Université de Neuchâtel
Faculté des Sciences
Institut d'Informatique

Towards Energy Efficiency by Leveraging Heterogeneity of Clusters and Applications

par

Isabelly Rocha

Thèse

présentée à la Faculté des Sciences

pour l'obtention du grade de Docteur ès Sciences

Acceptée sur proposition du jury:

Prof. Pascal Felber, directeur de thèse
Université de Neuchâtel, Suisse

Prof. Andrey Brito, rapporteur
Universidade Federal de Campina Grande, Brésil

Dr. Sonia Ben Mokhtar, rapporteure
CNRS/INSA, Lyon, France

Prof. Marcelo Pasin, rapporteur
Haute école Arc, Suisse

Dr. Valerio Schiavoni, rapporteur
Université de Neuchâtel, Suisse

Soutenue le 1 décembre 2021

UNIVERSITÉ DE NEUCHÂTEL

Faculté des Sciences
Secrétariat-décanat de Faculté
Rue Emile-Argand 11
2000 Neuchâtel – Suisse
Tél : + 41 (0)32 718 21 00
E-mail : secretariat.sciences@unine.ch

IMPRIMATUR POUR THESE DE DOCTORAT

La Faculté des sciences de l'Université de Neuchâtel autorise l'impression de la présente thèse soutenue par

Madame Isabelly LOUREDO ROCHA

Titre:

"Towards Energy Efficiency by Leveraging Heterogeneity of Clusters and Applications"

sur le rapport des membres du jury composé comme suit:

- Prof. Pascal Felber, directeur de thèse, Université de Neuchâtel, Suisse
- Dr Sonia Ben Mokhtar, CNRS/INSA, Lyon, France
- Dr Valerio Schiavoni, Université de Neuchâtel, Suisse
- Prof. Marcelo Pasin, HE-Arc, Neuchâtel, Suisse
- Prof. Andrey Brito, U.F. Campina Grande, Brésil

Neuchâtel, le 9 décembre 2021 Le Doyen, Prof. A. Bangerter

Abstract

The development of applications is becoming every day more diverse in terms of requirements and objectives. Considering this, major cloud providers offer a large variety of hardware types to better fit their customers' needs. Moreover, a common approach is to design applications in a distributed setup. This means that applications typically consist of building blocks performing individual tasks and communicating with each other to achieve a given objective. Similarly, in order to fulfill the overall requirements of the application, each of its building blocks could feature different instances (*e.g.*, GPUs and FPGAs) as their individual requirements might differ from each other.

In this work, we add energy efficiency on top of the multiple objectives that an application might have such as performance or security. Recent studies estimate the already alarming and still increasing energy consumption in the cloud computing sector, especially coming from data centers. We aim at tackling this problem from two different perspectives: 1) cloud providers, and 2) application developers.

Looking from the cloud providers point of view, we start by proposing a novel scheduling strategy focused on single-tenant scenarios which leverages the heterogeneity of both clusters and applications to better address the trade-off between performance and energy efficiency. Then, we extend this approach to the multi-tenant scenario where we rely on differential approximation and sprinting to avoid job eviction and resource waste leading to an overall energy gain.

In the second part of this thesis, we switch the view from cloud providers to the developers of applications. Here we show that peculiar characteristics of applications can also be leveraged to achieve more energy efficient approaches while maintaining or even further improving their remaining objectives. For that end, we use the automatization of parameter tuning for deep learning applications as use case.

We start by characterizing such applications to motivate how relevant the tuning approach taken can be on the end results for tuning, training, and inference. After that, we propose a novel auto tuning approach which takes advantage of the high parallelism and recurring characteristics of such application to simultaneously consider performance, accuracy, and energy efficiency. Once the tuning process is complete, the final model is typically deployed to be used for inference. Therefore, our next step is to extend our approach to perform inference-aware tuning. With this complementary step, we additionally output to the users recommendations on how to deploy their models such that inference-related objectives are also achieved.

Finally, independent of the perspective taken, security is one of the major concerns in this area. Cloud providers need to ensure data privacy to their customers, in particular for multi-tenant scenarios where resources are shared among several users. On the other hand, applications related to deep learning are often trained on user-sensitive data which also has to be protected. Recently, trusted executed environments (TEEs) became largely available in server machines as a mechanism to guarantee security. However, such approaches still have their limitations when it comes to usability and the overhead which could be added to applications. Considering this, we close this thesis by proposing an approach which can automate the process of ensuring security in the context of heterogeneous applications and clusters.

In summary, this thesis aims at achieving more energy efficient computing while respecting other application requirements such as performance, security, or accuracy. We propose new mechanisms that can be generally applied during the management of jobs as well as optimizations which are specific to a deep learning use case application. Finally, we conclude this work by exploring a TEE based strategy for adding a security layer on top of the proposed approaches.

Keywords: energy efficiency, heterogeneous environments, distributed systems, deep learning.

Résumé

Le développement d'applications se diversifie de jour en jour, dans ses objectifs comme ses contraintes. Pour cette raison, les fournisseurs de service Cloud proposent une grande variété d'offres permettant de répondre aux besoins de leurs utilisateurs. De plus, il devient fréquent de développer des applications pour des architectures distribuées. Cela signifie que les applications sont généralement constituées de blocs qui effectuent des tâches et communiquent les uns avec les autres pour atteindre un objectif commun. Aussi, pour répondre aux besoins de l'application, chacun de ses blocs peut être instancié plusieurs fois (par exemple, les GPU et FPGA) puisque leurs besoins respectifs peuvent différer d'une instance à l'autre.

Dans nos travaux, nous ajoutons l'efficacité énergétique comme une contrainte supplémentaire parmi les objectifs multiples déjà assurés par une application, au même titre que la performance ou la sécurité. De récentes études ont estimé la consommation d'énergie du secteur du Cloud qui, malgré l'inquiétude qu'elle soulève, continue d'augmenter, en particulier dans les centres de données. Nous nous attaquons à ce sujet sous les angles de vue différents de deux acteurs : 1) les fournisseurs de service Cloud, et 2) les développeurs d'applications.

Du point de vue des fournisseurs de service Cloud, nous commençons par proposer une nouvelle stratégie d'ordonnancement dédiée au cas d'un utilisateur unique, qui exploite l'hétérogénéité des centres de calcul et des applications, afin d'obtenir un meilleur compromis entre performance et efficacité énergétique. Puis, nous étendons cette approche aux cas d'utilisateurs multiples, pour lesquels nous nous appuyons sur l'approximation différentielle et la technique de sprint pour éviter l'éviction de tâches et les pertes inutiles d'énergie qui font augmenter la consommation globale.

Dans la seconde partie de cette thèse, nous quittons le point de vue des fournisseurs de service Cloud pour celui des développeurs d'applications. Nous montrons que les caractéristiques particulières de certaines applications peuvent être exploitées afin d'opter pour des approches énergétiquement plus efficaces, tout en maintenant voire en améliorant d'autres objectifs. Pour cela, nous utilisons une technique de paramétrage automatique que nous avons appliqué à des applications d'apprentissage profond.

Nous commençons par caractériser ces applications pour vérifier l'impact que le paramétrage choisi peut avoir sur les phases de paramétrage, d'apprentissage, et d'inférence. Ensuite, nous proposons une approche de paramétrage automatisé qui tire profit de la parallélisation et des caractéristiques récurrentes de ce type d'applications pour prendre en compte simultanément les performances, la précision, et l'efficacité énergétique. Une fois cette étape de paramétrage effectuée, le modèle obtenu est déployé pour faire de l'inférence. Ainsi, la prochaine étape consiste à étendre notre approche pour prendre en compte l'inférence durant le paramétrage. Grâce à cette étape complémentaire, nous proposons à l'utilisateur des recommandations pour le déploiement de leurs modèles de manière que les objectifs liés à l'inférence soient également atteints.

Enfin, indépendamment du point de vue choisi, la sécurité reste une des préoccupations majeures dans ce domaine. Les fournisseurs de service Cloud doivent pleinement assurer la confidentialité des données de leurs utilisateurs, en particulier dans les cas des utilisateurs partageant les mêmes infrastructures. D'autre part, les applications en lien avec l'apprentissage profond sont souvent entrainées avec des jeux de données sensibles qui doivent également être protégé. Les environnements d'exécution de confiance (TEEs) sont devenus des composants fréquemment présents dans l'architecture des serveurs, fournissant ainsi une garantie de sécurité supplémentaire. Cependant, ces mécanismes ont encore de nombreuses limitations en termes de facilité de mise en œuvre et de coûts supplémentaires en matière de performances. Pour cette raison, nous terminons cette thèse en proposant une approche permettant d'automatiser les garanties de sécurité dans le contexte d'applications et de centres de calcul hétérogènes.

En résumé, cette thèse a pour objectif d'apporter davantage d'efficacité énergétique aux centres de calcul, tout en préservant les contraintes habituelles que sont les performances, la sécurité ou la précision. Nous proposons de nouveaux mécanismes pouvant être appliqués lors de l'étape de la gestion des tâches, ainsi que des optimisations spécifiques au cas des application d'apprentissage profond. Enfin, nous concluons ce travail en proposant l'ajout d'une couche de sécurité supplémentaire sur ces différentes idées, en se basant sur les environnements d'exécution de confiance.

Mots clés : Efficacité énergétique, environnements hétérogènes, systèmes distribués, apprentissage profond.

Zusammenfassung

Betrachtet man die Entwicklung von Anwendungen, so stellt man fest, dass ihre Anforderungen und Ziele von Tag zu Tag anspruchsvoller werden. Aus diesem Grund bieten die grossen Cloud-Anbieter eine Vielzahl von Hardwaretypen an, um den Bedürfnissen ihrer Kunden gerecht zu werden. Darüber hinaus ist es üblich, Anwendungen als verteiltes System zu entwerfen. Dies bedeutet, dass die Anwendungen üblicherweise aus Bausteinen bestehen, die einzelne Aufgaben ausführen und dabei miteinander kommunizieren, um ein bestimmtes Ziel zu erreichen. Um die Anforderungen an die Anwendungen zu erfüllen, könnte jeder dieser Bausteine unterschiedliche Instanzen aufweisen (z.Bsp. GPU und FPGA), da ihre einzelnen Anforderungen möglicherweise voneinander abweichen.

In dieser Arbeit wollen wir neben der Vielzahl an Anforderungen, die an Anwendungen gestellt werden (z.Bsp. Leistung und Sicherheit), ein weiteres Augenmerk auf die Energieeffizienz legen. Neuesten Studien zufolge nimmt der Energieverbrauch im Bereich des Cloud Computing und besonders in Rechenzentren drastisch zu. Unser Ziel ist es, dieses Problem von zwei Blickrichtungen zu betrachten: 1) Cloud-Anbieter und 2) Anwendungsentwickler.

Aus der Sicht von Cloud-Anbieter schlagen wir eine neue Planungsstrategie vor, die sich auf Single-Tenant-Szenarien fokussiert. Hierbei soll die Heterogenität zwischen Clustern und Anwendungen genutzt werden, um den Kompromiss zwischen Leistung und Energieeffizienz besser anzugehen. Dann übertragen wir unseren Ansatz auf ein Multi-Tenant-Szenario. Dafür stützen wir uns auf eine differentielle Annäherung und Sprinting, um eine Job Evakuierung und Ressourcenverschwendung zu vermeiden, was insgesamt zu einem Energiegewinn führt.

Im zweiten Teil dieser Arbeit ändern wir unsere Perspektive von den Cloud-Anbietern zu den Anwendungsentwickler. Hier zeigen wir, dass bestimmte Eigenschaften von Anwendungen auch genutzt werden können, um energieeffizientere Ansätze zu erzielen und die übrigen Zielanforderungen erreicht oder sogar noch übertroffen werden. Für diesen Zweck nutzen wir die Automatisierung der Paramaterabstimmung für Deep-Learning-Anwendungen als Use-Case.

Wir beginnen mit der Charakterisierung solcher Anwendungen um zu verdeutlichen, wie relevant der gewählte Ansatz für das Tuning, Training und die Inferenz der Resultate sein kann. Im Anschluss schlagen wir einen neuartigen Auto-Tuning-Ansatz vor, der die hohe Parallelität und die wiederkehrenden Eigenschaften einer solchen Anwendung nutzt. Hierdurch ist es uns möglich gleichzeitig Leistung, Genauigkeit und Energieeffizienz zu berücksichtigen. Sobald der Tuning-Prozess abgeschlossen ist, wird das endgültige Modell in der Regel für die Inferenz verwendet. Daher besteht unser nächster Schritt darin, unseren Ansatz zu erweitern, um eine inferenzbewusstes Tuning durchzuführen. In diesem ergänzenden Schritt geben wir den Nutzern zusätzlich Empfehlungen, wie sie ihre Modelle so einsetzen können, damit auch inferenzbezogene Ziele erreicht werden.

Unabhängig der gewählten Perspektive zählt die Sicherheit zu einem der Hauptanliegen in diesem Bereich. Cloud-Anbieter müssen den Datenschutz für ihre Kunden gewährleisten, insbesondere in Multi-Tenant-Szenarien, bei denen Ressourcen zwischen mehreren Nutzern geteilt werden. Andererseits werden Anwendungen im Zusammenhang mit Deep Learning häufig mit benutzersensiblen Daten trainiert, die ebenfalls geschützt werden müssen. Seit einiger Zeit sind Trusted Executed Environments (TEEs) als Mechanismus zur Gewährleistung der Sicherheit in Server-Rechnern verfügbar. Solche Ansätze haben jedoch immer noch ihre Grenzen, wenn es um die Benutzerfreundlichkeit und den möglichen Overhead geht. In Anbetracht dessen schlagen wir zum Abschluss dieser Arbeit einen Ansatz vor, der den Prozess der Gewährleistung von Sicherheit heterogener Anwendungen und Cluster automatisieren kann.

Zusammenfassend lässt sich sagen, dass diese Arbeit darauf abzielt, eine energieeffizientere Datenverarbeitung zu erreichen, um dabei andere Anwendungsanforderungen wie Leistung, Sicherheit oder Genauigkeit zu berücksichtigen. Wir schlagen neue Mechanismen vor, die allgemein bei der Verwaltung von Aufträgen sowie bei der Optimierung von typischen Deep-Learning-Anwendungen eingesetzt werden können. Abschliessend untersuchen wir eine TEE-basierte Strategie, um den vorgeschlagenen Ansätzen als zusätzliche Dimension Sicherheit hinzuzufügen.

Stichwörter: Energieeffizienz, heterogene Umgebungen, verteilte Systeme, Deep Learning.

To my family.

Acknowledgements

I would like to start by expressing my gratitude to my advisor, Prof. Pascal Felber, for all his guidance and support. Furthermore, I would like to thank Prof. Andrey Brito, Dr. Sonia Ben Mokhtar, Prof. Marcelo Pasin and Dr. Valerio Schiavoni for being part of my committee. Thank you for taking the time to evaluate my work, providing me with valuable feedback which helped improving this manuscript. In particular, special thanks to Valerio who worked so closely with me in almost all my contributions. Even when working late at night, during weekends or holidays, you were always there motivating me. I can never thank you enough.

Special thanks also to Andrey, who almost 10 years ago convinced me to work on a research project about distributed systems. He patiently introduced me to this challenging but new and exciting area, which I since then did not stop exploring. I'm sure that I could only get this far because of the strong foundation which you helped me building back then.

During my PhD I also had the opportunity to collaborate with many inspiring people, including Prof. Romain Rouvoy, Robert Birk, Nathaniel Morris, Yasmine Djebrouni, Prof. Sara Bouchenak, Prof. Vania Marangozova, Xavier Martorell, and many others. In particular, Prof. Lydia Chen who advised me through a big part of these collaborations. I could not put into words how much I've learned from you and what a role model you became to me. Thank you for warmly hosting me at TU Delft, for all the meetings, discussions and advises you gave me in the last years.

I also would like to thank all my colleagues from the IIUN. Our lunch breaks, biking and running tours, nights out, talks, drinks and trips made this journey much more fun. Particular thanks to Christian, the best office mate, Dorian, the best running buddy, Rémi, the best music advisor, and Laurent, the best climbing partner. Thank you, guys!

Now, I would never have made it here without the support of my family and friends. So I also would like to say a special thanks to my mom, dad and sis (Iêda, Janildo and Jamilly) who always encouraged my dreams, even when they implied in a lot of *saudade*. Still about family, thank you to Thea for all the lightened up candles, Jonas, Benno, Ilse, Dana and Palina.

Thank you to all the friends who came visiting me during this time, Christian, Alex, Elena, Mariëlle, Ana, Peter, Alexandra, Thiago, Niloo joon, Neptune and Stefan. To the new friends who made me feel like home, especially Débora, Nicole, Carla and Bea. Thank you also to all the other friends who managed to support me even from far away, including Isabella, Rigel, Vivi, Rayane, Warlinho, Ingryd, Alberto, Lázaro, and many others. Particular thanks to Lili, who shared the struggles of this academic life with me. We made it, miga!

Last but (definitely) not least, thank you to my better half, Sylvan, who so often drove 1300km on the weekends just for a few hours visit. Thank you for believing in me, supporting my decisions and always having my back. Finally, to Yoshi, our four legged kid who will never read this but was the best home office buddy I could ever have wished for.

Contents

List of Acronyms

FCFS	first come, first served
FIFO	first in, first out
ML	machine learning
DL	deep learning
DAG	directed acyclic graph
DVFS	dynamic voltage and frequency scaling
SGD	stochastic gradient descent
VM	virtual machine
BM	bare metal
RaW	read after write
WaW	write after write
WaR	write after read
Ml	machine learning
DL	deep learning
CNN	convolutional neural network
LSTM	long short-term memory
MLP	multilayer perceptron

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Recent studies report alarming numbers regarding energy consumption coming from data centers [1]. In a yearly basis, cloud computing ranked fifth in the list of consumption by countries, consuming more than entire countries such as India or Germany. Considering this, we leverage characteristics of distributed systems to design energy efficient computing strategies. However, applications already have other requirements such as latency, throughput and security. Therefore, this work aims not only at achieving lower energy consumption but at doing so while still maintaining or improving other relevant requirements when compared against equivalent approaches.

1.1 Context and Motivation

Cloud providers usually offer diverse types of hardware for their users. Customers exploit this option to deploy cloud instances featuring GPUs, FPGAs, architectures other than x86 (*e.g.*, ARM, IBM Power8), or featuring certain specific extensions (*e.g.*, Intel SGX). With containers being the *de facto* standard for micro-services, or to execute computing tasks, the underlying container orchestrator (*e.g.*, Kubernetes) should be designed so as to take into account and exploit this hardware diversity. In addition, besides the feature range provided by different machines, there is an often overlooked diversity in the energy requirements introduced by hardware heterogeneity, which is simply ignored by default container orchestrator's placement strategies.

When it comes to multi-tenant scenarios, a common approach is to rely on user's priority to perform job placement or decide on which jobs to evict once the system reaches its saturation point. However, the latency advantage of high-priority jobs comes at the cost of severe latency degradation of low-priority jobs as well as daunting resource waste caused by repetitive eviction and re-execution of low-priority jobs. An alternative to that, which we advocate in this work, is to explore the idea of differential approximation to avoid the eviction of low-priority jobs and applying sprinting techniques to accelerate high priority jobs. This unique combination avoids the eviction of low-priority jobs and its consequent latency degradation as well as resource waste.

Considering this, in the first part of this dissertation, we wish to answer the following questions:

- How can the hardware heterogeneity of clusters be combined with the diverse requirements of applications to achieve a better placement strategy?
- Is there an alternative approach to low-priority jobs eviction in the multi-tenant scenario which leads to less resource waste?

Now, shifting the perspective from the cluster managing point of view to application developers, the next question we address in the second part our work is:

- Are there application specific optimizations which can be performed to achieve better energy constraints while preserving the application's requirements?

In order to address this question we explore the problem of Deep Neural Networks (DNNs) parameters tuning. DNNs have demonstrated impressive performance on many machine-learning tasks such as image recognition and language modeling, and are becoming prevalent even on mobile platforms. Despite so,

designing neural architectures still remains a manual, time-consuming process that requires profound domain knowledge.

Recently, Parameter Tuning Servers (*i.e.*, autotuners) have gathered the attention of industry and academia. Those systems allow users from all domains to automatically achieve the desired model accuracy for their applications. However, the iteration between system parameters and hyperparameters is often still overseen by state-of-the-art autotuners. Moreover, although the entire process of tuning and training models is performed solely to be deployed for inference, state-of-the-art approaches typically ignore system-oriented and inference-related objectives such as runtime, memory usage, and power consumption.

Finally, independent of the perspective taken (*i.e.*, clusters or applications), security is a major concern. In the specific example of tuning DNNs, for instance, both training and inference are often performed using user's sensitive data, which makes privacy preserving a top priority for such applications. From the cloud provider's perspective, it is crucial to preserve their customer's privacy - especially in the multi-tenant scenario where resources are shared among multiple users.

To address this problem, hardware assisted protection mechanisms are becoming a popular solution both in private and public data centers. However, using such mechanisms may result in energy and performance overheads. Moreover, the solution does not always come off the shelf for application developers. Therefore, the last question we wish to answer in this dissertation is the following:

- How can we automate the process of ensuring security to heterogeneous applications and clusters?

1.2 Contributions

The main contributions of this work are as following:

- Design, implementation and evaluation of **Heats**, a task-oriented and energy-aware orchestrator for containerized applications targeting heterogeneous single-tenant clusters. We design our system to allow customers to trade performance vs. energy requirements. First, we learn the performance and energy features of the physical hosts. Then, we monitor the execution of tasks on the hosts and opportunistically migrates them onto different cluster nodes to match the customer-required deployment trade-offs. As a consequence, our approach can yield considerable energy savings and only marginally affect the overall runtime of deployed tasks.

- Proposal of **DiAS**, a new resource management design that exploits the idea of differential approximation and sprinting, providing different levels of accuracy and computation frequency to jobs with different priority levels. Instead of evicting low-priority jobs, differential approximation deflates their workload, abiding by their minimum accuracy requirements, thereby making resources available for high-priority jobs. Differential approximation is thus able to reduce resource waste, improve the latency of low-priority jobs by avoiding evictions and re-executions, and, most importantly, lower energy consumption.

- We contribute with a thorough **characterization of learning applications** in the context of parameters tuning. We collect and perform an in-depth analysis of workload execution traces to compare the efficiency of different configuration strategies. We consider tuning only hyper-parameters, tuning only platform parameters, and jointly tuning both hyper-parameters and platform parameters. This shows the relevance of multi-level parameter configuration to provide better results in terms of model quality and execution time while also optimizing resource costs.

- **PipeTune**: a framework for DNN learning jobs that addresses the trade-offs between hyperparameter and system parameter tuning. We take advantage of the high parallelism and recurring characteristics of such jobs to minimize the learning cost via a pipelined simultaneous tuning of both hyper and system parameters. With that, we not only considerably reduce the energy consumption but also improve the training and overall tuning performance.

- **EdgeTune**: a novel inference-aware parameter tuning server. For this approach, we rely on inference estimated metrics collected from our emulation server running asynchronously from the main

tuning process. We propose a novel one-fold tuning algorithm that employs the principle of multi-fidelity and simultaneously explores multiple tuning budgets, which the prior art can only handle as suboptimal case of single type of budget. Besides of performance and energy improvements, this approach outputs to the user the optimal inference parameters which should be used for deploying the final achieved model.

- Finally, to handle the security aspect of applications, **SGX-OmpSs** is presented. We propose the unique combination of Intel Software Guard Extension (SGX) and OmpSs programming model to provide security while requiring minimal effort from application developers. Through OmpSs, our approach supports asynchronous task based parallelism and hardware heterogeneity by using the data dependencies between tasks of the application which can be specified by code annotations. Our approach leads to considerable energy and performance gains in comparison to approaches relying solely in SGX.

A list of scientific papers including these contributions can be found in Appendix A.

1.3 Outline

This manuscript is organized in 9 chapters. The first chapter is introduction and the remaining ones are organized as follows.

In **Chapter 2** we present the background and state-of-the-art literature for distributed applications and clusters. We then focus on the DNN parameter tuning problem, introducing the relevant concepts to the applications we later consider as use case. Finally, we introduce the concepts of hardware assisted mechanisms and particularly Intel SGX.

We start exploring the problem from the cloud providers perspective in **Chapter 3** by designing and evaluating a task-oriented and energy-aware orchestrator for containerized applications targeting heterogeneous single-tenant clusters. Then, in **Chapter 4**, we extend this idea to cover the multi-tenant scenario. Here we explore the multi-priority concept combined with differential approximation and sprinting to achieve our objectives.

With both single-tenant and multi-tenant scenarios covered, we shift the perspective to the application developers and explore how specific characteristics can be leveraged to also achieve energy efficiency. For that we take as use case the DNN parameter tuning problem which we characterize in **Chapter 5**. Here we demonstrate how considering system parameters and performing multi-level tuning can both lead to energy efficiency and improve application performance.

Once the relevance of the problem is presented, we start introducing our proposals in **Chapter 6**. After presenting the design and implementation details of our novel tuning approach, we also perform a thorough evaluation for single-tenant and multi-tenant scenarios. Until now we only consider the training phase of the DNN applications, however, the main objective of training and tuning such applications is to deploy the final model for production where it will be used for inference. Therefore, **Chapter 7** extends the idea presented to also consider the inference phase. For that, we explore the concept of batching inference and the principle of multi-fidelity to simultaneously explore multiple tuning budgets.

Independent of the perspective taken, security is a major concern of this area and, therefore, another requirement which we have to consider. For that, in **Chapter 8** we discuss a novel approach combining Intel SGX with OmpSs programming model. Here we discuss how this combination can ensure security on top of exploring the heterogeneity aspects of clusters and applications to be energy efficient.

Finally, **Chapter 9** revisits our contributions both from the perspective of cloud providers and application developers. Before concluding, we discuss some of the research perspectives opened by this thesis together with ideas on how to address them.

Chapter 2

Background and Related Work

In the introduction, we proposed two main perspectives from which we can tackle the problem of making energy efficient applications. The first one assumes that the cloud providers will perform optimizations based on customers requirements but independent of application details. In the second, application developers take the optimizers role and uses specific characteristics of their problems to perform the desired optimizations. In the latter case, cloud providers can also provide engines with such out-of-the-box optimizations in the application granularity. Further more, in both cases we assume applications consisting of building blocks (*e.g.*, tasks) which can be parallelized and distributed across a cluster.

We start this chapter by giving some preliminary concepts about distributed applications and their main characteristics which are relevant to our context. Then, we discuss the concept of heterogeneous clusters, the differences between single-tenant and multi-tenant environments, as well as their challenges. Furthermore, as security is a major concern from both sides of this approach (*i.e.*, cloud providers and application developers), we introduce the concept of hardware-assisted mechanisms which we later rely on to cover security requirements.

With these general concepts introduced, we then introduce the preliminaries for the parameter tuning problem, which we later consider as an use case to show how applications characteristics can be leveraged to achieve energy efficiency. Each of these main points also refer to their respective related work and draw a parallel between state-of-the-art systems and the solutions we propose here. Finally, we give an overview of the LEGaTO research project which a significant part of this thesis collaborated with.

2.1 Distributed Task-Based Applications

Throughout this thesis, we consider applications which are parallelizable and distributed across multiple machines (*e.g.*, on-premise clusters or public clouds). Each composing part of such applications have its own requirements and characteristics which vary depending on the task being performed. Next, we detail some of these characteristics, their differences, and ways to automatically handle them.

One of the most relevant characteristics of applications for this work is whether they are stateful or stateless. The main difference between stateful and stateless applications is that the former requires backing storage while the later does not. In stateful applications, the server processes requests based on the combination of information 1) embedded in the request, and 2) stored from earlier requests. Considering this, in the type of application we explore here, the same server must be used to process all requests linked to the same state information or the state information must be shared with all servers that need it. On the other hand, in stateless applications different servers can process different requests as the servers process requests based only on information containing in the request itself.

Independent of the application being stateful or stateless, the tasks consisting of a given application can also be categorized as synchronous or asynchronous. Asynchronous parallelism [2] is commonly used for tasks that need to do work independent from the current one but does not need to make the other tasks wait and be blocked by waiting for a response that is not relevant for them — as it is the case in synchronous parallelism. In such applications, if more than one task is started then all these tasks will immediately be started but completed independent of each other.

For instance, in a matrix multiplication, we split the work into different tasks, each taking care of different and independent segments of the input matrices. Their outputs do not depend on the completion of each other. In this example, the application needs to wait until all tasks are completed before returning the result. However, the computations can run in an asynchronous manner.

A common approach to automatically differentiate synchronous from asynchronous tasks as well as the dependencies between synchronous tasks is using data dependency graphs. A data dependency is generated when a program statement (in our context a task) reads from or writes into a value that is written by the data of a preceding statement. The technique of analyzing this information is used to discover data dependencies among statements (or tasks).

More specifically, when dealing with this type of applications, we rely on OmpSs [2], a programming model which is based on the OpenMP with some modifications to its execution model: it uses a thread-pool execution model instead of the traditional OpenMP fork-join model. A master thread starts the execution, while several other threads cooperate executing the work that the master thread creates from work sharing or task constructs. Therefore, there is no need for a parallel region. Nesting of constructs allows other threads to generate work as well.

OmpSs allows annotating function declarations or definitions with a task directive. In this case, any call to the function creates a new task that will execute the function body. The data environment of the task is captured from the function arguments. It integrates the StarSs dependency support and allows annotating tasks with the three clauses explained earlier: input, output, inout. They allow expressing, respectively, that a given task depends on some data produced before, that will produce some data, or both. The syntax in the clause allows specifying scalars, arrays, pointers and pointed data. Data addresses and sizes do not need to be constant at compile time since they are detected at runtime. Also, the `taskwait` construct (used as a barrier after parallel code) is extended as well with the on clause, which allows the encountering task to block until some data is produced. Finally, the OmpSs environment is built on top of Mercurium compiler and Nanos++ runtime system which we introduce later.

In our scenario, the task construct allows expressing data dependencies among tasks using the `in`, `out` and `inout` clauses, standing for input, output and input/output respectively. They allow to specify for each task in the program what data a task is waiting for and signaling is readiness. Whether the task really uses that data in the specified way it's the programmer responsibility. Each time a new task is created, its `in` and `out` dependencies are matched against those of existing tasks. If a dependency (*i.e.*, RaW, WaW or WaR) is found then the task becomes a successor of the corresponding tasks. This process creates a task dependency graph at runtime and tasks are scheduled for execution as soon as all their predecessors in the graph have finished or at creation if they have no predecessors.

Nanos++: parallel runtime library [3] with the aim of enabling easy development of different parts of the runtime so researchers have a platform that allows them to try different mechanisms. The scheduling policy, the throttling policy, the dependency approach, the barrier implementations, and work sharing mechanisms, the instrumentation layer and the architectural dependent level are examples of plugins that developers may easily implement using Nanos++. This extensibility does not come for free. The runtime overheads are slightly increased, but they should be low enough for results to be meaningful except for cases of extremely fine grain applications.

Mercurium: C/C++/Fortran source-to-source compilation infrastructure [4] used, together with the Nanos++ Runtime Library, to implement the OmpSs programming model. Mercurium has configuration flags which are used to define which profile to use, *i.e.*, the behavior specifying which phases of Mercurium to execute or which backend compiler to use.

Listing 1 shows the example of an OmpSs annotated application. The programmer can specify a task by simply using the `task` construct. This construct can appear inside any code block of the program, which still mark the following statement as a task. This statement can contain several task constructs such as `depend`, `priority`, `if`, and `final`. In our example we see some of the following two clauses being used: `label` and `firstprivate`. The `label` clause defines a string literal that can be used by any performance or debugger tool to identify the task with a more human-readable format, and `firstprivate` declares one or more list items to be private to a task (*i.e.*, the task receives a new list item).

Algorithm 1 Fragment of OmpSs annotated dot product app.

```
double dot_product (long N, long CHUNK_SIZE, double A[N], double B[N])
{
  ...
  for (long i = 0; i < N; i += CHUNK_SIZE) {
    actual_size = (N - i >= CHUNK_SIZE) ? CHUNK_SIZE : N - i;

    #pragma omp task label(dot_prod) firstprivate( j, i, actual_size )
    {
      C[j]=0;
      for (long ii = 0; ii < actual_size; ii++)
        C[j]+= A[i+ii] * B[i+ii];
    }

    #pragma omp task label(increment) firstprivate(j)
    result += C[j];

    j++;
  }
  ...
}
```

The last major characteristic of applications which we explore in this work is the heterogeneity. Heterogeneous programming consists of defining several versions of a given application to support hardware heterogeneity and define in the application which parts of it should run in each device. In OmpSs, the support for device heterogeneity comes through the `target` construct which is to specify that a given element can be run in a set of devices. The target construct can be applied to either a task construct, which means that the task can be executed on a device, or a function definition, which means that this function has to be present in the device code. For instance, if an application consists of tasks with an unbalanced load then you could designate the more computationally intense tasks to run in accelerators such as FPGAs or GPUs and the other tasks to run in less powerful devices.

An example of application with these characteristics would be a deep neural network consisting of layers with different types where some layers are more complex than others. More specifically, a convolutional layer size 4x4 with 100 filters has many more parameters than a convolutional layer size 3x3 with 50 filters and is, therefore, also more complex to train. In this case, the former layer could be trained using more resources or on a more powerful machine than the latter. This approach allows the application to achieve better performance but also to avoid unnecessary overheads.

2.2 Cluster Management

Whether deployed on-premises clusters or relying on public cloud providers, such types of applications discussed in Section 2.1 require complex management including scheduling of tasks, orchestration of the system, and monitoring. In this section we introduce the main concepts related to the perspective taken throughout this work together with a discussion on the state-of-the-art literature. We start by discussing the background of priority scheduling, big data processing engines, and computational sprinting.

A number of field studies [5]–[8] from publicly available big data cluster traces show that priority scheduling is widely adopted in production big data systems. Workloads are defined in the unit of jobs that are in turn composed of multiple tasks. Jobs are divided into multiple classes, each of which is assigned with a priority level. For example, Google clusters employ 12 priority levels [8]. Performance of high-priority jobs is typically enforced at the cost of low-priority jobs. Earlier studies [6] show that jobs with the lowest priority (priority 0) are repetitively evicted by the scheduler, due to the arrival of high priority workloads.

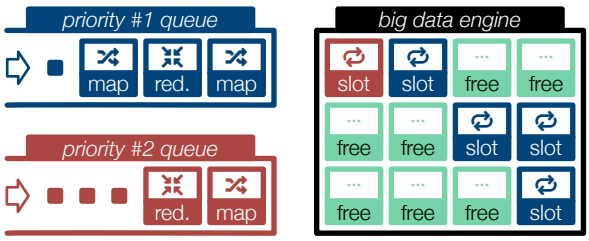


Figure 2.1: Schematics of priority scheduling for big data jobs.

As the analyzed Google traces show, the unfortunate consequences of eviction in production systems are two-fold:

1. high amount of resources, *i.e.*, 25% machine time (*i.e.*, including, for instance, waiting for input/output operations) and 30% CPU time (*i.e.*, amount of time the step utilizes CPU resources), are spent/wasted on subsequent evictions;

2. significant latency degradation for low priority jobs, *e.g.*, the slowdown of priority-0 jobs compared to the case of no eviction is 3x higher than that of priority-6 jobs.

Considering this, it is of paramount importance to optimize the priority scheduler, especially when encountering big data workloads with strong inter- and intra-priority dynamicity [8], [9]. Priority schedulers typically separate jobs by their priority levels and keep them in separate queues. Then, they determine when and which job to process next and allocate slots to jobs.

Figure 2.1 depicts the schematics of a big data cluster with multiple slots serving high- and low-priority jobs in two queues. Jobs arrive with a number of parallel tasks, following MapReduce [10] programming paradigm and having time-varying arrival rates. Jobs with the same priority class stay in the same queue and are typically served in a first-come-first-served (FCFS) manner. Across priorities, jobs with a higher priority have precedence over any job with a lower priority. Upon arrival of a high priority job, the scheduler ensures that it is served quickly by either evicting any lower priority jobs currently being processed, or by letting the current lower priority jobs to finish and immediately start the incoming job. The former is called preemptive, whereas the latter is termed non-preemptive priority scheduling. As an example [6], the Google production systems employ preemptive priority scheduling, causing significant resource waste due to evictions. After being evicted, low-priority jobs return to the head of the queue and wait for new scheduling opportunities, *i.e.*, until no high-priority job is queued or processed. These evictions are completely avoided by differential approximation as it employs non-preemptive scheduling, with the consequent resource savings.

In the context of our work, a job can comprise multiple map and reduce tasks/stages following the MapReduce parallel programming paradigm to process data at scale. Spark [11] is a popular open source implementation of this paradigm with additional support for fast iterative computations and fault tolerance mechanisms. A typical MapReduce job processes input data and returns analysis results via parallel tasks executed in multiple map and reduce stages. The input data are organized as blocks stored in a file system, such as the Hadoop File System (HDFS) [12], which splits data across the servers in the cluster. During a map stage, map tasks are spawned to process one input block each. Intermediate results are stored as key-value pairs. Afterwards, reduce tasks access and aggregate the intermediate key-value pairs for the final result. Specific policies supported by cluster job scheduler exist to allow one or multiple concurrent jobs in the engine, respectively for FCFS and weighted fair sharing.

Another common strategy to burst the performance of high priority jobs is to temporarily accelerate the execution of a job. Computational sprinting allows for bursts of peak performance under a sprinting budget. The sprinting budget can stem from thermal [13], power [14] or provisioning [15] constraints. Several sprinting mechanisms exist, from modifying the CPU performance via dynamic voltage and frequency scaling (DVFS) [13] to tuning the job parallelism [16]. Each sprinting mechanism is controlled via a corresponding sprinting policy which determines what and when to sprint. Time-based policies levering timeouts to control the sprinting are rather common [17], [18].

Finally, when it comes to the state-of-the-art systems in this area, there is a large body of literature on scheduling, deployment and migration policies, mainly driven by the strong momentum on green computing [19]. However, here we focus on energy-related scheduling policies in a container/task-based deployment setting, leaving out research on optimization problems specifically geared toward reducing energy costs [20]. We particularly highlight those focused on managing priority, approximation frameworks and sprinting strategies for Spark-like applications which are the concepts later explored by our approach. Moreover, we summarize efforts on computational sprinting and modeling that address latency distribution for multi-priority jobs composed of multiple parallel tasks.

Green Scheduling
Wang *et al.* [21] proposes two polynomial-time algorithms, one for energy-aware heterogeneous data allocation and another for task ratio greedy algorithm. The algorithms schedule real-time constrained tasks of applications on heterogeneous multiprocessor systems using integer linear programming. Simulations compare these two algorithms against a greedy algorithm on two heterogeneous multiprocessor systems. GenPack [22] proposes an energy-saving mechanism inspired by the JVM's garbage collector. It migrates containers from young (unstable) to old (stable) generations of machines. However, GenPack ignores the user-demanded trade-offs of the jobs, and containers are migrated across the cluster only by observing the *stability* of the jobs. Partial Optimal Slacking (POS) [23] is an energy-efficient scheduling approach based on the concept of *task slacking* with the objective to lower the processing speed of a processor executing a task without affecting other tasks. POS achieves this by using DVFS techniques [24].

Priority systems
Characterization studies [5], [9], [25] on production big data systems show that multi-task jobs are associated with multiple priorities and exhibit diverse workload characteristics [9]. To ensure the performance of (particularly high) priority jobs, the scheduler evicts low-priority jobs to make resources available for high priority ones, resulting in a significant resource waste [25] and latency penalties on the low priority jobs [5]. Indeed, modern large jobs processing engines as Hadoop [26] and Spark [11] also support multi-priority job scheduling. For instance, Hadoop's fair scheduler [27] can assign different weights on different workloads to achieve soft priority, *i.e.*, higher weights on higher priority and low weights on low priority. Mesos [28] is a cluster manager that support priorities across and within multiple processing engines with a focus on fairness. Omega [8] is a two-level priority scheduler designed for large-scale system. Recognizing the need of evictions in priority systems, Natjam [29] develops novel job and task eviction policies for a scenario of two priorities that have different deadlines.

Approximation big data engines
To process vast and fast amount of data influx, novel approximation-enabled systems are designed to meet the dual objectives of accurate analysis and resource efficiency. In the context of MapReduce paradigm, statistical sampling theory is commonly applied to selectively process a subset of data either at the level of input block [30] or task [31], before or after the execution starts. BlinkDB [30], an approximate query processing framework, provides accuracy guards in short response times by leveraging statistical sampling theory to choose the inputs. ApproxHadoop [32] develops a two-stage sampling strategy for Hadoop [26] by either dropping the tasks or amount of data per tasks, so as to minimize the overhead of data accessing. To overcome the accuracy loss of sampling, IncApprox [31] combines the task sampling and incremental computing, *i.e.*, memorizing intermediate historical result. Grass [33] is a scheduler that prioritizes jobs with higher approximation level over lower levels for approximate analytics engine. While approximate big data engines effectively trade accuracy for the latency target and resource efficiency, they only consider single-priority scenarios and often overlook the latency models of complex dependency of jobs arrival.

Computational sprinting
Computational sprinting enables bursts of peak processing in systems limited by dark silicon (*i.e.*, fraction of cores on a chip that can be powered on due to thermal limits). Sprinting mechanisms exist nowadays at all system levels, from processors [13] to computer systems [34], and datacenters [35]. Sprinting policies decide when and what to sprint, *e.g.*, phases within job executions [36] and particular queries [37]. Several approaches have been explored in the single priority scenario, from simple heuristics [17], to queueing [38] and machine learning [18] models.

Stochastic models for multi-priority jobs
Modeling the latency for multi-priority jobs is a long standing challenge by itself because of the complex workload dynamics across jobs and the interdependency among tasks. For priority systems, modeling studies can be categorized into single *vs.* multi-server setups, preemptive *vs.* non-preemptive, and resume *vs.* non-resume under preemptive scheduling. Some systems [39], [40] employ matrix analytics methods to analyze the jobs average latency in a non-preemptive multi-server system, whereas others [41] focus on the state probabilities of preemptive multi-server systems. Horváth [42] derives the latency distribution in both preemptive and non-preemptive setting for single server system. Jelenkovic [43] derives the stability conditions for non-resume preemptive systems, highlighting the high risk of instability.

Later in Chapter 3 and Chapter 4 we propose Heats and DiAS, two approaches which also explore the concepts introduced in this section. Heats leverages software and hardware monitors to gather energy-related metrics and perform energy-aware scheduling and migration on single-tenant clusters. Frequency scaling is supported in Heats only during the modeling phase. Then, DiAS covers the multi-tenant scenarios by extending the sprinting policies with multi-priority scheduling and approximation schemes such that the performance of all priorities can be improved.

2.3 Security

On top of the considerations we discussed until now, a major concern for all types of applications and management systems is security. This is a concern for cloud providers, especially in the multi-tenant scenario where resources are shared among several users. Moreover, applications such as deep learning often rely on user sensitive data and, therefore, preserving relevant user information is crucial. Considering this, next we introduce the general concept of hardware assisted mechanisms and the benefits of using them to ensure security. Then, we focus on Intel SGX, the hardware-assisted protection mechanism later used by our proposed solution. Finally, we introduce the state-of-the-art systems related to this problem and compare them with the approach we propose later in Chapter 8.

Sensitive data processing occurs more and more often on machines or devices out of user control. One well known example is the Internet of Things (IoT), where data security could be considered as being at risk, independently of the deployment used, either using edge or cloud services. In these systems, different categories of attacks exist such as physical bus sniffing [44] and cache side-channel [45]. Considering this, software-based countermeasures have been proposed. However, the severity and complexity of these attacks require a level of security that only the hardware support can ensure.

In the last years, major companies released a number of architectural extensions providing hardware-assisted security to software. These solutions provide security features such as isolated execution, integrity of applications executing within the trusted execution environment (TEE), and confidentiality of their assets. Some instances of hardware technologies used to support TEE implementations are, for instance, AMD Platform Security Processor (PSP) [46], Arm TrustZone [47], and Intel Software Guard Extensions (SGX) [48] which we detail next.

Intel Software Guard Extensions (SGX) is a set of security-related instruction codes that are built into modern Intel-based processors. With SGX, the root of protections against attacks is a series of memory access checks which prevents the currently running software from accessing memory that does not belong to it. A secured memory area, *i.e.*, Enclave Page Cache (EPC), is used by the processor to store enclave pages when they are a part of an executing enclave. From the application point of view, its entire code can be encapsulated by a single enclave, or it can be decomposed into smaller components, such that only security-critical components are placed into an enclave. The latter option is more common, considering that the limited memory used by SGX was initially limited to 128 MB, of which only 93.5 MB are used by user applications. More recent versions of SGX extend this limitation to 512 GB.

Multiple SGX threads can execute simultaneously inside an enclave. However, SGX threads cannot be created or destroyed on the fly. Instead, control pages (*i.e.*, thread control structures—TCSs) must be allocated and initialized beforehand. These cannot be changed throughout the enclave's lifecycle. Moreover, the number of concurrent SGX threads is limited to the number of logical CPUs available on the machine. This constraint exists because a running SGX thread is mapped to one of the logical CPUs, to which it is

Table 2.1: State-of-the-art systems related to OmpSs-SGX.

System	Security	Parallelism	Task-based	Asynchronous	Data Dependency	Heterogeneity	Legacy Systems Support
Haven [49]	✓	✓	✗	✓	✗	✗	✓
SCONE [52]	✓	✓	✗	✓	✗	✗	✓
Panoply [54]	✓	✓	✗	✓	✗	✗	✗
Graphene-SGX [53]	✓	✓	✗	✗	✗	✗	✗
Occlum [50]	✓	✓	✗	✗	✗	✗	✓
Occlumency [55]	✓	✓	✗	✗	✗	✗	✗
SGX-OmpSs	✓	✓	✓	✓	✓	✓	✓

fully dedicated. Next, we give an overview of existing systems on secure, performance and energy efficient multitasking, with focus on systems relying on Intel SGX for security.

Haven [49] implements shielded execution of unmodified Windows applications by leveraging Intel SGX. It implements a form of user-level scheduling where a fixed number of threads is created according to the desired level of parallelism. These operate as virtual CPUs supporting an arbitrary number of application threads inside the enclave.

Occlum [50] is a system that enables multitasking in a LibOS for Intel SGX. Authors implement the LibOS processes as SFI-Isolated Processes (SIPs). Software-based fault isolation (SFI) [51] is a software instrumentation technique for sandboxing untrusted modules. Occlum designs a SFI scheme named MPX-based, called Multi-Domain SFI, leveraging it to enforce the isolation of SIPs.

SCONE [52] is a secure container mechanism for Docker that uses the SGX trusted execution support of Intel CPUs to protect container processes from outside attacks. Similarly to Occlum, it provides a user-level threading implementation but combines this with an asynchronous system call mechanism in which OS threads outside the enclave execute system calls, thus avoiding the need for enclave threads to exit the enclave.

Graphene-SGX is an extension of Graphene, a lightweight library OS, designed to run a single application with minimal host requirements. Graphene-SGX [53] ports Graphene to Intel SGX as well and provides other improvements to make the security benefits of SGX more usable such as integrity support for dynamically-loaded libraries and secure multi-process support. Regarding threading, it uses a 1:1 model where users specify how many threads the application needs.

Panoply [54] provides an abstraction called a micro-container (*i.e.*, microns) which is a unit of code and data isolated in SGX enclaves. It gives the abstraction of an arbitrary number of threads dynamically created by calling the standard p-thread API. Different than SCONE which uses a M:N threading model, this on-demand threading model spawns new microns in separate enclaves to scale the number of threads. This way, each thread in the application is associated with a unique thread in the enclave which avoids multiplexing on a limited number of existing threads in a single enclave but introducing an overhead to create dedicated enclaves for each thread.

More than security, there are systems which additionally focus on energy-efficiency. An instance of such systems is Occlumency [55], a cloud-driven solution specifically designed for performing deep learning inferences. Occlumency also relies on Intel SGX to provide security, but tries to improve performance by storing model weights outside of enclaves and by implementing a pipeline parallelism approach. Differently than the approach we later propose (*i.e.*, SGX-OmpSs), it does not rely on any task-based asynchronous parallelization within the code running inside the enclave.

Table 2.1 summarizes the characteristics of each system compared to SGX-OmpSs, the approach we propose later in Chapter 8. SCONE, Panoply and Graphene-SGX support multitasking with Enclave-Isolated Processes (EIPs) but suffer from performance and usability issues. Haven and Occlum also support multitasking in a single-address-space architecture but no asynchronous parallelism like in OmpSs-SGX. Moreover, none of these systems support hardware heterogeneity or follows a data dependency approach for task parallelization.

2.4 Deep Learning Parameter Tuning

In this section we discuss how a parameter tuning job operates and describe the different types of workloads, tuning parameters, and tuning budgets typically considered by existing systems. We show, by means of a motivating example in the domain of image classification, how tuning server providers (*e.g.*, Google Vizier [56] and Amazon SageMaker [57]) and the final users can both benefit from the tuning phase if the process considers different types of parameter. Then, we discuss the state-of-the-art literature for the areas which we later explore in our use case applications and compare their main characteristics.

A tuning deep learning job takes as input a given workload, a set of parameters to be tuned, and outputs the set of optimal parameters' values found together with the model which trained these values. Throughout this work, we refer to workload as a tuple pairing a model and dataset. Typically, DNN workloads are used for training (*i.e.*, learning) or inference (*i.e.*, prediction). Training is the process of finding the model parameter's based on a given algorithm and objective. Once a model is trained, the inference phase deploys the model to perform prediction of data with the same structure used in the training but with unseen values. Here we explore both the training and the inference phase of deep learning workloads.

Before the training phase starts, there are several other parameters that must be configured (*e.g.*, hyperparameters and system parameters) and that have high impact on model performance both in terms of accuracy and runtime. Here we assume that the training phase includes parameter tuning on top of learning the weights of the model. The process of tuning these parameters is typically done by auto-tuning tools which rely on training trials to explore the search space of possible parameter configurations.

Tuning a given workload consists of multiple training trials, each divided into epochs. As the training trials we consider follow the stochastic gradient descent algorithm [58], each epoch involves one forward and one backward pass on the entire input dataset. Moreover, for ease of processing, the dataset is split into smaller batches, each one being propagated forward and backward once during an epoch (*i.e.*, iteration). At the end, the winning trial (*i.e.*, parameters leaving to best solution) is output to the user.

Once the model training is completed, it can be used for evaluation on an unseen chunk of the training data which is put aside at the beginning of training for this purpose (*i.e.*, 20% of the full dataset in our case). Finally, if the model training and evaluation performance are satisfied then the user deploys it to be used on production with unknown data from real applications. In this phase, the user still has to tune the parameters referent to inference before the final model is deployed. These parameters involve batch size for multi-image inference as well as system configurations.

In multi-tenant scenarios, it is common that the same model is trained on different datasets, as well as that different models are trained using the same dataset. We later explore this practice (depicted in Figure 2.2) to propose a warm start of applications parameters based on model or database similarities. This idea works under the assumption that the similarity between workloads is also reflected on their set of optimal parameters.

