memory-bound and I/O-bound applications, with complex and irregular communication patterns; specifically, the ones with collective communication and dynamic load balancing.

JOB INTERFERENCE CATEGORIZATION: In this thesis, we assumed that the application could be profiled offline. But, that may not always be possible or accurate. Then, a promising research direction is to perform scheduling and placement decisions through run-time monitoring using a technique such as a decision tree, statistical clustering, machine learning, among others. We expect that using more advanced prediction technique; one should be able to distinguish the overhead of different *virtual environments* more easily, and also distinguish the performance interference suffered from the collocation of a broad set of applications with different characteristics.

COLLOCATING HPC AND NON-HPC APPLICATIONS: In this work, we focus on non-interactive workloads, where the performance goal was relative to the completion time. Therefore, another interesting research direction is to analyze and extend the scheduling algorithms to include support for interactive workloads (e.g., web-services) in which the performance goals are relative to average or percentile response time or throughput over a short time interval. Moreover, the collocation of interactive and non-interactive workloads introduces a whole set of new challenges. Typically, interactive workloads have more restrict SLO with higher priority and are highly sensitive to collocation with other resource-intensive applications. The major challenges include sharing-induced performance interference, different performance metrics, and security concerns.

SCALING JOBS: In this thesis we do not schedule and place jobs considering that an application might have different phases. For example, some application does not use GPUs during all its execution, but only in some specific phases. An application can only use GPUs in its final phase, for example, without needing to keep all the idle GPUs allocated. In this sense, some applications can expand or shrink during the execution to further enable fine-grain improvements in the cluster resource-efficiency, and also, confer cost benefits in a cloud environment. Similarly, some applications could also be dynamically shrunk to open space to expand or place other high-priority applications. Therefore, a promising research direction is to take decisions based on the applications needs during its execution, instead of using an application-agnostic scheduling algorithm.

SECURITY IN CLOUD: The work in this thesis had the focus on improving performance and resource utilization of applications running on virtualized and disaggregated datacenters. But, another major challenge that typically discourages users to use a cloud-like environment is the security concerns. Some of the security challenges are providing isolation and stateless allocation of GPUs (i.e., the currently running jobs must not have access to local data from the previous running jobs) and network with the support of RDMA. Among other challenges mostly related to public clouds such as the loss of full control of the resources in the cluster and lack of trust on the cloud providers. Future research is needed to address these concerns to enable a more secure cloud environment.

BIBLIOGRAPHY

[1] Andrew V. Adinetz, Paul F. Baumeister, Hans Böttiger, Thorsten Hater, Thilo Maurer, Dirk Pleiter, Wolfram Schenck, and Sebastiano Fabio Schifano. "Performance Evaluation of Scientific Applications on POWER8." In: *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation: 5th International Workshop, PMBS 2014, New Orleans, LA, USA, November 16, 2014. Revised Selected Papers*. Ed. by Stephen A. Jarvis, Steven A. Wright, and Simon D. Hammond. Cham: Springer International Publishing, 2015, pp. 24–45. ISBN: 978-3-319-17248-4. DOI: 10.1007/978-3-319-17248-4_2. URL: http://dx.doi.org/10.1007/978-3-319-17248-4_2.

[2] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1993. ISBN: 0-13-617549-X.

[3] Marcelo Amaral, Jordà Polo, David Carrera, Iqbal I. Mohomed, Merve Unuvar, and Malgorzata Steinder. "Performance Evaluation of Microservices Architectures Using Containers." In: *2015 IEEE 14th International Symposium on Network Computing and Applications*. 2015, pp. 27–34. DOI: 10.1109/NCA.2015.49.

[4] Marcelo Amaral, Yurdaer N. Doganata, Iqbal I. Mohomed, Asser N. Tantaw, and Merve Unuvar. "Selecting resource allocation policies and resolving resource conflicts." Patent US9697045B2 (US). July 2017. URL: http://www.patentlens.net/patentlens/patent/US_9697045B2/.

[5] Marcelo Amaral, Jordà Polo, David Carrera, Seetharami Seelam, and Malgorzata Steinder. "Topology-aware GPU Scheduling for Learning Workloads in Cloud Environments." In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '17. Denver, Colorado: ACM, 2017, 17:1–17:12. ISBN: 978-1-4503-5114-0. DOI: 10.1145/3126908.3126933. URL: http://doi.acm.org/10.1145/3126908.3126933.

[6] Amazon. *Amazon EC2 Cluster Compute Instances*. Ed. by www.top500.org. [online] https://www.top500.org/system/177457. 2015. URL: https://www.top500.org/system/177457.

[7] Amazon. *Amazon Machine Learning*. 2017. URL: https://aws.amazon.com/machine-learning/.

[8] Soheil Bahrampour, Naveen Ramakrishnan, Lukas Schott, and Mohak Shah. "Comparative Study of Caffe, Neon, Theano, and Torch for Deep Learning." In: *CoRR* abs/1511.06435 (2015). URL: http://arxiv.org/abs/1511.06435.

[9] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. "Resource Containers: A New Facility for Resource Management in Server Systems." In: *Proceedings of the Third Symposium on Operating Systems Design and Implementation*. OSDI '99. New Orleans, Louisiana, USA: USENIX Association, 1999, pp. 45–58. ISBN: 1-880446-39-1. URL: http://dl.acm.org/citation.cfm?id=296806.296810.