Hyperparameters

A hyperparameter is a configuration external to the model. Its value cannot be estimated from data, it is set before the training starts, and does not change afterwards. Choosing the right hyperparameters

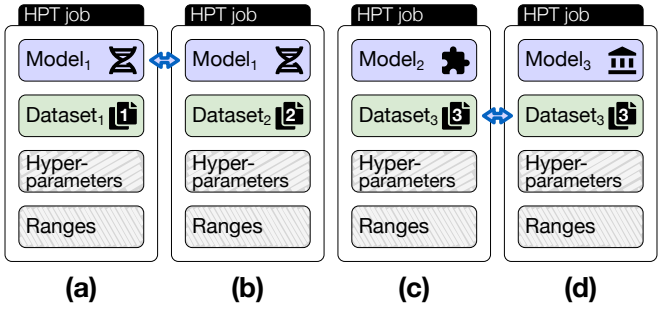
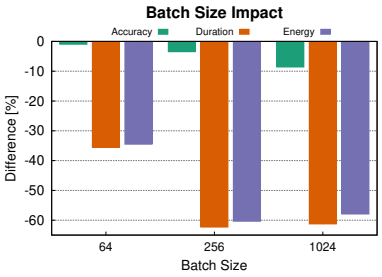
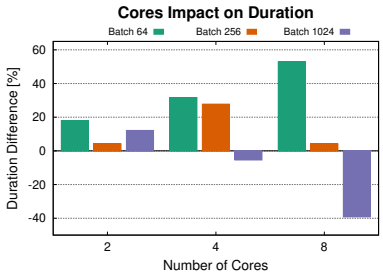


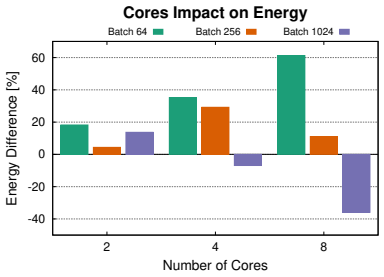
Figure 2.2: Workloads similarity of hyperparameter tuning jobs on the model and dataset level. Jobs (a) and (b) consist of the same model. Jobs (c) and (d) consist of the same dataset.



(a) Batch size impact on accuracy, runtime and energy. Baseline: batch size = 32.



(b) Cores impact on runtime per batch size. Baseline: sequential (*i.e.*, cores = 1).



(c) Cores impact on energy per batch size. Baseline: sequential (*i.e.*, cores = 1).

Figure 2.3: Impact of hyper and system parameters on accuracy, runtime and energy training of LeNet and MNIST workload.

during the *tuning* phase is key, as the output accuracy of the trained models can vary significantly. This phase is typically based on trial-and-error with model selection criteria. The complexity of this phase sparked several research efforts towards its automation and autotune frameworks [56], [59], [60]. As a result, hyperparameter optimization outputs a tuple of hyperparameters that yields an optimal model which minimizes a predefined loss function on given independent data [61].

The selection criterion typically considered is model accuracy. However, the hyperparameters values will impact model accuracy, its training time and the energy footprint. The first is typically related to the utility of the trained model, the latter two to its costs. Figure 2.3a shows this behavior by reporting the impact of varying one hyperparameter (*i.e.*, batch size) for the training of a LeNet model [62] on the MNIST [63] dataset. On the y-axis we show the measured differences for accuracy, duration and energy observed for 3 possible batch size values (*i.e.*, 64, 256, and 1024), against a default value of 32.

We can observe how larger values of batch size achieve worse accuracy, but shorter training time and lower energy footprints. However, these observed trends might present considerable variations for different applications as it strongly depends on the workload and the values of the other hyperparameters. Therefore, these trade-offs are not trivially predicted, making it challenging to handle multidimensional selection criteria.

System Parameters

We define system parameters as being the configurable resources of the underlying computing infrastructure where the training will execute (*e.g.*, memory, CPU cores, CPU frequency). Typically, the hyperparameter optimization fixes the same system parameters for each trial, although they might benefit from different configurations. To highlight this, we train again a LeNet model on the MNIST dataset. We vary the number of CPU cores used with different batch sizes. Figure 2.3b and Figure 2.3c depict our findings.

We observe how the increasing number of cores is beneficial for larger batch size values (*e.g.*, 1024), but not for smaller ones. In fact, for smaller values (*e.g.*, 64) the runtime increases as the number of cores increases. This behavior is explained by the synchronous mini-batch stochastic gradient descent (SGD) algorithm used to train the neural network model. Each N iterations, SGD first computes the gradients using the current mini-batch, and then makes a single update to the weights of the neural network model.

The *batch size* hyperparameter is divided by N to form these mini-batches, where N is the number of cores. When this value is too small, the overhead of model parameters synchronization is too high and ends up slowing down the training itself. This overhead can be amortized by using techniques such as the ones implemented by Drizzle [64] which schedules multiple iterations of computations at once, greatly reducing scheduling overheads even if there are a large number of tasks in each iteration [65].

Regarding the energy observations, we estimate the overall energy consumption of the cluster by calculating the trapezoidal integral of the power values collected every second during training. We observe a clear correlation between training runtime's gains (Figure 2.3b) and the energy variations (Figure 2.3c).

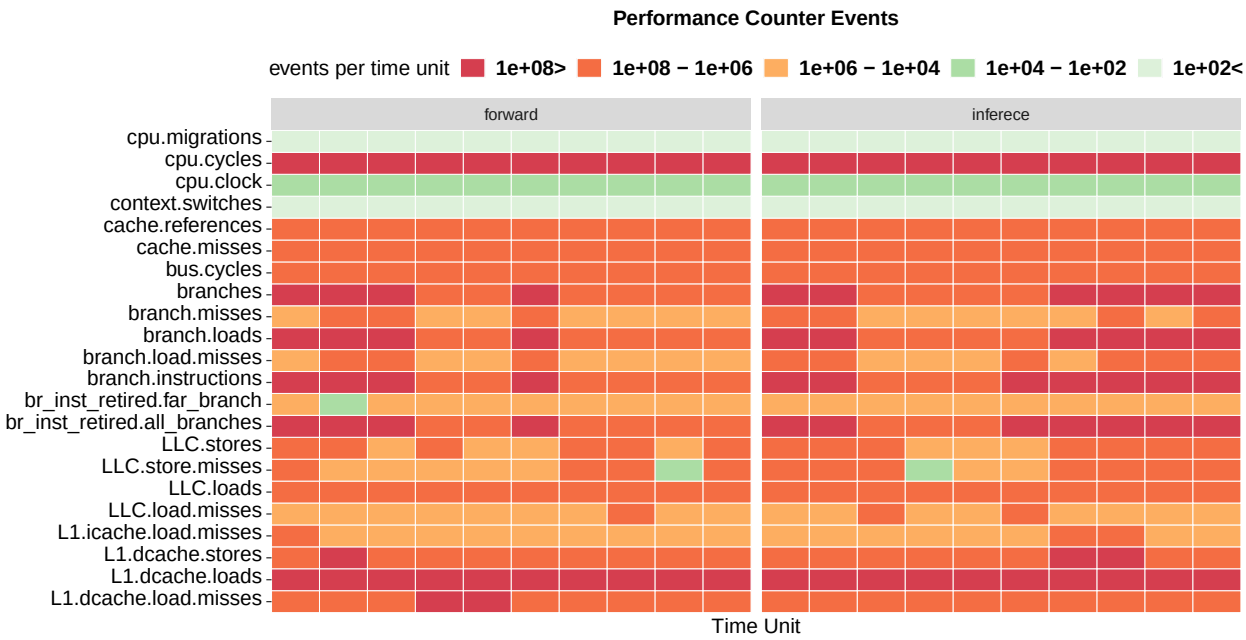


Figure 2.4: Performance counter events collected during the forward phase of training and inference.

These observations might however vary when the tuning is applied to different set of system parameters, *e.g.*, CPU frequency, or for different workloads.

In summary, these preliminary results show the delicate trade-offs between hyper and system parameters. One needs to balance them all towards optimal values, such that the underlying system achieves the best training performance without compromising the model accuracy.

Inference Parameters
When it comes to inference, there are also some parameters that one has to consider before deploying a model. First, in a multi-sample inference scenario, the batch size also plays a significant role. In this case, the batch size refers to how many samples should be aggregated before calling the deployed model. Most importantly, considering the performance requirements usually related to this phase, properly tuning the system parameters becomes even more critical.

In order to consider the inference during the process of tuning parameters, one could consider the following design options:

1. rely on the forward phase of training;
2. offload the model to inference device and collects; and
3. simulate the inference devices in the tuning server.

In theory, the inference is equivalent to the forward phase of training with the only difference being on the model weights. However, in practice this is not the case as the memory utilization during training is much higher than for the inference. In fact, while training, the weights are constantly being updated and therefore are kept in memory for performance reasons. Instead, during the inference phase the optimal weights are known and only consist of constant values. Finally, training is performed on relatively large batches of images, whereas inference is often done on single images.

To show that relying on the forward phase to make predictions about the inference phase is not the best approach, we profile the different phases using hardware performance counters [66]. These experiments were repeated in *(1)* an ARMv7 Processor rev 4 (v7l) with 4 cores and 4GB memory, *(2)* a Raspberry Pi 3 Model B Plus Rev 1.3 also with 4 cores but 1 GB memory, and *(3)* an Intel(R) Core(TM) i7-7567U CPU with 16GB. Figure 2.4 shows CPU and memory related events for the AlexNet [67] model with CIFAR10 [68] dataset during the forward phase of training and inference performed with the trained model. By applying

clustering techniques, we observe how CPU-related events are consistent between the forward phase and inference. However, memory-related events do not manifest similar patterns.

Hence, one is left with the options of offloading the model to actual edge devices, or simulating such devices in the tuning server. The first approach gives more precise results, but brings an overhead due the required data transferring, and limitations on the number of devices which we can try. Considering this, the approaches we detail later chooses the approach based on simulation of the edge devices for inference. Doing so, we can quickly try a large search space without adding an overhead as the combination of optimizations proposed in our approach actually reduces tuning time. Moreover, the error of the simulation results on inference with respect to the actual measurement in edge devices is really small (at most 20% according to our experiments).

Tuning Budget

As already mentioned, the general process of tuning parameters consists of evaluating different combinations of values and pick the most effective one. To optimize this otherwise costly search process, techniques such as early-stop [69] and multi-fidelity [70] budgets can greatly reduce the evaluation costs. With multi-fidelity budget, the promising range of values is identified using a model approximation which is cheap to evaluate by definition. Typical model approximations include a subset of the training data or a smaller number of epochs. This technique is able to explore a larger range of values combination, early identify and stop unpromising configurations, and, therefore, frugally perform tuning.

As explained earlier, a training trial consists of a certain number of epochs runs applied on a given dataset. These runs are an exploration of the search space and the chosen set of optimal configuration are only those from the winning trial. Hence, the majority of trials do not lead to useful results, and it is therefore highly desirable to quickly identify non-promising candidates as soon as possible.

By equally applying the same number of epochs on the full dataset to all the trials, one wastes a significant amount of time and resources on non-promising configurations. Considering this, trials typically run on a given budget, defined in terms of

1. number of epochs;
2. portion of training dataset; or
3. duration.

The tuning algorithm defines a minimum and a maximum budget used to set the bounds of resources allowed for tuning, and a reduction factor which determines the fraction of configuration that continues in the next iteration. For instance, consider the number of epochs as example. Assume the minimum number of epochs is 1, the maximum is 16, and the reduction factor is 2. We start tuning with 16 trials initialized on the minimal budget (*i.e.*, 1 epoch). The next iteration will consist of 8 trials with 2 epochs, then 4 trials with 4 epochs, 2 trial with 8 epochs and a final iteration containing only one trial with 16 epochs. Similar reasoning applies for other budgetable parameters, *i.e.*, dataset fraction or time.

When it comes to related work, there is a large body of work behind machine learning in general, and parameter tuning more specifically. We survey the most prominent ones in Table 2.2. We distinguish between systems that support CPU or GPU processing nodes, if they support hyper, system, or architecture parameters, if their objective focus is on tuning, training, or inference, and whether they exploit the multi-image inference aspect. Table 2.2 also contains PipeTune and EdgeTune, the two systems we propose as use cases later in Chapter 6 and Chapter 7, respectively. Next, we discuss these related systems in more detail.

Parameters Tuning

In the context of this thesis, tuning is the process of choosing optimal parameters for a given workload (*i.e.*, model and workload). The process of tuning parameters is crucial to find the best model performance of a given application. However, the best model performance can significantly vary depending on the performance definition, typically based on the interests of users. Moreover, the tuning process itself can take different perspectives into account such as searching algorithms [88]–[90], supported tools [91], and type of processing nodes (*e.g.*, GPUs, CPUs, FPGAs). Therefore, there are many proposed approaches and tools addressing this problem.

Table 2.2: State-of-the-art systems related to hyper and system parameter tuning.

System	CPU	GPU	Parameter			Objective			Multi-Sample Inference
			Hyper	System	Architecture	Tuning	Training	Inference	
Astra [71]	✗	✓	✓	✗	✗	✓	✓	✗	✗
AutoKeras [59]	✓	✓	✓	✓	✗	✓	✓	✗	✗
ByteScheduler [72]	✓	✓	✗	✓	✗	✓	✓	✗	✗
ChamNet [73]	✓	✓	✗	✗	✓	✓	✓	✓	✗
DPP-Net [74]	✓	✓	✗	✗	✓	✓	✓	✓	✗
FBNet [75]	✓	✓	✗	✗	✓	✓	✓	✓	✗
GRNN [76]	✓	✓	✗	✗	✗	✗	✓	✗	✗
Hop [77]	✓	✗	✗	✗	✗	✗	✓	✗	✗
HyperDrive [78]	✓	✓	✓	✗	✗	✓	✓	✗	✗
HyperPower [79]	✗	✓	✗	✗	✗	✗	✓	✗	✗
MnasNet [80]	✓	✗	✗	✗	✓	✓	✓	✓	✗
NeuralPower [81]	✗	✓	✗	✗	✓	✓	✓	✗	✗
Optimus [76]	✓	✓	✗	✗	✗	✗	✓	✗	✗
Orion [82]	✓	✗	✗	✗	✗	✗	✓	✗	✗
ProxylessNAS [83]	✓	✓	✗	✗	✓	✓	✓	✓	✗
Parallax [84]	✓	✓	✗	✗	✗	✗	✓	✗	✗
PipeDream [85]	✗	✓	✗	✗	✗	✗	✓	✗	✗
SageMaker [57]	✓	✓	✓	✗	✗	✓	✓	✗	✓
STRADS [86]	✓	✗	✗	✗	✗	✗	✓	✗	✗
STRADS-AP [87]	✓	✗	✓	✗	✗	✓	✓	✗	✗
Tune [60]	✓	✓	✓	✗	✗	✓	✓	✗	✗
Vizier [56]	✓	✓	✓	✗	✗	✗	✓	✗	✗
PipeTune	✓	✗	✓	✓	✓	✓	✓	✗	✗
EdgeTune	✓	✓	✓	✓	✓	✓	✓	✓	✓

Improving Training

GRNN [92] constructs a hybrid performance model that estimates the cost of a configuration according to the communication and computation needs. It ranks all the configurations and selects the first top-K to compile and run returning the fastest among them. Hop [77] is a heterogeneity-aware decentralized training protocol. It relies on a queue-based synchronization mechanism that can implement backup workers and bounded staleness in a decentralized setting. Optimus [76] uses online fitting to predict model convergence during training, and sets up performance models to estimate training speed as a function of allocated resources in each job. It estimates online how many more training epochs a job needs to run for convergence and how much time a job needs to complete one training epoch given a certain amount of resources. The speed model is computed based on a small sample set of training data, with possible combinations of parameter servers and workers.

Moreover, Orion [82] performs static dependence analysis to determine when dependence-preserving parallelization is effective and map a loop computation to an optimized distributed computation schedule. It automatically parallelizes serial imperative ML programs on distributed shared memory. Parallax [84] combines Hyperparameter Server [93] and AllReduce [94] architectures to optimize the amount of data transfers according to the data sparsity. It splits between a static phase for graphs with dense variables, and a sampling phase for fewer iterations. Finally, PipeDream [85] combines inter-batch pipelining and intra-batch parallelism to improve parallel training throughput, helping to better overlap computation with communication and reduce when possible the amount of communication.

Hyperparameter Tuning

As the process of tuning hyperparameters is, in most cases, crucial to find the best model performance of a given application, there are many proposed approaches and tools addressing this problem. Astra [71] is a framework for online fine-grained exploration of the optimization state space in a work-conserving manner while making progress on the training trials. STRADS [86] exposes parameter schedules and parameter updates as separate functions to be implemented by the user. A parameter schedule identifies a subset of parameters which a given worker should sequentially work on. STRADS-AP [87] extends STRADS to a distributed ML setting. These approaches leverage a runtime and API comprised of Distributed Data Structures (DDSs) and parallel loop operators. Moreover, AutoKeras [59] enables Bayesian optimization to guide the network morphism for efficient neural network architecture search. The framework develops a neural network kernel and a tree-structured acquisition function optimization algorithm to efficiently

explore the search space. Similarly, Tune [60] provides a narrow-waist interface between training and search algorithms.

Finally, we mention two auto-tuning tools used in industry. HyperDrive [78] is a package part of Azure Machine Learning which supports hyperparameter tuning. It follows POP's scheduling algorithm which combines probabilistic model-based classification with dynamic scheduling and early stop techniques. Amazon SageMaker [57] is a fully managed machine learning service. It supports automatic model tuning component that finds the best version of a model by running many training trials on the dataset using the algorithm and ranges of hyperparameters specified by the user. Google Vizier [56] is an internal service used to optimize machine learning models and other systems. It also provides core capabilities to Google's Cloud Machine Learning HyperTune subsystem.

System Parameter Tuning
ByteScheduler [72] is a Bayesian optimization approach. It specifically focuses on auto-tune tensor credit and partition size for different training models under various networking conditions. ByteScheduler uses auto-tune algorithms to find the optimal system related configurations. AutoKeras [59] supports a form of system parameter tuning, by means of an adaptive search strategy for different GPU memory limits.

Inference-Aware Tuning
In the process of tuning, the users define their objective function which is often to maximize model performance in terms of accuracy [95], [96]. However, some users are also interested in model performance from other perspective such as the inference throughput. In this case, the tuning servers must be inference-aware, meaning that either empirical or estimated measurements of the inference runtime is included in the objective optimization function under consideration.

Multi-Objective Tuning
Multi-Parameter Tuning consists of tuning servers which simultaneously consider more than one dimension in their objective optimization functions [97], [98]. For instance, when both the model accuracy and inference throughput are equally relevant, then the optimization must follow a multiple criteria decision making strategy [99]. Sometimes these objectives are conflicting, which makes the decision making process not trivial and leading to multiple possible Pareto optimal solutions [100].

Neural Architecture Search (NAS)
Neural architecture search (NAS) [101] is a technique for automating the design of artificial neural networks (ANN), a widely used model in the field of machine learning. NAS has been used to design networks that are on par or outperform hand-designed architectures. Methods for NAS can be categorized according to the search space [102], search strategy [103], [104] and performance estimation strategy used [105].

Multi-Image Inference
When the model has all parameters tuned, it is deployed to be used for inference on unseen data. Data arrives in batch or streaming form [106]. Although the most common approach for the inference phase is to apply the model for one data point at a time, this phase considers the hyperparameter *batch size* which determines how many data points can be aggregated before performing model inference. When the inference batch size is set to a value higher than one, then multi-image inference is being performed.

2.5 LEGaTO Project

A considerable part of this work was done in the context of the project Low Energy Toolset for Heterogeneous Computing (LEGaTO) [107], whose mission was to develop a software toolchain for heterogeneous hardware with energy efficiency as the main focus. Besides of that, the project also considers fault-tolerance, security and programmability. Led by Barcelona Super computing Center (BSC), LEGaTO consisted of 10 partners, including universities, industries, and research institutes (viz., Chalmers Tekniska Hoegskola AB (CHALMERS), Christmann Informationstechnik + Medien GmbH & Co. KG (CHR), Machine Intelligence Sweden AB (MIS), Helmholtz Centre for Infection Research (HZI), Technische Universitaet Dresden (TUD), Maxeler Technologies (MAXELER), Israel Institute of Technology (TECHNION), Bielefeld University (UNIBI), and University of Neuchatel (UNINE)).

Figure 2.5 shows an overall view of the various components used in the project, including the use cases (*e.g.*, smart home, smart city), programming model, compiler and high level synthesis languages, runtime,

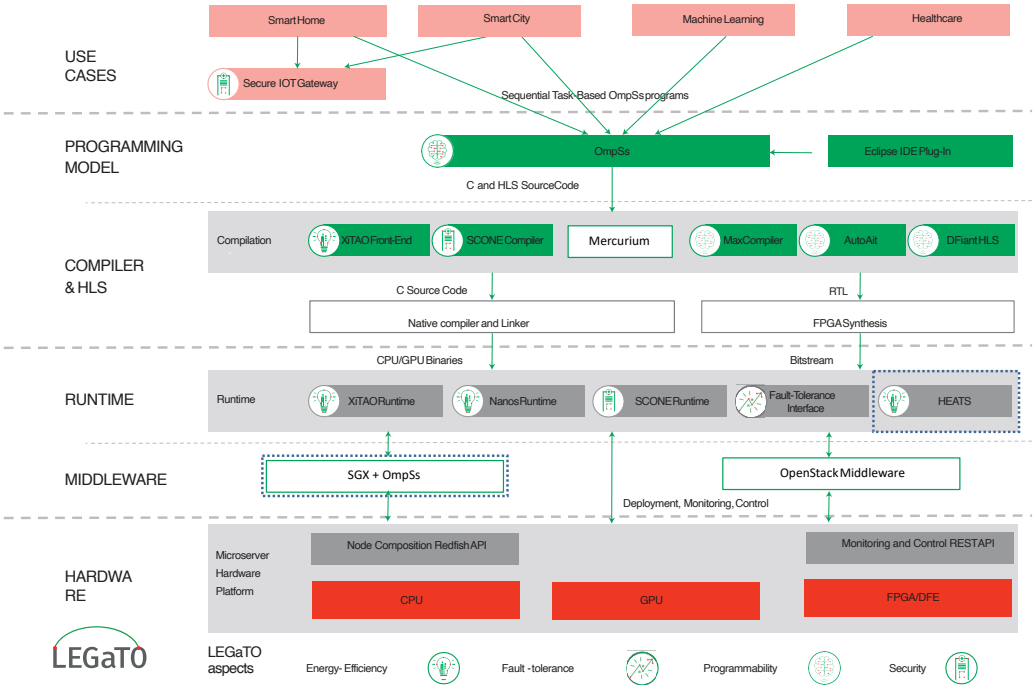


Figure 2.5: LEGaTO platform architecture.

middleware and hardware. In the runtime level we contributed with Heats introduced in Chapter 3 and its follow-up approach DiAS introduced in the Chapter 4. Moreover, the two use cases presented in Chapter 6 and Chapter 7 are part of the machine learning use cases. Finally, the security approach introduced in Chapter 8 is part of the programming model and compiler levels of this architecture.

Besides of the work presented here, the LEGaTO project produced a number of relevant scientific contributions. TZ4Fabric [108], for instance, is an extension of Hyperledger Fabric to leverage ARM TrustZone for the secure execution of smart contracts. Hermes [109] is an application-level protocol for the data-plane that can operate using generalized deduplication as well as classic deduplication. SpecFuzz [110] is the first tool that enables dynamic testing for speculative execution vulnerabilities (*e.g.*, Spectre). More information regarding the project, including deliverables, use case details, related work with list of publications can be found in the website [111].

2.6 Summary

In this chapter, we presented the main concepts of distributed applications and clusters which we use throughout this thesis. Special emphasis was given to the energy efficiency, security and heterogeneity aspects which we later leverage in our proposed approaches.

To contextualize the types of applications we focus in this work, we detailed the characteristics of distributed applications and introduced the differences between stateful and stateless applications as well as synchronous and asynchronous parallelism. Then, we presented the perspective of clusters which host such applications and the challenges laying behind doing so.

We discussed the security dimension of the problems we consider here and introduced the concept of TEEs to ensure security both for applications and clusters. More precisely, we have given focus to Intel SGX which is the TEE technology we further explore in one of our proposed approaches.

As we use the parameter tuning of learning applications as use cases of our work, we also introduced the main concepts of this area as well as a thorough analysis of the related state-of-the-art systems available. Finally, we have given a brief overview about the LEGaTO project to show part of our work motivations and achievements.

Chapter 3

Cloud Providers: Single-Tenant

In this chapter we start tackling the problem of making distributed applications more energy efficient from the perspective of cloud providers. To this end, we introduce Heats, a task-oriented and energy-aware orchestrator for containerized applications targeting heterogeneous single-tenant clusters. First, our system learns the performance and energy features of the physical hosts. Then, it monitors the execution of tasks on the hosts and opportunistically migrates them onto different cluster nodes to match the customer-required deployment trade-offs.

3.1 Introduction

Customers of cloud providers have access to a wide range of heterogeneous resources. The diversity of resources encourages application developers and deployers to program for, and offload even more workloads to the cloud. There, specialized hardware (*e.g.*, GPU, FPGA) can be rented for limited time, reducing upfront costs and allowing for better scalability. Moreover, applications designed to run distributed often consist of different building blocks each with its own requirements. These requirements vary from resources specification up to minimum latency or throughput. Applications with these characteristics can take advantage of a diverse infrastructure to frugally meet their goals.

To illustrate this diversity, Table 3.1 shows an overview of the commercial offering of heterogeneous resources at 6 major public cloud providers. For each, we list the CPU architecture (x86, IBM Power, ARM), and the availability of GPU, FPGA or ASIC units. We further indicate if such resources can be accessed using bare metal (BM) or virtual machine (VM) instances. Additionally, we show whether the operating frequency of the processor can be dynamically scaled up or down, a feature that could be leveraged to reduce the generated energy costs of a node. This quick survey reveals that it is possible to combine a large and heterogeneous ensemble of machines, each offering specific hardware feature sets. This capability represents the ideal case for applications that have different resource demands, as it is sometimes better to migrate the execution from a machine of one kind to a different one, in order to more closely match the expected trade-off between energy consumption and performance requested by the customer. Resource diversity can also be exploited to deploy applications and workloads of different nature.

Table 3.1: Heterogenous resources available at public cloud providers via bare metal (BM) or virtual machines (VM). * = frequency scaling enabled. ✓= feature available from VM or BM. ✗= not available.

Provider	**x86-64**		**POWER**		**ARM**		**GPU**		**FPGA**		**ASIC**	
	BM	**VM**	**BM**	**VM**	**BM**	**VM**	**BM**	**VM**	**BM**	**VM**	**BM**	**VM**
Amazon [112]	✓*	✓	✗	✗	✗	✗	✗	✓	✗	✓	✗	✗
Microsoft [113]	✗	✓	✗	✗	✗	✗	✗	✓	✗	✓	✗	✗
Google [114]	✗	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓
IBM [115]	✓*	✗	✓*	✗	✗	✗	✓	✗	✗	✗	✗	✗
Oracle [116]	✓*	✓*	✗	✗	✗	✗	✓	✓	✗	✗	✗	✗
Scaleway [117]	✓*	✗	✗	✗	✓*	✗	✗	✗	✗	✗	✗	✗

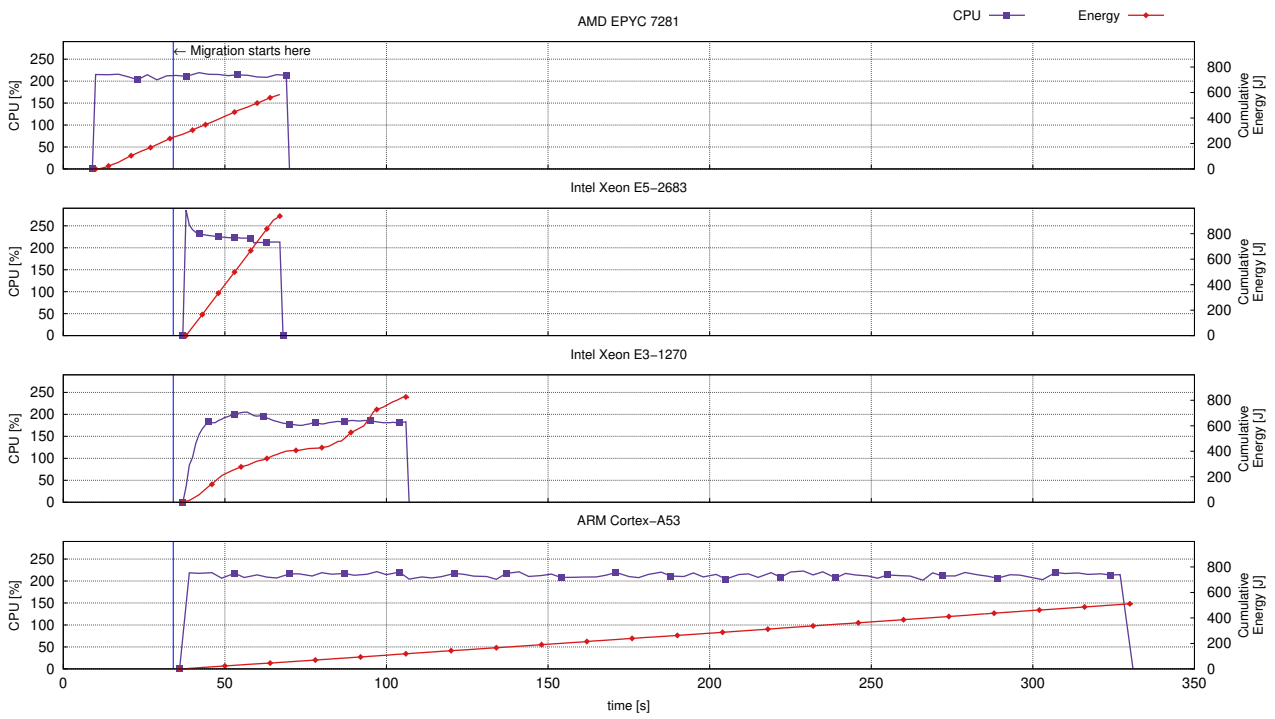


Figure 3.1: Migrating a task to a different host allows for energy savings but increased run time.

Containers (*e.g.*, Docker [118]) have recently become the de facto standard to deploy applications on the cloud, executed by specialized container orchestrators, such as Kubernetes [119]. Current policies of container orchestrators often ignore the diversity found in hardware, leading to subtle trade-offs between energy and performance. To better understand this aspect and motivate our work, we conducted a simple experimental study with results depicted in Figure 3.1.

We set up an on-premise cluster composed of 4 different types of nodes: three server-grade machines (two Intel and one AMD) and one ARM-based low-energy device (a RaspberryPi). Each machine has different hardware characteristics (*e.g.*, number and type of CPU cores, memory and operating frequency) and energy requirements. While these properties are known by the cluster owner (*e.g.*, cloud providers) at deployment time, neither the cluster nor the application owners are aware of the energy requirements and the raw computing power of the machines for a specific workload. Typically, customers are only able to evaluate those at runtime, while executing their applications. Because of that, they can face unexpected costs or missed deadlines upon completion of tasks.

In the scenario of our motivating example, we developed a containerized version of the popular *k-means* clustering algorithm [120] and deploy it in our on-premise cluster. At first, the task is deployed on the AMD node depicted in the top-most plot of Figure 3.1. Given our cluster settings, with the default Kubernetes scheduler, we observe the deployment on the machine with more cores and memory. When remaining in the same host, the task completes after 69 seconds, consuming 565 Joules.

Next, we consider customers wishing to compromise the running time for energy costs. This requires a dynamic container rescheduling policy that can migrate a task into the ARM node after it has made some progress but before completion (e.g., 30 seconds after startup, as highlighted by the vertical line in each plot). In doing so, the net energy savings are significant (up to 34%) but at the cost of a $5.4\times$ increase of the task's running time.

In the case of this specific classification application, for instance, the runtime is not critical. This is the case because we are focusing on the training phase of such algorithm which typically happens independent of the applications using the trained model for inference. In the later case the runtime performance of the task becomes critical but in the first case scenario trading the performance regression by the observed energy saving would make sense depending of the customer's requirements.

Algorithm 2 Scheduling of tasks.

```
1: function Schedule
2:     while pendingTasks ≠ ∅ do
3:         t = pendingTasks.poll()
4:         bestFit ← BestFit(t, e_w, p_w)
5:         Assign(t, bestFit)
```

Algorithm 3 Rescheduling of tasks.

```
1: function Reschedule
2:     for t ∈ runningTasks do
3:         bestFit ← BestFit(t, e_w, p_w)
4:         if bestFit ≠ currentHost then
5:             Migrate(t, currentHost, bestFit)
```

Algorithm 4 Find best fitting node for a task.

```
1: function BestFit(t, e_w, p_w)
2:     r ← RequiredResources(t)
3:     n ← AvailableNodes(r)
4:     scores ← ∅
5:     n_e, n_p ← Predict(nodes, r)
6:     for n ∈ nodes do
7:         n_s ← e_w(1 − n_e/max_e) + p_w(n_p/max_p)
8:         scores.add({n, n_s})
9:     return n | {n, s} ∈ scores ∧ s = max_s
```

Such trade-offs are often desirable (especially for deadline-free, low-priority workloads), but difficult to achieve in practice. As a matter of fact, a task (or container) orchestrator would need to be aware of several factors and able to:

1. know or learn the characteristics of the underlying cluster and its hardware resources;
2. understand the trade-off that a customer is willing to accept;
3. identify when a better placement opportunity exists for the currently executing tasks; and
4. migrate the task accordingly.

Therefore, in this chapter we introduce Heats, a scheduling system geared toward heterogeneous clusters that achieves these goals for the single-tenant scenario. The key mechanism used by Heats consists in offering to customers the ability of indicating, at deployment time, their intended energy-performance ratio (the acceptable *trade-off*), in the form of an H value. Thereafter, Heats continuously matches the demanded H value to the available resources, considering the resources themselves, pre-built performance and energy models, and the possibly conflicting requirements from other concurrent tasks. As shown in Section 3.4, this has consequences on the task throughput of the underlying cluster.

The rest of this chapter is organized as follows. The rational of the Heats scheduling policy is presented in Section 3.2. We describe the architecture of Heats in Section 3.3. We extensively evaluate the performance of our prototype in Section 3.4, where we also detail the synthetic traces used to show the benefits of Heats before concluding in Section 3.5.

3.2 Heats Scheduling Policy

First, the resource requirements of a task, as for instance memory or number of cores, are specified before submission. Resource availability in the hardware nodes is monitored (in our practical experiment we rely on Heapster [121]) and reported to Heats monitoring module. Then, Heats computes suitable nodes for execution considering the resource requirements for all previously running tasks as well as the availability reported by the underlying system. Next, the algorithm executes a profiling phase and estimates the performance and energy requirements of the given task in each of the previously computed available nodes. Finally, the scheduling module relies on these estimations to compute scores for each node, to be weighted by the energy/performance ratio defined by the customer. The best fitting node is chosen to deploy the given task.

When a customer submits an application then all tasks composing this application go into a queue of pending tasks (*i.e.*, submitted tasks waiting to be scheduled). Algorithm 2 describes the process of constantly checking if there are new tasks in this queue. When a task is pulled from the queue, we search for the best fitting node and deploy the task into it. If no node is available at this given point, then the task is

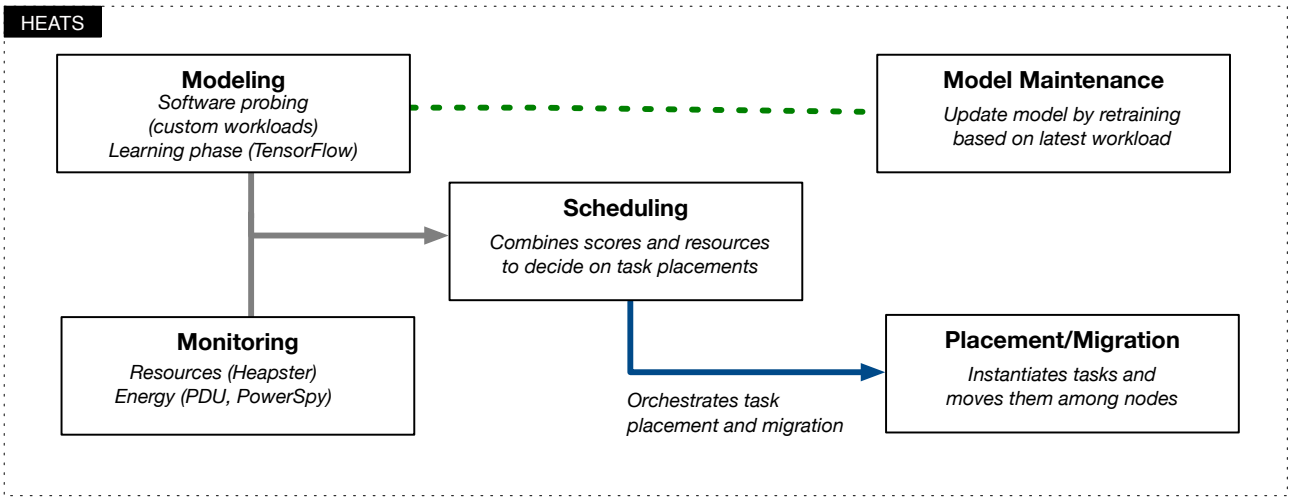


Figure 3.2: Heats's abstract components and interaction.

added to the queue again and this process will be eventually repeated until a node is found. As the queue follows a FIFO strategy, a task without matching host will not block other tasks to be scheduled as it goes to the end of the queue and a retry schedule for it will happen only when this process is completed for all the other tasks already present in the queue.

The Heats strategy will attempt to place tasks on the most efficient host that still has enough resources to run the given task. We define *most efficient* as the closest match to the demanded trade-off between energy and performance (*i.e.*, highest score after applying energy/performance ratios defined by the customer). However, the ideal node for a task will not always be available at scheduling time. Therefore, we systematically recompute our scheduling decision every x seconds. Whenever a better fit than the current host of a task is found, the scheduler performs a migration. This monitoring of running tasks combined with the re-evaluation of their best fitting host nodes is described in Algorithm 3.

Both these algorithms make use of a helper function to find the best fitting nodes to either schedule or reschedule a task. This function assumes that a task t and the customer defined ratios e_w and p_w for t are given as input. First, we identify the required resources of t and a list of nodes N which still have enough idle resources to host t. Then, for each node n in this list of nodes N we predict the duration and energy of t in n. Once this information is know for all nodes in N, then the ratios e_w and p_w are applied to calculate a score for each node. Finally, the node with the highest score is chosen as the best fitting node for task t. These steps are described in Algorithm 4.

In summary, putting all these steps together, the scheduling phase is triggered for the queue of all pending tasks. The algorithm starts by finding the best fit for the next task. It identifies its resource requirements, *e.g.*, CPU and memory, as well as the available nodes for these resources. Then, it computes the score for each of the nodes. The model (described in Section 3.3.1) is used for the profiling of nodes. The scores are computed by normalizing the predictions and adding the demanded weights. Every x seconds the rescheduling phase is triggered for the set of all running tasks. If the re-execution of the best fit decides on a different target node, the task is migrated to the new host and removed from the current one. We show in our evaluation that the chosen value of x, for our specific workload and cluster settings, has minimal impacts on the runtime or the energy efficiency of Heats.

3.3 Architecture

The architecture of Heats is composed of the following main interacting components: modeling, model maintenance, monitoring, scheduling, placement and migration. Modeling consists of a workload probing and a model learning phase, the monitoring collects the metrics of interest which are required to keep track of the system, scheduling orchestrates the placement and migration of tasks, and model maintenance uses the monitoring outcome to keep the model up-to-date. Figure 3.2 depicts an overview of these interacting components and we describe each of them in details in the remainder of this section.

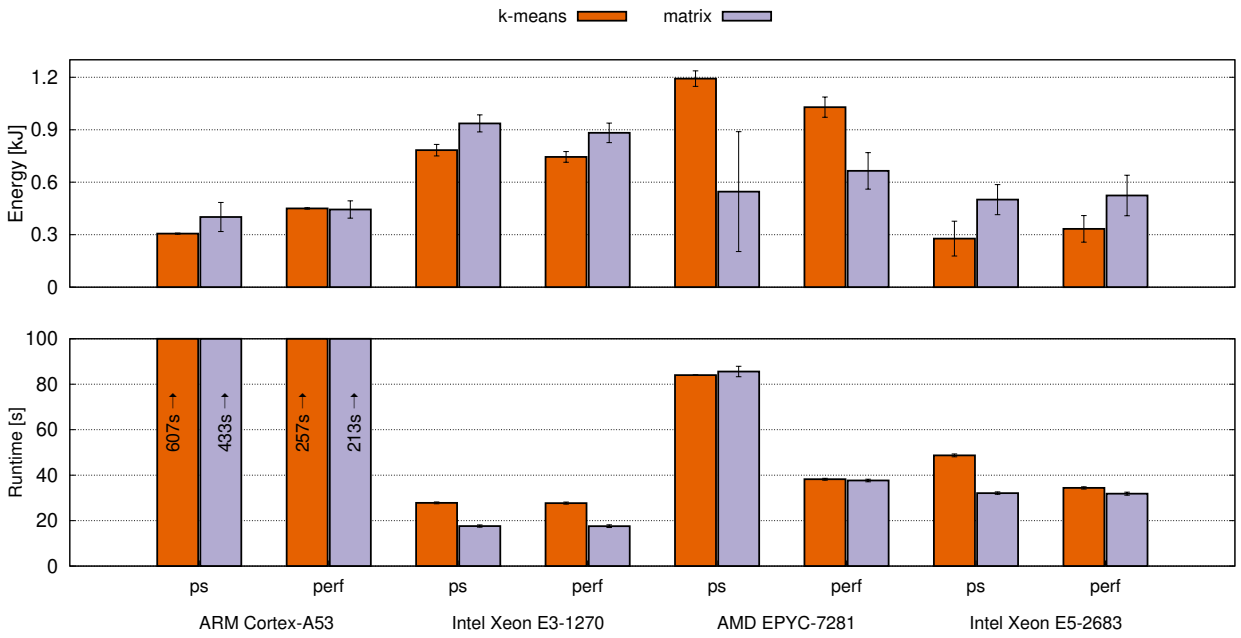


Figure 3.3: Runtime and energy spent by tasks executing `k-means` and a matrix multiplication with two different CPU governors: *powersave (ps)* and *performance (perf)*.

3.3.1 Modeling

The modeling component executes two main operations (*i.e.*, probing and learning) described below.

Probing. The probing phase discovers the properties and capabilities of the cluster, *i.e.*, the machines composing it. This probing phase is executed upon the initial setup of Heats, as well as for every major hardware reconfiguration (such as the integration of new machine types in the cluster pool). We implemented this probing so that it also takes care of exploring the performance of the nodes by scaling up and down the frequency of the CPUs [122]. We report that, in a typical setup, to produce an accurate model of a new machine usually requires a few hours. This comes from the cost of simulating applications to warm start the model considering that initially there is no historical data available.

Figure 3.3 shows the results of possible characterizations that this phase can produce, when applied to the machines of our cluster. In particular, it outputs the runtime and energy requirements of two different families of *probing tasks*. The energy requirements reported here do not consider the idle state of the machines but of the task itself only. In this way we can better understand the tasks energy requirements for the different types of hardware given. We show the results with two of such CPU-bound tasks: the aforementioned *k-means* clustering algorithm, as well as a matrix multiplication operation which is typically part of deep learning applications (*e.g.*, darknet [123]). Our architecture allows for easy integration of additional probing tasks, for instance to capture the behavior under different types of workloads.

The framework further executes these probing tasks by frequency scaling of the underlying CPUs. We achieve this by leveraging two different Linux's CPU governors [124], *powersave* and *performance*, respectively running the CPU at the minimum and maximum frequency. We observe that, within the same machine type, the energy and performance are largely affected by scaling the CPU frequency. The output of this phase is used next.

Learning. The data collected by the probing phase is used to train a multiple linear regression model [125]. Given a task, its CPU (number of cores) and memory (GBs) requirements, a fitted regression model is used to predict its energy and performance on a given machine. Each machine type available in the cluster has its own model trained on a subset of the data corresponding to the probing performed on it. Moreover, although the input features are the same in both cases, we train one model for runtime and another one for energy predictions. This results in a total of two models per type of machine in the cluster, where the type of machine is specified by the cluster owner.

The learning operation described here consists of training these models based on the data resulting from the initial probing. These two operations together result in a warm start for the model as at creation cluster time there is no historic data available. This is a typical approach used by cloud providers as probing data from other users is often available. However, the probing component constantly records data of new coming tasks and Heats relies on it to refine the model at a given frequency. This process makes sure that predictions stay accurate by reflecting the most recent state of the cluster.

For the development of the model we rely on TensorFlow [126], an open-source machine-learning framework. We did a preliminary analysis of different machine learning techniques and, for the workload used, a multiple linear regression model presented the best results with respect to model performance and cost. However, an evaluation of machine learning algorithms with different workloads may lead to a different result. Therefore, we prototype our framework in such a way that users can easily choose different methods to train their models.

3.3.2 Monitoring

Kubernetes is equipped with several tools to monitor resources: cAdvisor [127] has been partially integrated into Kubernetes' node agent kubelet [128], and it is capable of measuring resources used by containers. Heapster [129] exploits the measurements from cAdvisor, aggregates them and provides means to analyze and monitor the state of the Kubernetes cluster using Grafana [130]. Furthermore, Heapster allows us to store the aggregated data in InfluxDB [131], a time-series database that supports SQL-like queries to retrieve historical resource measurements of the Kubernetes cluster.

In order to decide whether a task has to be migrated from one node to a different heterogeneous node, the Heats scheduler has to be able to rely on a fine grained resource monitoring system. Despite the potential capability to gather resource measurements every $5\,\mathrm{s}$, we found out that Heapster cannot reliably deliver these resource measurements at a fixed rate. A custom resource measurement system was therefore implemented and installed on the Kubernetes nodes, which queries every second the local Docker instances for up-to-date resources used by the containers. These resource measurements can then be aggregated and used by the Heats scheduler to provide the needed support for migrating tasks.

The monitoring component is responsible for actively gathering information regarding the resources currently being consumed at each node by the tasks in execution. This information is required by the scheduling component (described below) to know which node has sufficient resources for the pending tasks. Heats leverages some of the default software probes from Heapster to continuously fetch the hardware resources available on any given node.

Additionally, to access in real-time the current power and energy levels of a node, we assume the availability of hardware monitors that are remotely accessible. We experimented with two different types of energy monitors: one for server-grade machines and one for low-energy profiles (see Section 3.4). These are two potential options which we use in our evaluation but there are many others (*e.g.*, pyRAPL [132]) and the energy monitoring choice is orthogonal to the results presented here.

3.3.3 Model Maintenance

In order to maintain the model up-to-date, the model maintenance component of Heats retrains the model on a sliding window of the data every x hours. In our evaluation we set the value of x to 24, but this parameter can easily be adapted to the customer's needs. For instance, if a cluster has a homogeneous trace of workloads with similar tasks coming in, then this number can be reduced without affecting the overall functionality of the system. On the other hand, if the type of tasks running in the cluster often varies, then a frequent update of the model is recommended for more accurate predictions.

Once a task is completed, the monitoring component stores in the database the actual runtime and energy measurements together with the respective metrics predicted by the current model. Additionally, we use this information to calculate and store the prediction error of both runtime and energy. Once a retraining of the model is triggered, we first use these error metrics to filter out all the entries which the error is below a given threshold. The filtered out entries correspond to workload types which the current model can already accurately predict and therefore does not need to be taken into account.

This module ensures that the trained models used for prediction reflect all types of workloads present in a cluster's trace. As the framework also allows different types of models to be trained, this component

can also be leveraged to test different types of models. This is particularly relevant as the training data is updated and different models can be better suited in different points of time. Finally, at the moment we use only the CPU and memory requirements as features but this could be extended to metrics collected during the first seconds of execution, resulting in a more accurate prediction and therefore assisting the migration component to make better decisions.

3.3.4 Scheduling

Finally, the scheduling component is in charge of orchestrating the inputs received by the modeling and monitoring components. To that end, it first ensures that a prediction for the resources used by the task on the different set of machines is completed. Then, it combines this prediction with the energy and performance trade-offs, as defined by the end-user, to decide on the best fitting node. Periodically, the scheduling component reconsiders its past decisions: when a better fitting node is found, a migration decision is taken and the corresponding task is moved to the target node.

In practice, this step monitors the following two main queues:

1. new coming tasks to be scheduled (*i.e.*, pending tasks), and
2. already scheduled and running tasks (*i.e.*, running tasks).

Both cases follow a FIFO strategy (*i.e.*, first in first out) such that older pending tasks are prioritized, reducing the average waiting time. Although the monitoring of these two queues happen independently, the scheduling of pending tasks is prioritized over the reschedule of already running tasks. This is ensured by holding the monitoring of running tasks when the pending tasks queue is not empty and the free capacity of the cluster is above the sum of required requests in all tasks of the pending queue. With this we make sure that no task is rescheduled unless there is enough resources available for new coming tasks to be scheduled.

3.3.5 Placement and Migration

In this work, we assume that the applications running in the cluster are stateless. The mains characteristic of stateless applications is that each building block of the application's architecture depends on third-party storage rather than sorting any kind of state in memory or on its disk. This means that any data the application requires is fetched from some other stateful service such as a database, or present in the request itself. Considering this, the migration of tasks in this scenario consists of simply starting the task in the new host, making sure that the service is up to avoid overheads, and finally remove it from the current host.

Our assumption of stateless applications is mainly motivated by the popularity of serverless computing services available at the major public cloud providers such as Amazon AWS Lambda [133], Microsoft Azure Functions [134] and Google Cloud Functions [135]. Some other reasons for applications to follow this architecture model is that it can automatically be scaled based on the load, adapted to the applications needed to avoid unnecessary costs and customers don't need to provision or manage anything. However, if the prototype is extended to support stateful migration then all the findings presented here also apply for stateful application.

3.4 Evaluation

This section presents the experimental evaluation of our Heats prototype. We start describing the prototype implementation followed by the experimental settings. Then, we describe the synthetic trace used to compare Heats against the default Kubernetes (k8s) settings. We compare both schedulers in terms of energy and resource utilization. Finally, we analyze how the user demands of energy/performance ratios affect the observed performances. Finally, we look at the impact of the rescheduling frequency on the overall job runtime.

3.4.1 Prototype Implementation

We base our implementation on Kubernetes (v1.8), itself implemented in Go [136]. Custom schedulers can however be implemented in any programming language and connected to the main orchestrator engine via the Kubernetes Scheduler API [137]. Heats is implemented in Python (v3.6.3) and leverages

Table 3.2: Hardware characteristics of our cluster.

	Architecture	#Cores	Frequency (GHz)	TDP (W)	Memory (GiB)
ARM Cortex-A53	big.LITTLE	4	1.4	5	1
AMD Epyc 7281	amd64	32	2.1	155	64
Intel Xeon E3-1270 v6	x86	4	3.8	72	64
Intel Xeon E5-2683 v4	x86	32	2.1	120	128

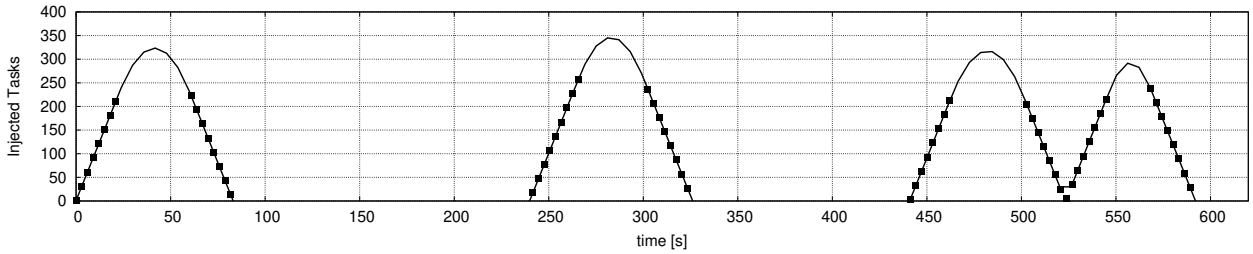


Figure 3.4: Workload injected by the synthetic trace of concurrent tasks arriving in 4 bursts.

the Kubernetes Python Client [138], a client library for the Kubernetes API [139]. The modeling component (described in Section 3.3.1) leverages the Python bindings for TensorFlow (v1.11). Heats is released as open-source and is readily available at `https://github.com/legato-project/heats-scheduler`.

3.4.2 Evaluation Settings

We deploy and conduct our experiments over a cluster composed of 4 different types of machines (see Table 3.2). Our cluster is composed of 9 machines, where one is the Kubernetes master, orchestrating the deployments and the remaining machines are workers nodes executing the tasks. The 8 worker nodes consist of one AMD, 3 Intel and 4 ARM machines. The energy consumption is measured using a LINDY iPower Control 2x6M power distribution unit (PDU) for the server type machines and PowerSpy [140] devices for the three Raspberry Pi. Both the PDU and PowerSpy records up-to-date measurements for the active power at a resolution of $1\,\mathrm{W}$. We query it up to every second and store the collected metrics in a InfluxDB instance also running in the master node.

3.4.3 Workload Trace

We use a synthetic workload trace to evaluate the gains and trade-offs of our system. Figure 3.4 shows the workload injected by this trace. We use it to deploy multithreaded tasks executing an iterative implementation of the `k-means` algorithm in the C programming language. The program, shipped as statically linked binary for Alpine Linux [141], executes over a predefined dataset of $65\,536$ data points over 32 dimensions.

Once deployed, the tasks will compute clusters by splitting the dataset into blocks processed by two worker threads for a specified maximum number of iterations, chosen randomly in the range of 500 to 1000. This step ensures the heterogeneity characteristic of the workload regarding the type of tasks running. At completion, the result of the application is stored in the form of a file inside the container's image. In total, 480 `k-means` jobs are deployed following four bursts over 10 minutes, executed randomly within a timeframe of 150 seconds. The same sequence of pseudo-random numbers is ensured upon every run of a trace by using a fixed random seed.

3.4.4 System Metrics

To evaluate our Heats scheduler we start by looking at some system metrics while using the previously described cluster, running the discussed synthetic workload trace and taking the default Kubernetes scheduler as baseline. Our objective here is to show that by leveraging existing orchestrators in heterogeneous contexts, we can ensure energy efficiency while still meeting customers requirements.

First, we compare the CPU load induced on the cluster by Heats against the default scheduling policy of Kubernetes. Figure 3.5 shows this for the default k8s scheduler as well as for two different Heats

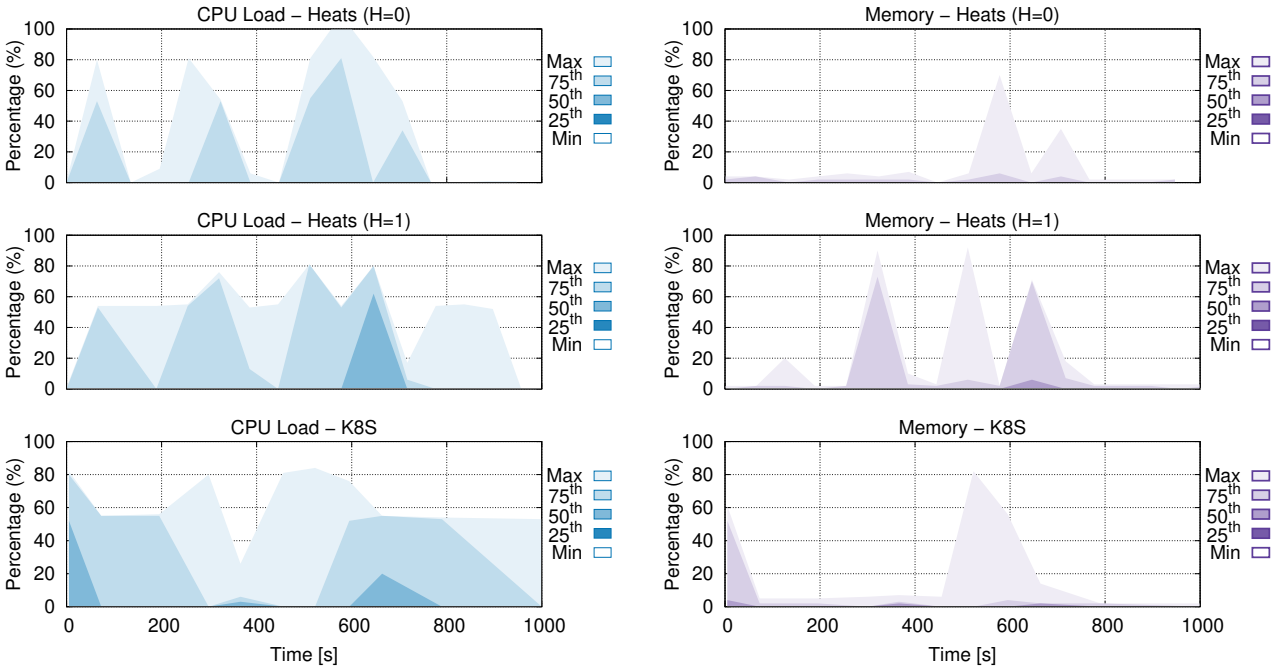


Figure 3.5: CPU and memory usage distribution (percentiles) across all machines in the cluster. We show these metrics with three different schedulers: Heats in two configurations (H=0, H=1) on the first and second row, Kubernetes in the third row.

configurations, one for performance ($H = 0$), the other for energy-efficiency ($H = 1$). We observe how the load patterns are very similar and follow the arrival pattern of the tasks in the trace Figure 3.4. As in the trace considered here, the tasks arrive in bursts, we can also observe peaks of utilization following this pattern. We conclude that the Heats scheduler does not deteriorate the lifetime of the processors by artificially stressing them [142].

Then, we look at the memory usage across the cluster for the default k8s scheduler as well as for the two different Heats configurations described above. The memory load of the k8s default scheduler and Heats with $H = 0$ induce similar patterns. On the other hand, Heats with $H = 1$ seems to have a higher memory utilization. This happens because, in this case, we have more tasks willing to be energy efficient than the cluster computing capacity available in energy efficient machines. Therefore, a migration happens every time a task is completed and some of this capacity is freed.

3.4.5 Energy vs. Performance Weights

The value chosen for the H parameter is of paramount importance, especially when considering the resulting energy costs and impact on the overall runtime of the jobs. To better understand this aspect, we compare the following 6 different approaches:

1. the default scheduler (`k8s`),
2. Heats configured to be as efficient as possible runtime-wise, ignoring any energy concerns ($H = 0$),
3. Heats trying to be as energy-efficient as possible ($H = 1$),
4. a fixed H value chosen out of our practical experience (`rand`, for $H = 0.618$),
5. a different random H value for each task (`rall`), and
6. other variations of the H-value (from 0 to 1, by increments of 0.2).

The fixed random scenario (`rand`) aims to mimic a customer with no particular requirements for all tasks to be deployed. On the other hand, the `rall` scenario reflects the case where tasks with mixed requirements

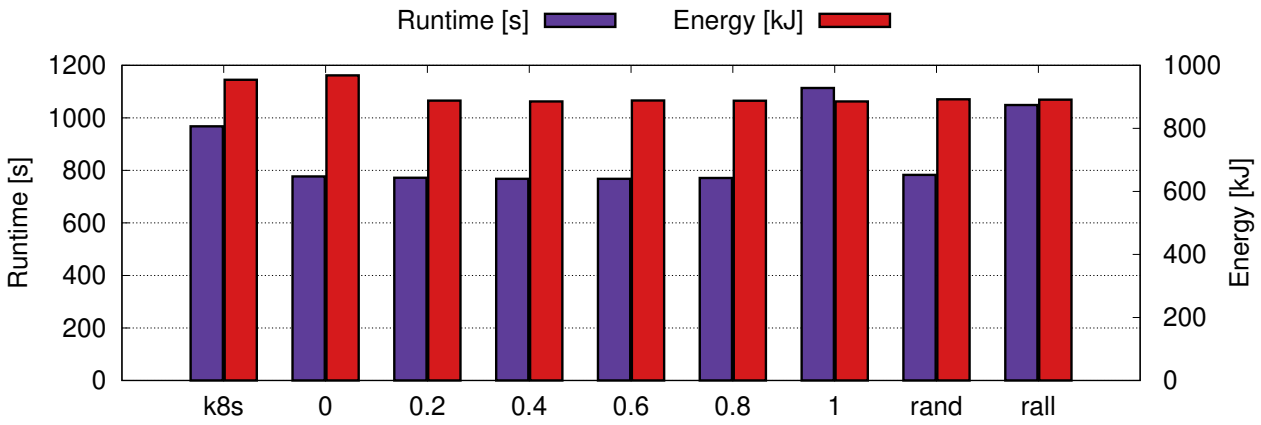


Figure 3.6: Energy efficiency and impact on the overall runtime of the trace for several scheduling policies.

are deployed in the cluster. These two scenarios reflect the deployment of homogeneous and heterogeneous applications, respectively.

Figure 3.6 shows our results. For each configuration, we show the cumulative energy costs (in kJ) and the achieved runtime (in seconds), respectively on the left and right vertical bars. In this case, we include the idle state requirements of the machines to have a macro view over the cluster. While these results require further investigations (*e.g.*, using real-world traces), we believe them to be of practical interest for end-users.

When compared to the default scheduler (`k8s`), (2) performs better on an energy cost of 1.5% while (3) performs worse but presents 7.1% of energy savings. Besides, when compared to each other, (2) performs better while (3) is more energy efficient. Regarding approach (4), we can observe that the runtime as well as the energy consumption are in between the observations for approach (2) and (3). Approach (5) doesn't have a clear overall priority over performance or energy efficiency as each task have a different requirement. Finally, for the other variations of the H-value in approach (6) we observe how the configurations achieve similar results, with a sensible deviation only with the less energy efficient variant. In order to observe a higher impact from varying the H-value, we could consider scaling up the size of cluster and scope of applications.

3.5 Summary

In this chapter, we presented Heats, a novel task-based scheduling system for heterogeneous clusters. Heats learns about the properties of the machines in the cluster, schedules and possibly migrates tasks to the best-fitting node currently available. Our experimental evaluation has revealed that Heats can yield considerable energy savings depending on the type of resources at hand, the workload and the desired energy/performance ratio.

In summary, the contributions presented in this chapter are as follows:

1. a probing framework, which we use to build a model of the underlying hardware resources;
2. the design and implementation of Heats prototype, a new container scheduler system that, by leveraging the underlying model, places application tasks onto the best matching nodes among the currently available hardware resources for the intended energy/performance ratio;
3. prototype evaluation by means of an in-depth experimental analysis.

Next, in Chapter 4, we start exploring a multi-tenant scenario which leverages per-core pinning and per-core frequency scaling as well as user priorities to further improve energy savings.

Chapter 4

Cloud Providers: Multi-Tenant

In this chapter we extend the work presented so far to cover the multi-tenant scenario. To this end, we advocate a new resource management design that exploits the idea of differential approximation and sprinting. This unique combination avoids the eviction of low-priority jobs and its consequent latency degradation and resource waste. Next, we discuss the design, implementation and evaluation details of DiAS, an extension of the Spark processing engine to support deflate jobs by dropping tasks and to sprint jobs.

4.1 Introduction

Big data production systems, *e.g.*, Google [7] and Facebook [9], implement priority scheduling to process job streams with different characteristics and latency requirements. Analysis jobs, *e.g.*, hive queries [143] and text mining, of different priorities arrive in streams and are executed as parallel jobs with varying numbers of map and reduce tasks. Trace studies [25] show that high-priority jobs are promptly served with little queueing time, while low-priority jobs suffer from repetitive evictions causing significant resource waste. This is mainly due to preemptive priority scheduling [8], where high-priority jobs are given the ability to preempt lower-priority jobs in execution. The average latency slowdown of low-priority jobs [5], *i.e.*, the end-to-end response time divided by the execution time excluding eviction, can be $3\times$ higher than for high-priority jobs. All in all, priority-enabled big data systems preserve the performance advantage of high-priority jobs at the cost of resource efficiency and performance of low-priority jobs.

It is extremely challenging to optimize the performance of big data engines with priority scheduling, due to performance conflicts across disparate job priorities or conflicting performance targets, *i.e.*, latency *vs.* resource efficiency. Meeting the latency targets in priority systems is a long standing challenge from both system [8], [29] and modelling [39], [42] perspectives due to the complex dynamics and performance requirements across diverse priority classes. This is partly because the processing order of low-priority jobs highly depends on the high-priority jobs, especially when low-priority jobs are evicted during periods of resource shortage. In additional to the inter-job dependency, big data jobs themselves have complex execution dynamics across their parallel tasks and synchronization stages.

Existing systems address the latency issue of big data engines mainly from the perspective of single job type, *i.e.*, one single priority. On the one hand, approximation-enabled processing engines, *e.g.*, BlinkDB [30] and ApproxHadoop [32], reduce the execution times by processing a fraction of input data. The performance advantage comes at the cost of accuracy losses. On the other hand, hardware features are increasingly being exploited to accelerate the executions of jobs. For example, Pupil [36] and Sprinting Game [14] temporarily sprint the CPU frequency during the slow execution phases of jobs. However, these engines do not readily apply to multi-priority scenarios, answering the performance and resource tradeoff among different priorities.

We advocate to differentially approximate and sprint CPU frequency for jobs of different priorities, termed *differential approximation and sprinting* (DiAS), to replace preemptive eviction in priority scheduling. DiAS improves the latency for all priorities and eliminates resource waste from re-executing the evicted low-priority jobs. To achieve this, DiAS reduces a fraction of data load for low-priority jobs and temporarily increase the CPU frequency for high-priority jobs.

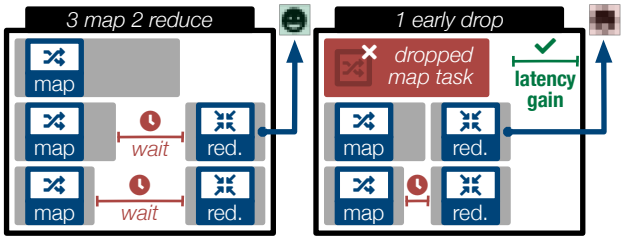


Figure 4.1: Approximation by task dropping: an example on a job of 3 maps and 2 reduce tasks.

The differential approximation adopts a controllable approximation level that discriminates among priority classes by dropping different fractions of data. It gives better latencies for low-priority jobs at the cost of their accuracy loss and minor latency increase for high-priority jobs, depending on the levels of input data dropping. The differential sprinting then adjusts the frequency levels such that the high-priority jobs can be accelerated after temporarily waiting behind the low-priority approximate jobs.

We design, implement and evaluate DiAS on top of Spark [144]. The DiAS extension module is composed of N priority buffers, and one deflator that assigns approximation and sprinting levels for each priority. To determine the dropping and frequency levels, we derive a set of stochastic models that can predict average response times of DiAS jobs. The models, based on matrix-analytic methods and parameterized via simple linear regressions, can effectively guide the choice of approximation levels for each priority.

We evaluate DiAS using benchmarks that process the contents of the *stackexchange* [145] network of question-and-answer websites as well as the Google web graph [146], with two and three job priority levels. To demonstrate the robustness of DiAS, we consider various workload profiles, *i.e.*, priority ratios, job sizes, and system loads. Evaluation results show that DiAS achieves remarkable reductions not only on the mean and tail latency of low-priority jobs, but also on the *tail latency of high-priority jobs*. With DiAS we achieve up to 90% and 60% reduction in the mean/tail latency for low-priority and high priority jobs, respectively, a preemptive priority system without approximation and sprinting. Moreover, the promising performance gain of DiAS comes with a noticeable energy reduction, *i.e.*, up to 30% , even after spending extra power to sprint the high priority jobs.

The rest of this chapter is organized as follows. The main design choices of DiAS are presented in Section 4.2. We describe the engine details of DiAS in Section 4.3. We extensively evaluate the performance of our prototype in Section 4.4, where we also detail the prototype implementation before concluding in Section 4.5.

4.2 DiAS Design

Motivated by the importance and complexity of tuning the performance of priority-enabled systems, we propose the idea of differential approximation and sprinting across different priority levels as a means to

1. reshape the workload demands of jobs in each priority level;
2. implicitly provision more resources to higher-priority jobs;
3. speed up the execution of lower-priority jobs by deflating their processing load, *i.e.*, number of tasks;
4. provide consistent performance guarantees on high-priority jobs; and
5. minimize resource waste.

The core goal of differential approximation and sprinting is to decide the approximation level (task dropping ratio), denoted by $0 \leq \theta_k \leq 1$, and sprinting timeout, denoted by T_k to be applied on arriving jobs given their priority class k, their tolerance to accuracy degradation, and the available sprinting budget. The expected outcome of differential approximation in a scenario of two job priorities, *i.e.*, high *vs.* low, is to minimize the resource waste and average/tail latencies of high/low priority jobs, while maintaining the relative error of low priority jobs within a given bound and fully use the available sprinting budget.

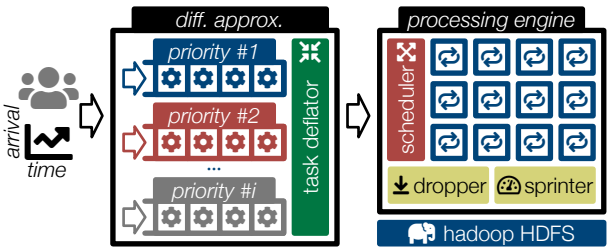


Figure 4.2: Schematic diagram of differential approximation.

In contrast to preemptive schedulers, we alter the resource demand of lower- and higher-priority jobs, instead of evicting lower-priority jobs upon the arrival of higher-priority jobs. DiAS is our prototype implementation of this design. It plugs into existing big data processing engines to support differential approximation, computational sprinting and workload deflation by means of dropping tasks.

4.2.1 Approximate Big Data Jobs

The aim of approximate computing in big data processing [30], [32] is to solve the performance conundrum between latency and accuracy requirements of analysis jobs. Instead of processing all the input data, only a subset is chosen to be processed to lower the overall computation demand and reduce latency. Existing systems (*e.g.*, ApproxHadoop [32]) put a significant engineering effort to enable dropping (map) tasks and their assigned input data prior or during execution. Figure 4.1 illustrates this task dropping strategy on a simple job with 3 map and 2 reduce tasks, with the goal of attaining a given approximation level. In this example, we randomly choose one map task and drop it before its execution. Task dropping saves the overhead of fetching data and avoids the execution of the dropped tasks. Nevertheless, while it reduces the computational demand of jobs, it unavoidably degrades the analysis accuracy. This precision loss depends on both the analysis performed and the data used, which can be estimated offline as shown in Section 4.4.

4.2.2 Architecture

Figure 4.2 depicts the architecture of DiAS consisting of the following key components:

1. a set of job buffers for each priority, indexed by $k \in \{0, \ldots, K\}$,
2. a task deflator that determines the approximation level θ_k and sprinting timeout T_k for each priority k,
3. a scheduler which orchestrates the arriving and running tasks,
4. a dropper which applies the θ_k to the respective tasks, and
5. a sprinter which temporarily sprints jobs based on the T_k.

The higher values of k indicate higher priority. Moreover, the deflator has two main functionalities:

1. to determine the approximation level and sprinting timeout based on empirical/stochastic models as well as on performance and budget thresholds, and
2. to dispatch jobs from the priority buffers into the processing engine.

Upon arrival, jobs are immediately dispatched to the corresponding buffer according to their priorities. Jobs queued at each buffer are processed in a FCFS manner (*i.e.*, first come, first served). The task deflator selects the job at the head of the highest non-empty priority buffer, say k. This priority-k job with its corresponding approximation level, θ_k, is sent to the processing engine, which splits the job into multiple tasks on the cluster. To avoid potential resource waste caused by eviction using preemption, the execution across priority buffers of DiAS is non-preemptive. DiAS only dispatches jobs from the head of the buffer to the processing engine when the previous job completes, independent of the priority of newly arrived jobs. Moreover, DiAS assumes that the processing engine is able to drop tasks to achieve the target ratio θ_k. We note that as DiAS aims to reshape the job workloads prior to entering the processing engines, the design of DiAS is general and compatible with different processing engines.

For our baseline results with preemption (Section 4.4), DiAS also provides the ability to evict jobs from the processing engine. In this case, as soon as any job arrives with a higher priority than the job currently being processed, the job in the engine is evicted back to the head of its buffer and the arriving job is immediately sent in for execution. In addition, DiAS can leverage computational sprinting to further counter the effects on higher priority jobs stemming from not preempting lower priority ones. If sprinting is enabled, the deflator in parallel to dispatching the job communicates to the sprinter the sprinting timeout T_k to use. Once the timeout elapses, the sprinter will temporarily accelerate the execution of the job.

4.3 DiAS Engine

To guide DiAS, we analytically derive the response time distribution offered by the cluster to the incoming multi-task jobs classified in multiple priorities. Jobs are classified in K priorities, where a priority-k job has precedence over jobs in priority levels $l < k$, for $1 \leq l, k \leq K$. According to the DiAS architecture, jobs are served in FCFS order and each job seizes all the resources in the cluster (or in the partition used by the corresponding engine) to execute. This can be viewed as a single server queue serving K priority classes. We thus opt to employ the recent method proposed in [42], which is capable of obtaining the response time distribution and its moments for a fairly general priority queue with K priority classes under both preemptive and non-preemptive scheduling.

One key reason to choose [42] as the latency model for DiAS is its support for Phase-Type (PH) job processing times [147], which is a class of distributions that can capture fairly general behavior. Further, PH distributions are closed under a number of operations, a feature that we exploit to model the detailed job processing times of concurrent tasks. Instead of using a given distribution to model the job processing time, we resort to a bottom-up approach and build a more detailed view at the task level or wave levels. For a description of waves and their role in job execution see Section 4.3.2 below. We thus exploit PH distributions to capture details of tasks and waves (*i.e.*, no. of waves $=\lceil \frac{no.\,tasks}{no.\,slots} \rceil$) within the job processing time and build upon recent results on priority queues with PH components [42].

Background on the MMAP[K]/PH[K]/1 priority queue

Horváth [42] analytically derived the latency distribution for an MMAP[K]/PH[K]/1 priority queue, where processing times follow PH distributions, differentiated for the K job classes, and arrivals follow a Marked Markovian Arrival Process (MMAP) with K different streams [147], one for each priority class. This class of arrival processes can capture fairly general behaviors, including correlations among arrival streams or general inter-arrival times. The parameters of an MMAP are $K+1$ $m_a \times m_a$ matrices $(\boldsymbol{D}_0, \boldsymbol{D}_1, ..., \boldsymbol{D}_K)$, where $\boldsymbol{D}_k$ holds the transition rates for class-k jobs, and $\boldsymbol{D}_0$ ensures that the matrix $\boldsymbol{D} = \sum_{k=0}^{K} \boldsymbol{D}_k$ is the generator of a Markov chain. The simplest non-trivial example is the marked Poisson arrival process, where $m_a = 1$, $\boldsymbol{D}_k = \lambda_k$, which is the arrival rate of class-k jobs, and $\boldsymbol{D}_0 = -\sum_{k=1}^{K} \lambda_k$.

Assumptions and notations on the cluster and approximate/sprinting jobs

We assume the cluster, or the allocated partition, is composed of C computing slots. Priority-k jobs have n_m^k map and n_r^k reduce tasks, both of which are discrete random variables with minimum value 1 and maximum value N_m^k and N_r^k, respectively. On average, the time to execute a map task is $1/\mu_m^k$ and to execute a reduce task is $1/\mu_r^k$. In addition, the job execution may include an initial setup time that lasts $1/\mu_o^k$ time on average, and an intermediate shuffle stage that requires on average $1/\mu_s^k$ time. We note that, when sprinting is enabled, the service rates can be approximately captured by the effective sprinting rates as a weighted average of the sprinted and non-sprinted execution times per task and class k. Predicting these rates is complex [18]. We assume that the effective sprinting rates are provided by an oracle for each class k and timeout value. Moreover, as the number of executors available is less than the number of parallel tasks and executors comprise multiple cores, each executor concurrently executes multiple tasks. Hence, our current approach sprints all available cores at the same time which is beneficial for applications consisting of tasks with equal workloads. Table 4.1 summarizes the notation.

4.3.1 Task-level Model

For priority-k jobs, we set the task drop ratio to θ_m^k for map tasks and to θ_r^k for reduce tasks. The effective number of map and reduce tasks is thus $\bar{n}_m^k = \lceil n_m^k(1-\theta_m^k) \rceil$ and $\bar{n}_r^k = \lceil n_r^k(1-\theta_r^k) \rceil$. Moreover, as the number of tasks is a random variable, we let $p_m^k(t)$ and $p_r^k(u)$ be the probabilities that a priority-k job has t

Table 4.1: Summary of DiAS's notation.

Symbol	Definition
C	Number of computing slots
N_m^k	Max. number of map tasks in a priority-k job
N_r^k	Max. number of reduce tasks in a priority-k job
$p_m(t)$	Prob. that a priority-k job has t map tasks
$p_r(u)$	Prob. that a priority-k job has t reduce tasks
$1/\mu_m^k$	Mean exec. time for map tasks in a priority-k job
$1/\mu_r^k$	Mean exec. time for reduce tasks in a priority-k job
$1/\mu_o^k$	Mean setup time for a priority-k job
$1/\mu_s^k$	Mean shuffle time for a priority-k job
θ_m^k	Approximation ratio for map tasks in a priority-k job
θ_r^k	Approximation ratio for reduce tasks in a priority-k job
$\mathcal{O}$	Overhead stage
$\mathcal{M}_t$	Map stage with t map tasks left to process
$\mathcal{S}$	Shuffle stage
$\mathcal{R}_u$	Reduce stage with u map tasks left to process

map and u reduce tasks, where $1 \leq t \leq N_m^k$ and $1 \leq u \leq N_r^k$. From this point on, we drop the super-index k for clarity, but the definitions apply to all job priorities making use of the appropriate index.

With the above definitions we can extend the model in [148] to incorporate the overhead $\mathcal{O}$ and shuffle stages $\mathcal{S}$, as well as to allow for a variable number of tasks. We thus let the processing phase i keep track of the job current execution step, where:

1. $i = \mathcal{O}$ indicates the job is in the initial setup (overhead) stage;
2. $i = \mathcal{M}_t$ indicates that t map tasks remain to be completed, for $1 \leq t \leq \bar{N}_m$;
3. $i = \mathcal{S}$ indicates the job is in the intermediate shuffle stage;
4. $i = \mathcal{R}_u$ indicates that u reduce tasks remain to be completed, for $1 \leq u \leq \bar{N}_r$.

All jobs start in stage $\mathcal{O}$ and their evolution is determined by the transition rates from phase i to the next phase j, $f(i,j)$, defined as:

$$f(i,j) = \begin{cases} \mu_o p_m(t), & i = \mathcal{O}, j = \mathcal{M}_{\bar{t}}, \\ C\mu_m, & i = \mathcal{M}_t, j = \mathcal{M}_{t-1}, t \geq C, \\ t\mu_m, & i = \mathcal{M}_t, j = \mathcal{M}_{t-1}, 2 \leq t < C, \\ \mu_m, & i = \mathcal{M}_1, j = \mathcal{S}, \\ \mu_s p_r(u), & i = \mathcal{S}, j = \mathcal{R}_{\bar{u}}, \\ C\mu_r, & i = \mathcal{R}_u, j = \mathcal{R}_{u-1}, u \geq C, \\ u\mu_r, & i = \mathcal{R}_u, j = \mathcal{R}_{u-1}, 1 \leq u < C, \end{cases} \tag{4.1}$$

where $\mathcal{R}_0$ denotes the end of all reduce tasks and the job completion.

In (4.1) the first row corresponds to a transition from the initial setup stage $\mathcal{O}$ to the map stage for a job with t map tasks, *i.e.*, it actually starts with $\bar{t}$ tasks due to early drop. The next two rows show that the maximum parallelism is C and that tasks finish one by one until the map stage is completed. The next transition is to the shuffle stage $\mathcal{S}$, after which the job moves on to the reduce stage, where, after dropping, a total of $\bar{u}$ tasks must be executed if the job has u reduce tasks. Since N_m and N_r are the maximum number of map and reduce tasks, the phase space is $\mathcal{P} = \{\mathcal{O}, \mathcal{M}_{\bar{N}_m}, \ldots, \mathcal{M}_1, \mathcal{S}, \mathcal{R}_{\bar{N}_r}, \ldots, \mathcal{R}_1\}$, and we can build a transition matrix $\boldsymbol{F}$ with entries $f(i,j)$ in (4.1) for $i, j \in \mathcal{P}$. Further, we define the vector $\phi = [1 \quad \mathbf{0}]$ as the initial phase distribution, where 1 indicates that all jobs start processing in phase $i = \mathcal{O}$. The pair $(\phi, \boldsymbol{F})$ is thus a PH representation [147] of the job processing time with $N_m + N_r + 2$ phases.

4.3.2 Wave-level Model

Whereas the just described model is very detailed in considering the evolution of jobs at the task level, it assumes that task execution times follow an exponential distribution. Generalizing this assumption at the task level is very challenging as it would require keeping track of individual tasks separately. We thus take a different approach. We observe that tasks tend to have fairly similar execution times [18], leading to an execution in waves. For instance, a job composed of 40 tasks executing in a cluster with 20 computing slots will start with a first wave of 20 tasks executing in parallel. If these tasks have fairly similar execution times, they will finish close to each other, allowing the next 20 tasks to execute almost at the same time, making up a second wave. This wave-level model captures this behavior, having the job processing time as a sequence of waves, each with a wave execution time.

Given C computing slots and a job made up of t and u map and reduce tasks, respectively, its effective number of map and reduce waves are $\bar{w}_m = \lceil \bar{t}/C \rceil$ and $\bar{w}_r = \lceil \bar{u}/C \rceil$, respectively. Recall that $\bar{t} = \lceil t(1-\theta_m) \rceil$ is the effective number of map tasks to execute once a task drop ratio θ_m is applied. Since waves are consecutive, we can model the execution time of the d-th map wave as a PH distribution with $v_{m(d)}$ phases and parameters $(\alpha_{m(d)}, A_{m(d)})$, avoiding the exponential assumption and allowing for fairly general behaviors. Moreover, we allow each wave to have a potentially different execution time, as we also observed in our experiments with state-of-the-art execution engines, *e.g.*, Spark. We similarly let the d-th reduce wave have a PH distribution with $v_{r(d)}$ phases and parameters $(\alpha_{r(d)}, A_{r(d)})$. Also, let the initial setup stage have v_o phases and parameters (α_o, A_o), and the intermediate shuffle stage have v_s phases and parameters (α_s, A_s). We further define the exit rate vector $a_x = -A_x \mathbf{1}$ for x representing any of the stages considered. Since the sum of independent PH random variables is also PH [147], we represent the job processing time as a PH distribution with $v = v_o + \sum_{d=1}^{w_m} v_{m(d)} + v_s + \sum_{d=1}^{w_r} v_{r(d)}$ phases and parameters (α, A).

For clarity, consider the case where $w_m = w_r = 2$, *i.e.*, both map and reduce stages are composed of 2 waves of execution. The transition matrix A of the job processing time for this case is:

$$A = \begin{bmatrix} A_o & a_o\alpha_{m(1)}q_{m(2)} & a_o\alpha_{m(2)}q_{m(1)} & & & \\ & A_{m(1)} & a_{m(1)}\alpha_{m(2)} & & & \\ & & A_{m(2)} & a_{m(2)}\alpha_s & & \\ & & & A_s & a_s\alpha_{r(1)}q_{r(2)} & a_s\alpha_{r(2)}q_{r(1)} \\ & & & & A_{r(1)} & a_{r(1)}\alpha_{r(2)} \\ & & & & & A_{r(2)} \end{bmatrix} \tag{4.2}$$

where $q_m(d)$ and $q_r(d)$ are the probabilities that a job requires d waves of execution in the map and reduce stages, respectively. These can be computed as

$$q_m(d) = \sum_{\bar{t}=(d-1)C+1}^{dC} \sum_{t:\lceil t(1-\theta)\rceil=\bar{t}} p_m(t),$$

where the inner sum accounts for the probability that a job has $\bar{t}$ effective map tasks after dropping, and the outer sum accounts for all cases where the $\bar{t}$ effective tasks can be executed in d waves. Finally, the initial probability vector can be written simply as $\alpha = \begin{bmatrix} \alpha_o & 0 \end{bmatrix}$, since all jobs start in the setup stage, completing the PH representation of the job processing time at the wave level.

4.3.3 Validation

We now illustrate the results obtained with the model against those observed experimentally. We first consider two different datasets, for which we obtain map and reduce task execution times from a profiling run. The details of experiments can be found in Section 4.4. We also collect samples of the overhead times, which we have observed to be dependent on the data size. To keep profiling at a minimum, we collect overhead times from two configurations only: one where no task drop is performed, and one where 90% of the tasks are dropped, which is the maximum drop ratio we consider. Then, for a given drop ratio we determine the associated mean overhead time by a simple linear interpolation between these two extreme scenarios.

Figure 4.3 shows the *observed* job execution times (*x* marks) for several drop ratios and two different datasets. It also shows the *predicted* job execution times (*o* marks) obtained with the model from estimations of the task execution time, the overhead, and setting the task drop ratio. The results show that

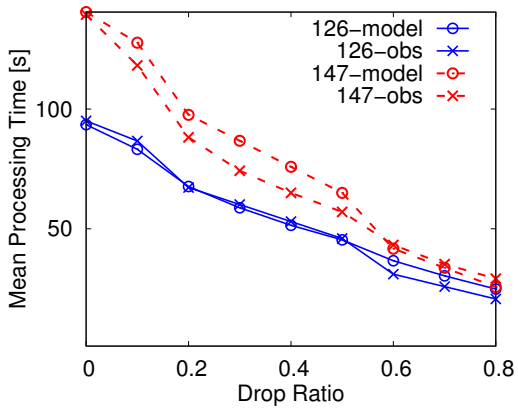


Figure 4.3: Job processing times and drop ratio.

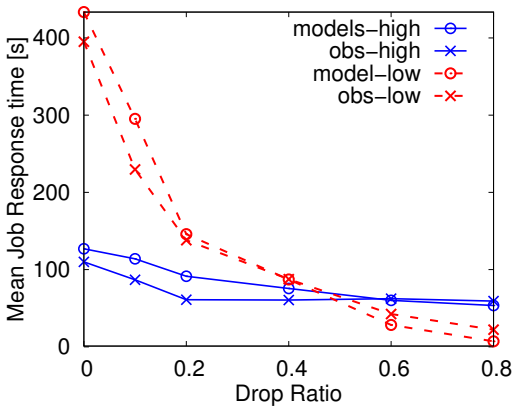


Figure 4.4: Job response times and drop ratio.

the model accurately predicts the job processing time as a function of the drop ratio, with mean errors of 11.1% and 7.8% for the two datasets shown.

We now employ the model to predict the job *response time*, parameterizing the model with the same information as above: mean task execution time and overhead. Also, we set the arrival rate to achieve an 80% cluster utilization and test several values for the drop ratio. Note that for low loads the response times are similar to the processing times, which we have shown above to be accurately predicted by the model. We are thus interested in testing a high load scenario where the model must be able to predict well both the processing and waiting times.

Further, we let high- and low-priority jobs process different datasets, such that the average low-priority job size is 2.36× larger (1117MB and 473MB, respectively), and the ratio between low- and high-priority jobs is set to 9 (i.e., more low-priority jobs). This setup is similar to the ones used in the experimental section. Figure 4.4 shows the observed and predicted mean response times for both low- and high-priority jobs. The model is clearly able to follow the decrease in response times as the drop ratio increases, with an average error of 18.7%.

We can therefore employ the model to determine whether a certain configuration under a given workload can achieve a preset latency objective. In fact, the model predictions can be used to determine a minimum value for the drop ratio, such that the latency degradation on the high-priority jobs is kept limited. Together with a constraint on the accuracy error, it is possible to provide the user with latency-accuracy pairs for feasible drop ratios, each of which presents a different tradeoff.

4.4 Evaluation

This section presents our extensive evaluation of the DiAS prototype atop the Spark engine. We compare it against priority systems, being preemptive or non-preemptive without approximation and sprinting.

The specific question we answer in this section is the following:

***RQ**: Given the accuracy requirement of multi-priority jobs, how much improvement can be obtained on the average and tail response time of low priority jobs without any resource waste and degradation of high priority jobs?*

We first describe the prototype implementation, then our experimental setup, the configuration used for Spark and the workload details. Specifically, we focus on big data applications of text and graph analytics. The number of priorities is defined based on the characteristics of Google trace which has 12 priorities but is is dominated by two to three classes that account for 89% of all tasks [25]. Therefore, although our proposed methodology can easily be extended to larger number of priorities, we will focus on the scenario of two and three priorities.

We first evaluate the design of differential approximation (§4.4.3), followed by the full fledged design of DiAS - combining differential approximation and sprinting. While the differential approximation improves the low priority jobs at a marginal degradation of the high priority jobs, the complete DiAS (§4.4.4) can

improve the performance of both priorities, compared to standard preemptive and non-preemptive systems.

4.4.1 Prototype Implementation

We implement the DiAS prototype in the Go programming language and use Spark as big data processing engine. In addition to priority buffers and the deflator, we also implemented a workload generator and augment Spark with the capability to drop tasks. To deliver the aforementioned functionalities, DiAS is designed to be multi-threaded.

Deflator

The deflator consists of one dispatcher and one monitor thread. When a job completes or a new job arrives, the dispatcher thread selects which job to run and dispatches it using the `os.exec` library. It does so by first creating a `cmd` structure and then launches the process with `Start()`. When evicting jobs, this thread sends the `SIGKILL` to the process using `cmd.Process.Kill()`. The monitor thread surveils the running job, collecting its exit status via `Wait()` and actively relays the completion/eviction of the job to the dispatcher thread using a golang channel.

Sprinter

If sprinting is enabled, the sprinter handles a sprinting timer for each dispatched job and tracks the remaining sprinting budget. When the timer fires, it uses DVFS (*i.e.*, dynamic voltage and frequency scaling) to temporarily accelerate the job execution by adjusting the frequency of the CPU on the cluster nodes via the `cpupower` utility. A job is accelerated until either its end or the depletion of the sprinting budget. The sprinting budget is replenished over time using a replenishing rate, *e.g.*, 6 sprinting minutes per hour [15]. The timeout is ignored if the job ends sooner.

Dropper

A Spark job typically analyzes a dataset stored as files in HDFS. Each Spark job is translated into a DAG of operations on Resilient Distributed Datasets (RDD) used as input/output. The job execution proceeds in stages (*i.e.*, periods of synchronization points). Each RDD is made of multiple partitions, the number of which indicates the parallelism achievable by Spark, as each partition can be concurrently executed by only one task. The size of a job is thus conventionally defined by the number of RDDs and their partitions (equivalently tasks). Each stage relies on the `findMissingPartitions()` function to get the number of partitions to be computed. To implement task dropping in Spark, we modify `findMissingPartitions()` to return only $\lceil n(1-\theta_k) \rceil$ partitions out of n following the specifications of the deflator.

4.4.2 Experimental Setup and Workloads

Next, we detail the setup used for the experiments here presented as well as the details about jobs forming the used workload.

Spark Processing Engine

We rely on Spark v2.1 and a cluster with one master and ten workers. Each worker uses 2 CPU cores, and 4 GB memory. Our machines consist of Dell PowerEdge R330 servers equipped with Intel Xeon E3-1270 v6 CPU, 64 hyper-threaded cores and 128 GB memory, interconnected by a 10G Ethernet switched network on a star topology. To store the data, we deploy HDFS (v2.8.0), using one `namenode` and three `datanodes` [149].

Text Analysis Jobs

We deploy jobs that perform text analysis on XML data dumps collected from 164 StackExchange websites [145] each dedicated to a different topic. The goal of the analysis is to find the popularity of different words in different topics by first parsing the XML to extract the posts of users followed by counting the frequency of words. Each Spark job processes one such data set. To take advantage of the 20 cores in our Spark cluster, following Spark's tuning suggestions [11], we split each dataset into 50 RDD partitions. Accordingly, Spark processes RDDs is split in multiple waves.

Jobs arrive following an exponentially-distributed inter-arrival time and enqueued before being dispatched to the Spark engine. We tune the arrival rate to obtain between 50% and 80% utilization of the system based on offline profiling of our scenario. The job response time is thus composed of the queueing time and processing time. The main metrics of interests include the average and tail response time, *i.e.*, 95^{th}, for each priority.

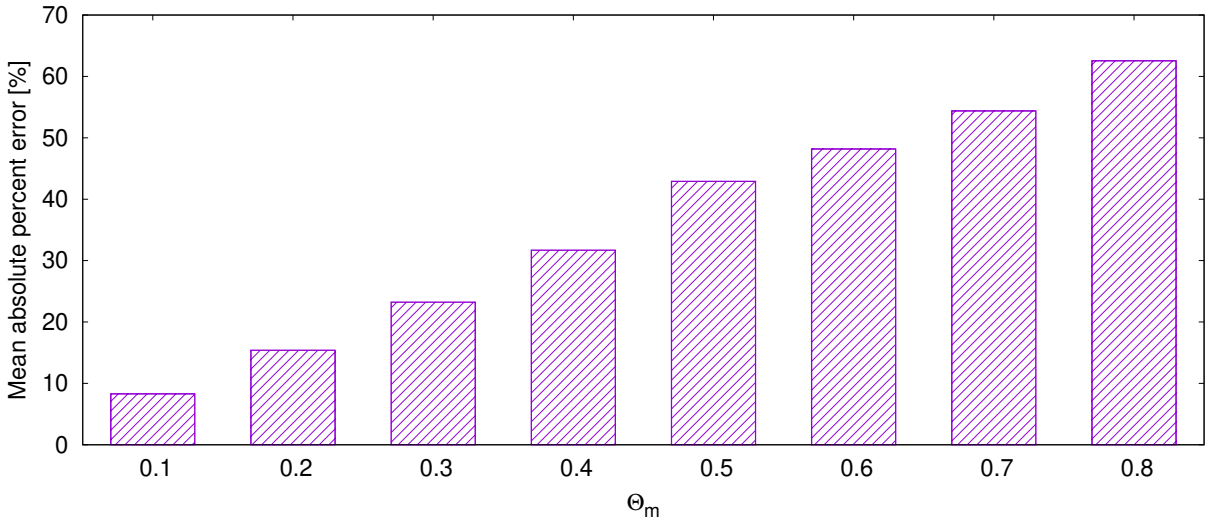


Figure 4.5: Impact of task dropping on accuracy loss: the trend of mean absolute error.

Graph Analysis Jobs

We run the triangle count algorithm implemented by the Spark's `graphx` [150] library. The input dataset consists on the public Google web graph [151], with 875.713 nodes and 5.105.039 edges. There are three types of jobs:

1. to build the edge RDD;
2. to build the vertex RDD;
3. the triangle count itself, composed by six ShuffleMap stages and one Result stage.

Differential Approximation

We specifically consider scenarios of two and three levels of job priorities, with different characteristics, *i.e.*, job sizes, arrival ratios across priorities, and overall system load. As for the accuracy loss, we compute the relative errors offline under different task dropping ratios for both workloads considered as shown in Figure 4.5. The mean absolute error in percentage increases sub-linearly with dropping ratios. When dropping 10% or 20% of map tasks, the relative errors are roughly 8.5% and 15%, respectively. Therefore, in the remainder of our evaluation section, we set the acceptable relative error to 0 for high-priority jobs and to 8.5%, 15% and 32% for lower-priority jobs. This corresponds to evaluating the latency impact of DiAS that drops 10%, 20% or 40% of tasks in lower-priority jobs.

Differential Sprinting

We use DVFS as the sprinting mechanism to change the speed of the CPU. The CPU clock frequency is initially set to 800MHz and temporarily increase it to 2.4GHz when sprinting. These frequencies were defined based on the limits supported by the machines used, which also corresponds to a common setup [152]. Sprinting reduces the execution time of high priority jobs by up to 60%, but increases the servers power consumption by 1.5x, from 180W to 270W.

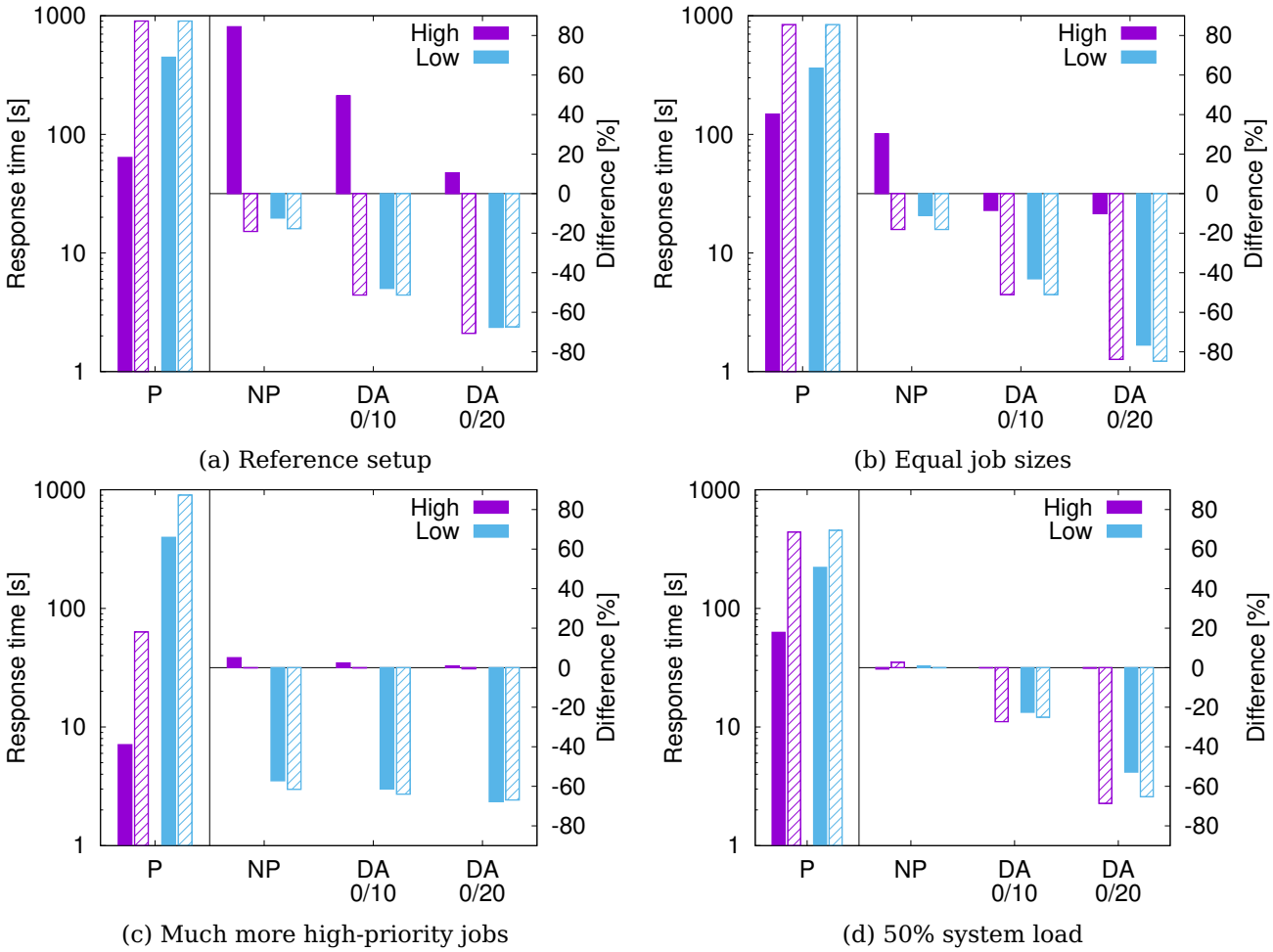
In the following, we consider two types of energy budgets:

1. limited sprinting, to sprint only 35% of the execution time of high priority jobs, and
2. unlimited sprinting, to sprint high priority jobs for their whole duration.

Following the budget strategy defined earlier, under limited sprinting high priority jobs sprint after 65 seconds while under unlimited sprinting they sprint as soon as they are dispatched.

Resource Waste

With DiAS, differential approximation and sprinting levels are applied on different priorities and lower-priority jobs are never evicted upon arrival of a higher-priority job. As a result, machine time is never wasted on reprocessing evicted jobs compared to a preemptive priority system. We define the resource



(a) Reference setup

(b) Equal job sizes

(c) Much more high-priority jobs

(d) 50% system load

Figure 4.6: Sensitivity analysis of differential approximation on mean (solid bars) and tail (shaded bars) latencies by changing the ratios between high and low priority jobs: sizes, arrival rates, and the overall system utilization.

waste as the percentage of machine time used to re-process evicted jobs compared to the total processing time.

4.4.3 Differential Approximation

Next, we evaluate the performance of DiAS under two and three priority systems.

Two-Priority System

We first show the effectiveness of differential approximation on a reference setup, highlighting the difference of mean and 95^{th} latency for both high- and low-priority jobs when compared to a preemptive and a non-preemptive priority system denoted as P and NP, respectively.

The three key parameters in the reference setup are:

1. the ratio between low- and high-priority jobs is 9 to 1;
2. the average sizes of low- and high-priority jobs are 1117MB and 473MB, respectively; and
3. the average system load is 80%

The parameters are set as close as possible to the workload characteristics of Google trace [7]. Figure 4.6 (a) summarizes the absolute results of the preemptive priority setup, and its relative difference compared to a non-preemptive priority setup, $DA_{(0,10)}$ and $DA_{(0,20)}$. The subscript pair of DA denotes the task dropping ratio for high- and low-priority, respectively. We use solid bars for the mean latency, while shaded bars are for the 95^{th} percentile latency. The resource waste is roughly 4% with the preemptive priority policy.

Under P, the mean latency of high-priority is better than the low-priority job. This stems from the unbalance in the queueing times: 0.03 seconds versus 310 seconds on average for high- and low-priority jobs, respectively. This difference is smaller for the 95^{th} percentile. When using the NP model, where preemption of low-priority jobs is not allowed, the performance of low-priority jobs improves roughly by 20% at the cost of increasing the latency of high-priority jobs by 80%. This is because high-priority jobs have to wait for the low-priority jobs in execution to finish before getting served. In contrast, $DA_{(0,20)}$ can significantly improve the performance (roughly 65%) of both mean and tail latency of low-priority at only a marginal (10%) increase in the mean latency of high-priority jobs and an accuracy loss for low-priority jobs of 15%.

We further consider a use case scenario where it is possible to tolerate a 30% accuracy loss for low-priority jobs while maintaining the latency of high-priority jobs under 100ms with no accuracy loss. The task deflator consults the results in Figure 4.5 to determine the maximum drop ratios to attain an accuracy target of 0% and 30% for high- and low-priority jobs, respectively. Likewise, the deflator runs the DiAS model (Section 4.3) and determines that a 20% drop ratio for low-priority jobs is already within the 100ms limit for the high-priority mean latency (as Figure 4.4 shows). We can thus choose to employ $DA_{(0,20)}$ to hold both accuracy and latency constraints, as confirmed by the experimental results in Figure 4.6 (a). This selection can be easily automated by assigning weights to the latency and accuracy targets to select among the feasible drop ratios.

Sensitivity Analysis
Our sensitivity analysis of differential approximation fiddles with following parameters in the reference setup one at a time:

1. high- and low-priority jobs of same size;
2. ratio between low- and high-priority jobs set 1 to 9; and
3. a total arrival rate resulting in a 50% system load.

Figure 4.6 (b-d) summarizes the results for these three scenarios. Due to the rich information embedded in the figure, we focus on comparing the latency gains of differential approximation between the reference and new setup.

Similar job size for both priorities
Comparing Figure 4.6 (b) and Figure 4.6 (a), the latency gain of differential approximation is significant, *i.e.*, for low-priority up to 80%. High-priority jobs improves too: both their mean and tail latencies have better improvement than the reference system. This can be explained by the fact that high-priority jobs have shorter waiting time here than in the reference system. In a non-preemptive setting, being in NP or DA, the maximum amount of queueing time for an arriving high-priority job is a single execution of low-priority job, assuming the high-priority queue is empty. Hence, smaller the low-priority jobs, better the gain of differential approximation used in the non-preemptive setup.