[10] Francine Berman and Lawrence Snyder. "On Mapping Parallel Algorithms into Parallel Architectures." In: *J. Parallel Distrib. Comput.* 4.5 (Oct. 1987), pp. 439–458. ISSN: 0743-7315. DOI: 10.1016/0743-7315(87)90018-9.

[11] Sergey Blagodurov, Sergey Zhuravlev, Mohammad Dashti, and Alexandra Fedorova. "A Case for NUMA-aware Contention Management on Multicore Systems." In: *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*. USENIXATC'11. Portland, OR: USENIX Association, 2011, pp. 1–1. URL: http://dl.acm.org/citation.cfm?id=2002181.2002182.

[12] Léon Bottou, Frank E. Curtis, and Jorge Nocedal. "Optimization Methods for Large-Scale Machine Learning." In: *CoRR* abs/1606.04838 (2016). URL: http://arxiv.org/abs/1606.04838.

[13] Jonathan D. Bright and John A. Chandy. "A Scalable Architecture for Clustered Network Attached Storage." In: *Proceedings of the 20 th IEEE/11 th NASA Goddard Conference on Mass Storage Systems and Technologies (MSS'03)*. MSS '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 196–. ISBN: 0-7695-1914-8. URL: http://dl.acm.org/citation.cfm?id=824467.824994.

[14] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. "hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications." In: *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*. Ed. by IEEE. Pisa, Italy, Feb. 2010. DOI: 10.1109/PDP.2010.67. URL: https://hal.inria.fr/inria-00429889.

[15] Horst Bunke and Xiaoyi Jiang. *Graph Matching and Similarity*. Ed. by Horia-Nicolai Teodorescu, Daniel Mlynek, Abraham Kandel, and H.-J. Zimmermann. Boston, MA: Springer US, 2000, pp. 281–304. ISBN: 978-1-4615-4401-2.

[16] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. "Borg, Omega, and Kubernetes." In: *Queue* 14.1 (Jan. 2016). ISSN: 1542-7730. DOI: 10.1145/2898442.2898444.

[17] Calico. *About Calico*. Accessed in: 29-December-2018. URL: https://docs.projectcalico.org/v3.4/introduction/.

[18] F. Callegati, W. Cerroni, C. Contoli, and G. Santandrea. “Performance of Network Virtualization in cloud computing infrastructures: The OpenStack case.” In: *Cloud Networking (CloudNet), 2014 IEEE 3rd International Conference on*. 2014, pp. 132–137. DOI: 10.1109/CloudNet.2014.6968981.

[19] David Carrera, Malgorzata Steinder, Ian Whalley, Jordi Torres, and Eduard Ayguadé. “Enabling Resource Sharing Between Transactional and Batch Workloads Using Dynamic Application Placement.” In: *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*. Middleware ’08. Leuven, Belgium: Springer-Verlag New York, Inc., 2008. ISBN: 3-540-89855-7.

[20] M. Casoni, C.A. Grazia, and N. Patriciello. “On the performance of Linux Container with Netmap/VALE for networks virtualization.” In: *Networks (ICON), 2013 19th IEEE International Conference on*. 2013, pp. 1–6. DOI: 10.1109/ICON.2013.6781957.

[21] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. “Rodinia: A Benchmark Suite for Heterogeneous Computing.” In: *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*. IISWC ’09. Washington, DC, USA: IEEE Computer Society, 2009. ISBN: 978-1-4244-5156-2. DOI: 10.1109/IISWC.2009.5306797.

[22] Jobs Run to Completion. *Kubernetes*. Accessed in: 22-December-2018. URL: hhttps://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion.

[23] CoreOS. *How it works*. Accessed in: 29-December-2018. URL: https://github.com/coreos/flannel#flannel.

[24] Paolo Costa, Hitesh Ballani, and Dushyanth Narayanan. “Rethinking the Network Stack for Rack-scale Computers.” In: *Proceedings of the 6th USENIX Conference on Hot Topics in Cloud Computing*. HotCloud’14. Philadelphia, PA: USENIX Association, 2014. URL: http://dl.acm.org/citation.cfm?id=2696535.2696547.

[25] Christina Delimitrou and Christos Kozyrakis. “Paragon: QoS-aware Scheduling for Heterogeneous Datacenters.” In: *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’13. Houston, Texas, USA: ACM, 2013, pp. 77–88. ISBN: 978-1-4503-1870-9. DOI: 10.1145/2451116.2451125. URL: http://doi.acm.org/10.1145/2451116.2451125.

[26] Christina Delimitrou and Christos Kozyrakis. “Quasar: Resource-efficient and QoS-aware Cluster Management.” In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’14. Salt Lake City, Utah, USA: ACM, 2014, pp. 127–144. ISBN: 978-1-4503-2305-5. DOI: 10.1145/2541940.2541941.

[27] Christina Delimitrou and Christos Kozyrakis. "Quasar: Resource-efficient and QoS-aware Cluster Management." In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 127–144. ISBN: 978-1-4503-2305-5. DOI: 10.1145/2541940.2541941. (Visited on 02/26/2016).

[28] Docker. *Swarm mode overview*. Accessed in: 18-December-2018. URL: https://docs.docker.com/engine/swarm/.

[29] Docker. *What is Docker?* Accessed in: 21-January-2015. URL: https://www.docker.com/.

[30] Rajdeep Dua, A Reddy Raja, and Dharmesh Kakadia. "Virtualization vs Containerization to Support PaaS." In: *Proceedings of the 2014 IEEE International Conference on Cloud Engineering*. IC2E '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 610–614. ISBN: 978-1-4799-3766-0. DOI: 10.1109/IC2E.2014.41. URL: http://dx.doi.org/10.1109/IC2E.2014.41.