Relatively increased high- to low-priority job ratio
Comparing Figure 4.6 (c) and Figure 4.6 (a), the latency gain of differential approximation is worse. Both the mean and tail latency of high-priority increase considerably. Though the average latency gain of low-priority remains the same as the reference case, the tail latency gain decreases from 60% to 20%. As differential approximation only applies approximation techniques on low-priority jobs which account for 10% of the total jobs, its effectiveness is limited. Hence, in the scenario of dominant high-priority jobs, one shall activate approximation for both priorities.

Relatively low system loads
Comparing Figure 4.6 (d) and Figure 4.6 (a), the latency gain of $DA_{(0,10)}$ is slightly worse, but $DA_{(0,20)}$ maintains a similar gain as the reference setup. Further, there is almost no performance degradation from preemptive to non-preemptive system, shown by the results of NP. When the system load is low, *e.g.*, 50%, there is no difference between preemptive and non-preemptive priority systems because the engine is rarely occupied when higher-priority jobs arrive. The gain of $DA_{(0,20)}$ on low-priority jobs is thus mainly attributed to the reduction of processing time, instead of queueing time. Moreover, the difference between $DA_{(0,10)}$ and $DA_{(0,20)}$ can be explained by the fact that dropping 20% of tasks reaches the critical mass to drop an entire wave. Overall, thanks to flexible approximation levels and stochastic models of deflator,

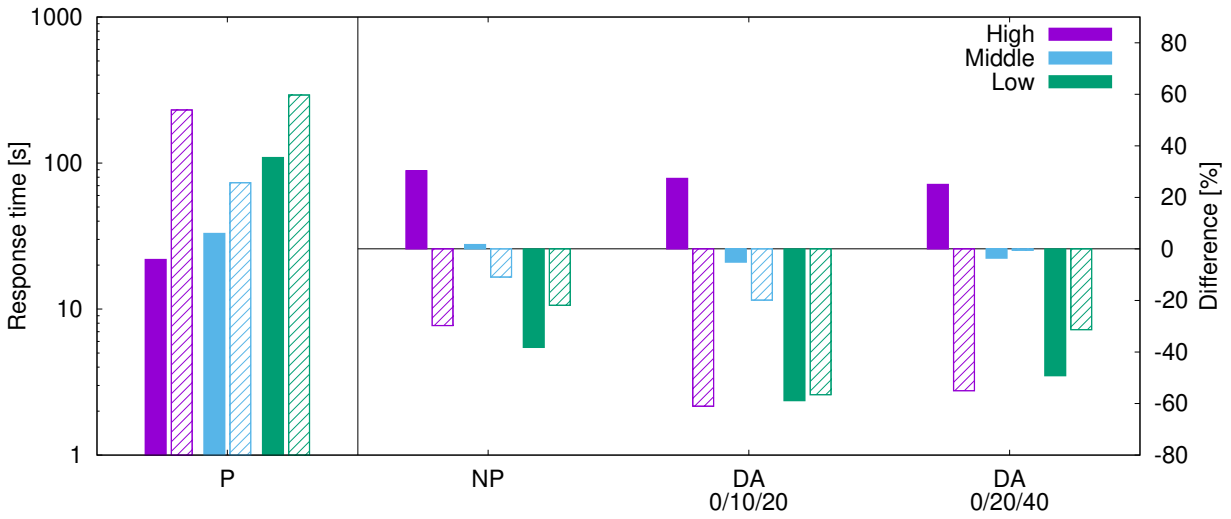


Figure 4.7: Differential approximation on three-priority system: relative difference in mean (solid bars) and tail (shaded bars) latency against preemptive priority (P).

differential approximation can effectively tradeoff analysis accuracy for improved mean/tail latencies of high/low-priority jobs, against complex system and workload dynamics.

Three-Priority System

We demonstrate the performance gains for differential approximation on a system with three priorities: high, medium and low (Figure 4.7). The total arrival rate is 2.3 jobs/min with rate ratio of high-medium-low priority of 1-4-5, resulting in roughly 80% system load. For drop rates, we use $DA_{(0,10,20)}$ and $DA_{(0,20,40)}$. According to Figure 4.5, $DA_{(0,10,20)}$ introduces 8.5% and 15% accuracy loss for medium- and low-priority jobs, respectively. Moreover, $DA_{(0,20,40)}$ introduces 15% and 32% accuracy loss for medium- and low-priority jobs, respectively.

Similar to the two-priority scenario, we use the mean/tail latency of the preemptive priority setup as comparison baseline. The resource waste under P is roughly 16%. The remaining three setups in Figure 4.7 incur in zero resource waste, due to their non-preemptive nature. In terms of latency improvement, differential approximation is able to significantly reduce the tail latency for all three priorities by up to 60%. Differential approximation reduces the average latency more for low-priority than medium-priority. However, such improvement of differential approximation comes at the cost of slightly higher average latency of high-priority jobs and accuracy loss of low- and medium-priority jobs. In this particular setup, $DA_{(0,10,20)}$ appears to achieve the most moderate tradeoff among accuracy/latency for high/lower-priority jobs.

Differential Approximation on Triangle Count

We further illustrate the gains of differential approximation when the computation requires several map and reduce stages. Specifically, we run the triangle count algorithm implemented in `graphx` library in Spark. Task dropping in this case is performed on every ShuffleMap stage, for which we consider drop ratios {1,2,5,10,20} for the low-priority jobs. The total effective drop ratio is thus the result of applying the stage drop ratio in each stage.

Figure 4.8 displays the gains obtained with differential approximation with respect to preemptive and non-preemptive scheduling. Clearly, with fairly limited task dropping ratios (5-10%) differential approximation is able to reduce the mean latency of low-priority jobs by over 50%. Moreover, differential approximation reduces by a similar factor the tail latency of *both* high and low-priority jobs.

4.4.4 Differential Approximation and Sprinting

Finally, we evaluate the complete design of DiAS, applying different CPU sprinting on the high priority jobs and approximation on the low priority jobs. We consider graph analytics jobs, which has high and low priorities of the same job size with a ratio of 3 to 7. We experiment the CPU sprinting policy under two different energy budgets resulting in two different scenarios. In the first scenario, *i.e.*, limited sprinting shown in Figure 4.9 (a), we consider a sprinting budget of 22kJ which roughly limits the jobs to run in

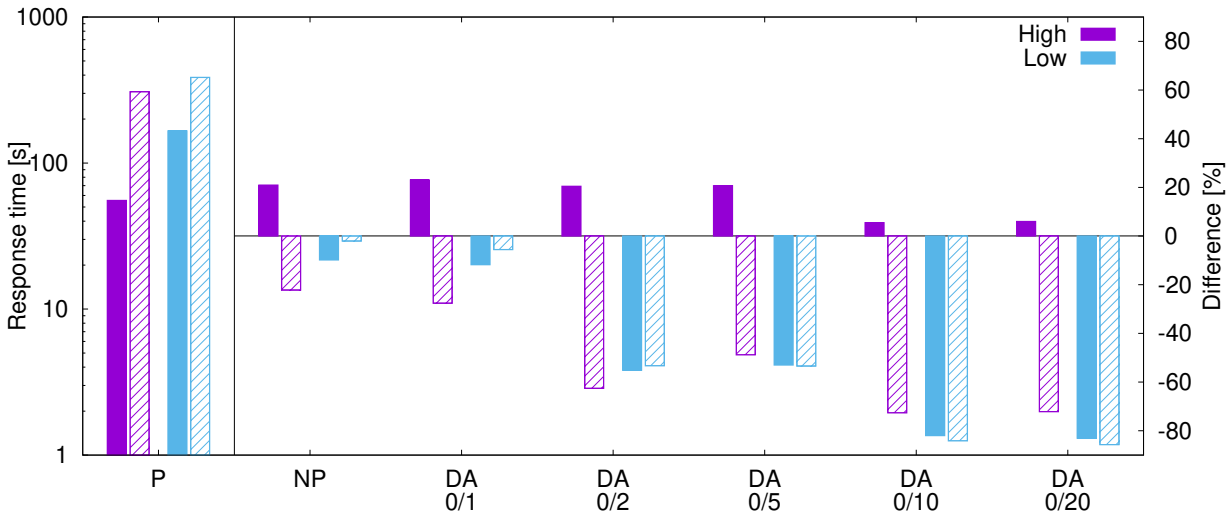


Figure 4.8: Differential approximation on triangle count: relative difference in mean (solid bars) and tail (shaded bars) latency against preemptive priority scheduler (P).

high frequency only for 35% of their execution time based on timeout. In the second one, *i.e.*, unlimited sprinting shown in Figure 4.9 (b), we set the budget high enough such that the high-priority jobs run at high frequency throughout their whole execution time. We use a non-sprinted P system as the baseline for the ease of comparison.

Latency gain

The complete DiAS of differential approximation and sprinting shows promising performance. First, the average and tail latency of *both priorities* improve under limited and unlimited sprinting budgets, ranging between 35% to 90%. Overall, the latency gain is more prominent for the tail latency, low-priority jobs, and unlimited sprinting. In terms of absolute comparison, the improvement for low-priority jobs is around 90%, whereas high-priority is between 40-60% depending on the sprinting budget. We stress that few cases with increased average latency of high-priority jobs observed in Section 4.4.3 are effectively countered by enabling differential sprinting. The performance gains are therefore more consistent for both tail and average latency, compared to the approximation-only results.

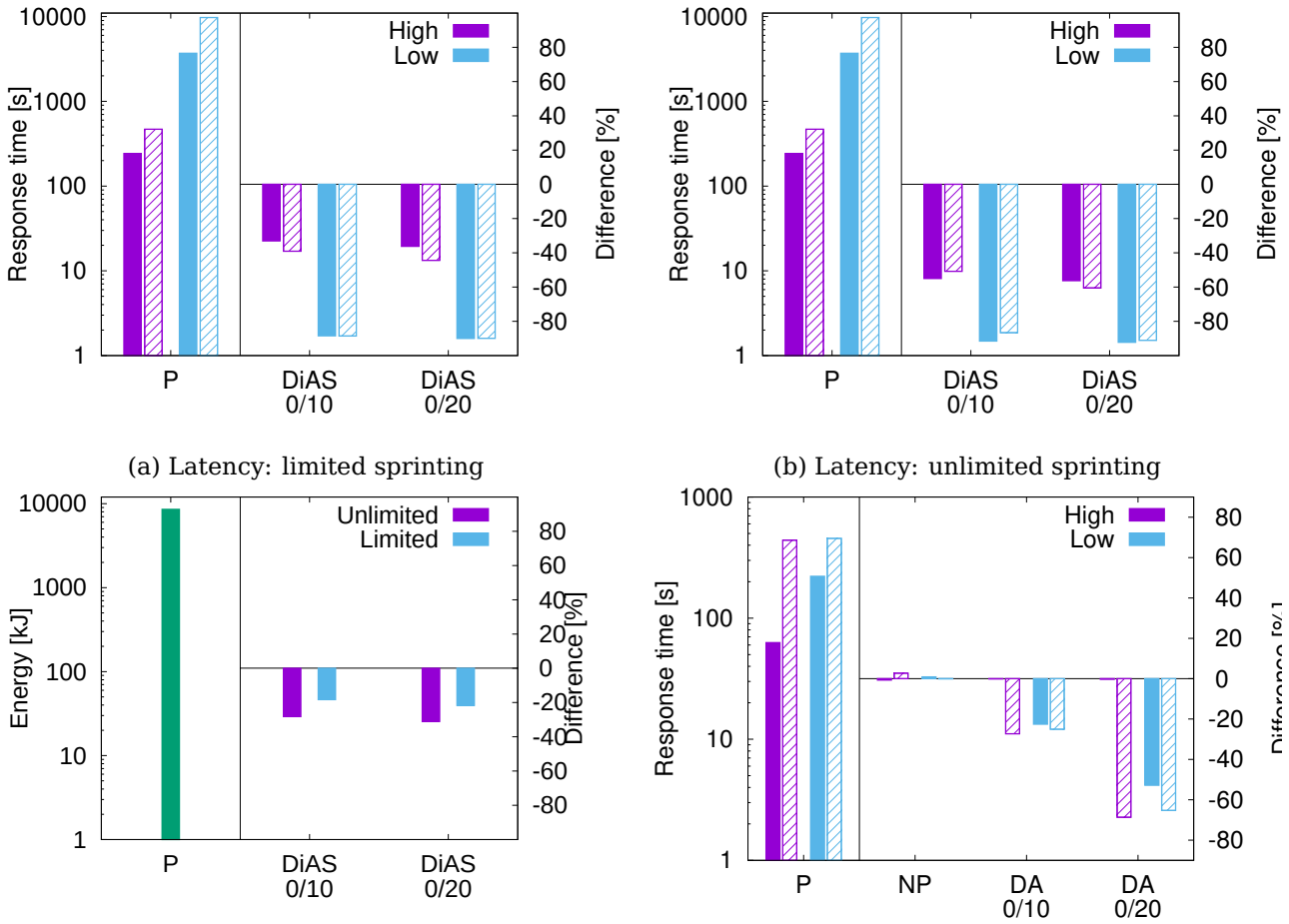
Despite the focus of sprinting being on high-priority jobs, the response times of low-priority jobs are also indirectly improved. When compared to differential approximation only, the average response time of DiAS$_{(0,20)}$ increases up to 55% for high-priority jobs but also up to 40% for low priority jobs. Similarly, for DiAS$_{(0,10)}$ the increase goes up to 50% and 53% for high- and low priority jobs, respectively. That is, by reducing the processing time of high-priority jobs, the queueing time of low-priority jobs is reduced, directly affecting the response time for both types.

Latency decomposition

To unveil the exact performance advantage of DiAS, we zoom into the performance of the limited sprinting case and present the average queueing and execution times for high- and low- priority jobs in Table 4.2. We also apply the same sprinting policy on the non-preemptive system, termed NPS, shown in Figure 4.9 (d). Due to sprinting, the execution times of high-priority jobs are lower than the low-priority jobs by at least 25%. Because of the 20% task dropping in DiAS$_{(0,20)}$, the average execution time of the low-priority jobs is the lowest among the three policies, *i.e.*, around 131 seconds. The percentage of time that low-priority jobs occupy the system thus reduces, avoiding longer waiting time for both high- and low-priority jobs. As such, the queueing times for both high- and low-priorities are lower than NPS and DiAS$_{(0,10)}$.

Energy gain

In Figure 4.9(c), we summarize the normalized energy consumption of DiAS against the P policy. For both unlimited and limited sprinting, we temporarily increase the CPU frequency for high-priority jobs. One would expect a slightly higher energy consumption, compared to the no-sprinting baseline. Surprisingly, for both unlimited and limited cases, DiAS reduces the overall energy consumption. The energy reductions stemming from differential sprinting alone for the limited and unlimited budgets are around 15% and



(a) Latency: limited sprinting

(b) Latency: unlimited sprinting

(c) Energy comparison

(d) Non-preemptive sprinting (NPS)

Figure 4.9: Complete DiAS on triangle count: latency and energy improvement against the preemptive priority scheduler (P). In (a),(b) and (d), the mean and tail latency are in solid bars and shaded bars, respectively.

26%, respectively. We explain this result by the significant reduction in execution times that outweighs the power increase during sprinting.

The energy gain of DiAS can also be amplified by the approximation ratios, *i.e.*, $\text{DiAS}_{(0,10)}$ and $\text{DiAS}_{(0,20)}$. With unlimited sprinting the gain increase to 28.2% and 31% for $\text{DiAS}_{(0,10)}$ and $\text{DiAS}_{(0,20)}$, respectively. Similarly, for limited sprinting we observe 18.3% and 21.6%. Higher reductions are observed for higher drop rates, as dropping reduces the computational load on the cluster. Overall, the design of combining differential approximation and differential sprinting can improve both the latency of both priorities and energy consumptions across diversified systems scenarios.

We suggest the following procedure to determine the static threshold used in the algorithm. To utilize the proposed models to predict the performance, one needs to first obtain the input parameters of the proposed models through workload profiling, that quantifies the relationship between file size and execution time under a constant CPU frequency. Then, one can exhaustively search through different combinations of dropping ratios, priorities, and frequency thresholds. Our proposed models can estimate the latency of such large combinations quickly. The values that optimize the tradeoff are then selected for a given set of workloads. We note that such searching procedure needs to be evoked upon every workload changes. DiAS considers a scenario where the workload set is given and hence only consider the static threshold.

4.5 Summary

In this chapter, we proposed a novel design of differential approximation and sprinting, DiAS, to trade off the accuracy and additional sprinting capacity for improving the efficiency of big data engines (*i.e.*,

Table 4.2: Average queueing and execution times of high- and low-priority jobs under sprinted non-preemptive scheduling (*NPS*), $\text{DiAS}_{(0,10)}$ and $\text{DiAS}_{(0,20)}$.

	NPS		**DiAS**$_{(0,10)}$		**DiAS**$_{(0,20)}$	
	Queue [s]	Exe.[s]	Queue [s]	Exe.[s]	Queue [s]	Exe. [s]
High	70.6	99.8	70.0	100.2	**55.1**	99.4
Low	378.9	148.5	286.42	139.0	**238.0**	**131.1**

reduction of mean/tail latency without resource waste). The design of DiAS supports different types of analyses and multiple priorities, and is compatible with existing MapReduce based processing engines that provide approximation mechanisms (*e.g.*, task dropping, and dynamic frequency scaling).

We have derived stochastic models to guide the control of approximation and sprinting levels of DiAS. We implemented the prototype of DiAS atop of Spark, with examples of text and graph analytics. Our extensive evaluation results showed that DiAS consistently reduces the mean/tail latency of both low- and high-priority jobs (by up to 90% and 60%, respectively) at roughly 15% relative error in the accuracy of low-priority jobs and more than 20% energy reduction, compared to state-of-art preemptive schedulers.

In summary, our main contributions were as follows:

1. we extended the idea previously presented in Chapter 3 to support multi-tenant scenarios by leveraging user priority and sprinting techniques;
2. based on that, we put forward a first of its kind design for differential approximation and sprinting that preserves the latency advantage of high-priority jobs and reverts the latency disadvantage of low-priority jobs for both mean and tail latencies;
3. we derived bottom-up stochastic models that capture the dynamics of big data jobs (at both the task and the stage levels) that implement different approximation and sprinting levels;
4. we implemented a prototype of DiAS on top of Spark, the state-of-the-art big data processing engine, by building a model-based job deflator and augmenting Spark with the approximation capability of dropping tasks; and
5. DiAS efficiently combined multiple knobs (task dropping, sprinting, and scheduling) to achieve significant latency and energy reduction from the state of the art.

With that, we conclude the proposed optimizations from the cloud providers point of view. Next, we start introducing applications which could benefit from such systems due to their heterogeneous characteristics and requirements. We first detail these aspects in Chapter 5 followed by Chapter 6 and Chapter 7 which go into the details of how the applications themselves could take care of leveraging such characteristics to achieve the desired optimizations.

Chapter 5

Applications Characterization

Until here, this thesis takes the perspective of cloud providers and proposes optimizations regarding the resource management which can generally be applied to every distributed application. Another possibility, however, is to leverage specific characteristics of a given application in order to achieve the same types of optimizations. For that end, in this thesis we focus on the tuning of parameters for deep learning applications which we start characterizing in this chapter.

5.1 Introduction

In typical machine learning applications, there are two ways of parallelizing the training process: model or data based. In model parallelism, each machine takes part of the model and performs a full pass on the data for that specific fraction of the model. However, as the size of training data grows, handling it on a single machine quickly becomes impractical due to memory and CPU limitations. In this case, a common practice is to rely on data parallelism, distributing the data across multiple machines and process it in parallel, facilitated by several distributed computing platforms (*i.e.*, Apache Spark [153], Hadoop [12]).

In these distributed platforms the data is represented by a directed acyclic graph (DAG) composed of multiple data-parallel operators. Spark stores data in resilient distributed datasets (RDD) [154]: each vertex of a Spark DAG represents an RDD, each edge represents an RDD transformation. The corresponding RDD operator directs from earlier to later stages in the data transformation. The DAG is submitted to the platform's scheduler to define the transformation stages and manages the corresponding computation tasks.

Once the dataset is in place, the next step is to prepare the model to use the data with. ML methods exist as distributed and parallel algorithms provided by several libraries like MLlib [155], BigDL [156], TensorFlow [157], PyTorch [158] or Caffe [159]. The most appropriate method for the domain of interest is to be chosen and applied out-of-the-box. Model training is performed automatically on the distributed platform. Once the model is trained, it is cached on each machine and can serve inference requests.

A typical distributed learning infrastructure includes a physical cluster, a distributed computing platform (*e.g.*, Spark [160]), a distributed data storage system (*e.g.*, HDFS [12]) and learning libraries (*e.g.*, MLlib [155] and BigDL [156]). Upon training, an interactive process iteratively executes the Spark jobs generated from the DML workloads. Jobs consist of a sequence of stages running several parallel and independent computation tasks. Tasks are scheduled and processed by executors, *i.e.*, processes running on the cluster nodes. For inference, the workflow is similar. The deployment of a pre-trained model across the cluster is managed by a single Spark job through its broadcast mechanism [161].

A workload running on a distributed learning system is a tuple consisting of a *learning method* and a *dataset*. Methods include classification and regression, such as linear models [162], naive Bayes [163] and decision trees [164]. Other examples are deep learning methods such as CNN [165], LSTM [166] and MLPs [167]. Datasets differ in terms of content, number of records and number of features. Their structure usually depends on the type of ML task to perform. Examples include handwritten digits images (MNIST [168]), or Higgs-boson-related measurements [169].

Table 5.1: Learning datasets.

Dataset	Description	#Records	#Features	Size
DDF (Driveface)	Images sequences of subjects while driving [173].	606	6 400	19.9 MB
GS (Drift)	Measurements from 16 chemical sensors utilized in a discrimination task of 6 gases [174].	13 910	129	40.3 MB
DHG (Higgs)	A collection of kinematic measures to detect signal processes which produce Higgs bosons [169].	11 000 000	28	7.5 GB
DN (News20)	Messages collected from 20 different newsgroups [175].	118 845	2	15 MB
DFM (fashion MNIST)	Images of fashion articles, associated with labels from 10 classes [176].	700 000	784	31 MB
DM (MNIST)	Handwritten digit images for ML research [168].	70 000	784	12 MB

Finally, these learning environments come with many configuration parameters which we classify into two main categories.

1. Platform parameters: The list of tunable parameters in distributed platforms is extensive. These parameters usually fall in one of the following types: *(i)* memory-related, *(ii)* data representation, *(iii)* scheduling, *(iv)* parallelization, and *(v)* data distribution. The two main aspects of platform parameters considered in most distributed computing systems for learning deal with parallelization degree and memory management. In Spark, one sets the number of executors an application should use, as well as the number of cores to assign to each of these executors.

2. Model hyperparameters: these impact directly the training process while being external to the model. Examples include the maximum number of iterations toward the convergence of the model [170]–[172], the maximum depth of decision trees, or the number of trees in random forests.

Next, we detail the methodology used in order to characterize these applications and the process of choosing their parameters configurations.

5.2 Trace Collection Methodology

In the following we describe the experimental setup and workloads used to collect our traces. We also detail the way we explored the configuration parameters space to understand the impact of different parameters on the workloads' performance.

5.2.1 Experimental Environment

We use Spark 2.4.0 as distributed computing platform and HDFS 2.7.7 as distributed file system. The used distributed learning libraries are MLlib (v2.4.0) [155] and BigDL (v2.4.0) [156]. We conduct our experiments on two clusters as described below.

Cluster 1. Consists of 4-nodes cluster equipped with a quad-socket Intel E3-1275 CPU processor, 8 cores per CPU, 64 GiB of RAM, 480 GB SSD drives, on a switched 1 Gbps Ethernet LAN, running Ubuntu Linux 16.04.1 LTS.

Cluster 2. To consider more hardware diversity and run larger workloads, we also deploy a 24-nodes cluster, dual-socket 8 core Intel Xeon E5-2630 CPU, 128 GB RAM, 600 GB HDD, $2\times$ 10 Gbps Ethernet, running Debian GNU/Linux 9.7.

5.2.2 Workloads

As stated earlier, workloads consist of a tuple dataset and learning method. Therefore, we detail the list of datasets and workloads consisting the workloads considered in this chapter.

Datasets. We consider six commonly used and publicly available datasets [168], [169], [173]–[176] shown in Table 5.1. We chose datasets to differ in terms of content type (*e.g.*, text, images), number of records, number of features and total size.

Methods. We test 13 state-of-art ML methods commonly used by data scientists including 9 methods implemented using MLlib and 4 using BigDL. The MLlib methods are based on different learning methods such as gradient descent, decision trees and neural networks. First, we have the algorithms K-Means (`KM`), Bisecting K-Means (`BKM`) and Gaussian Mixture Model (`GMM`) implement clustering. Then, Decision Tree (`DT`), Multilayer Perceptron (`MP`) and Binomial Logistic Regression (`BLR`) implementing classification. Liner Regression (`LR`), Random Forest Regressor `RFR` and Gradient-Boosted Tree (`GBT`) for regression. Finally, the deep neural networks include a `CNN` consisting of 9 layers, a `GRU` with 7 layers, a `LENET5` (a specific CNN for the MNIST dataset) with 5 layers and a `LSTM` with 7 layers.

In the remaining of this chapter, the names of the workloads are composed of the name of the dataset followed by the name of the learning method. For example, for the `DDF` dataset and the clustering methods we have the `DDF-KM`, `DDF-BKM` and `DDF-GMM` workloads.

5.2.3 Parameter Settings

In our experiments, we deploy each workload (*i.e.*, dataset and method) on the corresponding learning environment (*i.e.*, MLlib or BigDL) on a Spark cluster. For each deployed workload, we consider default values of hyperparameters, default Spark platform parameters, variations for hyperparameters and variations for different Spark parameters.

Hyperparameters. Based on previous literature [177]–[179], we choose hyperparameters that have a high impact on performance in terms of execution time and accuracy. These include the maximum depth of tree-based algorithms, the maximum number of iterations to reach model convergence and the learning rate and batch size of deep neural networks.

Platform Parameters. We leverage existing studies [160], [180] that evaluate which Spark parameters affect performance most. These include scheduling, data transfers, data storage and representation, parallelization and memory management. Most Spark parameters are left to their default values, except for executor memory. We changed the value from the default 1 GiB to 5 GiB to avoid out-of-memory issues. We used the same configuration values of Spark platform parameters for both MLlib and BigDL, except for executor configurations for BigDL. These configurations respect the constraints according to which the defined batch size has to be divisible by the total number of cores (*i.e.*, number of executors and the number of cores per executor are defined accordingly).

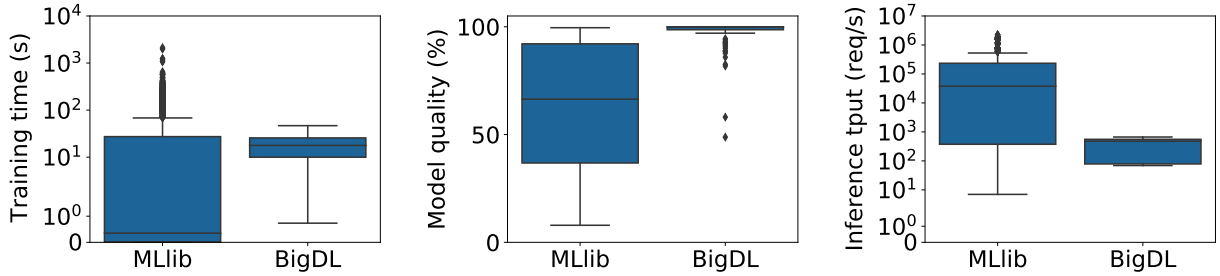
5.2.4 Metrics

We introduce here the characterization metrics at the application, platform and infrastructure level.

Application-level. Application-level metrics capture different aspects related to DML applications, including quality of the underlying model, execution times, throughput and costs. The quality of a classification model is typically measured through training accuracy, precision, recall and F1-core [181]. The *training accuracy* is the accuracy measures how well the trained model fits the training data. Some methods such as clustering algorithms (*e.g.*, K-Means) use the silhouette metric to evaluate the similarity of an object to its own cluster's objects. Other metrics used with regression algorithms include R-squared (R^2), root mean squared error (RMSE) and mean absolute error (MAE) [182].

When it comes to execution time, the *training duration* is given by the training time in seconds. We normalize the *training duration* based on the number of records and report the training duration per thousand records. We report the *inference execution time* in the form of *inference throughput* which gives the number of requests processed per second (reqs/s).

Platform-level. We rely on *SparkMeasure* [183] to collect metrics from the Spark cluster. We report the following metrics exposed by the *TaskMetrics* class: *(1)* task duration (ms) which corresponds to the total time to perform the task; *(2)* task deserialization time (ms) which corresponds to the time spent to deserialize a given task; *(3)* shuffle time (ms), *i.e.*, the time spent by tasks waiting for data to become available from remote machines; and *(4)* JVM garbage collection time (ms).



(a) Training time (b) Model quality (c) Inference tput.

Figure 5.1: Distribution of application-level metrics.

Infrastructure-level. For the infrastructure, we collect from each cluster node the CPU usage (extracted from `/proc/stat`), the memory usage (from `/proc/meminfo`), the network traffic (using `/proc/net/netstat` and filtering sent and received bytes) and the energy consumption. The power measurements collected every second via the processor counter monitor API [184].

5.2.5 Traces

We have harvested application-level, platform-level and the underlying infrastructure-level metrics. Our traces have been collected on both clusters 1 and 2. They consist of 16.2 GiB of data with more than 80 millions records. The traces and their detailed description are available on a public archive for the research community [185]. Throughout this section, we employ box-and-whiskers plots to report the statistical distribution of our metrics. For each metric, the values are grouped by the ML library used in the experimentation (MLlib or BigDL) and by the learning phase (training or inference).

Application-level. Figure 5.1 reports the statistical distribution for three of the collected application-level metrics, *i.e.*, training time, accuracy and inference throughput.

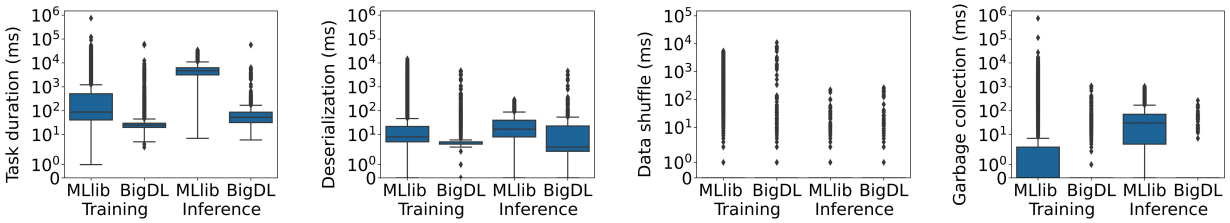
Figure 5.1 (a) shows ML model training times normalized for a training set of 1,000 records. In our experiments, MLlib exhibits high variation in training times. Workloads with very short training times (shorter than 0.4 s) represent 50% of the cases for MLlib whereas in 25% training times are between 25 s and 30 min. BigDL workloads' training times are more stable and span between 4 s and 47 s, We explain this by the greater heterogeneity of the datasets we used in our MLlib workloads as compared to the ones we used for BigDL workloads.

Figure 5.1 (b) gives the models quality metrics: accuracy for classification models, R^2 for regression models and silhouette for clustering models. The median model quality for MLlib workloads is 66.4%, and up to 99.5%. For BigDL, it is between 98% and 100% for 75% of the workloads.

Finally, Figure 5.1 (c) shows that the median inference throughput of our BigDL workloads (476 reqs/s) is less than that of MLlib workloads (around 37,747 requests/s). This corresponds to the fact that classical ML inference is cheaper and more time-efficient than deep-learning-based inference. The inference in deep learning methods consists in a complete pass through the neural network, and its cost is proportional to the network complexity.

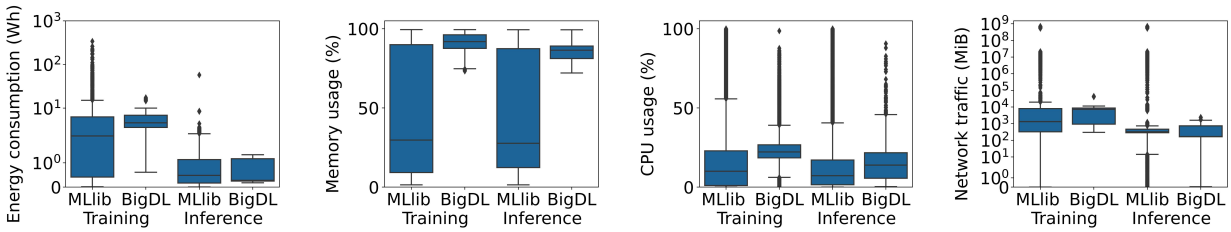
Platform-level. Figure 5.2 represents the distribution of 4 Spark metrics: *(a)* task duration *(b)* deserialization, *(c)* data shuffling and *(d)* garbage collection. Task serialization/deserialization is used upon assignment to and loading tasks by executors. Data shuffling characterizes data movements across executors and cluster nodes. Results include both the training and the inference phases.

As shown in Figure 5.2 (a), short tasks (in the 1–100 ms range) are very common in BigDL workloads with up to 79% of the tasks. Less common in the MLlib case, they are up to 50% for MLlib-Training, and even less for MLlib-Inference where 75% of tasks last at least 3,000 seconds. Longer tasks are more frequent in the inference phase for both MLlib and BigDL. Tasks that last more than 100 ms represent 21% of BigDL and up to 98% of MLlib inference tasks. Respectively, they represent only 3% of BigDL and 47% of MLlib training tasks.



(a) Task duration (b) Deserialization time (c) data shuffling (d) Garbage collection

Figure 5.2: Distribution of platform-level metrics



(a) Energy consumption (b) Memory usage (c) CPU usage (d) Network traffic

Figure 5.3: Distribution of infrastructure-level metrics.

Task deserialization (Figure 5.2 (b)) is very fast (*i.e.*, less than 10 ms) in 75% of BigDL-Training tasks, and in 25% to 50% for the other tasks. Longer deserializations, between 10 ms and 10,000 ms, represent 75% of MLlib-Inference tasks but at maximum 50% of the tasks in MLlib-Training, BigDL-Training and BigDL-Inference.

As shown in Figure 5.2 (c), data shuffling is negligible for 75% of all tasks, however, in some cases, it may be very costly and may reach up to 10,000 ms for training tasks and up to 100 ms for inference tasks, which is equal to the whole duration of some tasks as shown in figure 5.2 (a).

Finally, Figure 5.2 (d) shows that JVM garbage collection is very fast for BigDL-Training and BigDL-Inference tasks, *i.e.*, less than 1 ms) in 99% of them. Longer garbage collection, *i.e.*, more than and 100 ms, represent 75% of MLlib-Inference tasks but only 25% of the tasks in MLlib-Training.

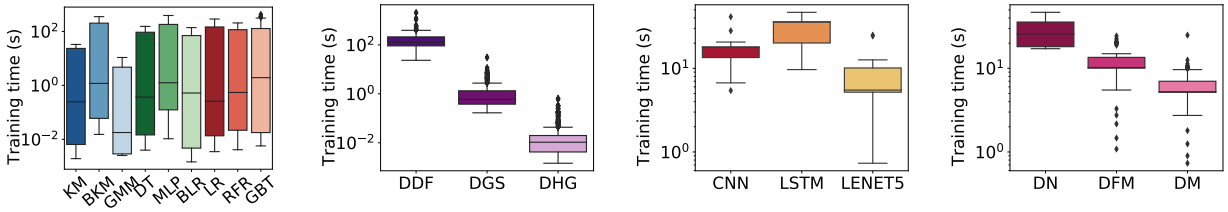
Infrastructure-Level. The infrastructure measurements concerning energy consumption, network traffic, CPU and memory usage are shown in Figure 5.3. If we consider energy consumption, Figure 5.3 (a) indicates that up to 25% of MLlib workloads consume very little energy *i.e.*, less than 0.4 Wh. BigDL-Inference executions consume at least 0.2 Wh and BigDL-Training at least 0.6 Wh. However, BigDL workloads consume at most 17 Wh whereas MLlib workloads' consumption reaches up to 340 Wh.

Regarding memory usage, Figure 5.3 (b) shows that our BigDL workloads are memory-intensive with at least 70% of memory usage reported by all measurements. CPU usage does not exceed 30% in 75% of all measurements (Figure 5.3 (c)) This indicates that our workloads in particular are memory-bound and not CPU-bound.

Finally, Figure 5.3 (d) shows collected measurements of network traffic. We observe that inference for both MLlib and BigDL workloads involves the lowest network traffic: 75% of inference executions consume at most 800 MiB. In contrast, around 50% of measurements in MLlib- and BigDL-Training correspond to a network traffic that exceeds 1 GiB. This happens because, after each iteration of training, the workers have to exchange local data in order to build the model.

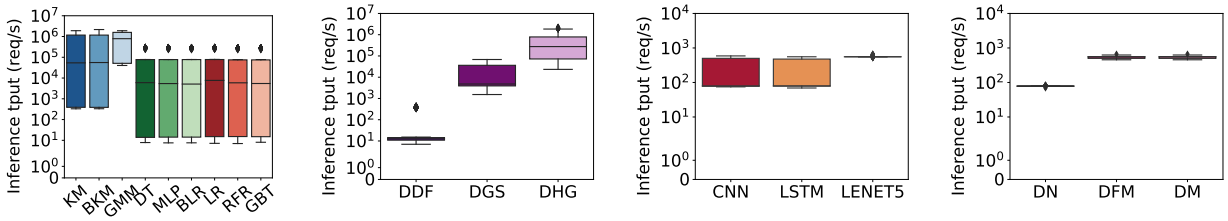
5.2.6 Datasets vs. Methods

To complete our analysis, we study how the nature of the datasets and the type of learning methods impact performance. Figures 5.4 and 5.5 represent performance variations. Subfigures (a) and (c) focus on variations due to the use of different datasets with the same learning method. Subfigures (b) and (d) report variations when working with the same dataset but with different methods.



(a) MLlib methods (b) MLlib datasets (c) BigDL methods (d) BigDL datasets

Figure 5.4: Training time variability within same DML method (normalized) vs. within same dataset.



(a) MLlib methods (b) MLlib datasets (c) BigDL methods (d) BigDL datasets

Figure 5.5: Inference throughput variability within same learning method vs. within same dataset.

Considering training times normalized to 1,000 records, Figure 5.4 (a) shows orders-of-magnitude variations for the methods. For instance, the minimum and maximum quartiles in the BLR case span up to 5 orders of magnitude. On the other hand, when we plot our measurements grouped per datasets (Figure 5.4 (b)), distributions are quite *compact*. The same effect is observed with inference throughput (Figure 5.5).

Considering these results, we observe that there are more performance variations between workloads with different datasets using similar learning methods than between workloads with different methods on similar datasets. Essentially, to efficiently configure the distributed computing platform, the specificities of data are more important than those of the method used.

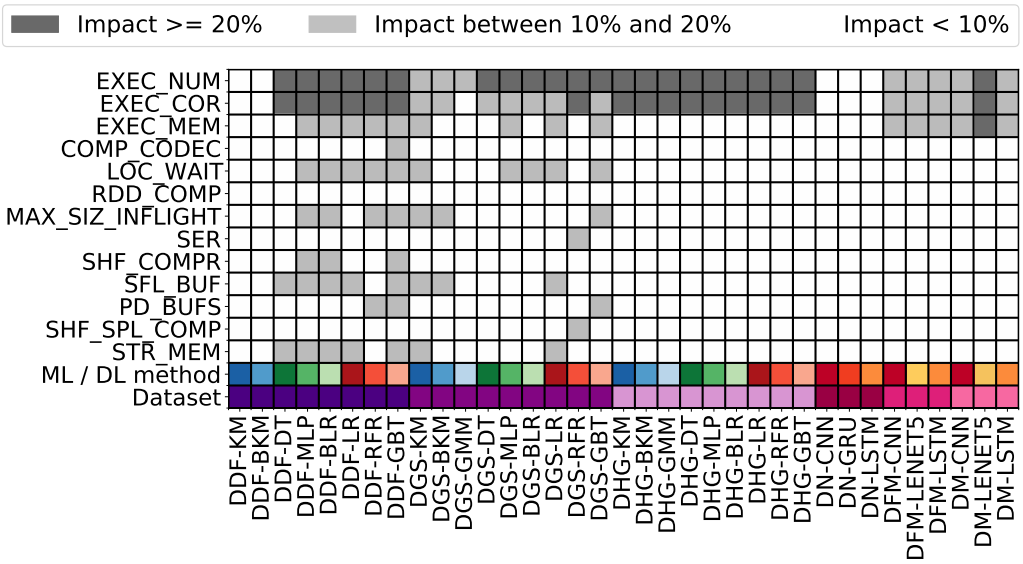
5.3 Parameter Tuning

To characterize the sensitivity of workloads to different configuration parameters and different configuration strategies, we start by studying the effect of varying only platform parameters and of varying only hyperparameters. We then compare single-level vs. multi-level parameter configuration strategies and analyze their behavior using different distributed learning workloads. Finally, we perform a multi-objective analysis in the context of AI-as-a-Service.

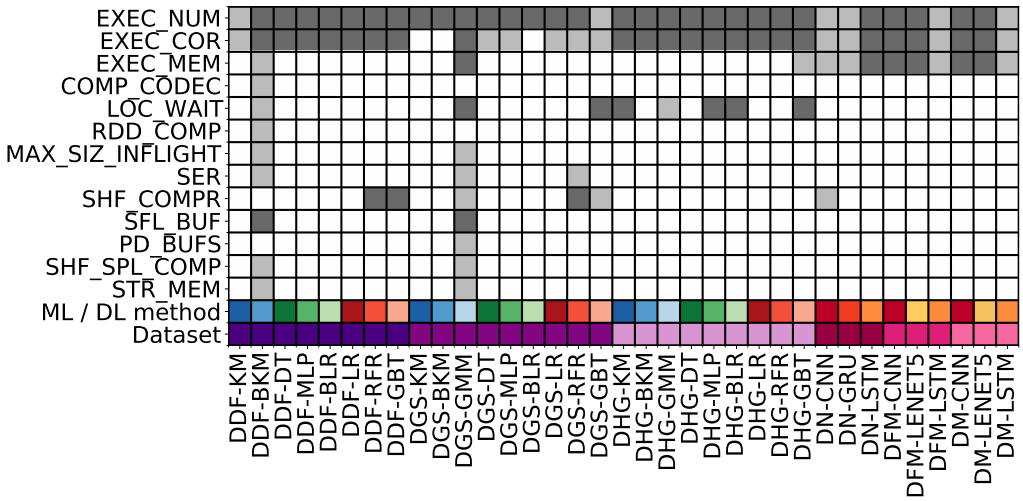
5.3.1 Platform Parameters

We characterize the impact of platform parameters on training time and inference throughput by varying each parameter individually for each workload. For each platform parameter, we measure the relative variations obtained in performance, *i.e.*, the difference between the lowest and highest performance, when different values of that parameter are considered. We then distinguish parameters with high impact on performance if the corresponding variation exceeds 20%. Parameters which incur variations between 10% and 20% have medium impact. Finally, parameters producing variations of less than 10% are the ones with low impact.

Figures 5.6 (a) and 5.6 (b) present our results using a heat map representation. For each platform parameter (vertical axis) and workload (horizontal axis), we use a three-color scheme to indicate its impact, from highest (dark grey in the figure) to the lowest (white). We see that for the majority of the workloads, the number of Spark executors (`EXEC_NUM`) and the number of cores per executor (`EXEC_COR`) play an important role. Intuitively, they improve parallelization and thus performance, in particular for the training phase.



(a) Training time.



(b) Inference throughput

Figure 5.6: Impact of platform parameter on performance (dark-grey for high: $> 20\%$, light-grey for medium: $10\% \leq 20\%$, white for low: $< 10\%$).

Moreover, Figure 5.6 (a) shows that for ensemble learning methods (*e.g.*, random forests and gradient-boosted trees) on datasets with many features (from 100 and beyond) the training phase triggers frequent shuffle operations (`SHF_COMPR` parameter). Examples of workloads are `DDF-RFR`, `DDF-GBT`, `DGS-RFR` and `DGS-GBT`, they have the highest impacts of `SHF_COMPR` (impact >19%) among all workloads. One can tune Spark's shuffle compression parameter to reduce the amount of data during shuffle. This will reduce network traffic and therefore reduce training time.

Our large datasets highlight that `LOC_WAIT` (the timeout after which a data-local task is launched on a distant node) can significantly affect the training time. Indeed, waiting for a nearby node to be available to do the job would avoid making huge data transfers and shuffles that consume time and bandwidth, and thus would significantly impact efficiency. Thus, jobs that deal with large amounts of data benefit from increasing the `LOC_WAIT` parameter. We observe this phenomenon in particular in Figure 5.6 (a), with the Higgs dataset (`DHG`) and four methods for clustering and classification (`DHG-KM`, `DHG-MLP`, `DHG-BLR` and `DHG-GBT`).

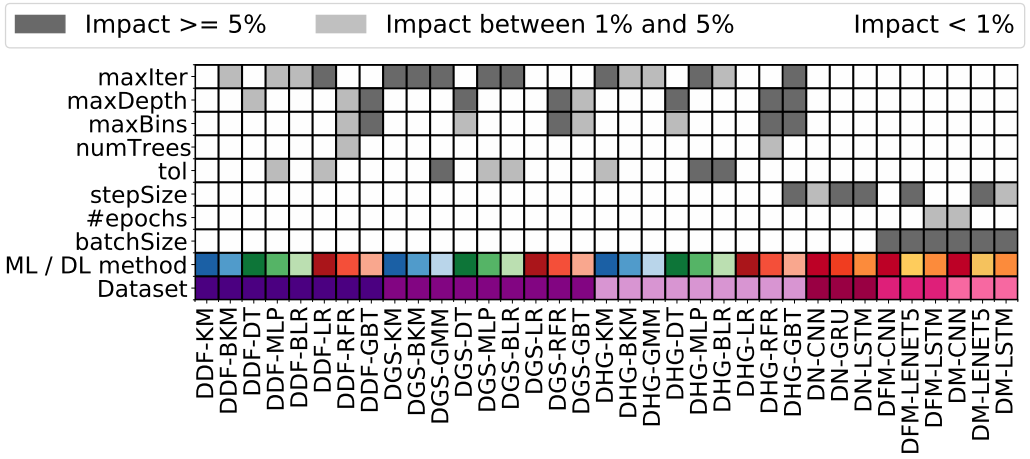


Figure 5.7: Impact of hyperparameter on performance (dark-grey for high: $> 5\%$, light-grey for medium: $1\% \leq 5\%$, white for low: $< 1\%$).

5.3.2 Hyperparameters

Now we investigate the impact of hyperparameters on our models quality (*i.e.*, accuracy for classification tasks, R^2 coefficient for regression tasks and silhouette score for clustering tasks). We vary individually several hyperparameters including the number of iterations, the number of classes for discretization of continuous variables, the depth of decision trees and others. As for platform parameters, for each hyperparameter we define its impact by the relative performance variations obtained when varying that hyperparameter.

In Figure 5.7 we distinguish high-impact hyperparameters when the variations exceed 5%, medium-impact parameters when variation is between 5% and 1%, and, finally, low-impact parameters when variations are less than 1%. Note that empty (*i.e.*, white) cells indicate that the corresponding hyperparameters have low impact or are irrelevant for the target learning methods. We observe that the *maxIter* hyperparameter, *i.e.*, the parameter giving the number of iterations for a given learning method, has high impact on the quality of several methods like `BKM` (Bisecting K-means used in the `DDF-BKM`, `DGS-BKM` and `DHG-BKM` workloads) and `MLP` (Multi-layer Perception, with `DGS-MLP` and `DHG-MLP` workloads). Further, *maxDepth* and *maxBins* are also impactful hyperparameters for decision-tree (`DT`) methods.

For the BigDL workloads considered here, the number of epochs does not significantly affect the methods accuracy. Instead, the *stepSize* and *batchSize* hyperparameters affect several BigDL workloads. Finally, we observe that for many MLlib and BigDL methods (*e.g.*, `KM`, `BKM`, `GMM`, `RFR`, `GBT`, `DT`, `LENET5`), the learning method is always impacted by the same hyperparameters, and this holds for any given dataset.

5.3.3 Multi-Level Tuning

Next, we compare the impact of single-level vs. the impact of multi-level parameter configuration strategies on workloads performance in terms of training time, inference throughput and model quality. To do so, we consider three different configuration strategies:

1. tuning hyper-only parameters;
2. tuning platform-only parameters; and
3. jointly tuning hyperparameters and platform parameters.

When tuning hyper-only parameters, we vary together parameters with high impact as identified in Figure 5.7. Similarly, tuning platform-only parameters involves varying together high-impact parameters from Figure 5.6. Finally, when tuning jointly hyperparameters and platform parameters, the previously considered platform and hyper parameters are varied together. Figure 5.8 presents the best raw values of training times, inference throughputs and model quality resulted from each of the three configuration strategies, for 3 different workloads. The error bars represent the 95% confidence intervals of the represented values. In some figures, the confidence interval is narrow and almost invisible on the bars due to low variation of data.

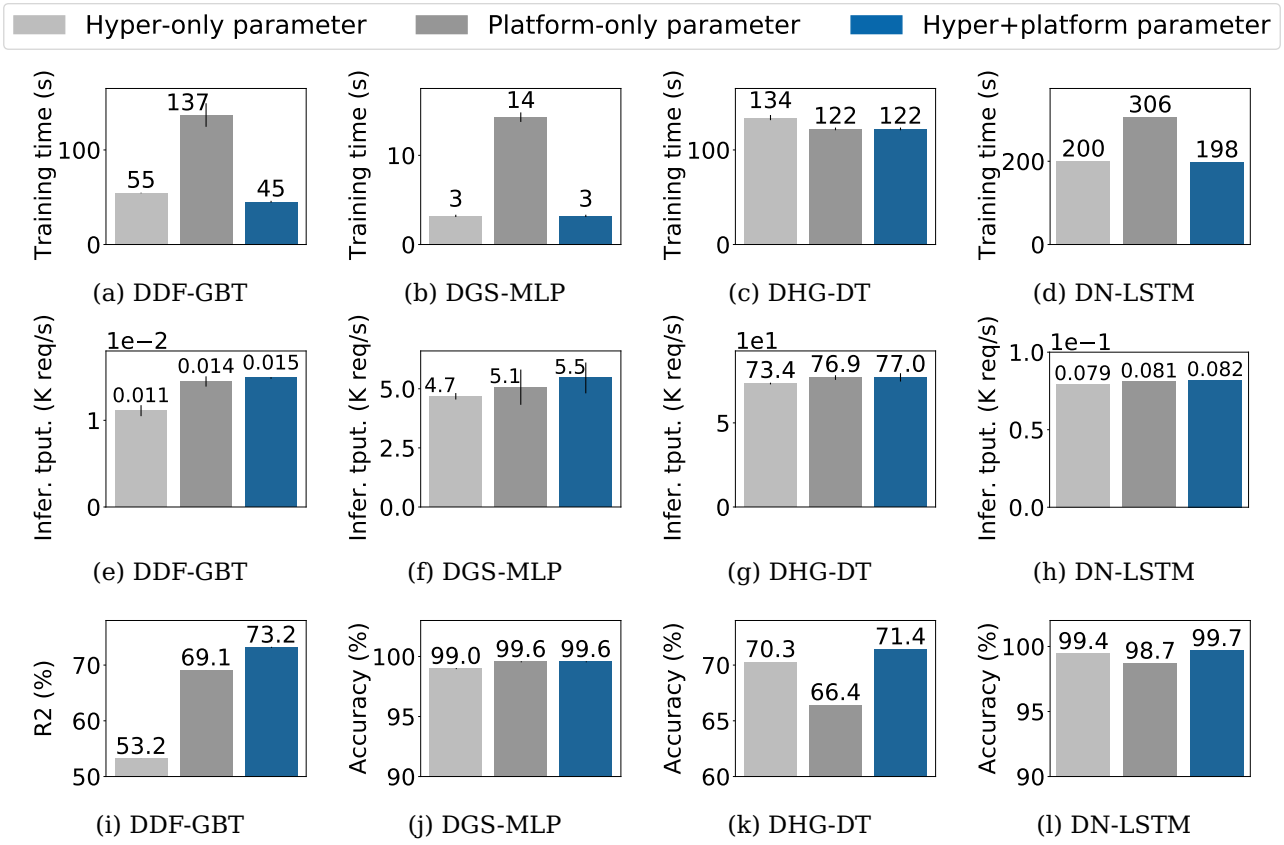
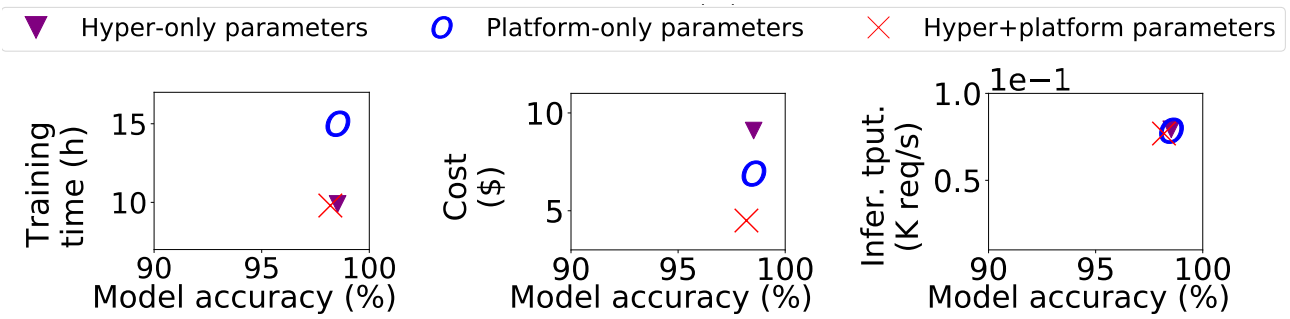


Figure 5.8: Single-level vs. multi-level configuration (training times in 1^{st} row, inference throughput in 2^{nd} row, model quality in last row).

In Figure 5.8 (a) we observe that the multi-level configuration strategy is the best configuration strategy for the training time of `DDF-GBT`. It improves training time by 71% compared to the platform-only approach (135 s) and by 27% compared to the hyper-only approach. Looking at Figures 5.8 (e) and 5.8 (i), we see that, for the same workload, it is also the best strategy for inference performance and model quality. Indeed, it improves the other approaches up to 45% for inference throughput and up to 37% for R-squared score. Multi-level configuration achieves the best improvements also for the training time of `DN-LSTM` (Figure 5.8 (d)), as well as for the inference throughput of `DGS-MLP`, `DHG-DT` and `DN-LSTM` (Figure 5.8 (f), (g) and (h)).

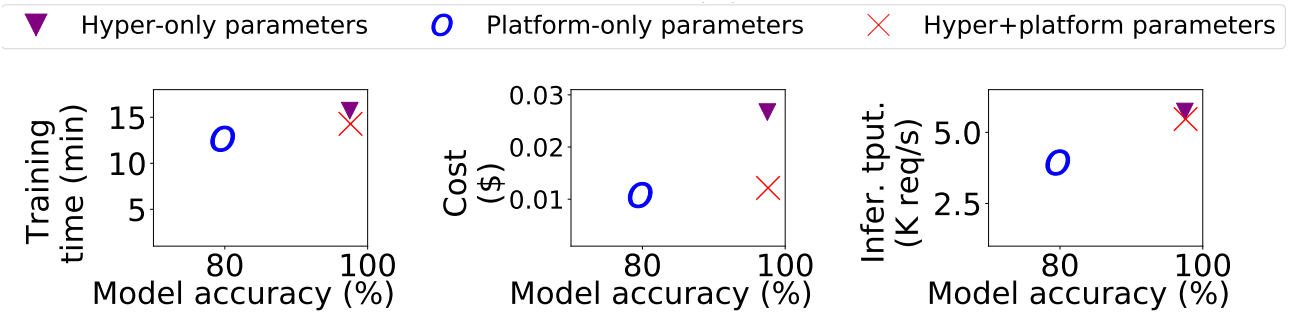
For the workloads shown in the figure, the multi-level configuration strategy achieves the best model quality. However, in some cases such as `DDF-GBT` and `DGS-MLP`, the platform-only approach is better than the hyper-only approach. For `DDF-GBT` the improvement goes up to 28% while for `DGS-MLP` it is up to 0.6%. This observation is explained by the way Spark manages data partitioning and jobs parallelization. Indeed, in Spark, the training data is distributed across partitions whose number is equal to the number of available cores. The different data partitions are used by parallel tasks for the training iterations. At the end of each iteration, the task's intermediate results are aggregated. As a consequence, changing the parallelization, *i.e.*, the number of cores, changes the data partitioning which in turn may change the intermediate results and thus impact the final model.

Given this analysis, we could conclude that the multi-level configuration strategy leads to the exploration of new configurations that outperform single-level configuration strategies. Besides of multi-level tuning, we have to consider that each user can have different objectives which could lead to different requirements. The multi-level configuration also allows us to tackle the multi-objective scenario more efficiently since it opens a lot of space for optimizations from different perspectives. In the next section we deep dive into this concept.



(a) Quality *vs.* train. time (b) Quality *vs.* cost (c) Quality *vs.* inference

Figure 5.9: Impact of different configuration strategies on different objectives (`DN-LSTM`).



(a) Quality *vs.* train. time (b) Quality *vs.* cost (c) Quality *vs.* inference

Figure 5.10: Impact of different configuration strategies on different objectives (`DGS-DT`).

5.3.4 Multi-Objective Tuning

In this section we consider examples of DML workloads in the context of AI-as-a-Service. We consider the problem of a service operator who needs to train a dataset of 2M records and searches for a configuration with the following threefold objective:

1. it provides the best accuracy;
2. it minimizes training time; and
3. it results in a minimal training cost.

We use this use-case to apply the previously presented strategies (platform-only parameters, hyper-only parameters and hyper+platform parameters) to AI-as-a-Service configuration and compare their impact on the actual service performance and cost. We consider an AI-as-a-Service as if deployed in Amazon EC2 (N. Virginia) [186]. Following AWS pricing scheme, we consider computing instances similar to our experimental setup, *i.e.*, 8 vCPU with 64 GB of memory, billed $0.46/hour. The data transfer is charged for outbound traffic for $0.05 per GB.

Figure 5.9 illustrates the case of the `DN-LSTM` workload. It compares the three configuration strategies in terms of model accuracy, training time (a), training cost (b), and inference throughput (c). In this specific example, for the strategy configuring hyper-only parameters, we include both `number of epochs` and `batch size` hyperparameters. For the strategy focusing on platform parameters, we consider the Spark parameters concerning the numbers of executors and cores. Finally, we use all four parameters for the configuration of both hyperparameters and platform parameters.

Here, the goal is to achieve at least 80% of model accuracy with the shortest possible training, and the cheapest possible training costs. We observe that the joint configuration strategy leads to an average model training time that is significantly faster than the platform-only strategy. It also has the best training cost as shown in Figure 5.9 (b). When it comes to model inference, the differences are negligible (Figures 5.9 (c)) as we do not tune any inference specific parameter in this scenario.

Figure 5.10 considers the same analysis but in the case of the DGS-DT workload. We consider *maxDepth* and *maxBins* hyperparameters and the same (executors and cores) platform parameters. Here we can see that, compared to the other two strategies, the platform-only parameters strategy has shorter training time but much lower accuracy. On the other hand, the hyper-only parameter and the multi-level configurations have similar accuracies, training times and inference throughputs, but the latter optimizes resources costs up to 54%, as shown in Figure 5.10 (b).

This last analysis shows that the combination of hyperparameter and platform parameter configurations allows the multi-objectives to be addressed in a more goal oriented manner. Moreover, tradeoffs such as cost and performance can easily be considered once the user's requirements are known. In the upcoming chapters, we present strategies to leverage these multi-level and multi-objective characteristics of such applications in the context of auto parameter tuning servers.

5.4 Summary

In this chapter, we performed a comprehensive study of the joint impact of both hyperparameters and platform parameters on different types of learning workloads. We have considered three strategies of tuning: only hyperparameters, only low-level platform parameters, and jointly tuning both. We studied the performance metrics in isolation as well as their trade-offs.

Moreover, we collected and released (see [185]) traces from our extensive experiments leveraging two popular learning libraries (*i.e.*, MLlib [155] and BigDL [156]) atop two Spark clusters [153]. Our statistical analysis can help the future development of modeling and simulation tools of learning workloads, as well as tools for synthetic trace generation.

Finally, we derived several observations and key takeaways concerning the characteristics of learning workloads. Based on these observations, we next focus on the automatization process of exploring these different levels of parameters, metrics, and objectives. Motivated by our finding in this chapter, we propose optimizations in the context of parameter auto tuning servers which leverage the characteristics of learning applications presented here.

Chapter 6

Use Case: PipeTune

In this chapter, we dive into the details of one specific use case application for the scenarios we discussed so far. For that, we take the context of DNN learning jobs which are common in today's clusters due to the advances in AI driven services such as machine translation and image recognition. The most critical phase of these jobs for model performance and learning cost is the tuning of hyperparameters. Existing approaches make use of techniques such as early stopping criteria to reduce the tuning impact on learning cost. However, these strategies do not consider the impact that certain hyperparameters and systems parameters have on training time. This chapter presents PipeTune, a framework for DNN learning jobs that addresses the trade-offs between these two types of parameters. PipeTune takes advantage of the high parallelism and recurring characteristics of such jobs to minimize the learning cost via a pipelined simultaneous tuning of both hyper and system parameters.

6.1 Introduction

Deep Neural Networks (DNN) are becoming increasingly popular, both in academia and industry [187], [188]. They are being adopted across a variety of application domains, including speech [189]–[191] and image recognition [192], self-driving vehicles [193], face-recognition [194], [195], genetic sequence modeling [196], natural language processing [197], e-health [198] and more. Several public cloud providers offer native support to deploy, configure and run them, providing tools to automatically or semi-automatically drive the DNN processing pipeline. One important factor is the choice of the DNN hyperparameters (*e.g.*, number of hidden layers, learning rate, dropout rate, momentum, batch size, weight-decay, epochs, pooling size, type of activation function, *etc.*). DNNs require careful tuning of the hyperparameters, and if done correctly, it can achieve impressive boosts in performance [199], [200]. However, misconfigurations can easily lead to wrong models and hence bad predictions [201], [202].

A naive approach to hyperparameter tuning is to perform a full exploration of the possible configuration variations. Such a tuning approach becomes quickly unpractical, costly and slow, as the number of variations grows exponentially [203]. We show this by estimating the tuning cost for 3 types of ML-optimized EC2 instances in Figure 6.1 for a small number of parameters. We take as example the tuning of a LeNet model on the MSNIT dataset and let it be tuned for different number of parameters (*i.e.*, varying from 1 to 6). In this case, each parameter was configured to take up to 3 different values. We measure the tuning

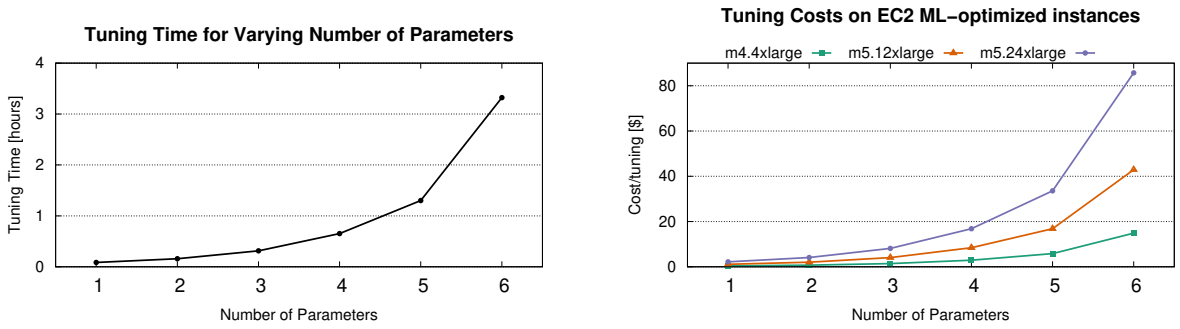


Figure 6.1: Clustering results grouped by workload type.

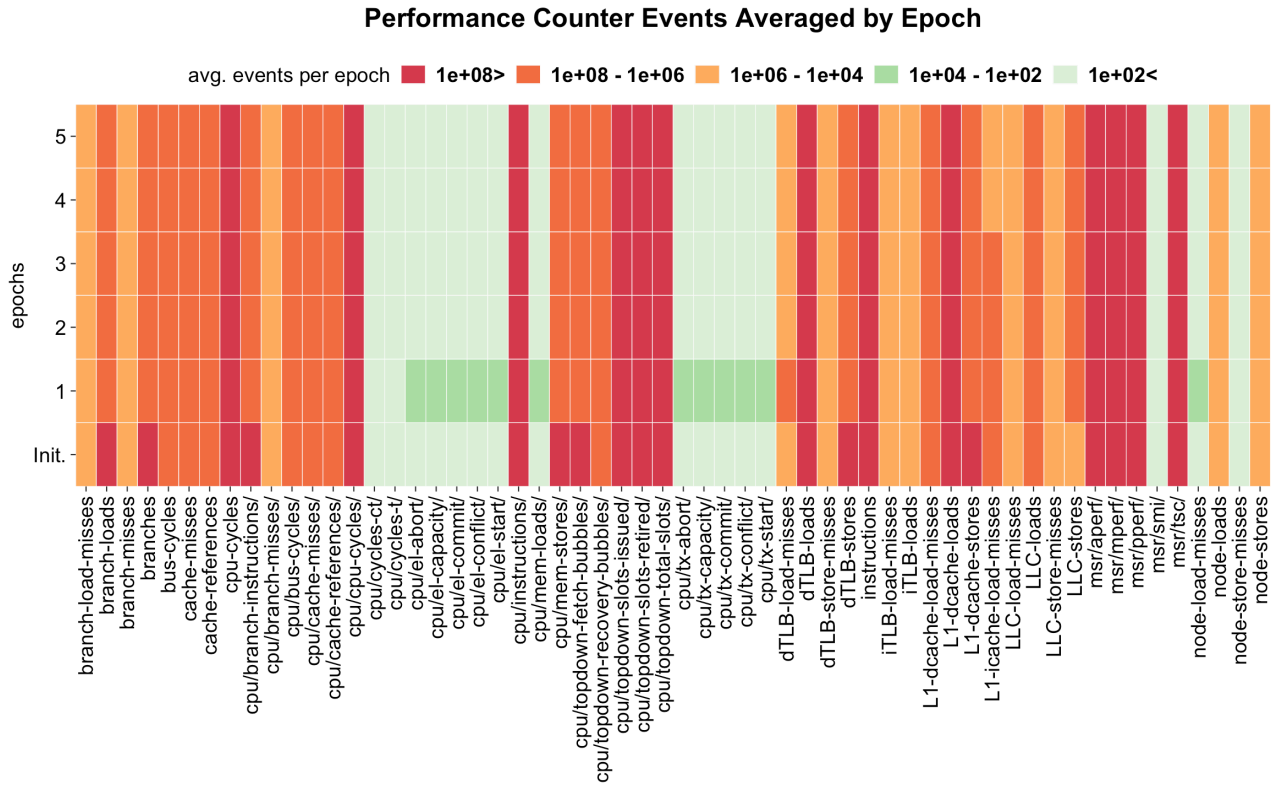


Figure 6.2: Profiling of training a CNN model on the News20 dataset [175] during the initiation phase and the 5 following epochs with 16 cores and 32GB memory.

time for each instance of this example and estimate the cost of doing so using a small, medium or large sized EC2 instance. We then observe that the cost of doing so grows exponentially with the number of parameters being tuned, becoming impractical.

Commercial platforms (*i.e.*, Google Vizier [56], Amazon SageMaker [57]), as well as on-premises solutions (*i.e.*, Auto-Keras [59]) help deployers by offering tuning services to mitigate (possibly avoid) misconfiguration. As a result of proper hyperparameters tuning, one should achieve fast convergence and high accuracy. Unfortunately, due to the tuning process length, this phase becomes expensive, and the situation exacerbates in cloud deployments [204]. Even using cheap cloud instances (*i.e.*, AWS EC2 Spot instances [205], as suggested for instance by AWS SageMaker [206]), the process can quickly lead to budget exhaustion. We observe that some hyperparameters (*e.g.*, number of epochs, batch size, dropout) can drastically reduce training time. Importantly, training a DNN by using different system resources (*e.g.*, number of CPU cores, allocated memory, number of GPUs) lead to different results.

However, handling system parameters as one of the hyperparameters is very time consuming, requiring in-depth knowledge of the workload, and it is often an intuition-driven process. In addition, doing so would directly affect training and tuning time, and therefore state-of-the-art DNN tuning systems [88] simply ignore this opportunity. Instead, the majority of the existing tuning solutions restrict themselves to the sole hyperparameter tuning using a variety of techniques, including grid search [207], random search [90], hyperband [69], bayesian optimization [208], [209], evolutionary algorithms [210], [211], population-based training (PBT) [212], *etc.* While a possible yet naive approach to treat system parameters is to consider them as possible hyperparameters, this leads to longer training periods.

PipeTune strives to optimize both accuracy and training time of DNNs, while simultaneously tuning hyper and system parameters. The key observation of PipeTune is that the backbone of popular training

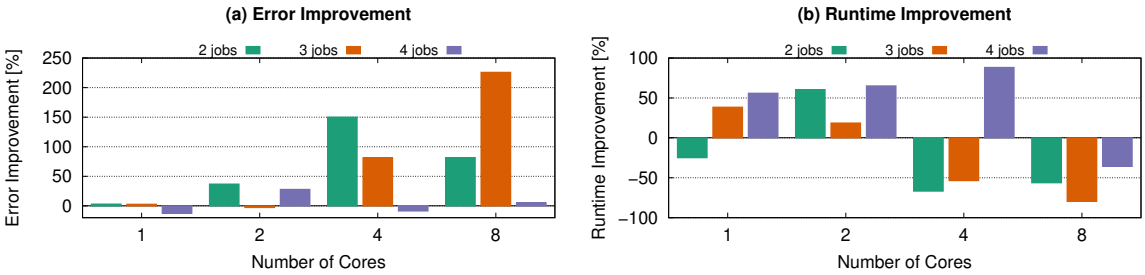


Figure 6.3: Characterizing Tune's performance under various system conditions (*i.e.*, system load, number of cores, and hyperparameters) during tuning.

algorithms for DNN is stochastic gradient decent [213], an iterative algorithm. PipeTune exploits such repetitive patterns as a unique opportunity to improve and achieve fast system parameter tuning. As an example, Figure 6.2 illustrates the typical repetitive behavior of a training process. We use a heatmap to show the hardware events happening through the training of a CNN model on the News20 dataset [175] during 5 epochs. On the y-axis we show 58 different hardware events, on the y-axis we show the initiation phase plus 5 training epochs. Each cell of the heatmap represents the average number of each event per single epoch. We see how certain events repeat throughput the epochs with the same occurrence.

Building on this observation, we design, implement and evaluate PipeTune, a middleware solution coordinating between the DNN training applications and systems. In a nutshell, PipeTune relies on low level metrics to profile the training trials on the epoch level and make quick decisions regarding the system parameters. The main research questions that PipeTune intends to answer, and the main contributions of this work are the following.

***RQ1**: Why system parameters must be taken into account in the process of DNN tuning?*

We show that by taking into account the system parameters, the overall tuning runtime can be greatly reduced while at the same time improving the model performance. Moreover, the training time can at the same time benefit from this approach, especially if the underlying system resources and their usage is exposed to the tuning phase.

***RQ2**: Can out-of-the-box hyperparameter optimization algorithms also cover system parameters?*

We show that it is possible to include system parameters in the tuning process and ask the algorithm to optimize the ratio of accuracy to performance. However, our experimental evidences (Section 7.5) highlight the following drawbacks. First, tuning runtime significantly increases (up to $\times 1.5$ in our experiments). Second, in doing so, the delicate equilibrium between performance and accuracy is negatively affected.

The remainder of this chapter is organized as follows. In Section 6.2, we present an alternative approach relying on state-of-the-art solutions and show the need for our novel approach. We present the design of PipeTune in Section 6.3. In Section 7.5, we describe our prototype implementation and present the results of our in-depth evaluation. Finally, we conclude in Section 8.5.

6.2 The "System *as* Hyperparameters" Case

The idea to consider system parameters as an additional set of hyperparameters is appealing. To verify its viability, we consider a state-of-the-art hyperparameter auto-tuning system, Tune [60], an open-source library implemented in Python supporting an extensive list of hyperparameters optimization algorithms. Note that the ideas shown next are nevertheless independent of the underlying tool used for the auto-tuning process of hyperparameters.

First, we consider two versions of Tune. In V1, it is used out-of-the-box to perform hyperparameters tuning with the objective of maximizing accuracy, without taking the system parameters into account. In this version all trials run with the same default system parameters. Then, in V2, the system parameters are included in the list of parameters to be tuned. This second version requires the resources used by

Table 6.1: Accuracy, training and tuning time taken by each considered approach for LeNet model on MNIST dataset.

Approach	Accuracy [%]	Training Time [s]	Tuning Time [s]
Arbitrary	84.47	445	-
Tune V1	91.54	272	4575
Tune V2	81.76	187	4817
PipeTune	92.70	188	3415

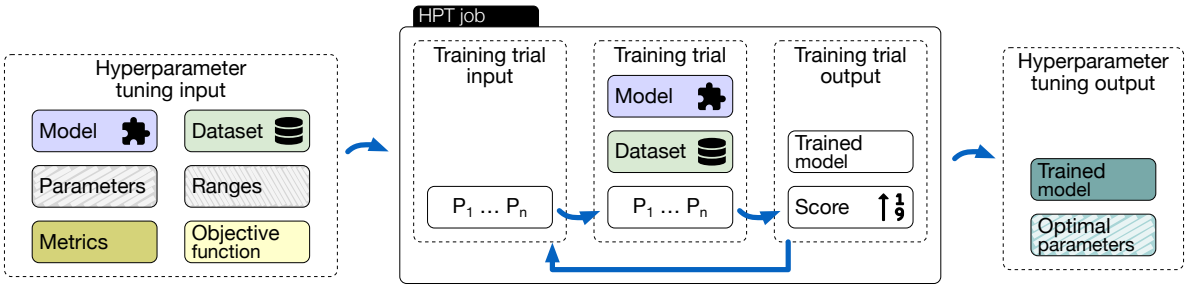


Figure 6.4: Hyperparameter tuning flow.

each trial to be manually controlled. Also, the objective function must be adapted to maximize the ratio accuracy to duration, rather than restricting it to accuracy only.

Figure 6.3 shows the results of Tune's performance characterization under various system conditions (*i.e.*, the number of cores assigned to the tuning job and the number of jobs assigned to the same logical cores). We used the V2 version of Tune to perform hyperparameter tuning. The tuning process was pinned to the same set of cores as the background jobs. For example, a configuration of 2 cores and 3 jobs meant a tuning job and 2 background jobs used the same 2 cores for execution. Figure 6.3 (a) illustrates the improvement in error relative to a single Tune V1 job. Figure 6.3 (b) is similar but shows training time improvement. Tuning under different system conditions significantly impacts the performance of the model being trained. There are only a few system configurations that yielded improvements over the baseline for error and training time. Some system configurations caused the tuning to trade better accuracy for faster training.

Hyperparameter tuning without system conditions can produce less efficient models. Table 6.1 shows the accuracy, training and tuning time achieved by different approaches for a LeNet model on MNIST dataset. These results show us the following. First, arbitrary values, if not correctly chosen, lead to both worse accuracy and training time. Second, if the user's focus is accuracy only, then PipeTune's accuracy results are comparable to Tune V1 however achieved in a lower tuning time. Third, if the user's focus is both accuracy and training time, then PipeTune's training time results are comparable to Tune V2 but with better accuracy and lower tuning time as well.

6.3 The PipeTune System

This section presents the system design of PipeTune. We begin clarifying the problem addressed by our system (Section 6.3.1). Then, we showcase its workflow (Section 6.3.2), the role of PipeTune's profiling phase (Section 6.3.3), the ground-truth phase (Section 6.3.4) and finally probing (Section 6.3.6).

6.3.1 Problem Statement

One of the first challenges of applying deep learning algorithms in practice is to find the appropriated hyperparameter values for a given workload. We assume that most DNN tuning jobs make use of some existing hyperparameter optimization solution. In the following we refer to these types of jobs as HPT Jobs (*i.e.*, Hyperparameters Tuning Jobs).

A given HPT Job takes as input a given workload, a set of parameters, its respective set of range values, an objective function and the metric of interest (*e.g.*, accuracy, performance, energy). This job spawns a collection of *training trials* based on the possible values of the parameters, following a given search

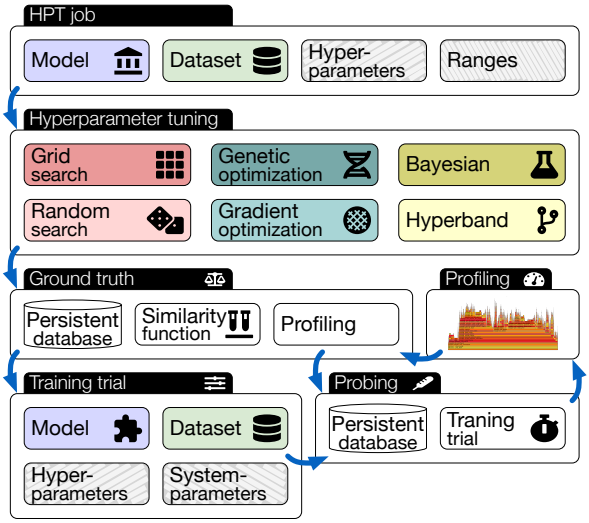


Figure 6.5: PipeTune architecture.

algorithm (*e.g.*, GridSearch, HyperBand). Each *training trial* takes as input the workload and a set of fixed values for the parameters of interest, where these values belong to their respective given ranges. These trials can run either sequentially or in parallel depending on the setup. They produce a trained model and a score for the given parameters values. Scores correspond to the metric of interest defined by the user. The optimal set of parameters values is chosen by applying the objective function to the scores. Figure 6.4 illustrates this process.

We consider a deep learning cluster consisting of N nodes, each containing C cores and M GB of memory. Note that despite a common trend to include GPUs in DNN clusters, we explicitly put aside this option. We do this given the (rather small) nature of workloads on which we focus, for which commodity machines are sufficient for training. HPT Jobs are scheduled in a FIFO manner. We categorize these jobs in the following two main types: Type-I: tuning the same model for different datasets (*e.g.*, recommendation engines), and Type-II: tuning different models for the same dataset (*e.g.*, computer vision).

Both types of tuning jobs can still be divided into two sub-types:

1. same set of hyperparameters and ranges, and
2. same set of hyperparameters but different ranges.

Each job, independent of its category, performs the earlier described tuning process from scratch. A key observation is that these jobs could benefit from previously computed results for other jobs in the same category to converge faster. Moreover, training trials spawned by the same HPT Job run all with the same system parameters even though they might require different resources configuration.

Another major limitation of the currently available approaches to hyperparameter auto-tuning is that only a single objective metric can be specified. This means that for a given HPT Job, one could choose to optimize either accuracy or performance, but not both simultaneously.

In summary, our problem's input consists of an HPT Job with the objective of achieving either maximum accuracy, or maximum accuracy with minimum training time. The former must output the best possible hyperparameters leading to the highest accuracy, independent of training time. For the latter, a combination of optimal hyper and system parameters is expected which leads to the highest accuracy and lowest training time. Note that for both scenarios, a shorter tuning times is beneficial, as allowed by our approach.

6.3.2 PipeTune Workflow

Figure 6.5 depicts the architecture components of PipeTune design and the main workflow. While training hyperparameters, a *trial* is a single training run with a fixed initial hyperparameter configuration. In order to find the best values for a given set of hyperparameters, the system executes a collection of trials,

Algorithm 5 PipeTune algorithm.

```
1: function Train(model, data, hyperparameters)
2:     job = async model.train(data, hyperparameters)
3:     async tuneSystem(model, data)
4:     job.wait()
5:     return model
6: function tuneSystem(model, data)
7:     profile = getProfile(job)
8:     (score, config) = getSimilarity(profile)
9:     if score > threshold then
10:        setSystemParameters(config)
11:    else
12:        for sp_v ∈ systemParameters do
13:            setSystemParameters(sp_v)
14:            wait until epoch finishes
15:            add collected metrics to m
16:        bestConfig = find best config in m
17:        setSystemParameters(bestConfig)
```

supervised by a given tuning library (*e.g.*, Vizier, Tune) and using one of the supported trial scheduling algorithms (*e.g.*, GridSearch, HyperBand).

PipeTune enhances the tuning of system parameters following a pipelined parallelism approach. That is, within each trial, a collection of sub-trials is executed, with the goal of defining the best system configurations for a given optimization function and metric of interest. This sub-trial consists of varying the system configuration on the epoch level and monitoring the system itself as well as the metrics of interest. The execution of sub-trials is controlled by PipeTune, which may also rely on different underlying scheduling algorithms.

Algorithm 5 details the pipelined approach. Function *train* (lines 1-5) is executed during a trial for a given workload (*i.e.*, model and dataset). After initiating the model training using the hyperparameter configuration given for that trial, *tuneSystem* (line 3) is invoked asynchronously. The *profiling* phase (lines 7) is initiated for this given trial with the objective of characterizing the workload properties and its systems requirements. This process is done at the granularity of epochs for the currently running trial. We rely on kernel performance counters (*e.g.*, cpu cycles memory stores, instructions) to gather hardware events corresponding to low-level metrics of the underlying system.

Once this *profiling* phase is over, its outcome is used as input to a *ground truth* phase. This process consists of applying a similarity function (line 8) on the job's profile. This is done to reuse optimal configurations known by the system for other jobs with similar characteristics. If the score of this similarity function is within a specific confidence level (line 9), then the optimal known configurations are applied (line 10) and no further system metric trials are required. However, if the score does not cross the threshold, a new *probing* phase starts, searching the optimal system configurations for that trial.

The probing requires each system configuration to be applied for a different epoch, following a given scheduling algorithm. We collect several meaningful metrics (*e.g.*, runtime, energy) plus low-level metrics (*e.g.*, hardware events). Then the optimization function is applied over these metrics (line 16) to identify the overall best system configuration. This process consists of iterating over the collected values for each tuple of system parameters, looking for the one which best fits the optimization function (*e.g.*, shortest runtime, lowest energy consumption). The complexity of this search is $O(n)$, where n is the number of distinct system parameters considered. Finally, the configuration identified as optimal is applied for the remaining iterations (line 17) and saved for further improving of the *ground truth* phase.

6.3.3 Profiling

The profiling component leverages hardware performance counters to collect low-level events of the system during the applications execution time. After an initial experiment campaign, we gathered a comprehensive list of such events. As the number of events collected per time unit is limited by the number of

actual hardware counters of the CPU, we filter out highly correlated as well as unsupported events. As result, our prototype deployed on x86 architectures current considers 58 measurable events, most of them being Performance Monitoring Unit (PMU) hardware events (*e.g.*, branch-instructions, cache-misses, cpu-cycles, mem-loads), reported by Linux's *perf* (v4.15.18). Although we have filtered the list of possible events to be collected, common Intel processors have only 2 generic and 3 fixed counters. Generic counters can measure any events while fixed counters can only measure one event.

When there are more events than counters (as it is in our case), then the kernel uses time multiplexing to give each event a chance to access the monitoring hardware. When this happens, an event might miss a measurement. If this happens, its occurrences are recomputed once the run ends, based on total time enabled *vs* time running [214], with:

$$final\_count = raw\_count * time\_enabled/time\_running.$$

This provides an estimate of what the count would have been, had the event been measured during the entire run. Considering that the output value is not an actual count, depending on the workload, there might be blind spots which can introduce errors during scaling. Although we profile workloads at the epochs granularity, each epoch runs for at least a few minutes and we measure the events of interest every second. To mitigate the potential profiling errors, we store the average of results during each epoch's time window.

6.3.4 Ground Truth

During this phase, new incoming HPT Jobs exploit the ground truth results from historical data collected during the previously completed jobs with similar system characteristics, to accelerate their system-parameter tuning phases. Our design allows the similarity function to be pluggable, and while we do settle on k-means [215] in the current implementation, PipeTune allows to easily switch to alternative techniques.

The implementation of *ground truth* is done as a separate module which is used by PipeTune. In this module, the user can point to a pre-trained similarity function for a warm start or let the system build a new one from scratch. For this, our implementation relies on the *scikit-learn* machine learning library for Python [216] which already supports several clustering algorithms (*e.g.*, affinity propagation, mean-shift, DBSCAN, OPTICS, Birch). The exhaustive list of supported models are then inherited by PipeTune and could be easily used as alternative similarity functions.

Regarding the currently used model (*i.e.*, k-means), it is trained over the low-level system metrics collected during the profiling phase. The datasets are then partitioned into $k = 2$ groups (*i.e.*, model and dataset). Extensions to other values of k, as well as to other similarities dimensions (*e.g.*, hyperparameters, ranges) are left for future work.

Figure 6.6 shows clustering results using k-means grouped by model and dataset labeled with their respective cluster's labels. We can observe that the majority of data fits into Type-I and Type-II are labeled as *cluster1* and *cluster2*, respectively. This result supports our assumptions regarding workloads similarities and shows that the chosen profiling technique can also capture the implicit characteristics of each workload. Finally, it shows that the clustering algorithm utilized can identify the similarities present in those characteristics and efficiently cluster them.

6.3.5 Privacy Concerns

Although the *ground truth* component of PipeTune makes use of customer's historical data, it does not require any information regarding the their workloads (*i.e.*, model or dataset). Instead, this process relies entirely on system events collected using the hardware performance counters. This profiling based on low level metrics allows PipeTune to characterize the applications while preserving user data privacy (*e.g.*, user parameters like model and dataset are not revealed).

Here, our assumption is that potential data, model and parameters similarities between workloads will affect the collected metrics in the same ways and therefore also be reflected in the similarity function. The results observed in Figure 6.6 supports this assumption. Relying on this, we can then identify similar workloads and reuse configurations without requiring any extra information from the user.

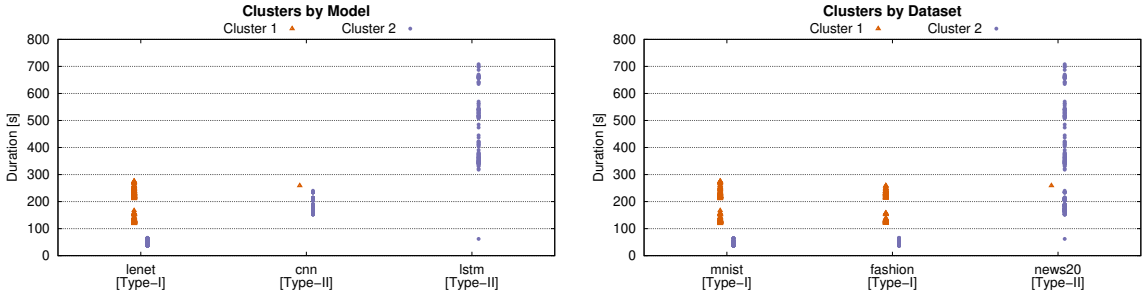


Figure 6.6: Clustering results grouped by workload type.

6.3.6 Probing

The probing phase profiles a given set of workloads in different system conditions, in order to collect sufficient data for a warm start of the *ground truth* component. In practice, the *ground truth* model is refined as the similarity of the incoming jobs with the historical data of the system starts to decrease. When this happens, we launch a grid search on the system-parameters at the epoch granularity, yet other search strategies are possible. In this case, the tuning of system parameters for the current job is performed directly on the analytical data collected. Moreover, this collected data is saved to be taken into account once re-clustering is applied.

We decide upon the necessity to launch a new probing or not for a given workload based on the similarity score outputted from the ground truth phase. When using k-means, the threshold matches the distance from the new set of data points to their current cluster's centroid. The distance is compared against the models' inertia, to measure the reliability of the prediction, or else if a re-clustering is needed.

6.4 Evaluation

In this section we present our in-depth evaluation of PipeTune both in a single-tenant and multi-tenant scenario using real-world datasets. The main findings we detail next are the following:

1. PipeTune achieves significant tuning speedups without affecting model performance (*i.e.*, accuracy);
2. By speeding up the tuning process, we also have a more energy efficient approach, not only due to the runtime reduction but also because of the more efficient utilization of system resources;
3. The proposed approach is sensitive to varying system loads as this is also reflected on the events used to profile and our system adapts on a fine granularity (*i.e.*, epochs level).

6.4.1 Prototype Implementation

PipeTune is implemented in Python (v3.5.2) and it consists of 947 LOC. We leverage two open-source projects, namely Tune and BigDL. Tune [60] is a Python library for hyperparameter search, optimized for deep learning and deep reinforcement learning [217]. Tune provides several trial schedulers based on different optimization algorithms. While we select HyperBand for the remainder of this work, Tune allows to switch among the available ones, as well as to implement new ones. As a consequence, PipeTune indirectly supports all its hyperparameter optimization algorithms.

The training applications are executed by BigDL [65], a distributed deep learning framework on top of Apache Spark. BigDL supports TensorFlow and Keras, hence PipeTune supports models defined using such frameworks. The Ground Truth module is based on a battle-tested k-means implementation openly available in the *scikit-learn* machine learning library for Python [216].

Finally, as storage backend, we leverage `InfluxDB` (v1.7.4), an open-source time series database. It offers a convenient `InfluxDB-Python` client for interacting with InfluxDB which we use to query information regarding the collected system metrics. PipeTune is released as open-source and can be found in the following repository[1].

[1] https://github.com/isabellyrocha/pipetune

Table 6.2: Workloads used for experiments.

	Model	Dataset	Datasize	Train Files	Test Files
Type-I	LeNet5	MNIST	12 MB	60 000	10 000
	LeNet5	Fashion-MNIST	31 MB	60 000	10 000
Type-II	CNN	News20	15 MB	11 307	7538
	LSTN	News20	15 MB	11 307	7538
Type-III	Jacobi	Rodinia	26 MB	1650	7538
	SPK-means	Rodinia	26 MB	1650	7538
	BFS	Rodinia	26 MB	1650	7538

6.4.2 Experimental Setup

Next we detail the experiment setup used for the experiments here presented.

Testbed

We deploy our experiments using Type-I and Type-II workloads on a cluster of 4 quad-socket Intel E3-1275 CPU processors with 8 cores per CPU, 64 GiB of RAM and 480 GB SSD drives. Experiments involving Type-III workloads are deploy on a single node containing an Intel E5-2620 with 8 cores, 24 GB of RAM and a 1 TB HDD. All machines run Ubuntu Linux 16.04.1 LTS on a switched 1 Gbps network. Power consumptions are reported by a network connected LINDY iPower Control 2x6M Power Distribution Unit (PDU), which we query up to every second over an HTTP interface to fetch up-to-date measurements for the active power at a resolution of 1W and 1.5% precision.

Workloads

We consider 7 state-of-the-art deep learning workloads for image classification, LLC-Cache computational sprinting and natural language processing. Table 6.2 summarizes their details.

LeNet5 [62] is a convolutional network for handwritten and machine-printed character recognition. Convolutional Neural Networks (CNNs) [218] are a special kind of multi-layer neural networks, trained via back-propagation. CNNs can recognize visual patterns directly from pixel images with minimal preprocessing. Long Short-Term Memory (LSTMs) [219] are artificial Recurrent Neural Networks (RNNs) architectures used to process single data points (such as images, connected handwriting recognition and speech recognition), as well as sequences of data (*i.e.*, speech, videos). Finally, Jacobi is a differential numerical solver, BFS is breath-first-search and spk-means is k-means implemented on top of Spark framework.

The MNIST dataset [63] of handwritten digits has a training set of $60\,000$ examples, and a test set of $10\,000$ examples. The digits have been size-normalized and centered in a fixed-size image. Fashion-MNIST dataset [220] is a dataset of article images consisting of a training set of $60\,000$ examples and a test set of $10\,000$ examples. Each example is a 28x28 grayscale image, associated with a label from 10 classes. Fashion-MNIST shares the same image size and structure of training and testing splits as the original MNIST dataset.

The News20 dataset [175] is a collection of $20\,000$ messages collected from 20 different netnews newsgroups. We sample uniformly at random 1000 messages from each newsgroup, and we partition them by name. The Rodinia Benchmark Suite [221] is a collection of profiling short-term resource allocation (*i.e.*, computational sprinting) policies which targets heterogeneous computing platforms with both multicore CPUs and GPUs. These workloads have the objective to classify or predict the original data reserved for testing purposes.

Hyperparameters

There are several potential hyperparameters to tune. For practical reasons, in our evaluation we select the 5 described below. Note that their recommended range is typically application-driven, and we settle on specific values without however generalizing for any workload.

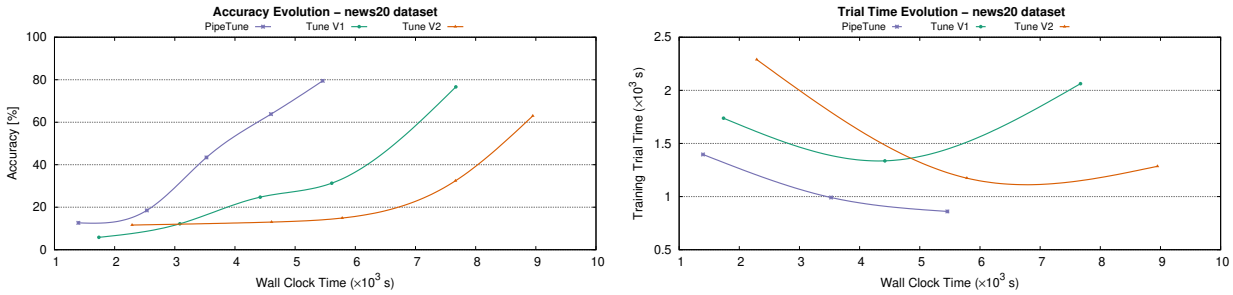


Figure 6.7: Accuracy and trial time convergence.

1. **Batch size.** Number of samples to work through before updating the internal model parameters. Large values for batch size have a negative effect on the accuracy of network during training, since it reduces the stochasticity of the gradient descent. Range: [32 - 1024].
2. **Dropout rate.** Dropout randomly selects neurons to be ignored during training. Dropout layers are used in the model for regularization (*i.e.*, modifications intended to reduce the model's generalization error without affecting the training error). The dropout rate value defines the fraction of input to drop to prevent overfitting [222]. Range: [0.0 – 0.5].
3. **Embedding dimensions.** Word embeddings provide a mean of transfer learning. This mechanism can be controlled by having word vectors fine-tuned throughout the training process. Depending on the dataset size on which word embeddings are being refined, updating them might improve accuracy [223]. Range: [50 – 300].
4. **Learning rate.** Rate at which the neural network weights change between iterations. A large learning rate may cause large swings in the weights, making impossible to find their optimal values. Low learning rates requires more iterations to converge. Range: [0.001 - 0.1].
5. **Number of epochs** Number times that the learning algorithm will work through the entire training dataset. Typically, larger number of epochs yields in longer runtimes but also higher training accuracy. However, the number of epochs required to achieve a given minimum desired accuracy depends on the workload. Range: [10 - 100].

System Parameters

For the purpose of this evaluation, we restrict the list of parameters to number of cores and memory. However, the same mechanisms can be applied to any other parameter of interest (*e.g.*, CPU frequency, CPU voltage). In our cluster, the ranges of valid values for system parameter tuning are [4 - 16] and [4 - 32] (GB) for for number of cores and memory, respectively.

Baselines

In order to better understand where our approach stands in comparison to other strategies, we take the following two baselines into account:

1. **Hyperparameters tuning:** Our first baseline system (*i.e.*, Tune V1) uses the tuning of hyperparameters ignoring any system parameter. We rely on HyperBand for the parameter optimization with the objective function set to maximize accuracy.
2. **System and hyper parameters tuning:** We further compare against Tune V2, where we include the list of system parameters to be considered in the list of parameters to be tuned by the HyperBand algorithm. We also include the training duration as part of the optimization function which in this baseline is set to maximize the ratio accuracy to duration (details in Section 6.2).

6.4.3 Convergence Evolution

In order to build our initial similarity model we rely on profiling data of the workloads described in Table 6.2. For each workload, we vary the system configurations as follows. Memory allocation can be 4GB, 8GB, 16GB, and 32GB. The total number of cores that could be allocated were 4, 8, or 16. Finally, batch size could take the values 32, 64, 512, or 1024. In total, this sums up to 48 different configurations

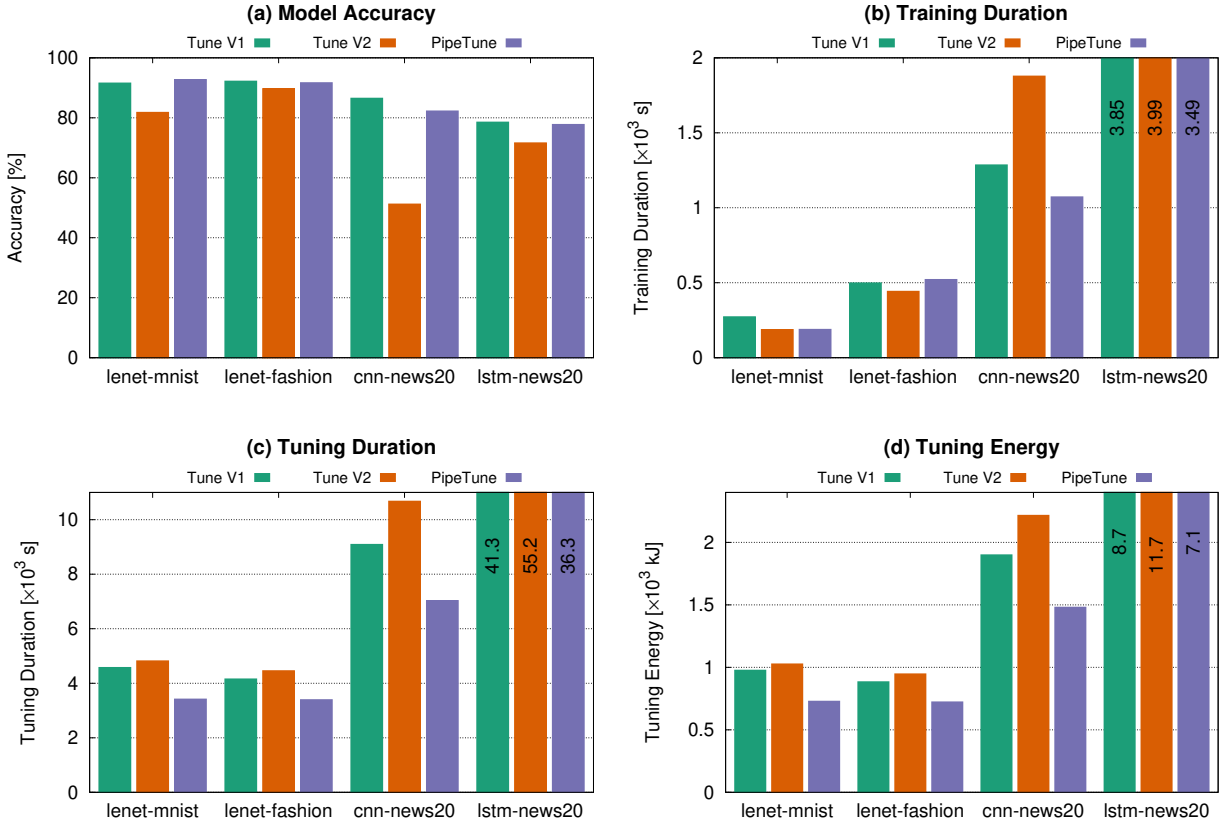


Figure 6.8: PipeTune analysis for Type-I and Type-II Jobs.

for each workload. There is no reason to expect variations in the data collected from different training instances using the exact same parameters. However, we repeat this process twice for each configuration to make sure that the achieved model is not affected by potential unseen variations.

We begin our evaluation by analyzing the convergence trajectory of PipeTune compared to Tune V1 and Tune V2. Figure 6.7 illustrates the accuracy evolution of the training trials over the tuning time of a CNN model on the News20 dataset. We observe that PipeTune converges to an accuracy value comparable to Tune V1 but at a much faster rate. For instance, PipeTune reaches a 60% accuracy after approximately 4500 seconds. On average our approach is $1.5\times$ and $2\times$ faster than Tune V1 and Tune V2, respectively.

The training time achieved shows similar behavior (see Figure 6.7). Interestingly, Tune V1 performs worse than Tune V2. Since Tune V1 optimizes only for accuracy, the most accurate model not necessarily achieves the shortest training time. On the other hand, as Tune V2 optimizes for the ratio accuracy to performance, the accuracy achieved might not be the highest possible. However, the training time in the given configurations might be lower (which is exactly what happens in this instance of the problem). Finally, we observe that PipeTune consistently presents shorter trial times than the other two approaches during the entire tuning process.

6.4.4 Single-Tenancy

We now consider a single-tenancy scenario, and assume each HPT Job runs in a dedicated cluster, where the required resources demanded by the system parameters are available and exclusive for a given tenant. This prevents interference caused by other jobs co-located on the same cluster. However, as a given HPT Job spawns several *training trials* asynchronously, the cluster still remains shared among these sub jobs. We evaluate how PipeTune performs in such stable setting, comparing it against Tune V1 and Tune V2, for all the workloads.

Comparison with baseline. Figure 6.8 presents the results of model accuracy, training and tuning

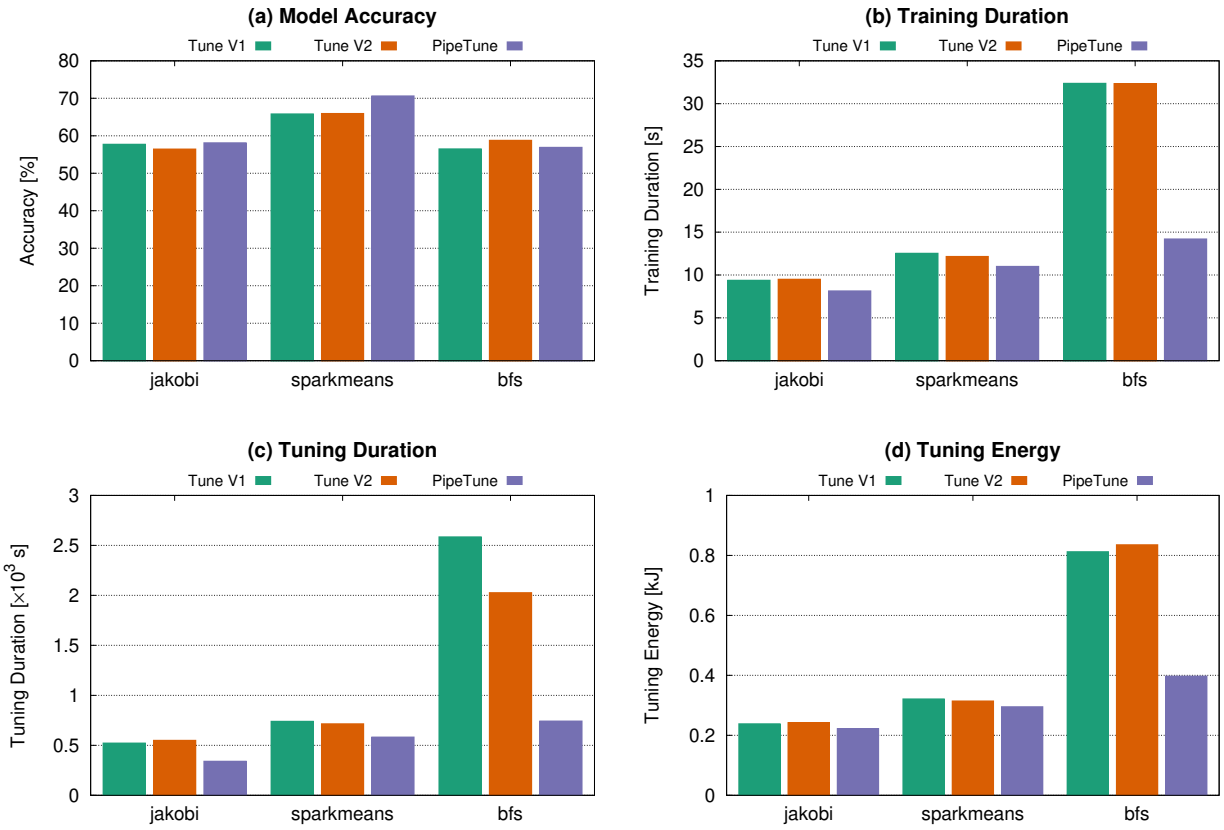


Figure 6.9: PipeTune's accuracy, runtime and energy consumption for Type-III Jobs.

runtime, and overall cluster energy consumption of offline HPT Jobs for the different workloads described in Table 6.2.