[31] Chris Edwards. "Growing Pains for Deep Learning." In: *Commun. ACM* 58.7 (2015), pp. 14–16. ISSN: 0001-0782. DOI: 10.1145/2771283. URL: http://doi.acm.org.recursos.biblioteca.upc.edu/10.1145/2771283.

[32] Yaakoub El-Khamra, Hyunjoo Kim, Shantenu Jha, and Manish Parashar. "Exploring the Performance Fluctuations of HPC Workloads on Clouds." In: *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science*. CLOUDCOM '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 383–387. ISBN: 978-0-7695-4302-4. DOI: 10.1109/CloudCom.2010.84. URL: http://dx.doi.org/10.1109/CloudCom.2010.84.

[33] F. Ercal, J. Ramanujam, and P. Sadayappan. "Task Allocation Onto a Hypercube by Recursive Mincut Bipartitioning." In: *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications: Architecture, Software, Computer Systems, and General Issues - Volume 1*. C3P. Pasadena, California, USA: ACM, 1988, pp. 210–221. ISBN: 0-89791-278-0. DOI: 10.1145/62297.62323.

[34] Iman Faraji and Ahmad Afsahi. "GPU-Aware Intranode MPI Allreduce." In: EuroMPI/ASIA'14 (Sept. 2014). DOI: 10.1145/2642769.2642773. URL: http://doi.acm.org/10.1145/2642769.2642773.

[35] Iman Faraji, Seyed Hessam Mirsadeghi, and Ahmad Afsahi. "Topology-Aware GPU Selection on Multi-GPU Nodes." In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2016, Chicago, IL, USA, May 23-27, 2016*. 2016, pp. 712–720. DOI: 10.1109/IPDPSW.2016.44.

[36] W. Felter, a. Ferreira, R. Rajamony, and Rubio J. *An Updated performance comparison of virtual machines and Linux containers*. Tech. rep. RC25482. IBM Research Division, 2014.

[37] Damon Fenacci, Björn Franke, and John Thomson. "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 &#38; Compilers for Embedded Systems*. SCOPES '10. St. Goar, Germany: ACM, 2010, 5:1–5:10. ISBN: 978-1-4503-0084-1. DOI: 10.1145/1811212.1811219. URL: http://doi.acm.org/10.1145/1811212.1811219.

[38] C. M. Fiduccia and R. M. Mattheyses. "A Linear-time Heuristic for Improving Network Partitions." In: *Proceedings of the 19th Design Automation Conference*. DAC '82. Piscataway, NJ, USA: IEEE Press, 1982, pp. 175–181. ISBN: 0-89791-020-6.

[39] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. "Network Requirements for Resource Disaggregation." In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'16. Savannah, GA, USA: USENIX Association, 2016. ISBN: 978-1-931971-33-1.

[40] Garth A. Gibson and Rodney Van Meter. "Network Attached Storage Architecture." In: *Commun. ACM* 43.11 (Nov. 2000), pp. 37–45. ISSN: 0001-0782. DOI: 10.1145/353360.353362. URL: http://doi.acm.org/10.1145/353360.353362.

[41] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. "Firmament: Fast, Centralized Cluster Scheduling at Scale." In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, 2016, pp. 99–115. ISBN: 978-1-931971-33-1.

[42] Andrew V Goldberg. "An Efficient Implementation of a Scaling Minimum-Cost Flow Algorithm." In: *Journal of Algorithms* 22.1 (1997). ISSN: 0196-6774. DOI: https://doi.org/10.1006/jagm.1995.0805.

[43] Andrew V. Goldberg and Robert E. Tarjan. "Finding Minimum-Cost Circulations by Successive Approximation." In: *Math. Oper. Res.* 15.3 (Aug. 1990), pp. 430–466. ISSN: 0364-765X. DOI: 10.1287/moor.15.3.430. URL: http://dx.doi.org/10.1287/moor.15.3.430.

[44] Google. *Kubernetes*. Accessed in: 21-January-2015. URL: https://github.com/googlecloudplatform/kubernetes.

[45] Google. *Google Cloud Prediction API Documentation*. 2017. URL: https://cloud.google.com/prediction/docs/.

[46] Samuel Greengard. "GPUs Reshape Computing." In: *Commun. ACM* 59.9 (Aug. 2016), pp. 14–16. ISSN: 0001-0782. DOI: 10.1145/2967979.

[47] Akhila Gundu, Gita Sreekumar, Ali Shafiee, Seth Pugsley, Hardik Jain, Rajeev Balasubramonian, and Mohit Tiwari. "Memory Bandwidth Reservation in the Cloud to Avoid Information Leakage in the Memory Controller." In: *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy*.

HASP '14. New York, NY, USA: ACM, 2014, 11:1–11:5. ISBN: 978-1-4503-2777-0. DOI: 10.1145/2611765.2611776. (Visited on 02/26/2016).

[48] A. Gupta, P. Faraboschi, F. Gioachin, L. V. Kale, R. Kaufmann, B. Lee, V. March, D. Milojicic, and C. H. Suen. "Evaluating and Improving the Performance and Scheduling of HPC Applications in Cloud." In: *IEEE Transactions on Cloud Computing* 4.3 (2016), pp. 307–321. ISSN: 2168-7161. DOI: 10.1109/TCC.2014.2339858.

[49] Anshul Gupta. *An Evaluation of Parallel Graph Partitioning and Ordering Softwares on a Massively Parallel Computer*. IBM T. J. Watson Research Center, 2010, All.