Figure 6.8 (a) presents the accuracy results. We can observe that the accuracy of PipeTune is not affected by the performance optimization. In fact, results are on par with Tune V1, where hyperparameters tuning is done with the only objective of maximizing accuracy. As expected, Tune V2 decreases accuracy up to 43%, since the objective function no longer tries to optimize accuracy but also takes the runtime into account.

Figure 6.8 (b) shows the training time of the achieved model. In this case, PipeTune presents comparable results to the baseline. In fact, we observe up to $1.7\times$ speed-up in comparison with Tune V2 which focuses exactly in reducing training runtime. We observe that Tune V2 increases tuning duration by up to 18% when compared to Tune V1. This happens for the following two reasons. First, the search space of Tune V2 is larger than of Tune V1, as it includes the system-parameters. Second, the optimization function consists of accuracy and runtime together. These two reasons make it harder for the search algorithm to find the optimal set of configurations, hence longer tuning times are observed.

On the other hand, PipeTune reduces tuning runtime by at least 18% when compared against Tune V1, as shown in Figure 6.8 (c). This performance gain is obtained because the search space and optimization function remains the same, and at the same time PipeTune finds and applies during runtime the optimal system configurations for each trial. Moreover, all the additional steps introduced by PipeTune are done in parallel, without impacting the hyperparameters tuning process.

Figure 6.8 (d) reports the energy results. The overall energy consumption of the cluster is directly affected both by the performance decays and gains. Compared against Tune V1, we observe up to 22% energy increase for Tune V2 and up to 29% energy decrease for PipeTune.

Figure 6.9 compares Tune V1, Tune V2 and PipeTune on a single node. The Type-III workloads used in these experiments have shorter epochs and each a different CNN model. Previous experiments deploy

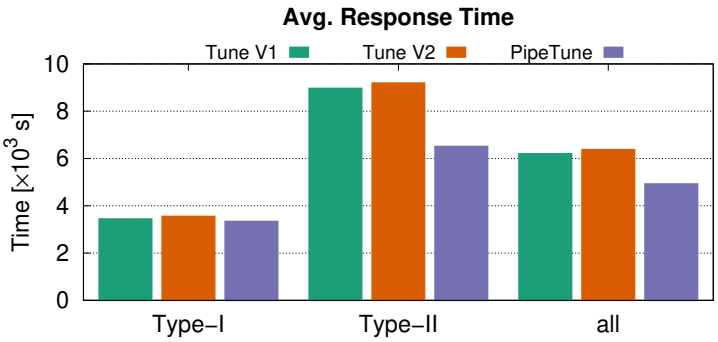


Figure 6.10: Average response time for Type-I and Type-II Jobs considered independently and all together.

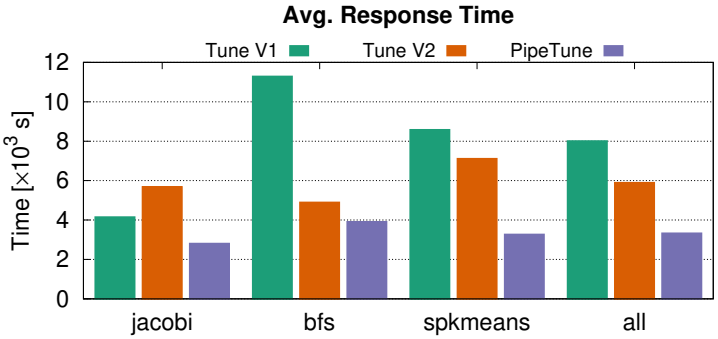


Figure 6.11: Average response time for Type-III Jobs.

PipeTune on workloads with epochs lasting minutes. Long epochs work in favor of PipeTune since low-overhead profiling is performed across the first couple of epochs to classify new workloads. Therefore, next we perform an extra analysis on Type-III Jobs which present this more challenging setup for PipeTune to observe how it behaves.

Figure 6.9 (a-d) plots the same metrics as seen in Figure 6.8. The goal is to test how well PipeTune can improve tuning for workloads with short but many epochs per trial. Here we can observe that PipeTune also achieves the expected results in this more challenging scenario and reduces both training and tuning time when compared to the baseline systems. Regarding model accuracy, we can also see that our approach achieves comparable or better results than the baseline. Finally, the energy results reflect the performance gains resulting in a more energy efficient approach as well.

To summarize, for these single-tenancy scenarios, PipeTune presents better performance with up to 23% reduction on tuning time, is more energy efficient reducing up to 29% the overall energy consumption of the utilized cluster, and does not affect model accuracy as the observed differences in this aspect are negligible.

Profiling overhead. Profiling is a fundamental part of our system design and essential for the decision making process. During the profiling of a given epoch, the extra computation introduces additional load, depending on the system configuration. However, as this profiling overhead only occurs in the epoch granularity and does not apply for all the epochs, the performance benefits resulting from tuning the system-parameters overtake the measured overhead. The experimental results presented above also support these assumptions as, otherwise, we would not observe performance gains when compared with the approaches Tune V1 and Tune V2 which do not perform any profiling.

6.4.5 Multi-Tenancy

Next, we evaluate PipeTune in a multi-tenancy scenario (*i.e.*, a shared cluster handling multiple HPT Jobs). In this case, we show the average response time of jobs as an indicator of performance. We consider that jobs arrive randomly with the interarrival times being exponentially distributed. For the case where two

workload types are considered together, each of them corresponds to 50% of the overall jobs (i.e., equally balanced). In all cases, within a given workload type, the workloads are chosen following a round-robin strategy. The portion of overall unseen jobs corresponds to 20%.

Figure 6.10 shows the results for the multi-tenancy scenario considering workloads of Type-I and Type-II grouped by type as well as the overall results. As in Section 6.4.4, this evaluation has been performed in a distributed environment. In this experiment we observe improvements similar to the ones in the single-tenancy scenario. Regarding response time, PipeTune results in up to 30% reduction when compared with Tune V1 and Tune V2.

Figure 6.11 shows the same results described above but considering workloads of Type-III. This trace was executed in a single node in contrast with the distributed environment of the previously described results. In this specific scenario we observe that the performance gain trends earlier observed becomes even more evident in such environment and workload type. In this case, PipeTune results in up to 65% reduction on the average response time in comparison with Tune V1 and Tune V2. This indicates that the overhead of computation added for the unseen jobs is compensated by the gain of future similar incoming ones.

6.5 Summary

This chapter presented PipeTune, an open-source system that leverages the repetitive behavior of DNN tuning jobs to quickly find the best set of parameters. Our approach is modular which makes it easy to swap between similarity functions and underlying search algorithms. We evaluated 7 different real-world datasets from different domains, including text classification and image recognition. When compared against state-of-the-art DNN tuning systems, PipeTune showed experimental evidence that the approach greatly reduces tuning and training time while achieving on-par accuracy. The observed optimizations come mainly from the combination of 1) efficiently including system parameters into the parameter tuning process, and 2) the proposed similarity function which allows us to leverage historical data.

In summary, our main contributions were:

1. design and prototype of PipeTune; and
2. state-of-the-art evaluation on single-tenant and multi-tenant scenarios showing.

Despite of the considerable runtime speedup and energy reductions achieved by PipeTune, our approach does not cover the inference phase of applications. Typically, once the tuning phase is complete and the model is trained with the optimal identified parameters, the application owner deploys this model for inference. The inference also has parameters which have to be tuned before deployment, a process often challenging for unexperienced users. Considering this, next we discuss ways of extending PipeTune to covering these aspects and further improve user experience.

Chapter 7

Use Case: EdgeTune

In the previous chapter we introduced a use case application focused on optimizing energy consumption of hyperparameters tuning. Although the entire process of tuning and training models is performed solely to be deployed for inference, state-of-the-art approaches do not provide any guidance for this follow-up phase. Therefore, in this chapter we present EdgeTune, a novel inference-aware parameter tuning server. It considers the tuning of parameters in all levels backed by an optimization function capturing multiple objectives. Our approach relies on inference estimated metrics collected from our emulation server running asynchronously from the main tuning process. The latter can then leverage the inference performance while still tuning the model. We propose a novel one-fold tuning algorithm that employs the principle of multi-fidelity and simultaneously explores multiple tuning budgets, which the prior art can only handle as suboptimal case of single type of budget.

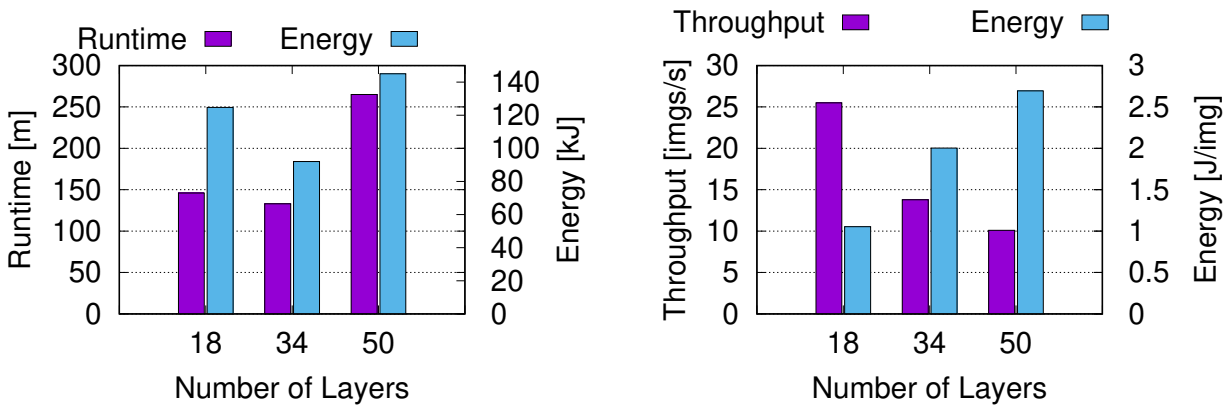
7.1 Introduction

As previously already discussed, commercial platforms as well as on-premises solutions help deployers by offering tuning services which automatizes the search for optimal parameters. The currently available tuning services focus in assisting users regarding model accuracy. However, they lack input regarding the inference phase, which is the ultimate goal of this process. Such tuning services typically output the optimal hyperparameters identified, paired with the model resulting of training using this set of parameter values. As the trained model is already given as output, the information of which optimal parameters identified during tuning is no longer useful for the users. If users plan on retraining the model with a different dataset, the set of tuned parameters could be reused but it is not guaranteed that the optimal parameters remain the same across different datasets. In fact, the next step for the user after having a fully trained model is to deploy it for inference use.

The deployment of inference model is not always straightforward and requires a vast domain knowledge, as well as extra experimentation, in order to find the best environment for achieving the desired inference performance. Even if the user has only a specific edge device available for model deployment, it is still crucial to either 1) configure the system parameters of this device, or 2) take the model inference performance on that specific device into account on the tuning process. However, the most common case is that the tuned model might be deployed across different edge devices and having these configurations suggested can assist users to take the most out of their tuned models.

Regarding tuning trials, a common approach is to define a budget which each trial might use. This is typically defined in terms of epochs, dataset, or time. Using a reduction factor will exclude unpromising trials and increase the budget of promising ones, resulting in a frugal usage of resources. This is also indeed the case when compared to fixed budged tuning or even algorithms which use no budget at all [224]. However, the current existing budgets (*e.g.*, epoch or dataset based) only consider one dimension which means we are still using either the entire dataset or the full number of epochs for each trial.

In addition to choosing the edge device to use or which configurations to set, a critical point is the number of samples in which inference should be applied to. Although single sampled inference is common in practice, there are scenarios (see §7.3.4) in which multi-sample inference is beneficial. In those scenarios, the batch size must be tuned carefully, as too-large values can lead to saturation of the system. For instance, in a server scenario where each inference query contains N samples arriving at fixed frequency,



(a) Training

(b) Inference

Figure 7.1: Model hyperparameters tuning.

the user should know what is the optimal way to split these samples. A similar pattern happens in a multi-stream system, where single sample inference queries arrive randomly following a certain distribution. In this case, if the optimal batch size is identified, aggregating samples for multi-sampled inference could improve the overall mean response time of the system.

Considering all these problems, in this chapter we propose EdgeTune, a novel edge-based tuning system capable of taking inference objectives into account during the tuning of hyperparameters. As result, EdgeTune outputs to users more useful information such as the optimal configurations of edge device for inference. This allows users to directly deploy their trained model for inference without further actions. Besides, as we show later, the additional optimizations of this approach also reduces tuning runtime and energy when compared to the state-of-the-art baseline systems.

The reminder of this chapter is organized as follows. First, we dive into the characteristics of our tuning system by means of a motivating example (Section 7.2). In Section 7.3, we present a detailed description of our design choices for each building block of our approach. We present the algorithm of EdgeTune in Section 7.4. In Section 7.5, we describe our prototype implementation and present the results of our in-depth evaluation. Finally, we conclude in Section 8.5.

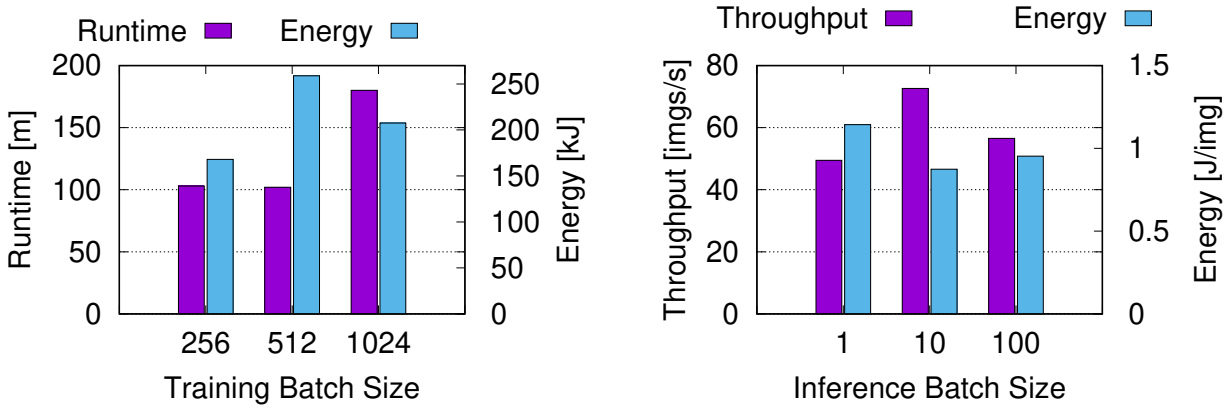
7.2 Tuning Space by Motivating Examples

The training phase of DNN workloads involves hyperparameters which are variables set before actually optimizing the model's parameters. Setting the values of hyperparameters can be seen as model selection, *i.e.*, choosing which model to use from the hypothesized set of possible models. Hyperparameters are often set by hand, selected by some search algorithm, or optimized by some auto-tuner tool [56], [57], [59]. DNN models can have many hyperparameters, including those which specify the structure of the network itself (*i.e.*, *model hyperparameters*) and those which determine how the network is trained (*i.e.*, *training hyperparameters*).

Moreover, depending on the inference setup chosen there are also hyperparameters to be tuned for this phase (*i.e.*, *inference hyperparameters*). Finally, in both phases the system configurations have a great impact on tuning and inference performance, and therefore also has to be considered (*i.e.*, *system parameters*). Next we describe each of these parameters type and motivate their relevance by means of a practical example using the tuple ResNet and CIFAR10 as workload being tuned to reach at least 80% model accuracy.

Model Hyperparameters

The structure of the neural network itself involves numerous hyperparameters in its design, including the size and nonlinearity of each layer. The numeric properties of the weights are often also constrained in some way, and their initialization can have a strong effect on model performance. Finally, preprocessing



(a) Training

(b) Inference

Figure 7.2: Training hyperparameters tuning.

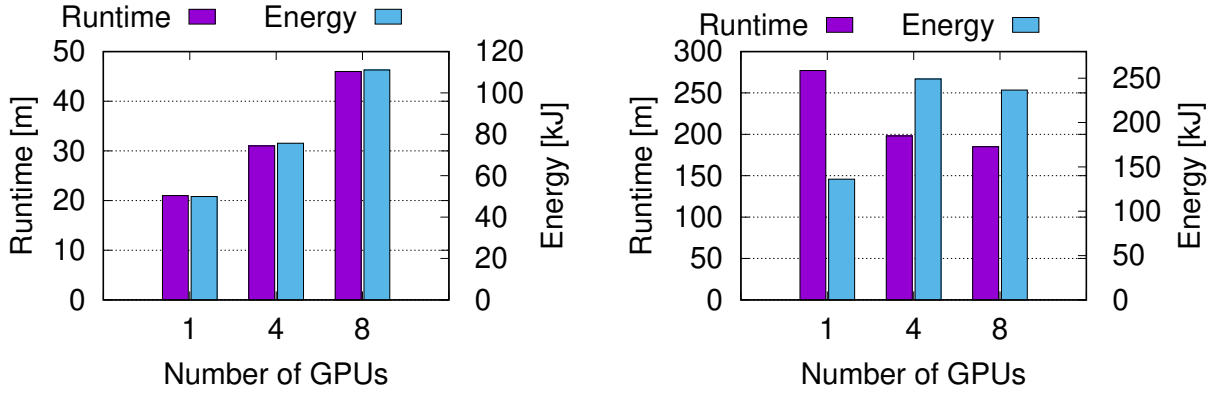
of the input data can also be important for ensuring convergence [225], [226]. Examples of such hyperparameters include number of hidden layers, weight decay, activation sparsity, weight initialization, and preprocessing input data.

We show how one of these model hyperparameters (*i.e.*, number of layers) impact the training (Figure 7.1a) and inference (Figure 7.1b) performance in terms of runtime and energy. We observe that performance does not directly relate to the number of layers, and that their relation is not straightforward to predict. For the inference phase we show the throughput (*i.e.*, images per second) and the energy consumption per single image inference in Joules (J). In this example, the throughput is inversely proportional to the number of layers and the energy consumption is proportional to it. However, the number of layers typically have a direct impact on accuracy which also has to be considered. These trade-offs can be exploited during tuning if the process is designed to be inference-aware. One way to achieve this is by simulating the inference phase during tuning and considering the inference-related results in the objective function.

Training Hyperparameters
When training a neural network, the resulting model depends not only on the chosen structure but also on the training method used to set the network's parameters. The training method itself can have many hyperparameters. Here we describe the hyperparameters of mini-batch gradient descent, which updates the network's parameters using gradient descent on a subset of the training data. Some examples of this type of hyperparameter are learning rate, loss function, mini-batch size, number of training iterations, and momentum. Figure 7.2a shows the impact of the batch size on training runtime and energy consumption. We observe how high batch sizes values (*i.e.*, 1024) result in high training times and energy consumption, while others (*i.e.*, 256 and 512) produce similar training times but different energy consumptions. This indicates that when energy is a concern, it should be explicitly taken into account while tuning. This observation supports the need of a multi-parameter tuning approach.

Inference Hyperparameters
In scenarios where multiple images are available, multi-image inference can be beneficial. In this case, the hyperparameter *batch size* must be defined for the inference phase and its value has a direct impact on performance. Figure 7.2b shows the throughput (*i.e.*, images per second) and the energy consumption per single image inference in Joules (J) when doing single-inference (*i.e.*, one image at a time) and multi-inference (*i.e.*, multiple images at a time). As show, both throughout and energy improve by performing multi-inference in comparison with single-inference. However, the choice of how many images to include in each batch to process at any given time is critical as the performance gains can quickly reach saturation and start decaying if the chosen batch size is too high.

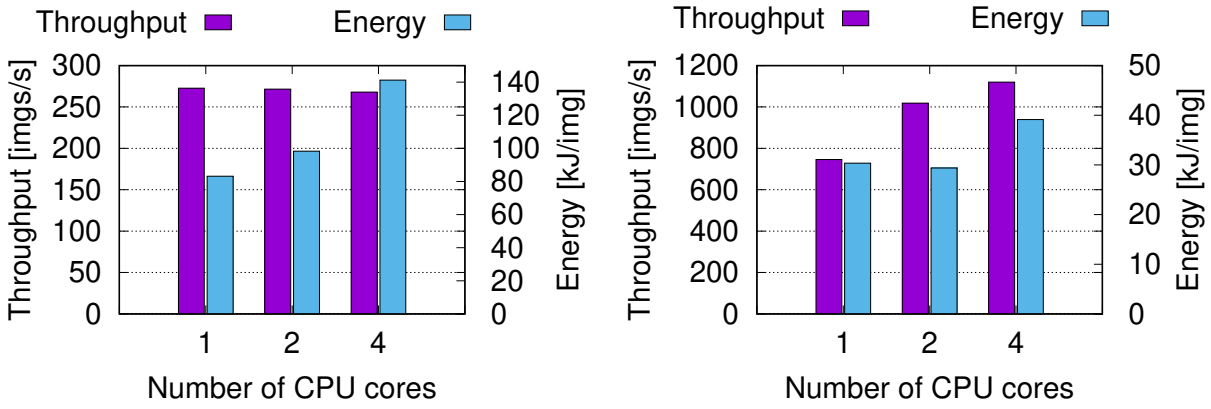
System Parameters
The configurable resources of the underlying computing infrastructure executing the training and inference (*e.g.*, memory, CPU cores, CPU frequency, number of GPUs, *etc.*) are the system parameters. Typically, the hyperparameter optimization fixes the same system parameters for each trial, although they



(a) Batch = 32

(b) Batch = 1024

Figure 7.3: Training system parameters tuning.



(a) Batch = 1

(b) Batch = 10

Figure 7.4: Inference system parameters tuning.

might benefit from different configurations. Moreover, while model and parameters are tuned for model accuracy, its impact on the inference performance is not considered. Figure 7.3 and Figure 7.4 show the impact of system parameters on the training and inference phases by varying the number of GPUs and the number of CPU cores.

Figure 7.3a and Figure 7.3b show the training results for batch 32 and 1024, respectively. We note that for smaller batch sizes, neither runtime nor energy are improved by increasing the number of GPUs. In fact, we actually observe the opposite, as the performance considerably decreases by up to 120%. With larger batch sizes, results become even more unpredictable. The running performance improves but not proportional to the number of GPUs, while the energy consumption actually increases even in the cases where runtime is lower. These results highlight the trade-offs to consider while tuning, specially if energy consumption is a concern for the user.

Regarding inference, Figure 7.4a and Figure 7.4b show similar results. For single image inference, increasing the number of cores does not increase throughput (as expected) and increases energy consumption. For multi-image inference, the throughout grows proportionally to the number of cores but the energy consumption of 4 cores is 33% higher than for 2 cores although the throughput is only 9% higher. Therefore, when energy savings are more important than inference performance, it becomes harder to find the sweet spot solution. For instance, in the shown example, the most energy-saving solution requires 2 CPU cores, which is however not the one with highest throughput.

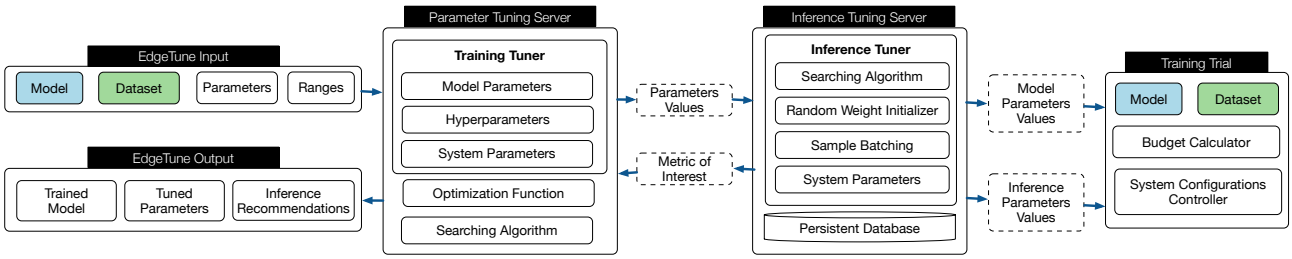


Figure 7.5: EdgeTune architecture.

7.3 EdgeTune: System Design

In this section we present the overall architecture of EdgeTune as well as a description of its main components, how they work individually to take care of specific aspects covered by the system, and how they communicate and interact with each other. Figure 7.5 depicts the components of this architecture design and its main workflow, which we describe in more details next.

7.3.1 Overview

EdgeTune consists of two main components: *Model Tuning Server* and *Inference Tuning Server*. The user gives as input 1) the workload to be tuned (*i.e.*, dataset and model), 2) the set of hyperparameters, 3) the set of system training parameters, 4) the set of inference training parameters, 5) the tuning objective (*e.g.*, tuning duration or energy, model accuracy), and 6) the inference objective (*e.g.*, throughput, energy). Each set of parameters to be tuned comes together with another set containing the corresponding range of values which each parameter can assume. As output the user receives the optimal trained model with respect to the tuning objective as well as the optimal system configurations to be used for inference deployment with respect to the inference objective.

We assume that the objective function given to the *Inference Tuning Server* component does not relate to model accuracy as the *Model Tuning Server* already takes care of this aspect (*e.g.*, maximize model accuracy as the objective of the tuning server and minimize inference energy for the inference server). Therefore, as soon as the values of a given trial are defined, the *Inference Tuning Server* can asynchronously be started with arbitrary weight values, and the inference tuning process can take place in parallel to model tuning. Moreover, some types of parameters such as training batch size and number of epochs do not affect the inference phase. Considering this, the *Inference Tuning Server* results can be reused for different parameters as long as they do not affect the architecture structure. Since we explicitly divide the hyperparameters into these two types (*i.e.*, model and training), we can easily identify for which parameters the results can be reused.

The *Model Tuning Server* and *Inference Tuning Server* components are implemented such that the user can also individually specify which tuning algorithm to be used by each of them (*e.g.*, Random Search [90], HyperBand [69], BOHB [227]). For instance, a user could choose to run the *Model Tuning Server* following the HyperBand approach while *Inference Tuning Server* following GridSearch. One setup where this configuration could make sense is where the range of inference parameters is not too large. In this case, trying all the parameters for inference would give more accurate results without necessarily affecting the overall tuning duration.

Once all the necessary input is defined by the user, the tuning process starts with the *Model Tuning Server* defining values for the first trial according to the tuning algorithm illustrated in Section 7.4. The chosen architecture structure is given as input to the *Inference Tuning Server*, which checks if the required information is already available for the defined architecture and model hyperparameters. This step consists of a table look-up based on historical data stored from previously processed trials. If results are already available, the *Inference Tuning Server* immediately returns it and no further action is required on this component. Otherwise, the inference-based tuning starts on the given architecture, model hyperparameters, and the set up inference parameters. In this case, the optimal configuration is stored and given back to the *Model Tuning Server* which takes it into account to update its metric of interest. This process is repeated for all the trials until the final result is reached and given as output to the user.

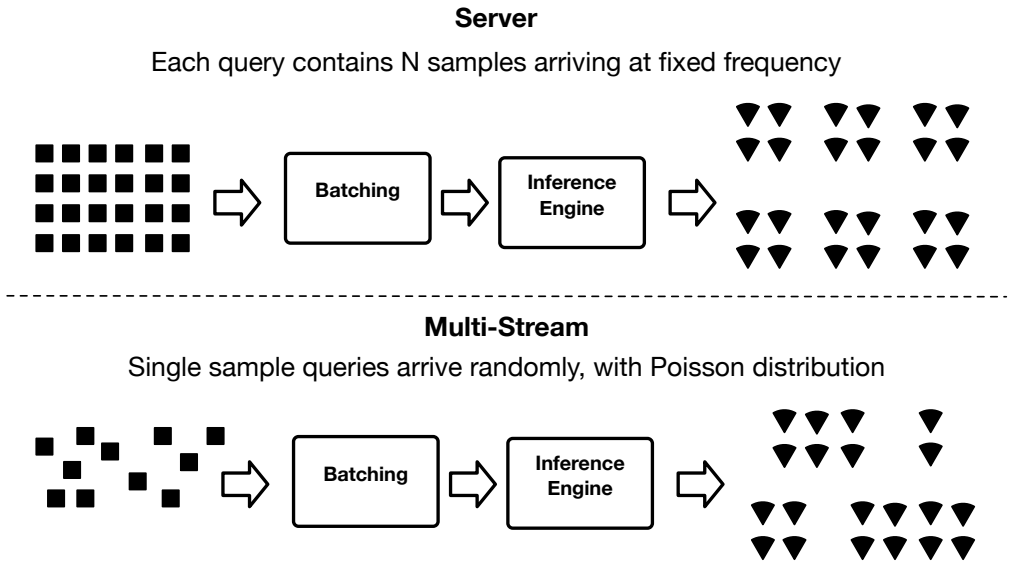


Figure 7.6: Illustration of scenarios where multi-image tuning is required.

7.3.2 Architecture

EdgeTune consists of two main components which we describe bellow in details (*i.e.*, model and inference tuning servers). The model tuning server can be executed using both CPUs or GPUs, while the inference server is only CPU based. The reason for the latter is that, first, the inference server simulates edge devices which typically do not contain any GPU card, and, second, the inference tuning is straightforward and therefore does not require any accelerator. Regarding the model tuning server on the other hand, although both scenarios are supported, it performs significantly better when used with GPUs. We also support multi GPUs training and tuning, which can improve the performance in some case but for others it does not scale as one might expect. As already shown in Figure 7.3, increasing the number of GPUs used for tuning does not positively affects runtime in every case. Therefore, the system parameters tuning takes care of finding the optimal architecture configurations for each tuple of workload and parameter's values.

7.3.3 Model Tuning Server

The *Model Tuning Server* receives as input the hyper (*e.g.*, number of layers, batch size, number of epochs) and system (*e.g.*, number of cores, memory, CPU frequency) parameters to be tuned regarding model training, together with the workload of interest (*i.e.*, dataset and model) and the metric of interest (*e.g.*, runtime, energy) and the optimization function (*e.g.*, min, max, threshold). Following a given search algorithm, the tuning server defines the search space of the given parameters and starts performing training trials on these values. For each trial, the *Inference Tuning Server* is asynchronous called which in parallel computes the information of the optimal inference system parameters together with its runtime and energy consumption. It is important that the *Inference Tuning Server* result comes back before the end of the ongoing trial, as it contains crucial information for the overall optimization function of the *Model Tuning Server*. This constraint is guaranteed as the entire *Inference Tuning Server* duration is contained in the duration of a *Model Tuning Server* training trial and, therefore, also does not add any overhead to the main process. Once the result is received and the training trial isfinished, the resulting metrics from both cases can be combined and considered together on the decision of promising configurations.

7.3.4 Inference Tuning Server

The *Inference Tuning Server* receives as input the model structure and model hyperparameter values defined by the trial which called it, together with the hyper (*i.e.*, batch size) and system (*i.e.*, number of cores, memory, frequency) parameters to be tuned regarding the inference phase. In this case, before starting the parameter search, the system verifies whether the optimal configurations are already known for the given model structure based on historical data. The feature of looking into historical data allows us to improve performance since it avoids retuning architectures and parameters twice, with the cost

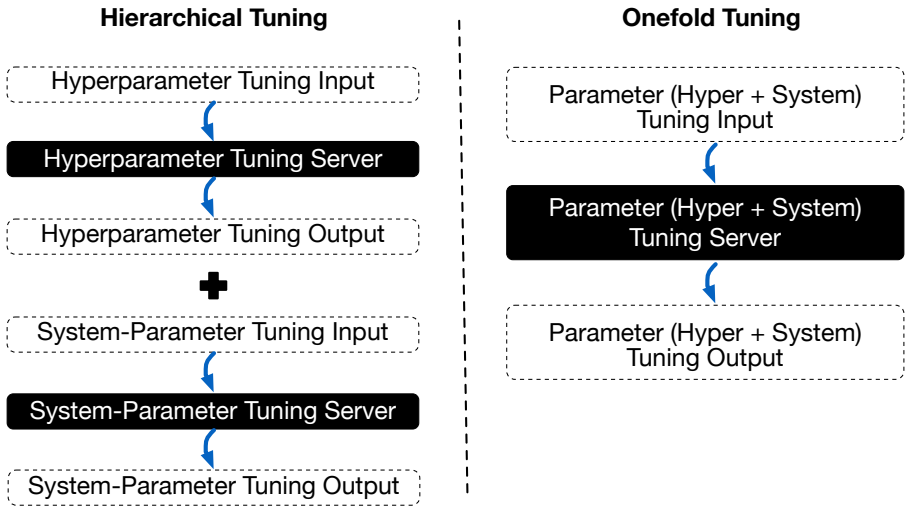


Figure 7.7: Difference on the execution flow between hierarchical and onefold tuning approaches.

of a small storage overhead [228]. If this is the case, then the found parameters together with their metric of interest (*e.g.*, runtime and energy consumption) are directly returned and no further action is required. Otherwise, a process similar to the one described for *Model Tuning Server* takes place but this time with focus on inference instead of training. Following a given search algorithm (*e.g.*, BOHB [227]), the tuning server defines the search space of the given parameters and start performing inference trials on these values. The optimization function defined by the user is applied to the performed trials and the optimal set of parameters is then identified, saved for later usage and returned together with the metrics of interest.

As this server takes as input the network structure from the *Model Tuning Server*, the definition of model hyperparameters are already taken care of in this step. Therefore, the focus of this component is on the inference hyperparameters and system parameters for the inference phase. Each workload might have particular hyperparameters to be tuned but batch size is a hyperparameter of common interest and therefore considered for all workloads by a subcomponent named *Batching*. Figure 7.6 illustrates two scenarios where the *Batching* subcomponent is crucial for inference performance. The first scenario is a server where each query contains N samples arriving at fixed frequency. In this case, it is important to define how many samples should be processed at a time to achieve the expected performance. The second scenario is a multi-stream where single sample queries arrive randomly, following a Poisson distribution. In this case, aggregating the individual samples to perform batch inference can also optimize the overall mean response time and therefore *Batching* is also relevant.

Finally, the systems parameters for inference are the second main aspect playing a role in the inference throughput and therefore has to be carefully considered. As we have seen in the motivating example of Section 7.2, different batch sizes might require different system configurations. Therefore, in order to reach optimal inference tuning performance, the batch size and the system configurations are tuned in combination. Although this adds an extra step to the main tuning process, this component runs in parallel to the training trials which in principle takes much longer than the inference trials. Moreover, in the worst case, the inference trial is performed for the first training trial of a given configuration and all the further training trials will reuse these results.

7.4 Onefold Tuning Algorithm

We describe here the proposed onefold tuning algorithm and its novel features, including search algorithm, training budget and objective functions. Before going into these details, we make a comparison between a onefold and twofold (*i.e.*, hierarchical) approach to highlights the motivations which backed our main design choice.

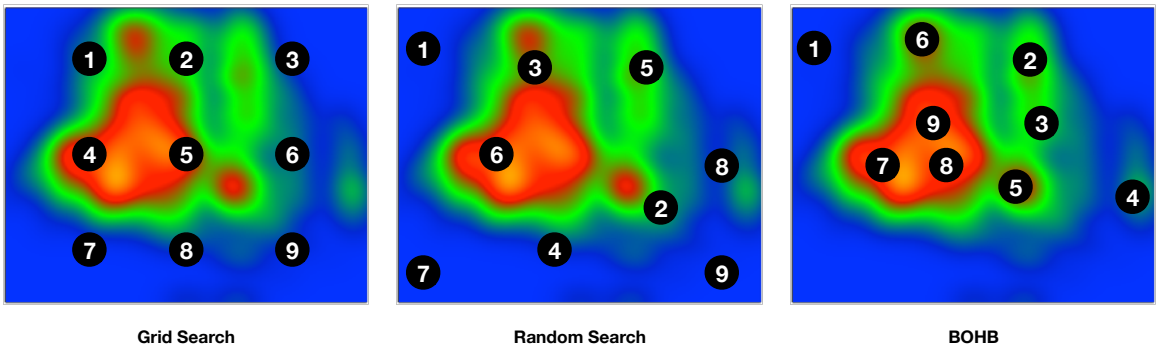


Figure 7.8: Flow of training trials during parameter tuning following 3 different searchings algorithms.

7.4.1 Hierarchical vs. Onefold Approach

The process of tuning hyper and system parameters can be solved in two different ways: 1) non-hierarchically (*i.e.*, onefold), where both parameters are tuned together in a one-tier manner, and 2) hierarchically, initially tuning the hyperparameters, and then the system parameters are tuned only for the set of optimal hyperparameters values found by the first tuning step. The main drawback of non-hierarchical tuning approaches lies in the search space size, as it increases significantly when two sets of parameters are considered together. However, these approaches allow to take the system parameters impact on model training and inference performance into account during the process of finding hyperparameters. Opposed to that, hierarchical approaches treat these two types of parameters independently and tune the hyperparameters in a manner which does not consider the strong dependency of hyper and system parameters. Figure 7.7 depicts the execution flow difference between hierarchical and non-hierarchical (*i.e.*, onefold) tuning approaches. We implement a prototype for each strategy, and compared the results to support our assumption.

7.4.2 Search Algorithm

The core of parameters tuning is the search algorithm (*e.g.*, Random search, HEBO, BOHB, HyperBand). Each of these algorithms follow a specific strategy to define the search space, the most basic ones for parameter search being via random and grid search. Random search is backed up by a variant generator which randomly picks a value in the given interval, while grid search exhaustively tries all the possible values. Optimized algorithms (*i.e.*, Bayesian Optimization HyperBand - BOHB-) implement early termination of bad trials and uses Bayesian Optimization to improve the parameter search. In practice, each type of parameters to be tuned may be specified to follow its own searching algorithm. For instance, a user can choose to tune the number of cores following random search and the batch size following BOHB in the same tuning run. In our context, while user can freely choose the strategy, the default behavior of the current prototype picks the BOHB strategy for all parameters. We focus on BOHB as our novel strategies (*i.e.*, multi-budget) can easily be integrated.

Figure 7.8 shows the parameter tuning problem with three different searching algorithms: grid search, random search, and BOHB. Each point represents a specific parameter configuration. Warmer colors indicate a performance following a given metric of interest. The circled numbers from 1 to 9 indicate the training trials performed in each case. We observe that the trials following BOHB concentrate on the most promising regions of the search space, differently than grid and random search. Given these results, the default behavior in our current implementation is that all parameters are tuned following the BOHB strategy.

7.4.3 Training Trial Budget

We design a novel multi-budget approach and compare it against an epoch based and a dataset based budget.

The state of the art tuning algorithms are based on the concept of multi-fidelity [70], *e.g.*, successive halving, using low-fidelity data to explore the entire system performance and then high-fidelity data to exploit the optimal configuration. Specifically, the small amount of "budget" is used to explore a large number of configurations and then big amount of budget is spent on a small number of promising configurations.

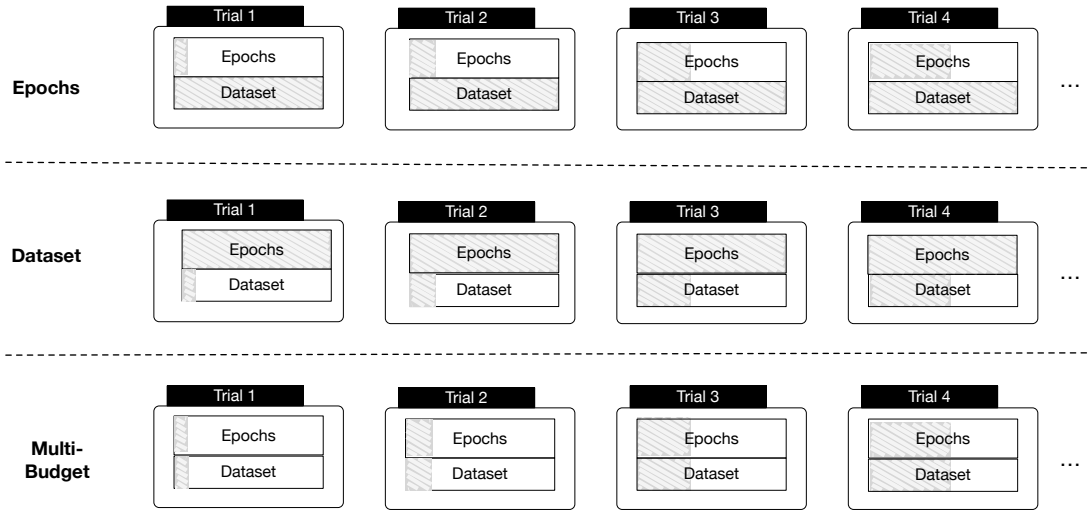


Figure 7.9: Flow of trials for 3 budget approaches: epochs, dataset, and multi-budget.

Typically budgets are defined through training iterations, *i.e.*, epoch, or training data size. While most of the prior art overlooks the impact of different budgets, [229] shows that an appropriate choice of budget type can improve the tuning efficiency significantly. In epoch based approaches, the number of epochs performed in each training trial is proportional to the current iteration (*i.e.*, epochs is equal to two times the iteration level). In a dataset based approach, only one epoch is performed per training trial but the amount of data used in each trial is proportional to the iteration level (*i.e.*, percentage of dataset used is equals to *min(1, iteration_id*0.1)*).

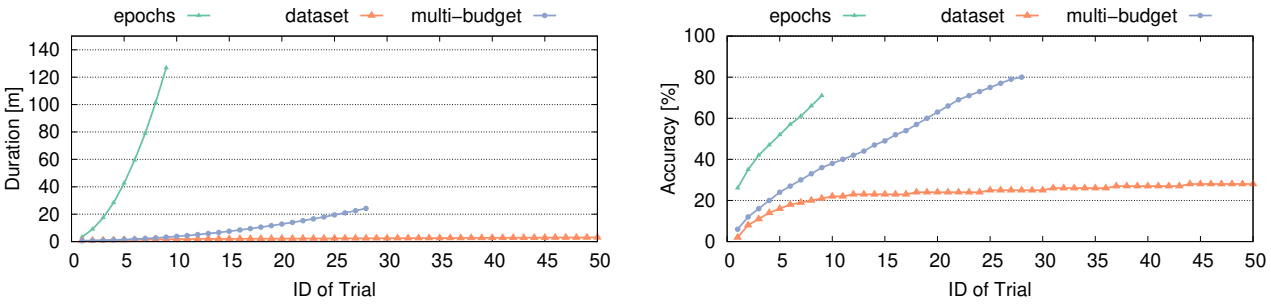
Finally, the multi-budget approach combines both strategies, *i.e.*, each training trial uses a number of epochs and a chunk of dataset which is proportional to its iteration. By combining both approaches, we have a trial which does not take as long as of we would use the entire or run for a fixed number of epochs. At the same time, we run it for long enough or with enough number of samples to make the accuracy of a trial representative. Figure 7.9 depicts the trial execution flow of the 3 above described budget approaches.

In the multi-budget approach (see Algorithm 6), we start with the minimum number of epochs and minimum fraction of the dataset. Then, in every new iteration, the budget for number of epochs and portion of the dataset grows simultaneously and proportionally to the current iteration. For instance, if we start with minimum epochs equals to 2 and minimum dataset fraction 10%, the next iteration will consist of 4 epochs on 20% of the dataset, the third iteration will take 6 epochs on 30% of the dataset, and so on. Although both dimensions grow simultaneously, their maximum limits are set independently. Once the maximum limit for one of the dimension is reached, the maximum value is used further and the other one continues to grow until both reaches their limits. In our example, if the maximum number of epochs is 10, then from the 5^{th} iteration onward, the number of epochs is fixed to 10 and the dataset keeps growing until the 10th iteration when both dimensions reach their maximum values. In summary, the budget for both epochs and dataset is given by *min(max, iteration_id*min_fraction)*).

Figure 7.10 shows the model accuracy convergency over the trials for each of the three approaches described above. In this example we can see that the epoch based approach reaches the target accuracy (*i.e.*, 80%) within few trials but the execution time per trial is extremely high. On the other hand, the dataset based approach has a very low execution time in comparison to the others but the model accuracy does not go higher than 40% even after 100 trials. Finally, the multi-budget approach presents the perfect balance between these two approaches, reaching the target accuracy with more trails than the epoch based approach but with significantly lower trials execution times. Hence, we show how the multi-budget approach is the most appropriated for tuning servers as it finds the sweet-spot between trial duration and accuracy. Having this balance is crucial: the trial duration is critical for the overall performance of the process, but the model accuracy given by a trial needs to be representative enough to avoid advancing the training on non-promising trials.

Algorithm 6 Trial multi-budget algorithm.

```
1: function trial(model, data, hyperparameters, it)
2:     epochs = min(min_epochs*it, max_epochs)
3:     data_frac =min(min_data*it, 1)
4:     data = data.subset(data_frac)
5:     for epoch in epochs do
6:         model.train(data)
```



(a) Training trial time

(b) Accuracy

Figure 7.10: Model accuracy and training time convergency over the trials execution with the 3 budget approaches. Workload: Resnet18 on CIFAR 10.

Now, combining this multi-budget approach with the halving process of algorithms such as BOBH would mean that, besides of the specifications above, the algorithm defines a reduction factor represented by η. This reduction factor defines the fraction of hyperparameters which continues to the next iteration at the end of each cycle. For instance, if $\eta = 2$, then half of the configurations being considered continues to the next iteration at each cycle. This, combined with the multi-budget strategy allows the tuning to try many configuration at low cost and, more importantly, spend more time and resources in the most promising configurations.

7.4.4 Objective Functions and Metrics

The *ModelTuning* objective function is to maximize model accuracy while still considering tuning and inference performance.

As tuning is composed by several training trails, optimizing training performance directly impacts the tuning performance. In current implementation, performance can be defined either in terms of runtime or energy consumption. Therefore, the final optimization function of this server to minimize the metric of interest (time or energy) defined as the ratio performance to accuracy:

1. $ratio = \frac{training\_time * inference\_time}{accuracy}$

2. $ratio = \frac{training\_energy * inference\_energy}{accuracy}$

In the case of the *Inference Tuning Server*, the objective function is defined only in terms of inference performance (*i.e.*, inference runtime or energy consumption). This mean that the objective function is set to minimize the metric of interest defined either as runtime or energy consumption of the inference phase alone.

Those objective function and metrics are used out-of-the-box, as our implementation already takes care of collecting and configuring all the necessary metrics. However, both the objective function and the metric of interest are configurable and can be easily set up. For instance, if the user would like to focus on the inference performance only then another metric of interest could be defined where the training performance is not included.

Table 7.1: Workloads used for experiments.

Type	ID	Model	Dataset	Datasize	Train Files	Test Files
Image Classification	IC	ResNet	CIFAR10	163 MB	50.000	10.000
Speech Recognition	SR	M5	Speech Commands	8.17 GiB	85.511	4.890
Natural Language Processing	NLP	RNN	AG News	60.10 MB	120.000	7.600

7.5 Evaluation

This section presents our extensive experimental evaluation of the EdgeTune prototype, including different considerations from 3 state-of-the-art application domains.

7.5.1 Prototype Implementation

The EdgeTune prototype is implemented in Python (v3.6.8), and consists of 817 LOC. We leverage Tune library from Ray (v1.4.0) to reuse existing implementations of state-of-the-art search algorithms. Our prototype also leverages Tune [60], a Python library for experiment execution and hyperparameter tuning, which provides tuning schedulers based on different optimization algorithms (*i.e.*, BOHB).

For the training and inference of workloads we rely on PyTorch [158] (v1.9.0). This is an imperative-style high-performance deep learning Python library, supporting data structures for multi-dimensional tensors and mathematical operations over these tensors. We rely on torchaudio (v0.7.0) for speech recognition workloads and on torchtext (v0.10.0) for natural language processing ones. Finally, for image classification and object detection workloads, we use torchvision (v0.8.1+cu101).

CUDA support for PyTorch [230] allows these tensors to run on an NVIDIA GPU [231]. We exploit this option, using CUDA (v11.3) combined with NVIDIA driver (v465.19.01) to accelerate the training part of our framework and therefore further optimize overall tuning duration.

7.5.2 Experimental Setup

We begin by detailing our experimental setup, including testbed, baseline, workloads, and as well as the considered parameters and their ranges.

Testbed

We use Titan RTX GPU, Turing architecture with 24GB of Memory and 7.5 compute capability running CentOS Linux (v7.9). Power metrics are collected using the library PyRAPL (v0.2.3.1) **pyRAPL** which is a software toolkit to measure the energy footprint of a host machine along the execution of a piece of Python code.

Baseline

Our baseline system (*i.e.*, Tune) uses the tuning of hyperparameters ignoring all system parameters and the inference phase. For a fair comparison, we configure Tune to use the same searching algorithm as EdgeTune (*i.e.*, BOHB).

Workloads

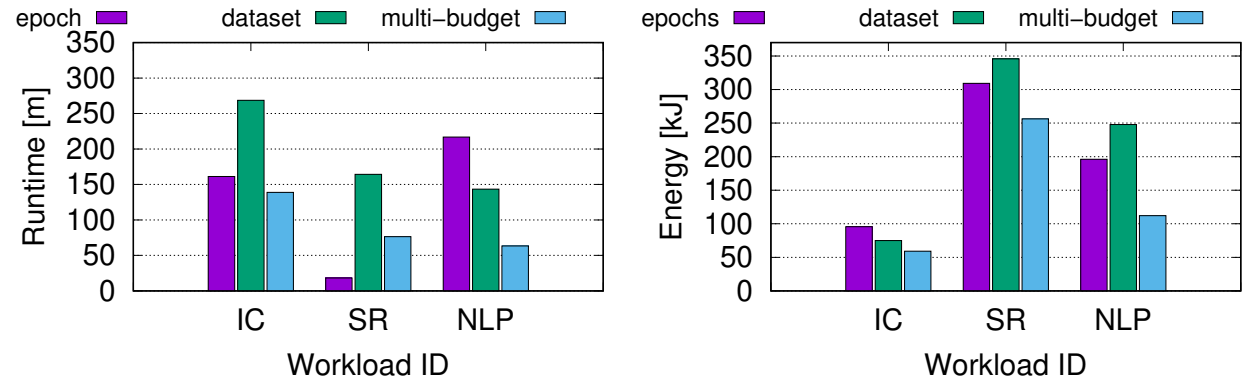
Table 7.1 summarizes the properties of the selected workloads for evaluation including image classification, speech recognition, and natural language processing problems. The CIFAR10 [68] dataset consists of 32x32 colour images labeled in 10 mutually exclusive classes. Speech Commands [232] is an audio dataset of spoken words designed to help train and evaluate keyword spotting systems. AG News [233] is a collection of news articles gathered from more than 2000 news sources spanning 1 year of activity.

Training Hyperparameters

We study the impact of the training batch size, *i.e.*, the number of samples are aggregated to perform the training. The range of values considered in our setup vary from 32 to 512, across all the workloads.

Model Hyperparameters

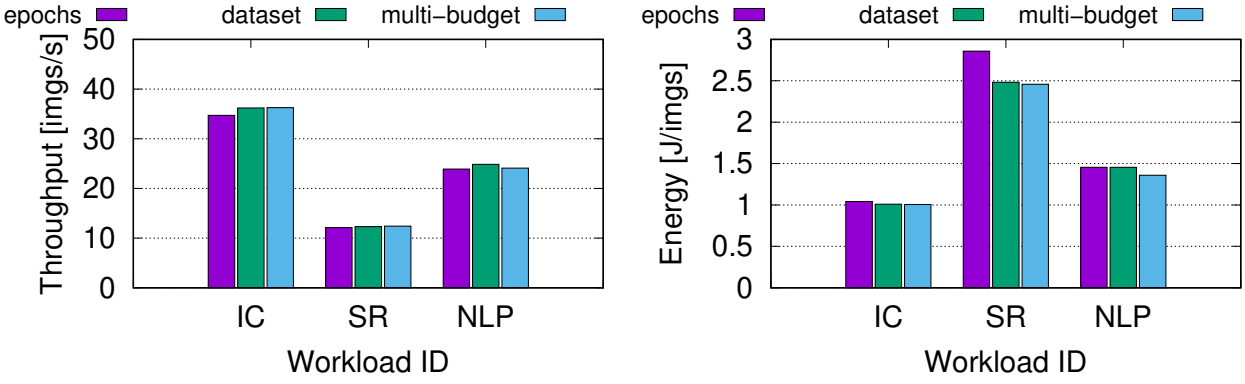
The model hyperparameter considered for ResNet was *number of layers*, with the possible values being 50, 34, and 18. In the case of M5 (*i.e.*, the speech recognition case), the model hyperparameter considered



(a) Tuning duration

(b) Tuning energy

Figure 7.11: Tuning duration and energy consumption of the 3 budget approaches for 3 different workloads.



(a) Throughput

(b) Inference energy

Figure 7.12: Throughput and sample energy consumption of the inference phase using the suggested parameters and configurations given by each of the 3 budget approaches.

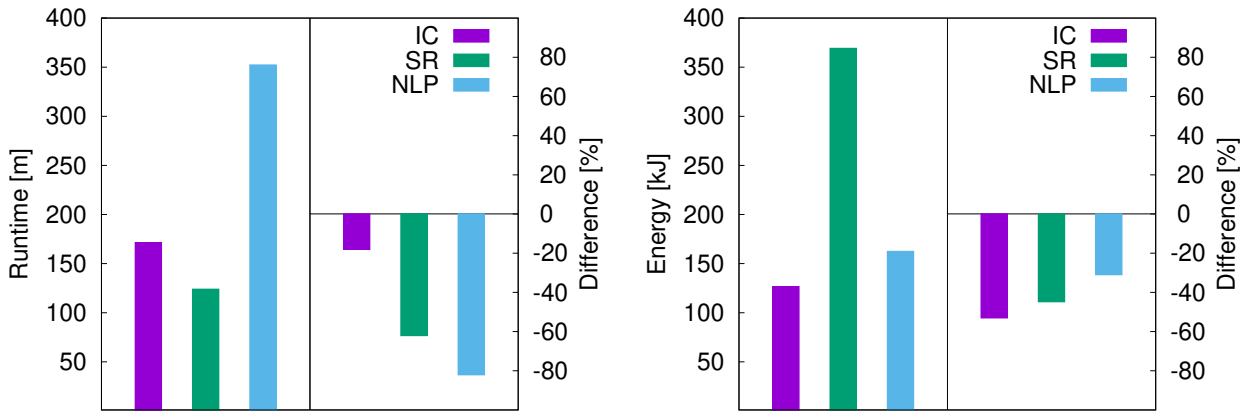
is the *embedded dimension*, assuming values 32, 64, or 128. For the RNN model (*i.e.*, for natural language processing), we tune the *stride* parameter, as it maps to the amount of movement over the image or video. For example, if a neural network's stride is 1, the filter moves one pixel, or unit, at a time. In our setup, the stride value vary from 1 to 32.

System-parameters

Regarding training system parameters, we consider the number of GPUs to be used by the trials which can vary from 1 to 8. For the inference system parameter, we consider number of CPU cores and inference batch size. The inference batch size, from 1 to 100 in our experiments, represents the number of samples used during inference.

7.5.3 Tuning Budget Choice

Given the three possible budget options mentioned earlier (*i.e.*, epoch, dataset, or multi-fidelity), we evaluate our novel multi-fidelity approach against the former two. Figure 7.11 and Figure 7.12 compare these approaches under the tuning and inference perspectives, respectively. For workload IC, we notice that the inference configuration of these 3 approaches are very similar. This is expected: there are different possible optimal solutions, and we run enough trials such that each approach can converge to one of these. However, there are significant differences regarding tuning runtime and energy consumption. We observe



(a) Tuning duration

(b) Tuning energy

Figure 7.13: Tuning duration and energy overhead of EdgeTune in comparison with *Tune* baseline which does not make use of the inference tuning server component.

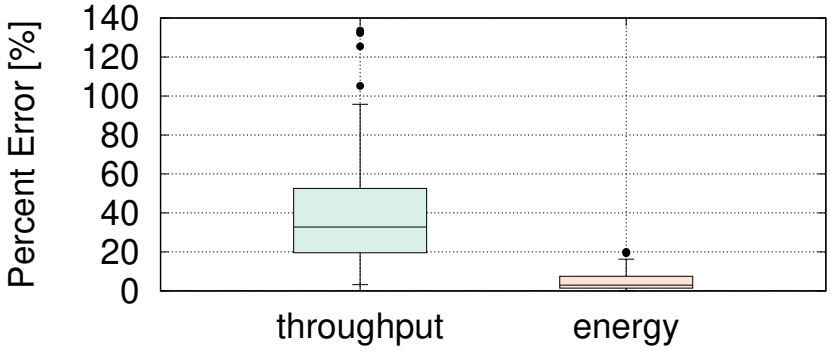


Figure 7.14: Error of throughput and energy consumption estimated by the *Inference Tuning Server* component in comparison with actual values collected on edge device.

that the epoch based approach is better than the dataset approach in terms of runtime but worse for energy consumption. Instead, the multi-budget approach performs consistently better (*i.e.*, shorter runtime and smaller energy consumptions) than the other two in both aspects. This suggests our novel approach can reach comparable inference results to state-of-the-art approaches while being more efficient.

7.5.4 Inference Tuning Server

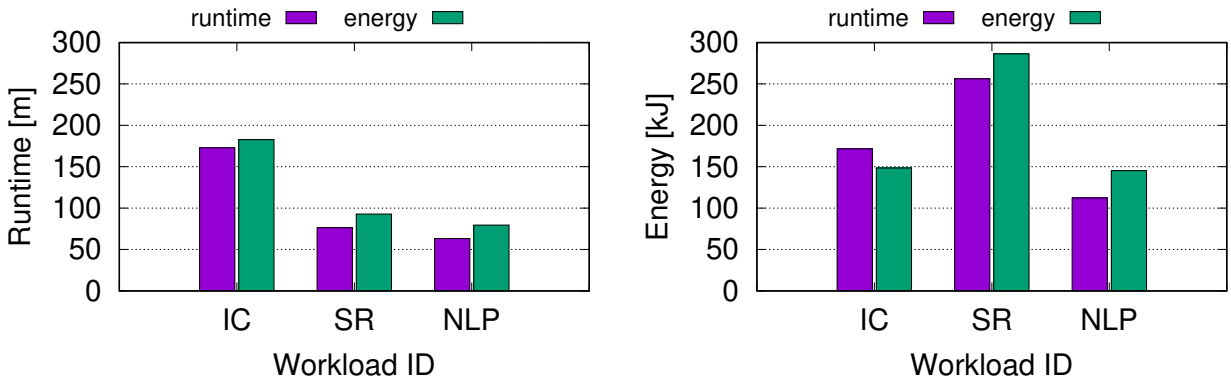
Next, we study the overhead and precision of the *Inference Tuning Server* component, one of EdgeTune's key contributions.

Overhead

Figure 7.13 shows the overhead of the *Inference Tuning Server* component of EdgeTune with respect to Tune which does not have such a component integrated. We can observe that for the workload IC, the tuning duration and energy are reduced by 18% and 53%, respectively. This indicates that the performance gains achieved by considering a multi-objective optimization function compensates the minimal overhead of the *Inference Tuning Server* component. One of the objectives of the optimization function used in our implementation is to minimize the duration on training trials which indirectly also minimizes the overall tuning time.

Precision

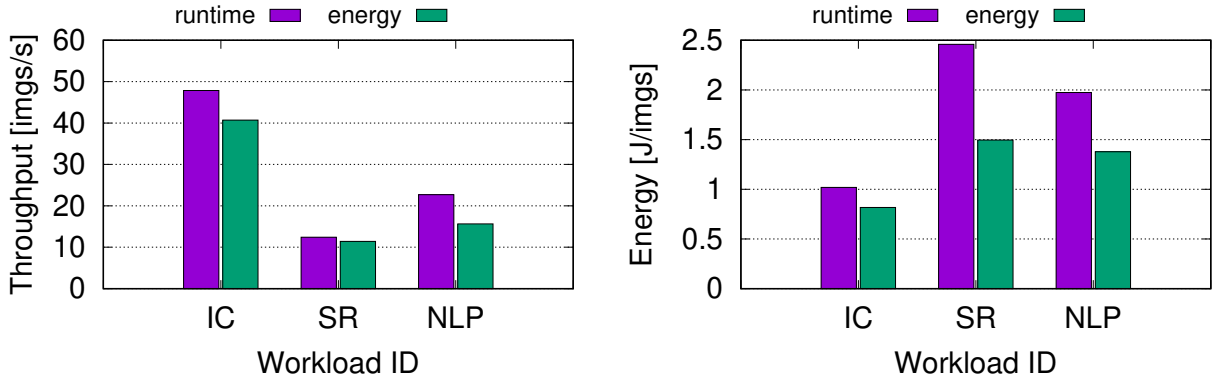
Figure 7.14 uses a box-and-whiskers representation to show the percent error between the runtime and energy estimated by the *Inference Tuning Server* component and the actual metrics collected on an edge device of the simulated configurations.



(a) Tuning duration

(b) Tuning energy

Figure 7.15: Overall tuning duration and energy consumption of the runtime and energy based objective functions.



(a) Throughput

(b) Inference energy

Figure 7.16: Throughput and sample energy consumption of the runtime and energy based objective functions.

The percent error (*i.e.*, PE) is:

$$PE = \left(\frac{|empirical\_value - estimated\_value|}{empirical\_value} \right) \times 100,$$

where $empirical\_value$ is the metric collected on the edge device and $estimated\_value$ is the metric collected in simulation mode.

7.5.5 Objective Function: Runtime *vs.* Energy

Beyond model performance, runtime and energy performance are EdgeTune's main concerns. hence, we have implemented EdgeTune using two different versions of objective function, one optimized for runtime and one for energy, respectively. Figure 7.15 and Figure 7.16 show a comparison between the utilization of these two functions under the tuning and inference perspectives, respectively. For the workload IC, we notice that tuning time is slightly lower and the energy is higher for the function based on runtime than for the energy based function. Moreover, when comparing the inference results of the configurations suggested in each case, both throughput and energy are higher for the runtime function than for the energy one. Although the differences are not significantly large (at most 20% for runtime and 29% for energy), we can notice the focus of the functions affecting the direction of the achieved results. We explain this by the fact that energy is often strongly correlated with runtime and, by optimizing runtime we are also indirectly optimizing energy consumption.

7.6 Summary

This chapter presented the design, implementation and evaluation of EdgeTune, a novel edge-based parameter tuning framework that simultaneously considers multiple parameters and is inference-aware. To achieve the one-fold tuning, EdgeTune is composed of tuning and inference servers which can asynchronously and parallel explore large space of model and inference system parameters. By tuning multiple interlaced parameters, EdgeTune can achieve higher model accuracy and energy-efficiency inference at a lower tuning cost, compared to the existing solutions, e.g., non-inference aware tuning and hierarchical tuning. We have designed a novel multi-budget tuning algorithm that flexibly exploit the pros and cons of different budget types, overcoming the limitations of the state of the art multi-fidelity tuning algorithms. We evaluated EdgeTune on three popular AI workloads on three edge systems, against different combinations tuning solutions. The exhaustive evaluation has shown that EdgeTune can also achieve 20% tuning improvement on runtime and 50% on energy.

To summarize, the main contributions of this chapter were:

1. EdgeTune, a novel tuning system which is inference-aware, striking the right balance between model performance and inference latency;
2. The proposed approach covers multi-parameters tuning in onefold solution which succeeds to achieve multiple layers of optimization objectives combined;
3. We introduce a multi-budget approach for the training trials which reduces their runtime but maintains the level of accuracy necessary for efficient convergency;
4. We demonstrate performance gains on runtime and energy measurements using state-of-the-art workloads.

This work fill the inference gap from the first use case discussed in Chapter 5. Combined, these two use case applications close the tuning, training and inference cycle of learning applications in an energy aware manner by leveraging their heterogeneous characteristics and hardware requirements. However, another major concern of these applications is regarding the security and privacy of customer data. This is a general concern when it comes to cloud providers, but, besides, learning applications usually rely on user's data to achieve accurate models which makes these type of application even more sensitive to the problem. In the next chapter we discuss this perspective of the problem in more details and propose one approach to also cover this requirement.

Chapter 8

In the Light of Security

As we discussed in the first chapters, a common way of improving performance is by developing applications in terms of tasks building blocks. However, this is not a straightforward process which also raises several orchestration challenges. Later, we discuss training and inference applications which are often performed using user's sensitive data. Therefore, preserving privacy becomes another major concern of such applications which until now is overseen by our work. Considering that, in this chapter we propose SGX-OmpSs, a novel approach to fill this gap by combining task-level parallelism with hardware assisted protection mechanisms.

8.1 Introduction

The ability to efficiently execute applications, *i.e.*, performance and energy-wise, while offering security guarantees, is a cross-cutting concern that spans across several types of domains. For consistency, in this chapter we focus on deep learning motivating scenarios. As discussed earlier, deep learning applications achieve high accuracy on non-trivial prediction problems, including speech recognition [191] or machine translation [234]. However, these applications are known to be resource-eager and time-consuming, in particular during the training phase of their models. Note however that our contributions, discussions, frameworks and results are easily translatable to applications domains beyond deep learning.

Efficiency Factors

There are three key factors of paramount importance to provide *efficiency* in this context and that, when combined, make the problem complex:

1. pure performance (*i.e.*, execution time, latency, throughput),
2. security guarantees (*i.e.*, how much protection a system require), and
3. energy requirements.

First, training can take too much time due to its compute intensive nature coupled with the need to go through a large number of input data sets in batches. This is particularly relevant for server-grade deployments (on-premises or outsourced over public clouds), where the energy costs must be kept under control. For instance, training natural language understanding models on a cluster of commodity hardware requires 11.5 hours [235]. When it comes to inference, the resource demands are much lower, in particular if executed on edge devices, where usually the inference latency becomes critical.

Secondly, DL system must increasingly respect strong security requirements. While deep learning has the potential to be applied in several privacy-sensitive domains (*i.e.*, health, finance, *etc.*), strict guarantees regarding data access control must be preserved. In addition, under certain circumstances, such system must be resilient to compromised cloud providers [236].

Thirdly, the energy footprint is a major concern, especially due to the long training time [237]. The energy consumption of training models is directly related to the minimum accuracy required by users. The more accurate a model has to be, the more resources are needed to achieve the desired results. Considering this together with the fact that model accuracy is a critical measure for most users, the

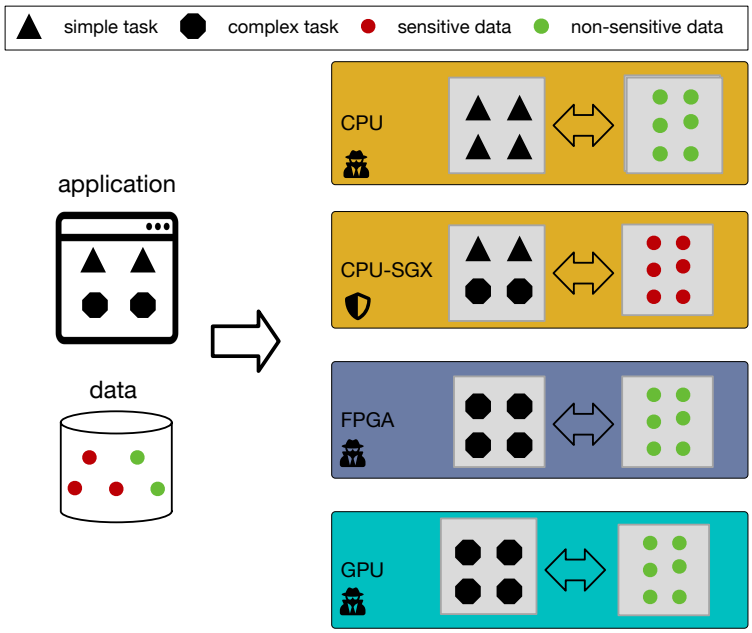


Figure 8.1: The key features of SGX-OmpSs: security, asynchronous task and data based parallelism, and hardware heterogeneity.

energy footprint of these applications quickly becomes a concern. Moreover, the energy consumption on the inference side is also critical if we consider that the battery life of IoT devices can be extended with such optimizations.

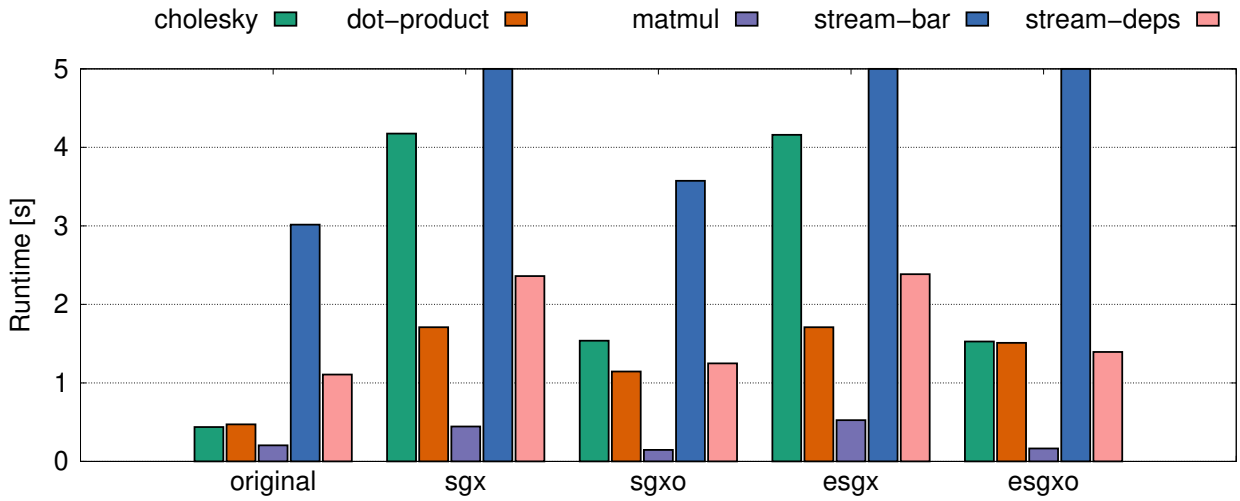
These three key factors vary in relevance and weight according to the type of application executed, as well as the context and environment. Our key observation is that these requirements can be efficiently matched all together by exploiting a high degree of hardware heterogeneity that is commonly found in modern computing facilities. In fact, certain types of applications, or well-identified sub-components within the same application, perform the best if executed by specific hardware components. In essence, a set of requirements for a given application or ensemble of applications can be better fulfilled by an heterogeneous set of hardware resources.

Parallelism

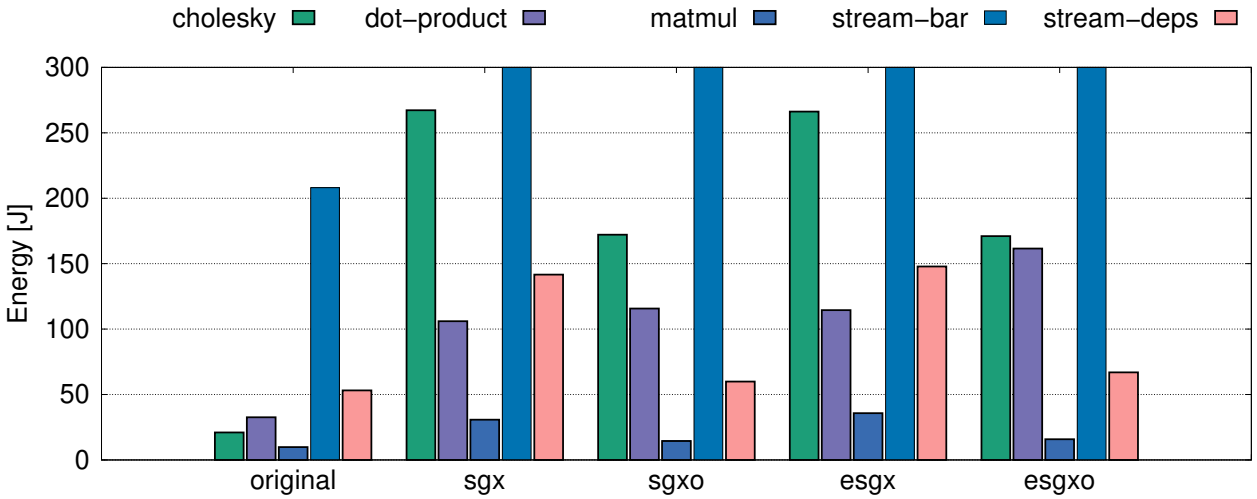
Task-level parallelism, sometimes referred to as functional, DAG, unstructured, or thread parallelism [238], is an efficient approach to address the performance requirements of applications. In this context, a task refers to a program statement of arbitrary granularity. As tasks running in parallel do not depend on each other, this paradigm natively support hardware heterogeneity and asynchronous data movement which efficiently improves performance. However, manually applying task parallelism to existing applications is not a straightforward process [239] and requires in-depth and time-consuming domain knowledge from developers. Existing programming models, *e.g.*, OpenMP [240], charm++ [241], OmpSs [2], facilitate this process by automatizing the management of data dependencies and shared memory environments.

Security

When it comes to addressing the security requirements of applications, a recent approach is to leverage the newly available off-the-shelf hardware-assisted memory protection mechanisms [242]. Examples of such mechanisms include AMD's Secure Encrypted Virtualization (SEV) [243] or Intel Software Guard Extensions (SGX) [244]. These mechanisms allow user-level as well as operating system code to define private regions of memory whose contents are transparently encrypted, providing protection (*i.e.*, preventing either read or writes) from any process outside of those regions.



(a) Runtime.



(b) Energy.

Figure 8.2: Comparison of (a) runtime and (b) energy results for different applications and variants. We include results for: the original version (`original`), Intel SGX SDK (`sgx`), `sgx` with OmpSs (`sgxo`), `sgx` with encryption (`esgx`) and `esgx` with OmpSs (`esgxo`).

Considering that, we propose an approach that combines task based programming models with hardware-assisted memory protection mechanisms. By combining these two approaches, we cover a larger range of applications requirements in a flexible and convenient way for developers. More specifically, we design, implement and evaluate an approach leveraging Intel SGX and the OmpSs programming model. The main idea behind this approach is that we can provide secure tasks backed by Intel SGX while at the same time reduce the overhead of doing so by adding task parallelism and heterogeneity via OmpSs. Figure 8.1 depicts the key features behind this idea.

Motivating Example

In order to understand the potential benefits of the proposed approach, and quickly assess the validity of our idea, we execute a suite of micro-benchmarks with a fine-grained analysis of 5 applications:

1. Cholesky factorization (*i.e.*, a decomposition of a Hermitian, positive definite matrix into the product of a lower triangular matrix and its conjugate transpose);
2. dot-product;

3. matrix multiplication; and
4. two versions of Stream, a synthetic benchmark that measures sustainable memory bandwidth and the corresponding computation rate for a simple vector kernel.
   (a) version I makes uses of barriers to ensure that the dependent functions execute one after another; and
   (b) version II does not make use of barriers, instead, keeps the correctness by means of data dependency detection.

These applications are further detailed in Section 8.3.2. Nonetheless, in a preliminary analysis we intend to compare the level of parallelism achieved by OmpSs alone (*i.e.*, the original version) and in the applications' versions which have Intel SGX integrated. For that, we vary several parameters such as task input size or the scheduling policies (*i.e.*, a scheduling policy has to decide the order of execution of tasks and the resource where each task will be executed). Then we perform a coarse-grained analysis where we focus more on the overall performance and show that our observations also hold for more complex applications, in particular dealing with Deep Learning.

We compare the results between 5 different variants of the same program:

1. the `original` one, without any modification and without using either OmpSs or SGX, and running a single thread outside protected memory;
2. a plain port of the original one using the Intel SGX SDK (`sgx`);
3. an SGX version adapted to leverage OmpSs (`sgxo`);
4. `esgx`, in which we use a single thread inside the SGX enclave, and for which all input data is encrypted;
5. `esgxo`, which further integrate `esgx` with the OmpSs programming model.

We note that `sgx` and `esgx` pay the additional overhead of decrypting the data before processing and re-encrypting it once the computation is complete. Additionally, `sgxo` and `esgxo` execute in a multithreaded environment, and require the applications to be annotated with custom OmpSs and SGX annotations.

While an in-depth discussion on the design and implementation of such variants is provided in Section 8.2, the results in Figure 8.2 allow to derive some immediate conclusions. First, they show a clear overhead of adding security to an application, in our experiments in the range of 113-854%. Secondly, if we leverage task based parallelism strategies such as OmpSs then we can minimize this cost by up to 73%. Importantly, in some cases we can achieve performance comparable with or better than the original application with no security, parallelization or heterogeneity support. Finally, we note that in this context the energy consumption is a direct reflection of the performance. Even though the multitasked version consumes more resources, we can also make similar conclusions regarding this spectrum of the problem.

The reminder of this chapter is organized as follows. In Section 8.2, we present our proposed approach to integrate security to the applications discussed until here. Then, we present evaluation results of the approach in the form of microbenchmarks (Section 8.3) and microbenchmarks (Section 8.4) to compare our solution with state-of-the-art approaches. Finally, we conclude in Section 8.5 by summarizing our contributions.

8.2 SGX-OmpSs Approach

We propose an approach that combines the OmpSs programming model and Intel SGX. Figure 8.3 depicts the proposed approach, which we describe next. First, the programmer has to adapt the tasks to SGX which means defining the tasks as C/C++ functions which will be declared and implemented using the Enclave's interface (❶). This process could be automatized but in the current version of our approach it is still a manual step.

Then, the parallelization pragmas can be added to the secure application in the same way as it is done for non-secure application except that all the pragmas have to be placed in the unprotected side of the

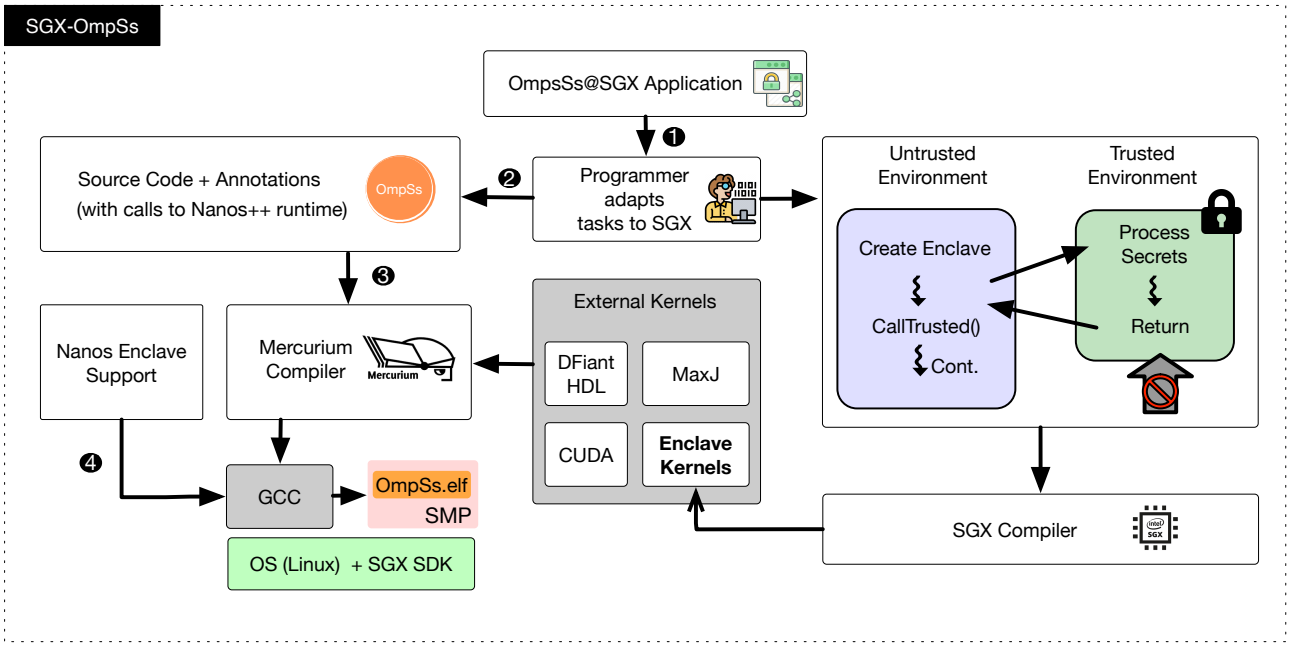


Figure 8.3: The proposed approach, integrating Intel SGX and OmpSs.

application (❷). For instance, if the user wants to annotate a function declared inside the Enclave then the pragmas have to be placed on the function call instead of its declaration. Next, the annotated code is compiled using the Mercurium compiler (❸). As Mercurium is a source-to-source compiler, it generates source code to be ultimately compiled with a backend compiler which is usually a native compiler although cross compilation is also supported. This means that Mercurium itself does not transform any OmpSs pragma but instead it processes the code and generates it again to be compiled by the backend compiler defined by the user (*e.g.*, C, C++ and Fortran). Finally, the Nanos++ runtime schedules the application tasks on the platform (❹).

Algorithm 7 shows the main fragment of a matrix multiplication application integrated with Intel SGX SDK and annotated with OmpSs pragmas. The code refers to the previously defined matrices A, B and C, where A and B are input and C is the matrix which will store the calculated result. Lines 4-6 iterate through dimension (`DIM`) of these matrices which by definition are all the same. Line 7 defines the first OmpSs pragma where we define which variables are input, output or both. In this same line we also use the flag `no_copy_deps` which avoids the OmpSs default behavior of copying the specified task dependencies with separate memory address spaces. This step is crucial to our approach as the memory management of protected tasks has to be entirely handled by Intel SGX. This pragma is then applied to the function call in Line 8 which is responsible for calculating the multiplication of a specific block (`BSIZE`) of the matrices A and B, and storing it directly in the final matrix C which is shared among all the tasks.

Although the result matrix is shared, each task accesses a different block of it so no synchronization among the tasks is needed. However, before returning the final result we need to make sure that all the calculations have been completed and therefore a second OmpSs pragma is used (Line 10) as a barrier waiting for all the tasks to finish. Finally, the variable `global_eid` also passed as parameter to the function `ecall_matmul` refers to the ID of the application enclave which is going to be attested. This is the crucial step for providing security to the operations performed in the ecall function being called. In this context, the authentication mechanism used for attestation uses a symmetric key system where only the enclave verifying the report structure and the enclave hardware creating the report know the key, which is embedded in the hardware platform.

Once the application is ready to be compiled, the next step is to properly configure the enclaves. Several configuration options exist, including: the number of Thread Control Structures (`TCSNum`), the minimum and maximum stack size per thread (`StackMinSize` / `StackMaxSize`), and the initial heap size for the process (`HeapInitSize`) [245]. With the application configured and compiled, scaling the number of tasks

Algorithm 7 Main fragment of annotated matrix multiplication application.

```
int SGX_CDECL main(int argc, char *argv[])
{
  ...
  double *A, B, C = (double *) malloc(DIM * DIM * sizeof(double));
  fill_random(A); fill_random(B); fill_random(C);

  for (i = 0; i < DIM; i++)
    for (j = 0; j < DIM; j++)
      for (k = 0; k < DIM; k++) {
        #pragma omp task in(A[i][k], B[k][j]) inout(C[i][j]) no_copy_deps
        ecall_matmul(global_eid, &A[i][k], &B[k][j], &C[i][j], BSIZE);
      }
  #pragma omp taskwait
  ...
}
```

is a matter of setting the runtime parameter `NX_SMP_WORKERS` to the desired number of tasks. Apart from the number of threads, there is a number of other parameters which could be set at runtime regarding scheduling and throttling policies, instrumentation modules, dependency managers, and others.

The Mercurium Compiler also supports the utilization of external kernels such as DFiant HDL [246], MaxJ [247], CUDA [248], and the native Enclave Kernels which is the one we use in this work. In this case, the Nanos++ installation also has to be configured with the external compiler (*i.e.*, represented by *Nanos Enclave Support* in our approach figure) and the driver simply calls this compiler generating additional files.

8.3 Microbenchmarks

In this section we evaluate the proposed approach by means of micro-benchmarks consisting of the following applications: Cholesky factorization, Dot Product, Matrix Multiplication and two versions of the Stream benchmark. We analyze each of these applications in the following 3 versions: a pure OmpSs implementation, an implementation combining OmpSs and SGX, and a third one adding encryption on top of the OmpSs and SGX combination. All the applications used here take as input a matrix and the parallelization is done based on the blocks of this input matrix. Considering that, the size of a given input matrix indirectly defines the graph size (*i.e.*, the number of total tasks spawned). Moreover, the block size in which we split the matrix defines the input size of each task as the larger a given block is the more data has to be processes by a given task. We then evaluate how each version performs when we vary the matrix and block size as well the scheduling algorithm. Finally, We also vary the number of threads to see how the applications scale for multiple threads.

8.3.1 Environment

We deployed our experiments on an Intel E3-1275 server machine with 4 CPU cores and 2 threads per core, 64 GiB of RAM and 480 GB SSD drives. The machines ran Ubuntu Linux 20.04.1 LTS on a switched 1 Gbps network. Power consumptions are reported by a network-connected Lindy iPower Control 2x6M Power Distribution Unit (PDU), which we query up to every second over an HTTP interface to fetch up-to-date measurements for the active power at a resolution of 1W and 1.5% precision.

8.3.2 Applications

In order to perform this evaluation, we have implemented 3 versions (*i.e.*, original, *sgx* and *esgx*) of 5 different applications (*i.e.*, Cholesky kernel, Dot Product, Matrix Multiplication, and Stream with and without barriers). Next we describe these applications in more details as well as how the task parallelism has been implemented for each of them.

Cholesky Kernel
Cholesky Kernel is a decomposition of a Hermitian, positive-definite matrix into the product of a lower triangular matrix and its conjugate transpose. The kernel uses four different (traditional) linear algebra algorithms: POTRF, TRSM, GEMM, and SYRK. The way we parallelize the code is by annotating these kernel functions so that each call in the previous loop becomes the instantiation of a task.

Dot Product
Dot Product is an algebraic operation that takes two equal-length sequences of numbers and returns a single number obtained by multiplying corresponding entries and then summing those products. A common implementation of this operation accumulates the result of each iteration on a single variable. This kind of operation is called reduction, and it is a common pattern in scientific and mathematical applications. There are several ways to parallelize operations that compute a reduction but in our implementation we use a vector to store intermediate accumulations. This means that tasks operate on a given position of the vector and the parallelism is determined by the vector length. Finally, when all the tasks are completed then the contents of the vector are summed up.

Matrix Multiplication
The Matrix Multiplication application receives matrix A and matrix B as input, performs the multiplication calculations between A and B, and outputs the result in a third matrix C. One of the constraints of this operation is that the number of columns in the first matrix has to be be equal to the number of rows in the second matrix as a given resulting cell C_{ij} is given by the dot product of the ith row in A and the jth column in B. As each resulting cell in the matrix C is calculated independent of the others, the parallelization is done by splitting the input matrices in pre-defined blocks which to be calculated in parallel.

Stream benchmark
The Stream benchmark [249] is a simple synthetic benchmark program that measures sustainable memory bandwidth (in GB/s) and the corresponding computation rate for simple vector kernel. We present the following two versions of this application: one that inserts barriers and another without barriers. The behavior of the version with barriers resembles the OpenMP version, where the different functions (*e.g.*, copy, scale) are executed one after another for the whole array. In the version without barriers, functions that operate on one part of the array are interleaved and the OmpSs runtime keeps the correctness by means of the detection of data dependencies.

8.3.3 Performance
In the following, we consider 3 versions of the above described applications:

1. `original`, without any security;
2. `sgxo`, with Intel SGX integration; and
3. `esgxo`, an extension of the second adding encryption to it.

Before diving into the configurations details, a generic observation we can make is regarding performance. It is clear for all the applications and configurations used that adding a layer of security based on Intel SGX brings a performance overhead.

In the Cholesky application, the maximum runtime overhead observed was 845% which occurred in the case where *breadth first* was used as scheduling policy (see Subsection 8.3.5 below), the task graph size was large and the tasks' inputs were small. In the dot-product we observe an overhead of up to 261% for the SGX version when compared with the original version. In this context of Matrix Multiplication, we observe at least 117% and 156% overhead for the SGX and SGX plus encryption versions when compared to the pure OmpSs version, respectively. Finally, in the Stream application, independent of the version used we could observe that the dependency graph performs better than using barriers. For the SGX versions this observation becomes even more evident as the overhead of running all the tasks inside the enclave, encrypting and decrypting input and output data add an extra cost. Finally, another observation is that the encryption overhead is much larger in the barrier version than in the dependency graph version. In fact, in the latter, we do not observe a significant variation between the SGX versions with and without encryption.

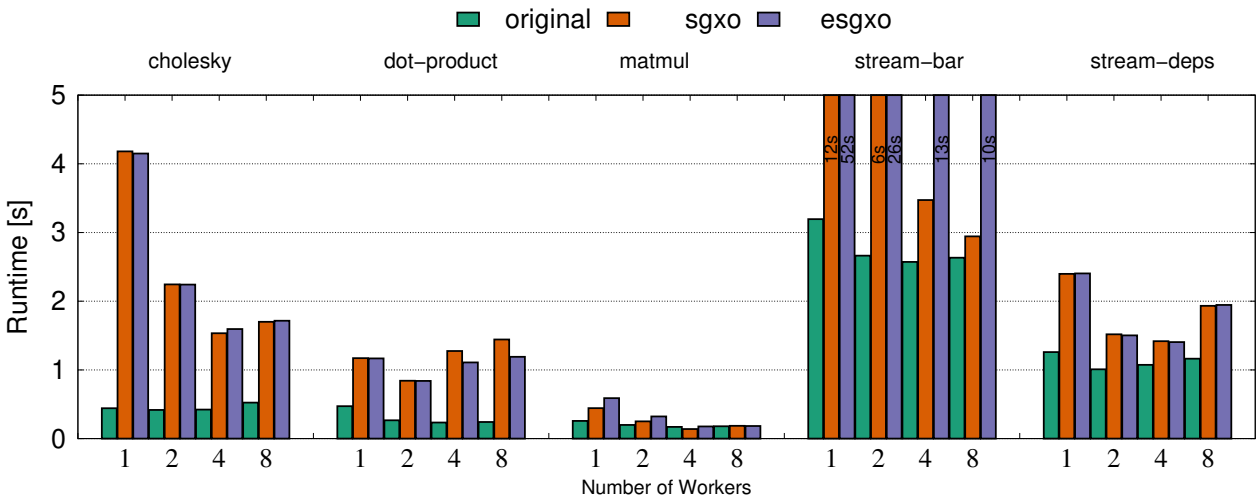


Figure 8.4: Comparison of runtime for different number of workers using original, `sgx` and `esgx`.

Although we observe a significant overhead for all the secure versions, next we discuss a way of overcoming this problem by leveraging the OmpSs annotated code to increase the number of workers. With this we can conclude that although there is a clear tread-off between performance and security, we can minimize the extra cost of adding a security layer by combining SGX with OmpSs.

8.3.4 Number of Workers

Here, all the three version of the applications have been annotated to run with OmpSs. We are then interested to see how much they can scale once the number of workers is increased. In order to increase this number we make use of the parameter `NX_SMP_WORKERS` at runtime which sets the number of OMP threads in the master. In Figure 8.4 we show the results of this experiment varying the number of workers from 1 up to 8.

In Cholesky, the overhead of the secure version could he reduce by 64% using 4 workers. On the other hand, for the Dot Product application the optimal configuration was 2 workers which is 28% faster than the sequential version. Moreover, with 4 and 8 threads the performance starts to decrease again and this can be due to the overhead added when compared with the size of tasks. The Matrix Multiplication presents a runtime improvement of 43% when using only two workers and it increases to 68% when using 4 workers. With the Stream application, both SGX versions present an improvement of at least 36% when using multiple workers when compared with their respective sequential versions. This improvement goes up to 77% depending on the configurations used.

In general lines, the sgx based versions perform worse than the original versions of the application as there is an overhead which comes from the security layer added. In the original version, there is no communication synchronisation required between secure and insecure tasks as no task is placed inside of the enclave. However, we can also observe that this overhead is amortized once the number of workers is leveraged.

8.3.5 Scheduling Policy

Here we evaluate how the scheduling policy being used can affect the efficiency of the considered applications. In this context, a scheduling policy defines how ready tasks are executed. Ready tasks are those whose dependencies have been satisfied and, therefore, their execution can immediately start. The scheduling policy has to decide the order of execution of tasks and the resource where each task will be executed. Nanos++ implements several scheduling policies but for the purpose of this evaluation we will focus on two of them, named *breadth first* and *work first*.

The breadth first policy only implements a single/global ready queue. When (1) creating a task with no dependencies, or (2) a task becomes ready after all its dependencies has been fulfilled, it is placed in this ready queue. The ready queue is ordered following a FIFO (*i.e.*, first in, first out) algorithm by default,

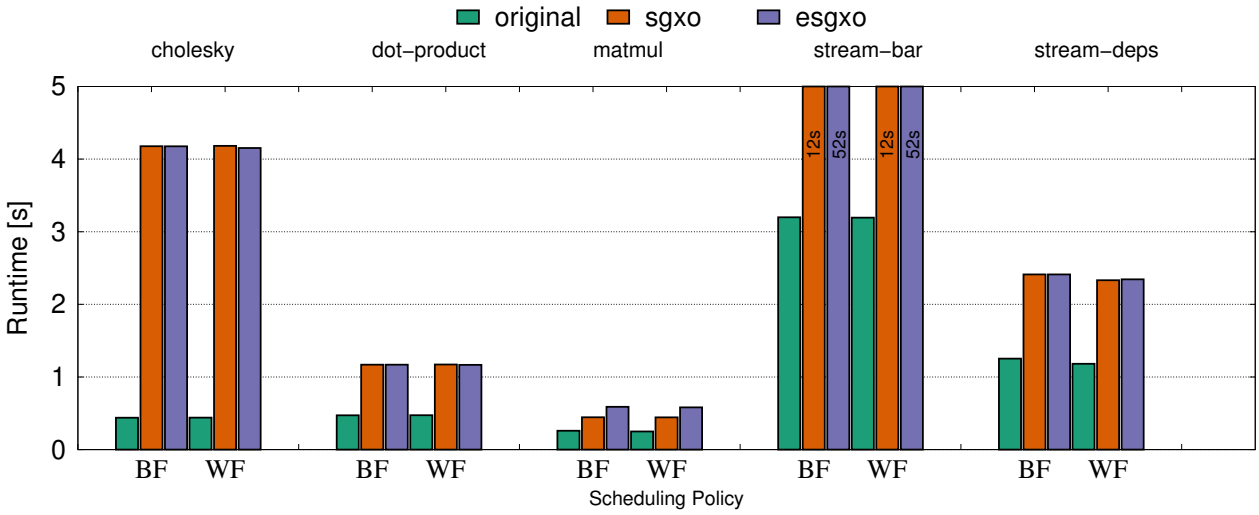


Figure 8.5: Comparison of runtime for two different scheduling policies using original, `sgx` and `esgx`.

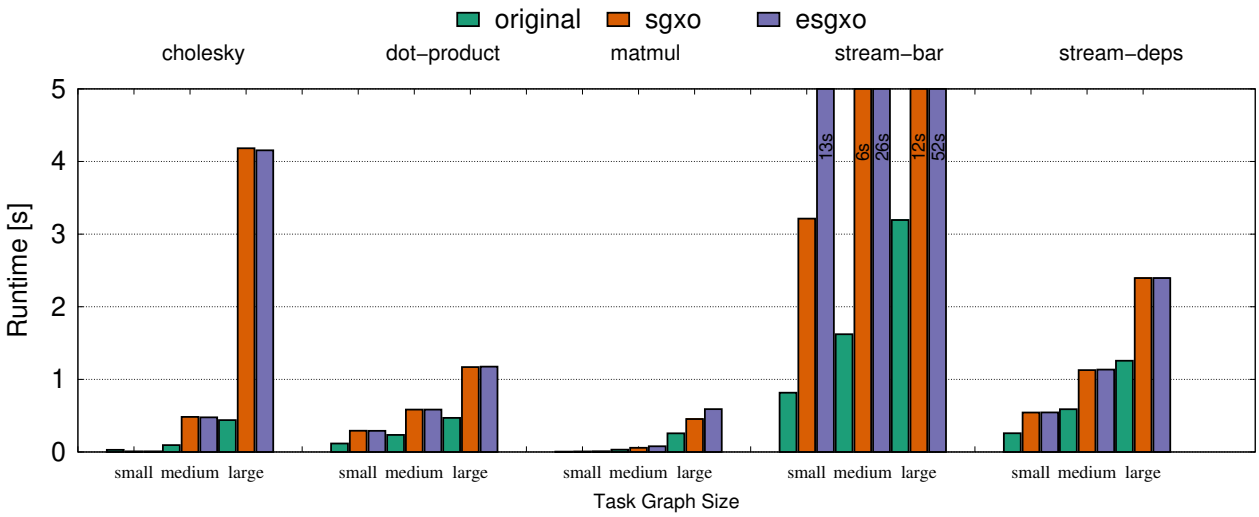


Figure 8.6: Comparison of runtime for different task graph sizes using original, `sgx` and `esgx` versions.

but it can be changed through parameters. In short, breadth first implements an immediate successor mechanism by default, unless in conflict with priority.

The work first policy implements a local ready queue per thread. Once a task is created it chooses to continue with the new created task, leaving the current task (creator) into the current thread's ready queue. The default behavior is implemented through FIFO access to the local queue. If local ready queue is empty, it tries to steal parent task if available through LIFO.

Figure 8.5 shows the results observed for these two policies. Although they have different behavior implementations, in the context of our applications we could not observe any performance difference between them. The reason for this is that when tasks don't have a considerable dependency between them then they cannot take advantage of the work first policy which is indeed the case of applications here considered. These results suggest that the scheduling policy does not have a high impact in this kind of applications.

8.3.6 Graph Size

Here we exploit the size of the task graph. The parent task, which is the task being run when the child task is created, contains the task graph with the relations of its children. This limits the tasks to depend

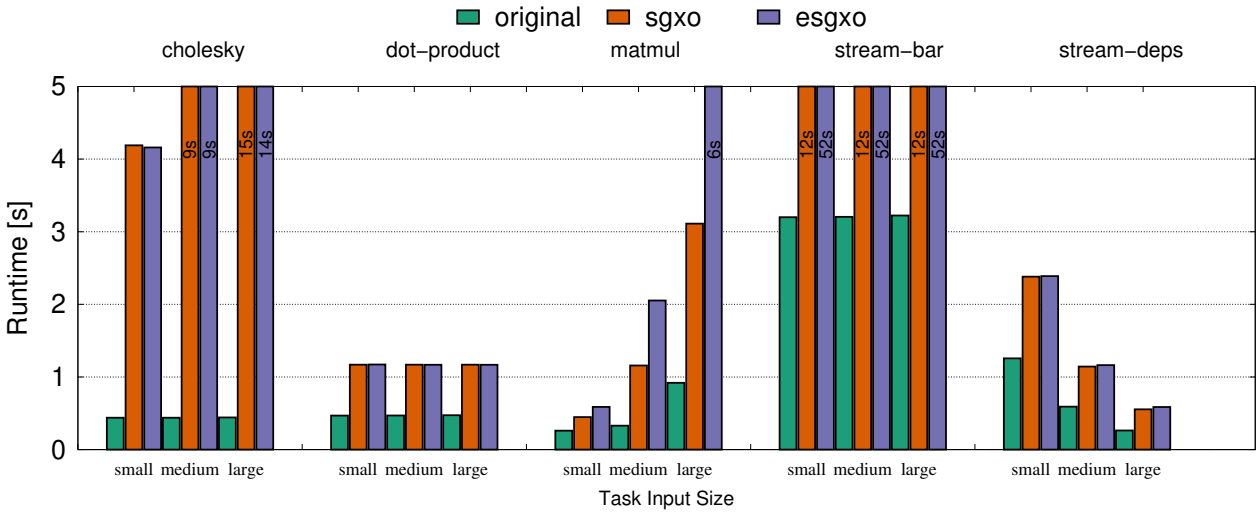


Figure 8.7: Comparison of runtime different task input sizes using original, `sgx` and `esgx` versions.

on only sibling tasks, but the global order is guaranteed because parent task dependencies must be a super-set of its child tasks dependencies. In this context, we increase the global graph size by keeping the input of tasks fixed and increasing the overall number of tasks. Figure 8.6 depicts the results we observe. The general observation we can derive from these results is that the larger the graph size is the more time it takes to be processed. This is also the expected behavior as the larger the graph is, the more tasks have to be completed and the more complex the dependencies between them gets.

8.3.7 Input Size

Now we focus on the input size of tasks and vary it from small to medium and large. In order to vary the input size we fix the task graph and use different values for the blocks of input given to each tasks. As shown in Figure 8.7, this time there is no correlation which we observed for all the applications and therefore the conclusions cannot be generalized.

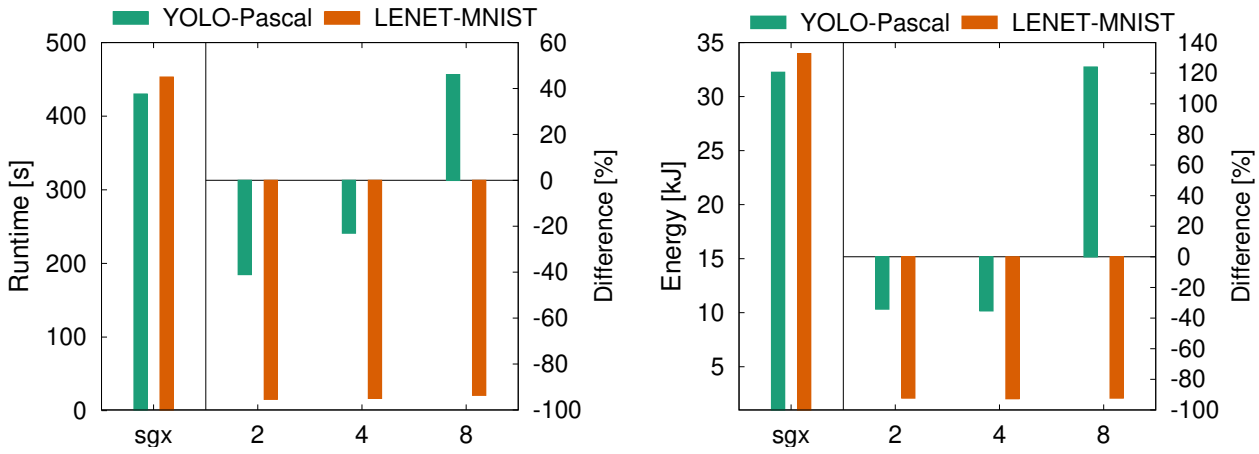
For the applications Cholesky and Matrix Multiplication we observe that the larger the task input is the longer they take to finish. However, for Dot Product and Stream without barriers the performance stays constant, independently of the task input size. Finally, for the Stream applications without barriers the runtime decreases as the task input increases. This happens because the larger the input is the less dependencies exist among the tasks. As this application is based on data dependency, its complexity is directly affected by the task input size.