[50] Vishakha Gupta, Ada Gavrilovska, Karsten Schwan, Harshvardhan Kharche, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. "GViM: GPU-accelerated Virtual Machines." In: *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*. HPCVirt '09. Nuremburg, Germany: ACM, 2009, pp. 17–24. ISBN: 978-1-60558-465-2. DOI: 10.1145/1519138.1519141. URL: http://doi.acm.org/10.1145/1519138.1519141.

[51] Qiming He, Shujia Zhou, Ben Kobler, Dan Duffy, and Tom McGlynn. "Case Study for Running HPC Applications in Public Clouds." In: *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. HPDC '10. Chicago, Illinois: ACM, 2010, pp. 395–401. ISBN: 978-1-60558-942-8. DOI: 10.1145/1851476.1851535. URL: http://doi.acm.org/10.1145/1851476.1851535.

[52] Hewlett-Packard. "HP The Machine." In: *http://www.hpl.hp.com/research/ systems-research/themachine/*.

[53] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. "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. Boston, MA: USENIX Association, 2011, pp. 295–308. URL: http://dl.acm.org/citation.cfm?id=1972457.1972488.

[54] Christopher Hollowell, Costin Caramarcu, William Strecker-Kellogg, Antonio Wong, and Alexandr Zaytsev. "The Effect of NUMA Tunings on CPU Performance." In: *Journal of Physics: Conference Series* 664.9 (2015), p. 092010. DOI: 10.1088/1742-6596/664/9/092010. URL: https://doi.org/10.1088%2F1742-6596%2F664%2F9%2F092010.

[55] Cheol-Ho Hong, Ivor Spence, and Dimitrios S. Nikolopoulos. "GPU Virtualization and Scheduling Methods: A Comprehensive Survey." In: *ACM Comput. Surv.* 50.3 (June 2017), 35:1–35:37. ISSN: 0360-0300. DOI: 10.1145/3068281. URL: http://doi.acm.org/10.1145/3068281.

[56] Kenneth Hoste, Aashish Phansalkar, Lieven Eeckhout, Andy Georges, Lizy K. John, and Koen De Bosschere. "Performance Prediction Based on Inherent Program Similarity." In: *Proceedings of the 15th International Conference on Parallel Ar-*

chitectures and Compilation Techniques. PACT '06. Seattle, Washington, USA: ACM, 2006, pp. 114–122. ISBN: 1-59593-264-X. DOI: 10.1145/1152154.1152174. URL: http://doi.acm.org/10.1145/1152154.1152174.

[57] Yang Hu, Mingcong Song, and Tao Li. "Towards "Full Containerization" in Containerized Network Function Virtualization." In: *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '17. Xi'an, China, 2017. ISBN: 978-1-4503-4465-4. DOI: 10.1145/3037697.3037713.

[58] N. Huber, M. von Quast, M. Hauck, and S.. Kounev. "Evaluating and modeling virtualization performance overhead for cloud environments." In: *CLOSER* (2011), pp. 563–573.

[59] J. Hwang, S. Zeng, F. Wu, and T. Wood. "A component-based performance comparison of four hypervisors." In: Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium, pp. 269–276.

[60] IBM. *Go beyond artificial intelligence with Watson*. 2017. URL: https://www.ibm.com/watson/.

[61] ImageNet. "Large Scale Visual Recognition Challenge (ILSVRC)." In: *http://www.image-net.org/challenges/LSVRC/*. image-net.org, 2012.

[62] Intel. "Facebook Disaggregated Rack." In: *http://goo.gl/6h2Ut*.

[63] Intel. "White Paper: Resource Pooling Using Intel Rack Scale Architecture." In: *https://goo.gl/kE3snYS*. 2017.

[64] Alexandru Iosup, Nezih Yigitbasi, and Dick Epema. "On the Performance Variability of Production Cloud Services." In: *Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. CCGRID '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 104–113. ISBN: 978-0-7695-4395-6. DOI: 10.1109/CCGrid.2011.22.

[65] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. "Quincy: Fair Scheduling for Distributed Computing Clusters." In: *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*. SOSP '09. Big Sky, Montana, USA: ACM, 2009. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629601.

[66] Sergio Iserte, Adrián Castelló, Rafael Mayo, Enrique S. Quintana-Ortí, Federico Silla, Jose Duato, Carlos Reaño, and Javier Prades. "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*. SBAC-PAD '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 318–325. ISBN: 978-1-4799-6905-0. DOI: 10.1109/SBAC-PAD.2014.49. URL: http://dx.doi.org/10.1109/SBAC-PAD.2014.49.

[67] Sergio Iserte, Francisco J. Clemente-Castelló, Adrián Castelló, Rafael Mayo, and Enrique S. Quintana-Ortí. "Enabling GPU Virtualization in Cloud Environments." In: *Proceedings of the 6th International Conference on Cloud Computing and Services Science - Volume 1 and 2*. CLOSER 2016. Rome, Italy: SCITEPRESS - Science and Technology Publications, Lda, 2016, pp. 249–256. ISBN: 978-989-758-182-3. DOI: 10.5220/0005780502490256. URL: https://doi.org/10.5220/0005780502490256.

[68] Keith R. Jackson, Lavanya Ramakrishnan, Krishna Muriki, Shane Canon, Shreyas Cholia, John Shalf, Harvey J. Wasserman, and Nicholas J. Wright. "Performance Analysis of High Performance Computing Applications on the Amazon Web Services Cloud." In: *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science*. CLOUDCOM '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 159–168. ISBN: 978-0-7695-4302-4. DOI: 10.1109/CloudCom.2010.69. URL: http://dx.doi.org/10.1109/CloudCom.2010.69.

[69] Rick Jones. In: Accessed in: 24-March-2015. Hewlett-Packard Company, 2012. URL: http://www.netperf.org/svn/netperf2/trunk/doc/netperf.pdf.

[70] Sangeetha Abdu Jyothi et al. "Morpheus: Towards Automated SLOs for Enterprise Clusters." In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'16. Savannah, GA, USA: USENIX Association, 2016. ISBN: 978-1-931971-33-1.

[71] K. Katrinis et al. "Rack-scale disaggregated cloud data centers: The dReDBox project vision." In: *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2016, pp. 690–695.

[72] Onur Kayiran, Nachiappan Chidambaram Nachiappan, Adwait Jog, Rachata Ausavarungnirun, Mahmut T. Kandemir, Gabriel H. Loh, Onur Mutlu, and Chita R. Das. "Managing GPU Concurrency in Heterogeneous Architectures." In: *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-47. Cambridge, United Kingdom: IEEE Computer Society, 2014, pp. 114–126. ISBN: 978-1-4799-6998-2. DOI: 10.1109/MICRO.2014.62. URL: http://dx.doi.org/10.1109/MICRO.2014.62.

[73] Michael Kerrisk. *Control Group configuration unit settings*. Accessed in: 29-January-2015. URL: http://man7.org/linux/man-pages/man5/systemd.cgroup.5.html.

[74] Michael Kerrisk. *Control Group configuration unit settings*. Accessed in: 29-January-2015. URL: http://man7.org/linux/man-pages/man7/systemd.namespaces.7.html.

[75] V. V. Kindratenko, J. J. Enos, G. Shi, M. T. Showerman, G. W. Arnold, J. E. Stone, J. C. Phillips, and W. m. Hwu. "GPU clusters for high-performance computing." In: *2009 IEEE International Conference on Cluster Computing and Workshops*. 2009, pp. 1–8. DOI: 10.1109/CLUSTR.2009.5289128.

[76] Morton Klein. *A primal method for minimal cost flows, with applications to the assignment and transportation problems*. 1967.

[77] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. "Flash Storage Disaggregation." In: *Proceedings of the Eleventh European Conference on Computer Systems*. EuroSys '16. London, United Kingdom: ACM, 2016, 29:1–29:15. ISBN: 978-1-4503-4240-7. DOI: 10.1145/2901318.2901337. URL: http://doi.acm.org/10.1145/2901318.2901337.

[78] Alexey Kopytov. *SysBench manual*. Accessed in: 21-January-2015. 2009. URL: http://imysql.com/wp-content/uploads/2014/10/sysbench-manual.pdf.

[79] P. Lama, Y. Li, A. M. Aji, P. Balaji, J. Dinan, S. Xiao, Y. Zhang, W. Feng, R. Thakur, and X. Zhou. "pVOCL: Power-Aware Dynamic Placement and Migration in Virtualized GPU Environments." In: *2013 IEEE 33rd International Conference on Distributed Computing Systems*. 2013, pp. 145–154. DOI: 10.1109/ICDCS.2013.51.

[80] Philipp Leitner and Juergen Cito. "Patterns in the Chaos - a Study of Performance Variation and Predictability in Public IaaS Clouds." In: *arXiv:1411.2429 [cs]* (Nov. 2014). arXiv: 1411.2429. (Visited on 02/24/2016).

[81] Baptiste Lepers, Vivien Quéma, and Alexandra Fedorova. "Thread and Memory Placement on NUMA Systems: Asymmetry Matters." In: *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC '15. Santa Clara, CA: USENIX Association, 2015, pp. 277–289. ISBN: 978-1-931971-225. URL: http://dl.acm.org/citation.cfm?id=2813767.2813788.

[82] Eliezer Levy and Abraham Silberschatz. "Distributed File Systems: Concepts and Examples." In: *ACM Comput. Surv.* 22.4 (Dec. 1990), pp. 321–374. ISSN: 0360-0300. DOI: 10.1145/98163.98169. URL: http://doi.acm.org/10.1145/98163.98169.

[83] C. S. Li et al. "Software defined environments: An introduction." In: *IBM Journal of Research and Development* 58.2/3 (2014), 1:1–1:11. ISSN: 0018-8646. DOI: 10.1147/JRD.2014.2298134.

[84] Chung-Sheng Li, Hubertus Franke, Colin Parris, Bulent Abali, Mukil Kesavan, and Victor Chang. "Composable architecture for rack scale big data computing." In: *Future Generation Computer Systems* 67 (2017), pp. 180 –193. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2016.07.014. URL: http://www.sciencedirect.com/science/article/pii/S0167739X16302631.

[85] T. Y. Liang and Y. W. Chang. "GridCuda: A Grid-Enabled CUDA Programming Toolkit." In: *2011 IEEE Workshops of International Conference on Advanced Information Networking and Applications*. 2011. DOI: 10.1109/WAINA.2011.82.

[86] A. D. Lin, C. S. Li, W. Liao, and H. Franke. "Capacity Optimization for Resource Pooling in Virtualized Data Centers with Composable Systems." In: *IEEE Transactions on Parallel and Distributed Systems* (2018). ISSN: 1045-9219. DOI: 10.1109/TPDS.2017.2757479.

[87] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. “Heracles: Improving Resource Efficiency at Scale.” In: *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*. ISCA ’15. Portland, Oregon: ACM, 2015, pp. 450–462. ISBN: 978-1-4503-3402-0. DOI: 10.1145/2749469.2749475.

[88] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. “Improving Resource Efficiency at Scale with Heracles.” In: *ACM Trans. Comput. Syst.* 34.2 (May 2016), 6:1–6:33. ISSN: 0734-2071. DOI: 10.1145/2882783.

[89] Virginia Lo, Kurt J. Windisch, Wanqian Liu, and Bill Nitzberg. “Noncontiguous Processor Allocation Algorithms for Mesh-Connected Multicomputers.” In: *IEEE Trans. Parallel Distrib. Syst.* 8.7 (July 1997), pp. 712–726. ISSN: 1045-9219. DOI: 10.1109/71.598346.

[90] L. Lovász and M.D. Plummer. “1 Matchings in Bipartite Graphs.” In: *Matching Theory*. Vol. 121. North-Holland Mathematics Studies. North-Holland, 1986. DOI: https://doi.org/10.1016/S0304-0208(08)73637-5.

[91] Nathan Luehr. “Fast Multi-GPU collectives with NCCL.” In: *https://devblogs.nvidia.com/fast-multi-gpu-collectives-nccl/*. NVIDIA Developer Blog. NVIDIA, 2016.

[92] Guillaume Mercier and Emmanuel Jeannot. “Improving MPI Applications Performance on Multicore Clusters with Rank Reordering.” In: *Recent Advances in the Message Passing Interface*. Ed. by Yiannis Cotronis, Anthony Danalis, Dimitrios S. Nikolopoulos, and Jack Dongarra. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 39–49. ISBN: 978-3-642-24449-0.

[93] Alexander M. Merritt, Vishakha Gupta, Abhishek Verma, Ada Gavrilovska, and Karsten Schwan. “Shadowfax: Scaling in Heterogeneous Cluster Systems via GPGPU Assemblies.” In: *Proceedings of the 5th International Workshop on Virtualization Technologies in Distributed Computing*. VTDC ’11. San Jose, California, USA: ACM, 2011, pp. 3–10. ISBN: 978-1-4503-0701-7. DOI: 10.1145/1996121.1996124. URL: http://doi.acm.org/10.1145/1996121.1996124.

[94] Mesos. *Mesos*. Accessed in: 25-March-2015. URL: http://mesos.apache.org.

[95] Hugo Meyer, José Carlos Sancho, Josue V. Quiroga, Ferad Zyulkyarov, Damian Roca, and Mario Nemirovsky. “Disaggregated Computing. An Evaluation of Current Trends for Datacentres.” In: *International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland*. 2017, pp. 685–694. DOI: 10.1016/j.procs.2017.05.129. URL: https://doi.org/10.1016/j.procs.2017.05.129.

[96] Paulius Micikevicius. “Multi-GPU Programming.” In: *https://www.nvidia.com/docs/IO/116711/sc1 multi-gpu.pdf*. Supercomputing 2011. NVIDIA, 2011.

[97] Microsoft. *Project Oxford - Cognitive Services APIs*. 2017. URL: https://www.microsoft.com/cognitive-services/.

[98] Raffaele Montella. *Introducing GVirtuS*. Accessed in: 06-February-2019. URL: https://github.com/RapidProjectH2020/GVirtuS.

[99] Raffaele Montella, Giulio Giunta, Giuliano Laccetti, Marco Lapegna, Carlo Palmieri, Carmine Ferraro, Valentina Pelliccia, Cheol-Ho Hong, Ivor T. A. Spence, and Dimitrios S. Nikolopoulos. "On the Virtualization of CUDA Based GPU Remoting on ARM and X86 Machines in the GVirtuS Framework." In: *International Journal of Parallel Programming* 45.5 (2017), pp. 1142–1163. DOI: 10.1007/s10766-016-0462-1. URL: https://doi.org/10.1007/s10766-016-0462-1.

[100] NNVIDIA. "Multi-GPU and multi-node collective communication primitives." In: *https://developer.nvidia.com/nccl*. NVIDIA Collective Communications Library (NCCL). NVIDIA, 2019.

[101] NVIDIA. *Multi-Process Service (MPS)*. Accessed in: 29-January-2019. URL: https://docs.nvidia.com/deploy/mps/index.html.

[102] NVIDIA. "The Power of Cloud Gaming." In: *http://www.nvidia. com/object/cloud-gaming.html*.

[103] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. "Q-clouds: Managing Performance Interference Effects for QoS-aware Clouds." In: *Proceedings of the 5th European Conference on Computer Systems*. EuroSys '10. New York, NY, USA: ACM, 2010, pp. 237–250. ISBN: 978-1-60558-577-2. DOI: 10.1145/1755913.1755938. (Visited on 02/26/2016).

[104] Marco A. S. Netto, Rodrigo N. Calheiros, Eduardo R. Rodrigues, Renato L. F. Cunha, and Rajkumar Buyya. "HPC Cloud for Scientific and Business Applications: Taxonomy, Vision, and Research Challenges." In: *ACM Comput. Surv.* 51.1 (Jan. 2018), 8:1–8:29. ISSN: 0360-0300. DOI: 10.1145/3150224. URL: http://doi.acm.org/10.1145/3150224.

[105] Dejan Novaković, Nedeljko Vasić, Stanko Novaković, Dejan Kostić, and Ricardo Bianchini. "DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments." In: *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*. San Jose, CA: USENIX, 2013. ISBN: 978-1-931971-01-0.

[106] Stanko Novakovic. "Rack-Scale Memory Pooling for Datacenters." In: *PhD. thesis*. ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE. switzerland: ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE, 2017. URL: https://infoscience.epfl.ch/record/228332/files/EPFL_TH7612.pdf?version=1.

[107] M. Oikawa, A. Kawai, K. Nomura, K. Yasuoka, K. Yoshikawa, and T. Narumi. "DS-CUDA: A Middleware to Use Many GPUs in the Cloud Environment." In: *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. 2012, pp. 1207–1214. DOI: 10.1109/SC.Companion.2012.146.

[108] OpenVZ. Accessed in: 21-January-2015. URL: http://openvz.org/main_page.

[109] OpenvStack. Accessed in: 21-January-2015. URL: http://www.openstack.org/.

[110] OpenvSwitch. Accessed in: 21-January-2015. URL: http://openvswitch.org/.

[111] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. “Sparrow: Distributed, Low Latency Scheduling.” In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP ’13. Farminton, Pennsylvania: ACM, 2013, pp. 69–84. ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522716. URL: http://doi.acm.org/10.1145/2517349.2522716.

[112] Albert Pagés, Rubn Serrano, Jordi Perell, and Salvatore Spadaro. “On the Benefits of Resource Disaggregation for Virtual Data Centre Provisioning in Optical Data Centres.” In: *Comput. Commun.* 107.C (July 2017). ISSN: 0140-3664. DOI: 10.1016/j.comcom.2017.03.009.

[113] François Pellegrini. *Scotch and libScotch 3.4 User’s Guide*. 2001.

[114] François Pellegrini and Jean Roman. *Experimental Analysis of the Dual Recursive Bipartitioning Algorithm for Static Mapping*. Tech. rep. Univ. Bordeaux I, 1996.

[115] Perfmon2. *Improving performance monitoring on Linux*. 2016. URL: http://perfmon2.sourceforge.net.

[116] S. Potluri, H. Wang, D. Bureddy, A. K. Singh, C. Rosales, and Dhabaleswar K. Panda. “Optimizing MPI Communication on Multi-GPU Systems Using CUDA Inter-Process Communication.” In: *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. IPDPSW ’12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 1848–1857. ISBN: 978-0-7695-4676-6. DOI: 10.1109/IPDPSW.2012.228.

[117] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. “Adaptive NUMA-aware Data Placement and Task Scheduling for Analytical Workloads in Main-memory Column-stores.” In: *Proc. VLDB Endow.* 10.2 (Oct. 2016), pp. 37–48. ISSN: 2150-8097. DOI: 10.14778/3015274.3015275. URL: https://doi.org/10.14778/3015274.3015275.

[118] J. R. Quinlan. “Induction of Decision Trees.” In: *Mach. Learn.* 1.1 (Mar. 1986), pp. 81–106. ISSN: 0885-6125. DOI: 10.1023/A:1022643204877. URL: http://dx.doi.org/10.1023/A:1022643204877.

[119] Moinuddin K. Qureshi and Yale N. Patt. “Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches.” In: *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 39. Washington, DC, USA: IEEE Computer Society, 2006, pp. 423–432. ISBN: 0-7695-2732-9. DOI: 10.1109/MICRO.2006.49.

[120] Mohammad Javad Rashti, Jonathan Green, Pavan Balaji, Ahmad Afsahi, and William Gropp. “Multi-core and Network Aware MPI Topology Functions.” In: *Proceedings of the 18th European MPI Users’ Group Conference on Recent Advances in the Message Passing Interface*. EuroMPI’11. Santorini, Greece: Springer-Verlag, 2011,

pp. 50–60. ISBN: 978-3-642-24448-3. URL: http://dl.acm.org/citation.cfm?id=2042476.2042484.

[121] C. Reano and F. Silla. "A Comparative Performance Analysis of Remote GPU Virtualization over Three Generations of GPUs." In: *46th International Conference on Parallel Processing WGorkshops (ICPPW)*. 2017, pp. 121–128. DOI: 10.1109/ICPPW.2017.29.

[122] Shuja ur Rehman Baig, Marcelo Amaral, Jordà Polo, and David Carrera. "Performance Characterization of Spark Workloads on Shared NUMA Systems." In: *2018 IEEE Fourth International Conference on Big Data Computing Service and Applications (BigDataService)*. 2018, pp. 41–48. DOI: 10.1109/BigDataService.2018.00015.

[123] David K. Rensin. *Kubernetes - Scheduling the Future at Cloud Scale*. 1005 Gravenstein Highway North Sebastopol, CA 95472, 2015, All. URL: http://www.oreilly.com/webops-perf/free/kubernetes.csp.

[124] Rocket. *Rocket - App Container runtime*. Accessed in: 25-March-2015. URL: https://github.com/coreos/rocket.

[125] Romana. *Just the Basics*. Accessed in: 29-December-2018. URL: https://romana.io/how/romana_basics/.

[126] Olga Russakovsky et al. "ImageNet Large Scale Visual Recognition Challenge." In: *International Journal of Computer Vision* 115.3 (2015), pp. 211–252. ISSN: 1573-1405. DOI: 10.1007/s11263-015-0816-y. URL: https://doi.org/10.1007/s11263-015-0816-y.

[127] Omar Sefraoui, Mohammed Aissaoui, and Mohsine Eleuldj. "OpenStack: Toward an Open-Source Solution for Cloud Computing." In: *International Journal of Computer Applications* 55.3 (2012), pp. 38–42. DOI: 10.5120/8738-2991.

[128] Lin Shi, Hao Chen, and Jianhua Sun. "vCUDA: GPU accelerated high performance computing in virtual machines." In: *2009 IEEE International Symposium on Parallel Distributed Processing*. 2009, pp. 1–11. DOI: 10.1109/IPDPS.2009.5161020.

[129] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. "The Hadoop Distributed File System." In: *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. MSST '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10. ISBN: 978-1-4244-7152-2. DOI: 10.1109/MSST.2010.5496972. URL: http://dx.doi.org/10.1109/MSST.2010.5496972.

[130] Singularity. *User Guide*. Accessed in: 19-December-2018. URL: https://www.sylabs.io/guides/3.0/user-guide/.

[131] Slurm. *HPC cluster workload management*. 2017. URL: http://slurm.schedmd.com/gres.html.

[132] Fabrizio Soppelsa and Chanwit Kaewkasi. *Native Docker Clustering with Swarm*. Packt Publishing, 2017. ISBN: 1786469758, 9781786469755.

[133] Shivram Srivastava. "The Poseidon an add-on Kubernetes scheduler for Firmament scheduler framework." In: *https://github.com/kubernetes-sigs/poseidon*.

[134] Philip Stoev. "SysBench: System evaluation benchmark." In: *https://github.com/nuodb/sysbench*. Git, 2012.

[135] Roberto Tamassia. "On Embedding a Graph in the Grid with the Minimum Number of Bends." In: *SIAM J. Comput.* 16.3 (June 1987), pp. 421–444. ISSN: 0097-5397. DOI: 10.1137/0216030.

[136] Stig Telfer. *The Crossroads of Cloud and HPC: OpenStack for Scientific Research*. Tech. rep. StackHPC Ltd. with the support of Cambridge University, 2016.

[137] Ozan Tuncer, Vitus J. Leung, and Ayse K. Coskun. "PaCMap: Topology Mapping of Unstructured Communication Patterns Onto Non-contiguous Allocations." In: *Proceedings of the 29th ACM on International Conference on Supercomputing*. ICS '15. Newport Beach, California, USA: ACM, 2015, pp. 37–46. ISBN: 978-1-4503-3559-1. DOI: 10.1145/2751205.2751225.

[138] Vinod Kumar Vavilapalli et al. "Apache Hadoop YARN: Yet Another Resource Negotiator." In: *Proceedings of the Symposium on Cloud Computing*. SOCC '13. Santa Clara, California: ACM, 2013. ISBN: 978-1-4503-2428-1. DOI: 10.1145/2523616.2523633.

[139] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. "Large-scale Cluster Management at Google with Borg." In: *Proceedings of the Tenth European Conference on Computer Systems*. EuroSys '15. Bordeaux, France: ACM, 2015. ISBN: 978-1-4503-3238-5. DOI: 10.1145/2741948.2741964.

[140] Akshat Verma, Puneet Ahuja, and Anindya Neogi. "Power-aware Dynamic Placement of HPC Applications." In: *Proceedings of the 22Nd Annual International Conference on Supercomputing*. ICS '08. New York, NY, USA: ACM, 2008, pp. 175–184. ISBN: 978-1-60558-158-3. DOI: 10.1145/1375527.1375555. (Visited on 02/26/2016).

[141] Linnan Wang, Wei Wu, George Bosilca, Richard W. Vuduc, and Zenglin Xu. "Efficient Communications in Training Large Scale Neural Networks." In: *CoRR* abs/1611.04255 (2016). URL: http://arxiv.org/abs/1611.04255.

[142] Weaveworks. *Network conatainers across any environment*. Accessed in: 29-December-2018. URL: https://www.weave.works/oss/net/.

[143] Kurt Windisch, Virginia Lo, and Bella Bose. "Contiguous And Non-Contiguous Processor Allocation Algorithms For K-Ary n-Cubes." In: *IEEE Transactions on Parallel and Distributed Systems* 8 (1995), pp. 712–726.

[144] J. Wu and B. Hong. "Collocating CPU-only Jobs with GPU-assisted Jobs on GPU-assisted HPC." In: *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. 2013. DOI: 10.1109/CCGrid.2013.19.

[145] M.G. Xavier, M.V. Neves, F.D. Rossi, T.C. Ferreto, T. Lange, and C.A.F. De Rose. “Performance Evaluation of Container-Based Virtualization for High Performance Computing Environments.” In: *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*. 2013, pp. 233–240. DOI: 10.1109/PDP.2013.41.

[146] S. Xiao, P. Balaji, Q. Zhu, R. Thakur, S. Coghlan, H. Lin, G. Wen, J. Hong, and W. Feng. “VOCL: An optimized environment for transparent virtualization of graphics processing units.” In: *2012 Innovative Parallel Computing (InPar)*. 2012, pp. 1–12. DOI: 10.1109/InPar.2012.6339609.

[147] Katherine Yelick, Susan Coghlan, Brent Draney, and Richard Shane Canon. *The Magellan Report on Cloud Computing for Science*. Tech. rep. [online] https://goo.gl/VfJNSH. Office of Advanced Scientific Computing Research (ASCR): U.S. Department of Energy, 2011. URL: https://goo.gl/VfJNSH.

[148] Seetharami R. Seelam Yu Bo Li IBM Research. *Speeding up Deep Learning Services: When GPUs meet Container Clouds, NVIDIA GPU Technology Conference*. Accessed in: 5-August-2017. 2017. URL: http://on-demand.gputechconf.com/gtc/2017/presentation/s7258-seetharami-seelam-speed-up-deep-learning-service.pdf.