8.4 Macrobenchmarks

In this section we evaluate the proposed approach by means of macrobenchmarks using two state-of-the-art deep learning workloads (*i.e.*, YOLO-Pascal and LENET-MNIST). In both cases we focus on the inference phase of the workloads which consist of a pre-trained network for real-time object detection. Here we rely on the observations made in Section 8.3 to define scheduler, graph and tasks input size for each workload, trying only different values for the number of workers. Therefore, we only compare a pure SGX (`sgx`) implementation against our SGX-OmpSs approach (`sgxo`) in its optimal configurations. We note that, given the known overhead of encryptions highlighted by our microbenchmarks, the observations for versions `esgx` and `esgxo` can be derived from the results of version `sgx` and `sgxo`, respectively. For these reasons, we omit the results for `esgx` and `esgxo`. The experiments use the same environment previously described in Section 8.3. The optimization percentages reported here correspond to the relative percentage difference between our approach and the baseline (*i.e.*, the difference between our approach and the baseline divided by the baseline).

8.4.1 Applications

To perform the experiments in this section we rely on an object detection and a classification application which we detail next.



(a) Runtime.

(b) Energy.

Figure 8.8: Macrobenchmark results for inference with YOLO-Pascal and LENET-MNIST. The `sgx` version of these workloads compared against the version `sgxo` with 2, 4 and 8 workers.

YOLO-Pascal: Real-Time Object Detection
We use YOLO [250], a state-of-the-art pre-trained network for real-time object detection system. Different than other approaches which apply the model to an image at multiple locations and scales, this approach applies a single neural network to the full image. This network divides the image into regions and predicts bounding boxes and probabilities for each region and these bounding boxes are weighted by the predicted probabilities. Specific to the YOLOv3 version, some tricks (*i.e.*, multi-scale predictions, a better backbone classifier, *etc.*) are used to improve training and increase performance The YOLO system detects objects on the Pascal VOC 2012 dataset [251], for 20 different classes of objects, including person, bird, cat, bicycle, *etc.* The Pascal VOC project provides standardized image data sets for object class recognition as well as a common set of tools for accessing the data sets and annotations.

LENET-MNIST: Classification of Handwritten Digits
The MNIST database of handwritten digits [63] is a subset of a larger set available from NIST, consisting of a training set of 60,000 examples and a test set of 10,000 examples. The original black and white (bilevel) images from NIST are size normalized to fit in a 20x20 pixel box while preserving their aspect ratio. The resulting images contain grey levels as a result of the anti-aliasing technique used by the normalization algorithm, and the digits have been size-normalized and centered in a fixed-size image. In order to perform the classification on this database we rely on LeNet [62], [252], a convolutional neural network.

8.4.2 Experimental Results

These experiments focus on the inference phase of the previously described workloads. Figure 8.8 depicts the achieved results, specifically for runtime (Figure 8.8a) and energy (Figure 8.8b). On the left half of each plot, we show our baseline, *e.g.*, SGX without OmpSs. On the right half, we plot the difference of SGX-OmpSs against the baseline as a ratio, for an increasing number of workers (from 2 to 8). A negative difference indicates a reduction of the runtime, *e.g.*, a faster runtime.

We observe a runtime improvement of up to 41% and 94% for the YOLO-Pascal and LENET-MNIST workloads, respectively for 2 and 4 workers. Power consumption using multiple tasks/workers is higher than for the original version of the workloads. However, in terms of energy consumption, the improved runtime compensates this overhead. Therefore, the performance gains are also reflected in their energy consumption which could be reduced by up to 92% using the SGX-OmpSs approach.

The reason why LENET-MNIST presents a higher improvement than YOLO-Pascal lies in the nature of the models being used. Given an image as input, the approach used by LENET-MNIST applies the model to the image at multiple locations and scales the different results collected. On the other hand, YOLO-Pascal applies a single network to the full image. Considering this, the first scenario makes it easier for the workload to benefit from task parallelism which our approach is based on.

Finally, for the YOLO-Pascal workload we clearly observe that 2 workers configuration achieves the best

configuration, regarding both runtime and energy. Although the difference is not as significant for the LENET-MNIST workload we still observe a slight difference (*i.e.*, approximately 1%) between using 2 and 8 workers. These results support our previous observations indicating the performance gain is not necessarily proportional to the number of workers and has to be analyzed individually for each case.

8.5 Summary

In this chapter we presented SGX-OmpSs, a novel approach to accelerate secure applications based on hardware-assisted protection mechanisms and asynchronous task parallelism. SGX-OmpSs relies on Intel SGX for providing security and on OmpSs for supporting task based parallelism, hardware heterogeneity, and data dependency. For applications which are already SGX-based, the manual modifications are minimal and all the other features are available by using predefined OmpSs pragmas. Using a number of small benchmarks, we were able to clearly identify the performance impact of placing OmpSs tasks inside enclaves, depending of the number of workers, the scheduling policies, the task graph size, and the data input size. Further, our evaluation analysis using state-of-the-art deep learning applications has shown performance gains regarding both runtime and energy consumption. More specifically, we were able to improve 94% in runtime with a real-time object detection application, and 92% in energy consumption with an application for classification of handwritten digits.

To summarize, the main contributions of this chapter were:

1. a novel approach combining OmpSs with Intel SGX;
2. an in-depth evaluation using state-of-the-art applications, including deep learning ones, demonstrating the trade-offs of our approach; and
3. secure applications performance and cost optimization regarding runtime and energy consumption.

Most importantly, with this work we have covered the heterogeneity aspect of clusters and applications also from the security perspective which can be leveraged both by cloud providers and application developers.

Chapter 9

Conclusion

Making the computing process of distributed applications energy efficient is a complex task. These types of applications already have many other requirements and objectives (*e.g.*, performance and security) which cannot always be compromised. However, a starting point to tackle this problem is identifying cases in which energy-related trade-offs are feasible. Another step is to understand the energy profile of different types of hardware available at hand and leverage this information to better allocate resources. Finally, specific characteristics of individual applications can also be used to perform optimizations leading to a more energy efficient version of such applications.

9.1 Summary of Contributions

In this thesis, we proposed approaches to achieve energy efficiency for distributed applications looking from two different perspectives: 1) cloud providers, and 2) application developers. We start in **Chapter 3** with a focus on single-tenant scenarios. Here we proposed a novel task-oriented and energy-aware scheduling policy which targets heterogeneous clusters. Our scheduler keeps track of system metrics and energy utilization of the nodes composing the cluster. Based on this information together with a model trained on historical data and the user specified task requirements, our system performs scheduling and migration of tasks in an energy-aware manner.

Then, in **Chapter 4**, we expand this idea to also cover multi-tenant scenarios. For that, we considered the different priorities which each user might assume and proposed strategies to avoid the eviction of low priority jobs. This strategy is based on the idea of combining differential approximation to deflate low priority jobs instead of evicting them due to the lack of resources to process high priority jobs. This, combined with applying sprinting techniques, allow us to meet the performance requirements of high priority jobs. Therefore, our approach reduces resource waste resulting in a more overall energy efficient approach.

With both single-tenant and multi-tenant scenarios covered, we switch the view from cloud providers to our use case application which we start characterizing in **Chapter 5**. We started by exploring the process of tuning parameters for machine learning applications. Particularly, we showed how these parameters highly impact the overall performance, accuracy and energy consumption of the applications. Then, we deep dive into this problem with focus on deep learning to propose energy efficient approaches by leveraging specific application's characteristics.

In **Chapter 6** we proposed a novel parameter tuning approach which also takes system parameters into account. By making use of the repetitive and highly parallalizable characteristics of deep learning tuning applications, our approach not only avoids overheads but also optimizes the entire process. Besides overall tuning optimizations, as energy efficiency is included in the list of tuning objectives, the training itself also presented reduced energy consumption. However, until this point we have not considered any inference related parameter, although the ultimate goal of these applications is to deploy the final model for performing inference.

Considering this, in **Chapter 7** we extended the previously presented approach to be inference-aware. We achieved this by simulating edge devices in the tuning server to collect performance and energy related metrics for different configurations. By relying on the fact that the system related metrics are entirely

independent of the network's weights, our approach manages to gather this data in parallel with weights tuning and at a very early stage. Our approach also explores a novel budget strategy which manages to converge earlier than other state-of-the-art strategies. The combination of all these ideas together, allowed our approach to output inference-based parameters to the users once the tuning process is complete without any additional cost.

Finally, as mentioned earlier, the applications explored here have already existing requirements apart from energy efficiency. Security is one of these requirements which applications can hardly compromise on. Therefore, in **Chapter 8** we proposed one possible approach to combine the security requirements of applications with the other concepts introduced in this thesis. We relied on SGX to ensure security to the parts of applications which require user privacy. SGX combined with OmpSs programming model offered an automated and efficient way of ensuring security to heterogeneous clusters and applications.

9.2 Research Questions

Now, looking back at our objectives enlisted in Chapter 1 together with the contributions presented throughout this work, we are able to draw the following conclusions.

How can the hardware heterogeneity of clusters be combined with the diverse requirements of applications to achieve a better placement strategy?

First, it is important to understand the energy profile of the different types of hardware forming a cluster. However, this profile is particular to the types of tasks which are being processed. Therefore, it is essential that this profiling phase is combined with the applications running to get a better overall picture of the system. With these profiles and user's objectives well defined, the tasks can be managed in an energy-aware manner using strategies to handle trade-offs between energy and other existing task requirements.

Is there an alternative approach to low-priority jobs eviction in the multi-tenant scenario which leads to less resource waste?

Alternatively to evicting low-priority job, we explore the idea of combining differential approximation with sprinting techniques to reduce resource waste. The main idea is that low priority jobs could be deflated instead of evicted. We show that by approximating the deflation ratio of tasks we can still ensure their required accuracy. Moreover, by sprinting high priority jobs, their performance requirements is also guaranteed. These two ideas combined demonstrate to be a viable alternative to reduce resource waste.

Are there application specific optimizations which can be performed to achieve better energy constraints while preserving the application's requirements?

This is a point specific to the application and environment which has to be analyzed case by base. For the application we considered as use case, we demonstrated such optimizations to be possible. This process requires a deep characterization and understanding of the application being considered. However, once the optimizations are identified and implemented, they can be integrated into existing systems to be used out-of-the-box by other users.

How can we efficiently automate the process of ensuring security to heterogeneous applications and clusters?

Here we propose an approach based on TEEs and OmpSs programming model to ensure security in the context of heterogeneous applications and clusters. We rely on Intel SGX to secure the parts of application with user sensitive data combined with OmpSs programming model to reduce the SGX overhead and improve its usability. OmpSs gives users the possibility of annotating the code to automatically parallelize independent tasks and allocate them to the required hardware devices.

9.3 Future Directions

This work opens a number of interesting research directions which could be explored in future work. First of all, the approaches proposed here were built in the form of academic prototypes which might need enhancements before they are turned into production systems. Besides, most of the experiments using these prototypes were performed in an on-premise cluster with a limited number of available resources. Therefore, another interesting aspect to consider is scalability tests to explore how the proposed approaches would perform in a production system. Next, we briefly discuss ideas for possible future work specific to each of the perspectives taken here.

Regarding the approaches proposed from the perspective of cluster providers, in our Heats implementation we assumed applications to be stateless. This could be extended to also support stateful migrations, covering a larger domain of applications. Another possible idea would be to do a finer grained scheduling and consider per-core pinning combined with per-core frequency scaling to further improve the energy savings. More specifically, in our current multi-tenant scenario approach, DiAS sprints all available cores at the same time. However, this could be improved by estimating an effective sprinting rate which could be used by more complex sprinting policies.

When it comes to our use case application, the implementation focused on the most popular learning frameworks available but does not support all the existing ones. More generally, as we mentioned earlier in this chapter, the feasibility of application based energy optimizations has to be analyzed individually for each case. Therefore, although learning applications are currently a popular topic, there is a vast range of domains which could be explored. Covering use cases for different domains would be an interesting direction to further consolidate the ideas proposed here.

Finally, our proposal to add a security layer of top of the other approaches was based in a combination of SGX and OmpSs. However, considering that we are in a heterogeneous context, it would be interesting to support TEEs technologies for other than Intel devices. For instance, we could explore TrustZone for Arm devices and Platform Security Processor (PSP) or Secure Extension Mode (SEM) for AMD devices. This would allow the data sensitive parts of application to also be flexible in the scheduling phase, probably resulting in further energy savings.

To conclude, we believe that the energy efficiency of distributed applications is a topic which is just starting to be noticed both by academia and industry and, therefore, still has a lot to be explored. The reported numbers on the energy consumption of data centers combined with the projections of continuous growth is alarming. As demonstrated throughout this thesis, this is a diverse problem which has to be tackled from different perspectives and can only be solved if all dimensions are factored in. With the advance on the types of application and hardware available, more and more requirements have to be fulfilled. Therefore, we also believe that solutions for energy efficient computing have to continue evolving in order to catch up with these advances.

Appendices

Appendix A

Publications

2018

1. Christian Göttel, Rafael Pires, Isabelly Rocha, Sébastien Vaucher, Pascal Felber, Marcelo Pasin, and Valerio Schiavoni. **Security, performance and energy trade-off of hardware-assisted memory protection mechanisms**. 2018 IEEE 37th International Symposium on Reliable Distributed Systems (SRDS). Salvador, Brazil, October 2018. doi: 10.1109/SRDS.2018.00024.

2. Isabelly Rocha, Christian Göttel, Pascal Felber, Marcelo Pasin, Romain Rouvoy and Valerio Schiavoni. **HEATS: Heterogeneity- and Energy-Aware Task-based Scheduling**. 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2019, Pavia, Italy, February 2019. doi: 10.1109/SRDS.2018.00024.

2019

3. Isabelly Rocha, Gabriel Vinha, Andrey Brito, Pascal Felber, Marcelo Pasin and Valerio Schiavoni. **ABEONA: an Architecture for Energy-Aware Task Migrations from the Edge to the Cloud**. 38th Symposium on Reliable Distributed Systems, SRDS 2019, Lyon, France, October 2019. doi: 10.1109/SRDS47363.2019.00056.

4. Robert Birke, Isabelly Rocha, Juan Perez, Valerio Schiavoni, Pascal Felber and Lydia Y. Chen. **Differential Approximation and Sprinting for Multi-Priority Big Data Engines**. 20th International Middleware Conference, Middleware 2019, Davis, CA, USA, December 2019. doi: 10.1145/3361525.3361547.

2020

5. Isabelly Rocha, Nathaniel Morris, Robert Birke, Valerio Schiavoni, Pascal Felber and Lydia Y. Chen. **PipeTune: Pipeline Parallelism of Hyper and System Parameters Tuning for Deep Learning Clusters**. 21st International Middleware Conference, Delft, The Netherlands, December 2020. doi: 10.1145/3423211.3425692.

2021

6. Yasmine Djebrouni, Isabelly Rocha, Sara Bouchenak, Lydia Y. Chen, Pascal Felber, Vania Marangozova, Valerio Schiavoni. **Characterizing Distributed Machine Learning and Deep Learning Workloads.** *(under review)*

7. Isabelly Rocha, Pascal Felber, Valerio Schiavoni, Marcelo Pasin, Xavier Martorell, Osman Unsal. **Secure Application Acceleration with Asynchronous Task Parallelism (how to integrate Intel SGX with OmpSs).** *(under review)*

8. Isabelly Rocha, Lydia Y. Chen, Valerio Schiavoni, Pascal Felber. **EdgeTune: Inference-Aware Multi-Parameter Tuning.** *(under review)*

Bibliography

[1] G. Cook, *How clean is your cloud?*, Posted at https://www.greenpeace.org/static/planet4-international-stateless/2012/04/e7c8ff21-howcleanisyourcloud.pdf, 2012.

[2] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell and J. Planas, "Ompss: A proposal for programming heterogeneous multi-core architectures", *Parallel processing letters*, vol. 21, no. 02, pp. 173–193, 2011.

[3] *Nanos++ Runtime Library*, https://github.com/bsc-pm/nanox, Last accessed on: 02-13-2022.

[4] *Mercurium C/C++/Fortran source-to-source compiler*, https://github.com/bsc-pm/mcxx, Last accessed on: 02-13-2022.

[5] A. Rosà, L. Y. Chen and W. Binder, "Understanding the dark side of big data clusters: An analysis beyond failures", in *IEEE/IFIP DSN*, 2015, pp. 207–218.

[6] A. Rosà, L. Y. Chen, R. Birke and W. Binder, "Demystifying casualties of evictions in big data priority scheduling", *ACM SIGMETRICS Performance Evaluation Review*, vol. 42, no. 4, pp. 12–21, 2015.

[7] J. Wilkes, *More Google cluster data*, Google research blog, Posted at http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html., Nov. 2011.

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

[9] Y. Chen, S. Alspaugh and R. H. Katz, "Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads", *PVLDB*, vol. 5, no. 12, pp. 1802–1813, 2012.

[10] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters", *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[11] *Apache Spark*, http://spark.apache.org/, 2019.

[12] K. Shvachko, H. Kuang, S. Radia, R. Chansler *et al.*, "The Hadoop Distributed File System", in *MSST*, vol. 10, 2010, pp. 1–10.

[13] E. Rotem, A. Naveh, A. Ananthakrishnan, E. Weissmann and D. Rajwan, "Power-management architecture of the intel microarchitecture code-named sandy bridge", *IEEE Micro*, vol. 32, no. 2, pp. 20–27, 2012.

[14] S. Fan, S. M. Zahedi and B. C. Lee, "The computational sprinting game", in *ASPLOS*, 2016, pp. 561–575.

[15] *Aws burstable ec2 instances*, https://aws.amazon.com/blogs/aws/low-cost-burstable-ec2-instances, 2019.

[16] M. E. Haque, Y. H. Eom, Y. He, S. Elnikety, R. Bianchini and K. S. McKinley, "Few-to-many: Incremental parallelism for reducing tail latency in interactive services", in *ASPLOS*, 2015, pp. 161–175.

[17] C. Hsu, Y. Zhang, M. A. Laurenzano, D. Meisner, T. F. Wenisch, J. Mars, L. Tang and R. G. Dreslinski, "Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting", in *HPCA*, 2015, pp. 271–282.

[18] N. Morris, C. Stewart, L. Y. Chen, R. Birke and J. Kelley, "Model-driven computational sprinting", in *EuroSys*, ACM, 2018, 38:1–38:13.

[19] P. Kurp, "Green computing", *Commun. ACM*, vol. 51, no. 10, pp. 11–13, Oct. 2008. doi: 10.1145/1400181.1400186.

[20] K. Li, "Power and performance management for parallel computations in clouds and data centers", *JCSS*, vol. 82, no. 2, pp. 174 –190, 2016. doi: https://doi.org/10.1016/j.jcss.2015.07.001.

[21] Y. Wang, K. Li, H. Chen, L. He and K. Li, “Energy-Aware Data Allocation and Task Scheduling on Heterogeneous Multiprocessor Systems With Time Constraints”, *IEEE Transactions on Emerging Topics in Computing*, vol. 2, no. 2, pp. 134–148, 2014. doi: 10.1109/TETC.2014.2300632.

[22] A. Havet, V. Schiavoni, P. Felber, M. Colmant, R. Rouvoy and C. Fetzer, “Genpack: A generational scheduler for cloud data centers”, in *Cloud Engineering (IC2E), 2017 IEEE International Conference on*, IEEE, 2017, pp. 95–104.

[23] S. Su, Q. Huang, J. Li, X. Cheng, P. Xu and K. Shuang, “Enhanced energy-efficient scheduling for parallel tasks using partial optimal slacking†”, *The Computer Journal*, vol. 58, no. 2, pp. 246–257, 2015. doi: 10.1093/comjnl/bxu002. eprint: /oup/backfile/content_public/journal/comjnl/58/2/10.1093/comjnl/bxu002/2/bxu002.pdf.

[24] V. Shekar and B. Izadi, “Energy aware scheduling for dag structured applications on heterogeneous and dvs enabled processors”, in *Green Computing Conference, 2010 International*, IEEE, 2010, pp. 495–502.

[25] D. Çavdar, A. Rosà, L. Y. Chen, W. Binder and F. Alagöz, “Quantifying the brown side of priority schedulers: Lessons from big clusters”, *ACM SIGMETRICS Performance Evaluation Review*, vol. 42, no. 3, pp. 76–81, 2014.

[26] *Apache Hadoop*, http://hadoop.apache.org/, 2019.

[27] *Hadoop Fair Scheduler*, https://hadoop.apache.org/docs/r1.2.1/fair_scheduler.html, 2019.

[28] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker and I. Stoica, “Mesos: A platform for fine-grained resource sharing in the data center”, in *NSDI*, vol. 11, 2011, pp. 22–22.

[29] B. Cho, M. Rahman, T. Chajed, I. Gupta, C. Abad, N. Roberts and P. Lin, “Natjam: Design and evaluation of eviction policies for supporting priorities and deadlines in mapreduce clusters”, in *SOCC*, ACM, 2013, pp. 1–17.

[30] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden and I. Stoica, “Blinkdb: Queries with bounded errors and bounded response times on very large data”, in *Eurosys*, ACM, 2013, pp. 29–42.

[31] D. Krishnan, D. Quoc, P. Bhatotia, C. Fetzer and R. Rodrigues, “Incapprox: A data analytics system for incremental approximate computing”, in *WWW 16*, 2016, pp. 1133–1144.

[32] I. Goiri, R. Bianchini, S. Nagarakatte and T. D. Nguyen, “Approxhadoop: Bringing approximations to mapreduce frameworks”, in *ASPLOS*, 2015, pp. 383–397.

[33] G. Ananthanarayanan, M. C.-C. Hung, X. Ren, I. Stoica, A. Wierman and M. Yu, “Grass: Trimming stragglers in approximation analytics”, in *NSDI*, 2014, pp. 289–302.

[34] T. Komoda, S. Hayashi, T. Nakada, S. Miwa and H. Nakamura, “Power capping of CPU-GPU heterogeneous systems through coordinating DVFS and task mapping”, in *IEEE ICCD*, 2013, pp. 349–356.

[35] W. Zheng and X. Wang, “Data center sprinting: Enabling computational sprinting at the data center level”, in *IEEE ICDCS*, 2015, pp. 175–184.

[36] H. Zhang and H. Hoffmann, “Maximizing performance under a power cap: A comparison of hardware, software, and hybrid techniques”, in *ASPLOS*, 2016, pp. 545–559.

[37] M. Jeon, Y. He, H. Kim, S. Elnikety, S. Rixner and A. L. Cox, “TPC: target-driven parallelism combining prediction and correction to reduce tail latency in interactive services”, in *ASPLOS*, 2016, pp. 129–141.

[38] Z. Qiu, J. F. Pérez and P. G. Harrison, “Variability-aware request replication for latency curtailment”, in *IEEE INFOCOM*, 2016, pp. 1–9.

[39] M. Harchol-Balter, T. Osogami, A. Scheller-Wolf and A. Wierman, “Multi-server queueing systems with multiple priority classes”, *Queueing Syst.*, vol. 51, no. 3-4, pp. 331–360, 2005.

[40] A. Wierman, T. Osogami, M. Harchol-Balter and A. Scheller-Wolf, “How many servers are best in a dual-priority M/PH/k system?”, *Performance Evaluation Review*, 2006.

[41] A. Sleptchenko, A. Harten and M. Heijden, “An exact solution for the state probabilities of the multi-class, multi-server queue with preemptive priorities”, *Queueing Systems*, vol. 50, no. 1, pp. 81–107, 2005.

[42] G. Horváth, “Efficient analysis of the MMAP[K]/PH[K]/1 priority queue”, *European Journal of Operational Research*, vol. 246, no. 1, pp. 128–139, 2015.

[43] P. R. Jelenkovic and E. D. Skiani, “Is sharing with retransmissions causing instabilities?”, in *ACM SIGMETRICS Performance Evaluation Review*, vol. 42, 2014, pp. 167–179.

[44] A. Kulshrestha and S. K. Dubey, “A literature reviewon sniffing attacks in computernetwork”, *International Journal of Advanced Engineering Research and Science (IJAERS)*, vol. 1, no. 2, 2014.

[45] F. Liu, Y. Yarom, Q. Ge, G. Heiser and R. B. Lee, “Last-level cache side-channel attacks are practical”, in *2015 IEEE symposium on security and privacy*, IEEE, 2015, pp. 605–622.

[46] *AMD Secure Processor (Built-in technology).* https://www.amd.com/en-us/innovations/software-technologies/security, Last accessed on: 02-13-2022.

[47] S. Pinto and N. Santos, “Demystifying arm trustzone: A comprehensive survey”, *ACM Computing Surveys (CSUR)*, vol. 51, no. 6, pp. 1–36, 2019.

[48] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd and C. Rozas, “Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave”, in *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, 2016, pp. 1–9.

[49] A. Baumann, M. Peinado and G. Hunt, “Shielding applications from an untrusted cloud with haven”, *ACM Transactions on Computer Systems*, vol. 33, no. 3, pp. 1–26, 2015.

[50] Y. Shen, H. Tian, Y. Chen, K. Chen, R. Wang, Y. Xu, Y. Xia and S. Yan, “Occlum: Secure and efficient multitasking inside a single enclave of intel sgx”, in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 955–970.

[51] R. Wahbe, S. Lucco, T. E. Anderson and S. L. Graham, “Efficient software-based fault isolation”, in *Proceedings of the fourteenth ACM symposium on Operating systems principles*, 1993, pp. 203–216.

[52] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O’keeffe, M. L. Stillwell *et al.*, “Scone: Secure linux containers with intel sgx”, in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 689–703.

[53] C.-C. Tsai, D. E. Porter and M. Vij, “Graphene-sgx: A practical library os for unmodified applications on sgx”, in *2017 USENIX Annual Technical Conference USENIX ATC 17)*, 2017, pp. 645–658.

[54] S. Shinde, D. Le Tien, S. Tople and P. Saxena, “Panoply: Low-tcb linux applications with sgx enclaves.”, in *NDSS*, 2017.

[55] T. Lee, Z. Lin, S. Pushp, C. Li, Y. Liu, Y. Lee, F. Xu, C. Xu, L. Zhang and J. Song, “Occlumency: Privacy-preserving remote deep-learning inference using sgx”, in *The 25th Annual International Conference on Mobile Computing and Networking*, 2019, pp. 1–17.

[56] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. Karro and D. Sculley, “Google vizier: A service for black-box optimization”, in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Halifax, NS, Canada, August 13 - 17, 2017*, ACM, 2017, pp. 1487–1495. doi: 10.1145/3097983.3098043.

[57] E. Liberty, Z. S. Karnin, B. Xiang, L. Rouesnel, B. Coskun, R. Nallapati, J. Delgado, A. Sadoughi, Y. Astashonok, P. Das, C. Balioglu, S. Chakravarty, M. Jha, P. Gautier, D. Arpin, T. Januschowski, V. Flunkert, Y. Wang, J. Gasthaus, L. Stella, S. S. Rangapuram, D. Salinas, S. Schelter and A. Smola, “Elastic machine learning algorithms in amazon sagemaker”, in *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini and H. Q. Ngo, Eds., ACM, 2020, pp. 731–737. doi: 10.1145/3318464.3386126.

[58] L. Bottou, “Large-scale machine learning with stochastic gradient descent”, in *Proceedings of COMPSTAT’2010*, Springer, 2010, pp. 177–186.

[59] H. Jin, Q. Song and X. Hu, "Auto-keras: An efficient neural architecture search system", in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2019, Anchorage, AK, USA, August 4-8, 2019*, A. Teredesai, V. Kumar, Y. Li, R. Rosales, E. Terzi and G. Karypis, Eds., ACM, 2019, pp. 1946–1956. doi: 10.1145/3292500.3330648.

[60] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez and I. Stoica, "Tune: A research platform for distributed model selection and training", *CoRR*, vol. abs/1807.05118, 2018. arXiv: 1807.05118.

[61] M. Claesen and B. D. Moor, "Hyperparameter search in machine learning", *CoRR*, vol. abs/1502.02127, 2015. arXiv: 1502.02127.

[62] Y. LeCun *et al.*, "Lenet-5, convolutional neural networks", *URL: http://yann. lecun. com/exdb/lenet*, vol. 20, p. 5, 2015.

[63] Y. LeCun, "The mnist database of handwritten digits", *http://yann. lecun. com/exdb/mnist/*, 1998.

[64] S. Venkataraman, A. Panda, K. Ousterhout, M. Armbrust, A. Ghodsi, M. J. Franklin, B. Recht and I. Stoica, "Drizzle: Fast and adaptable stream processing at scale", in *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, ACM, 2017, pp. 374–389. doi: 10.1145/3132747.3132750.

[65] J. J. Dai, Y. Wang, X. Qiu, D. Ding, Y. Zhang, Y. Wang, X. Jia, C. L. Zhang, Y. Wan, Z. Li, J. Wang, S. Huang, Z. Wu, Y. Wang, Y. Yang, B. She, D. Shi, Q. Lu, K. Huang and G. Song, "Bigdl: A distributed deep learning framework for big data", in *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019*, ACM, 2019, pp. 50–60. doi: 10.1145/3357223.3362707.

[66] G. Ammons, T. Ball and J. R. Larus, "Exploiting hardware performance counters with flow and context sensitive profiling", *ACM Sigplan Notices*, vol. 32, no. 5, pp. 85–96, 1997.

[67] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size", *arXiv preprint arXiv:1602.07360*, 2016.

[68] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images", 2009.

[69] L. Li, K. G. Jamieson, G. DeSalvo, A. Rostamizadeh and A. Talwalkar, "Hyperband: A novel bandit-based approach to hyperparameter optimization", *J. Mach. Learn. Res.*, vol. 18, 185:1–185:52, 2017.

[70] K. Kandasamy, G. Dasarathy, J. Schneider and B. Póczos, "Multi-fidelity bayesian optimisation with continuous approximations", in *International Conference on Machine Learning*, PMLR, 2017, pp. 1799–1808.

[71] M. Sivathanu, T. Chugh, S. S. Singapuram and L. Zhou, "Astra: Exploiting predictability to optimize deep learning", in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, I. Bahar, M. Herlihy, E. Witchel and A. R. Lebeck, Eds., ACM, 2019, pp. 909–923. doi: 10.1145/3297858.3304072.

[72] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu and C. Guo, "A generic communication scheduler for distributed DNN training acceleration", in *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, T. Brecht and C. Williamson, Eds., ACM, 2019, pp. 16–29. doi: 10.1145/3341301.3359642.

[73] X. Dai, P. Zhang, B. Wu, H. Yin, F. Sun, Y. Wang, M. Dukhan, Y. Hu, Y. Wu, Y. Jia *et al.*, "Chamnet: Towards efficient network design through platform-aware model adaptation", in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 11 398–11 407.

[74] J.-D. Dong, A.-C. Cheng, D.-C. Juan, W. Wei and M. Sun, "Dpp-net: Device-aware progressive search for pareto-optimal neural architectures", in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 517–531.

[75] B. Wu, X. Dai, P. Zhang, Y. Wang, F. Sun, Y. Wu, Y. Tian, P. Vajda, Y. Jia and K. Keutzer, "Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search", in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 10 734–10 742.

[76] Y. Peng, Y. Bao, Y. Chen, C. Wu and C. Guo, “Optimus: An efficient dynamic resource scheduler for deep learning clusters”, in *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, R. Oliveira, P. Felber and Y. C. Hu, Eds., ACM, 2018, 3:1–3:14. doi: 10.1145/3190508.3190517.

[77] Q. Luo, J. Lin, Y. Zhuo and X. Qian, “Hop: Heterogeneity-aware decentralized training”, in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, I. Bahar, M. Herlihy, E. Witchel and A. R. Lebeck, Eds., ACM, 2019, pp. 893–907. doi: 10.1145/3297858.3304009.

[78] J. Rasley, Y. He, F. Yan, O. Ruwase and R. Fonseca, “Hyperdrive: Exploring hyperparameters with POP scheduling”, in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference, Las Vegas, NV, USA, December 11 - 15, 2017*, K. R. Jayaram, A. Gandhi, B. Kemme and P. R. Pietzuch, Eds., ACM, 2017, pp. 1–13. doi: 10.1145/3135974.3135994.

[79] D. Stamoulis, E. Cai, D.-C. Juan and D. Marculescu, “Hyperpower: Power-and memory-constrained hyper-parameter optimization for neural networks”, in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2018, pp. 19–24.

[80] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard and Q. V. Le, “Mnasnet: Platform-aware neural architecture search for mobile”, in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 2820–2828.

[81] E. Cai, D.-C. Juan, D. Stamoulis and D. Marculescu, “Neuralpower: Predict and deploy energy-efficient convolutional neural networks”, in *Asian Conference on Machine Learning*, PMLR, 2017, pp. 622–637.

[82] J. Wei, G. A. Gibson, P. B. Gibbons and E. P. Xing, “Automating dependence-aware parallelization of machine learning training on distributed shared memory”, in *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, G. Candea, R. van Renesse and C. Fetzer, Eds., ACM, 2019, 42:1–42:17. doi: 10.1145/3302424.3303954.

[83] H. Cai, L. Zhu and S. Han, “Proxylessnas: Direct neural architecture search on target task and hardware”, *arXiv preprint arXiv:1812.00332*, 2018.

[84] S. Kim, G. Yu, H. Park, S. Cho, E. Jeong, H. Ha, S. Lee, J. S. Jeong and B. Chun, “Parallax: Sparsity-aware data parallel training of deep neural networks”, in *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, G. Candea, R. van Renesse and C. Fetzer, Eds., ACM, 2019, 43:1–43:15. doi: 10.1145/3302424.3303957.

[85] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons and M. Zaharia, “Pipedream: Generalized pipeline parallelism for DNN training”, in *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, T. Brecht and C. Williamson, Eds., ACM, 2019, pp. 1–15. doi: 10.1145/3341301.3359646.

[86] J. K. Kim, Q. Ho, S. Lee, X. Zheng, W. Dai, G. A. Gibson and E. P. Xing, “STRADS: a distributed framework for scheduled model parallel machine learning”, in *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*, C. Cadar, P. R. Pietzuch, K. Keeton and R. Rodrigues, Eds., ACM, 2016, 5:1–5:16. doi: 10.1145/2901318.2901331.

[87] J. K. Kim, A. Aghayev, G. A. Gibson and E. P. Xing, “STRADS-AP: simplifying distributed machine learning programming without introducing a new programming model”, in *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, D. Malkhi and D. Tsafrir, Eds., USENIX Association, 2019, pp. 207–222.

[88] J. Bergstra, R. Bardenet, Y. Bengio and B. Kégl, “Algorithms for hyper-parameter optimization”, in *Advances in Neural Information Processing Systems 24: 25th Annual Conference on Neural Information Processing Systems 2011. Proceedings of a meeting held 12-14 December 2011, Granada, Spain*, J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. C. N. Pereira and K. Q. Weinberger, Eds., 2011, pp. 2546–2554.

[89] T. Bartz-Beielstein and S. Markon, "Tuning search algorithms for real-world applications: A regression tree based approach", in *Proceedings of the 2004 Congress on Evolutionary Computation (IEEE Cat. No. 04TH8753)*, IEEE, vol. 1, 2004, pp. 1111–1118.

[90] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization", *J. Mach. Learn. Res.*, vol. 13, pp. 281–305, 2012.

[91] F. Madrigal, C. Maurice and F. Lerasle, "Hyper-parameter optimization tools comparison for multiple object tracking applications", *Machine Vision and Applications*, vol. 30, no. 2, pp. 269–289, 2019.

[92] C. Holmes, D. Mawhirter, Y. He, F. Yan and B. Wu, "GRNN: low-latency and scalable RNN inference on gpus", in *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, G. Candea, R. van Renesse and C. Fetzer, Eds., ACM, 2019, 41:1–41:16. doi: 10.1145/3302424.3303949.

[93] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita and B. Su, "Scaling distributed machine learning with the parameter server", in *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, J. Flinn and H. Levy, Eds., USENIX Association, 2014, pp. 583–598.

[94] A. R. Mamidala, J. Liu and D. K. Panda, "Efficient barrier and allreduce on infiniband clusters using multicast and adaptive algorithms", in *2004 IEEE International Conference on Cluster Computing (CLUSTER 2004), September 20-23 2004, San Diego, California, USA*, IEEE Computer Society, 2004, pp. 135–144. doi: 10.1109/CLUSTR.2004.1392611.

[95] A. Rojas-Domínguez, L. C. Padierna, J. M. C. Valadez, H. J. Puga-Soberanes and H. J. Fraire, "Optimal hyper-parameter tuning of svm classifiers with application to medical diagnosis", *Ieee Access*, vol. 6, pp. 7164–7176, 2017.

[96] A. Moschitti, "A study on optimal parameter tuning for rocchio text classifier", in *European Conference on Information Retrieval*, Springer, 2003, pp. 420–435.

[97] D. Horn and B. Bischl, "Multi-objective parameter configuration of machine learning algorithms using model-based optimization", in *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, IEEE, 2016, pp. 1–8.

[98] M. Baldeon-Calisto and S. K. Lai-Yuen, "Adaresu-net: Multiobjective adaptive convolutional neural network for medical image segmentation", *Neurocomputing*, vol. 392, pp. 325–340, 2020.

[99] J. Lin, "Multiple-objective problems: Pareto-optimal solutions by method of proper equality constraints", *IEEE Transactions on Automatic Control*, vol. 21, no. 5, pp. 641–650, 1976.

[100] P. K. Shukla and K. Deb, "On finding multiple pareto-optimal solutions using classical and evolutionary generating methods", *European Journal of Operational Research*, vol. 181, no. 3, pp. 1630–1652, 2007.

[101] T. Elsken, J. H. Metzen and F. Hutter, "Neural architecture search: A survey", *The Journal of Machine Learning Research*, vol. 20, no. 1, pp. 1997–2017, 2019.

[102] H. Liu, K. Simonyan and Y. Yang, "Darts: Differentiable architecture search", *arXiv preprint arXiv:1806.09055*, 2018.

[103] P. Liashchynskyi and P. Liashchynskyi, "Grid search, random search, genetic algorithm: A big comparison for nas", *arXiv preprint arXiv:1912.06059*, 2019.

[104] B. Baker, O. Gupta, R. Raskar and N. Naik, "Accelerating neural architecture search using performance prediction", *arXiv preprint arXiv:1705.10823*, 2017.

[105] C. Ying, A. Klein, E. Christiansen, E. Real, K. Murphy and F. Hutter, "Nas-bench-101: Towards reproducible neural architecture search", in *International Conference on Machine Learning*, PMLR, 2019, pp. 7105–7114.

[106] C.-J. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. Hazelwood, E. Isaac, Y. Jia, B. Jia *et al.*, "Machine learning at facebook: Understanding inference at the edge", in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2019, pp. 331–344.

[107] A. Cristal, O. S. Unsal, X. Martorell, P. Carpenter, R. De La Cruz, L. Bautista, D. Jimenez, C. Alvarez, B. Salami, S. Madonar *et al.*, "Legato: First steps towards energy-efficient toolset for heterogeneous computing", in *Proceedings of the 18th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, 2018, pp. 210–217.

[108] C. Muller, M. Brandenburger, C. Cachin, P. Felber, C. Gottel and V. Schiavoni, "Tz4fabric: Executing smart contracts with arm trustzone:(practical experience report)", in *2020 International Symposium on Reliable Distributed Systems (SRDS)*, IEEE Computer Society, 2020, pp. 31–40.

[109] C. Göttel, L. Nielsen, N. Yazdani, P. Felber, D. E. Lucani and V. Schiavoni, "Hermes: Enabling energy-efficient iot networks with generalized deduplication", in *Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems*, 2020, pp. 133–136.

[110] O. Oleksenko, B. Trach, M. Silberstein and C. Fetzer, "Specfuzz: Bringing spectre-type vulnerabilities to the surface", in *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, S. Capkun and F. Roesner, Eds., USENIX Association, 2020, pp. 1481–1498.

[111] *Legato is a low energy toolset for heterogeneous computing*, Posted at https://legato-project.eu., Nov. 2021.

[112] Amazon Web Services, Inc., *Amazon EC2 Instance Types*, https://aws.amazon.com/ec2/instance-types/, Last accessed on: 02-13-2022, Sep. 2018.

[113] Microsoft Corporation, *Pricing - Linux Virtual Machines*, https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux/, Last accessed on: 02-13-2022.

[114] Google LLC, *Google Compute Engine Pricing*, https://cloud.google.com/compute/pricing, Last accessed on: 02-13-2022.

[115] IBM, *Bare metal servers*, https://www.ibm.com/cloud/bare-metal-servers, Last accessed on: 02-13-2022, Sep. 2018.

[116] Oracle Corporation, *Bare Metal Cloud Computing*, https://cloud.oracle.com/compute/bare-metal/features, Last accessed on: 02-13-2022.

[117] Scaleway, *BareMetal SSD Cloud Servers*, https://www.scaleway.com/en/bare-metal-servers, Last accessed on: 02-13-2022.

[118] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment", *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.

[119] V. Medel, O. Rana, J. Á. Bañares and U. Arronategui, "Modelling performance & resource management in kubernetes", in *2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC)*, IEEE, 2016, pp. 257–262.

[120] A. Likas, N. Vlassis and J. J. Verbeek, "The global k-means clustering algorithm", *Pattern recognition*, vol. 36, no. 2, pp. 451–461, 2003.

[121] *Compute Resource Usage Analysis*, https://github.com/kubernetes/heapster, Last accessed on: 02-13-2022.

[122] E. Le Sueur and G. Heiser, "Dynamic voltage and frequency scaling: The laws of diminishing returns", in *Proceedings of the 2010 international conference on Power aware computing and systems*, 2010, pp. 1–8.

[123] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement", *arXiv*, 2018.

[124] *Linux cpu governors*, Last accessed on: 02-13-2022.

[125] R. H. Myers and R. H. Myers, *Classical and modern regression with applications*. Duxbury press Belmont, CA, 1990, vol. 2.

[126] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning.", in *OSDI*, vol. 16, 2016, pp. 265–283.

[127] *cAdvisor*, https://github.com/google/cadvisor, Last accessed on: 02-13-2022.

[128] *Kubelet*, https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/, Last accessed on: 02-13-2022.

[129] *Heapster*, https://github.com/kubernetes/heapster, Last accessed on: 02-13-2022.

[130] *Grafana*, https://grafana.com/, Last accessed on: 02-13-2022.

[131] *InfluxDB*, https://www.influxdata.com/time-series-platform/influxdb/, Last accessed on: 02-13-2022.

[132] *PyRAPL*, https://github.com/powerapi-ng/pyRAPL, Last accessed on: 02-13-2022.

[133] AWS, *Aws lambda*, AWS website, Posted at https://aws.amazon.com/lambda., Nov. 2021.

[134] Microsoft, *Azure functions - process events with serverless code*, Microsoft Azure website, Posted at https://azure.microsoft.com/en-us/services/functions., Nov. 2021.

[135] Google, *Google functions*, Google Cloud website, Posted at https://cloud.google.com/functions., Nov. 2021.

[136] *Kubernetes*, https://github.com/kubernetes/kubernetes, Last accessed on: 02-13-2022.

[137] *Kubernetes Scheduler API*, https://kubernetes.io/docs/reference/command-line-tools-reference/kube-scheduler/, Last accessed on: 02-13-2022.

[138] *Kubernetes Client Python*, https://github.com/kubernetes-client/python, Last accessed on: 02-13-2022.

[139] *Kubernetes API*, https://kubernetes.io/docs/concepts/overview/kubernetes-api/, Last accessed on: 02-13-2022.

[140] K. S. Banerjee and E. Agu, "Powerspy: Fine-grained software energy profiling for mobile devices", in *Wireless Networks, Communications and Mobile Computing, 2005 International Conference on*, IEEE, vol. 2, 2005, pp. 1136–1141.

[141] *Alpine linux*, https://www.alpinelinux.org/, Last accessed on: 02-13-2022.

[142] M. Shafique, S. Garg, J. Henkel and D. Marculescu, "The eda challenges in the dark silicon era: Temperature, reliability, and variability perspectives", in *Proceedings of the 51st Annual Design Automation Conference*, 2014, pp. 1–6.

[143] A. Ganapathi, Y. Chen, A. Fox, R. Katz and D. Patterson, "Statistics-driven workload modeling for the cloud", in *IEEE ICDEW*, 2010, pp. 87–92.

[144] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, Franklin *et al.*, "Apache Spark: a unified engine for big data processing", *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[145] *StackExchange*, https://anime.stackexchange.com, 2019.

[146] *SNAP: Network datasets: Google web graph*, https://snap.stanford.edu/data/web-Google.html, 2019.

[147] G. Latouche and V. Ramaswami, *Introduction to matrix analytic methods in stochastic modeling*. SIAM, 1999.

[148] J. F. Pérez, R. Birke and L. Y. Chen, "On the latency-accuracy tradeoff in approximate mapreduce jobs", in *IEEE INFOCOM*, 2017, pp. 1–9.

[149] *HDFS Architecture Guide*, https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html, 2019.

[150] *GraphX*, https://spark.apache.org/graphx/, 2019.

[151] J. Leskovec, K. Lang, A. Dasgupta and M. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters", *Internet Mathematics*, vol. 6, pp. 29–123, 2009.

[152] *Property: TDP down frequency*, https://en.wikichip.org/wiki/Property:tdp_down_frequency, 2019.

[153] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin *et al.*, "Apache Spark: a unified engine for big data processing", *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[154] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing", in *NSDI*, 2012, pp. 2–2.

[155] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen *et al.*, "Mllib: Machine learning in apache spark", *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 1235–1241, 2016.

[156] J. J. Dai, Y. Wang, X. Qiu, D. Ding, Y. Zhang, Y. Wang, X. Jia, C. L. Zhang, Y. Wan, Z. Li *et al.*, "Bigdl: A distributed deep learning framework for big data", in *Proceedings of the ACM Symposium on Cloud Computing*, 2019, pp. 50–60.

[157] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu and X. Zheng, "Tensorflow: A system for large-scale machine learning", in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. Osdi'16, Savannah, GA, USA: USENIX Association, 2016, pp. 265–283.

[158] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library", *Advances in neural information processing systems*, vol. 32, pp. 8026–8037, 2019.

[159] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding", in *Proceedings of the 22nd ACM international conference on Multimedia*, 2014, pp. 675–678.

[160] Apache Spark, *Spark configuration*, https://spark.apache.org/docs/2.4.3/configuration.html, 2021.

[161] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker and I. Stoica, "Shark: Sql and rich analytics at scale", in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of data*, 2013, pp. 13–24.

[162] J. A. Nelder and R. W. Wedderburn, "Generalized linear models", *Journal of the Royal Statistical Society: Series A (General)*, vol. 135, no. 3, pp. 370–384, 1972.

[163] I. Rish *et al.*, "An empirical study of the naive bayes classifier", in *IJCAI 2001 workshop on empirical methods in artificial intelligence*, vol. 3, 2001, pp. 41–46.

[164] J. R. Quinlan, "Induction of decision trees", *Machine learning*, vol. 1, no. 1, pp. 81–106, 1986.

[165] T. N. Sainath, A.-r. Mohamed, B. Kingsbury and B. Ramabhadran, "Deep convolutional neural networks for lvcsr", in *2013 IEEE international conference on acoustics, speech and signal processing*, IEEE, 2013, pp. 8614–8618.

[166] S. Hochreiter and J. Schmidhuber, "Long short-term memory", *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[167] M. W. Gardner and S. Dorling, "Artificial neural networks (the multilayer perceptron) – a review of applications in the atmospheric sciences", *Atmospheric environment*, vol. 32, no. 14-15, pp. 2627–2636, 1998.

[168] L. Deng, "The MNIST Database of Handwritten Digit Images for Machine Learning Research [Best of the Web]", *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012. doi: 10.1109/MSP.2012.2211477.

[169] P. Baldi, P. Sadowski and D. Whiteson, "Searching for exotic particles in high-energy physics with deep learning", *Nature Communications*, vol. 5, no. C, Jul. 2014.

[170] M. Yedla, S. R. Pathakota and T. Srinivasa, "Enhancing k-means clustering algorithm with improved initial center", *International Journal of computer science and information technologies*, vol. 1, no. 2, pp. 121–125, 2010.

[171] X. Zou, Y. Hu, Z. Tian and K. Shen, "Logistic regression model optimization and case analysis", in *2019 IEEE 7th International Conference on Computer Science and Network Technology (ICCSNT)*, IEEE, 2019, pp. 135–139.

[172] J. Gaudart, B. Giusiano and L. Huiart, "Comparison of the performance of multi-layer perceptron and linear regression for epidemiological data", *Computational Statistics and Data Analysis*, vol. 44, pp. 547 –570, 2004.

[173] K. Diaz-Chito, A. Hernández-Sabaté and A. M. López, "A reduced feature set for driver head pose estimation", *Appl. Soft Comput.*, vol. 45, no. C, pp. 98–107, Aug. 2016. doi: 10.1016/j.asoc.2016.04.027.

[174] A. Vergara, S. Vembu, T. Ayhan, M. A. Ryan, M. L. Homer and R. Huerta, "Chemical gas sensor drift compensation using classifier ensembles", *Sensors and Actuators B: Chemical*, vol. 166-167, pp. 320–329, May 2012. doi: 10.1016/j.snb.2012.01.074.

[175] *20 Newsgroups*, http://qwone.com/~jason/20Newsgroups, Last accessed on: 02-13-2022.

[176] H. Xiao, K. Rasul and R. Vollgraf, "Fashion-mnist: A novel image dataset for benchmarking machine learning algorithms", *arXiv preprint arXiv:1708.07747*, 2017.

[177] P. Fränti and S. Sieranoja, "How much can k-means be improved by using better initialization and repeats?", *Pattern Recognition*, vol. 93, pp. 95–112, 2019.

[178] A. K. Reyes, J. C. Caicedo and J. E. Camargo, "Fine-tuning deep convolutional networks for plant recognition.", *CLEF (Working Notes)*, vol. 1391, pp. 467–475, 2015.

[179] P. Probst, M. N. Wright and A.-L. Boulesteix, "Hyperparameters and tuning strategies for random forest", *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 9, no. 3, e1301, 2019.

[180] O. Marcu, A. Costan, G. Antoniu and M. S. Pérez-Hernández, "Spark Versus Flink: Understanding Performance in Big Data Analytics Frameworks", in *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, 2016, pp. 433–442. doi: 10.1109/CLUSTER.2016.22.

[181] C. V. Rijsbergen, "Information retrieval.", *Journal of the American Society for Information Science*, vol. 30, no. 6, pp. 374–375, 1979. doi: https://doi.org/10.1002/asi.4630300621. eprint: https://asistdl.onlinelibrary.wiley.com/doi/pdf/10.1002/asi.4630300621.

[182] S. G. Makridakis and M. Hibon, *Evaluating accuracy (or error) measures.* Fontainebleau: INSEAD, ser. 1995). Evaluating accuracy (or error) measures. Fontainebleau: INSEAD. 1995.

[183] *Sparkmeasure, a tool for performance troubleshooting of apache spark workloads*, https://db-blog.web.cern.ch/blog/luca-canali/2018-08-sparkmeasure-tool-performance-troubleshooting-apache-spark-workloads, Last accessed on: 02-13-2022.

[184] PCM, *Processor Counter Monitor (PCM)*, https://software.intel.com/content/www/us/en/develop/articles/intel-performance-counter-monitor.html, 2021.

[185] D. Blind, *Anonymous dataset middleware'21 submission #15*, https://doi.org/10.5281/zenodo.4805581, version 0.0.1, May 2021. doi: 10.5281/zenodo.4805581.

[186] AWS, *Amazon EC2 On-Demand Pricing*, https://aws.amazon.com/fr/ec2/pricing/on-demand/.

[187] M. Kahng, P. Y. Andrews, A. Kalro and D. H. P. Chau, "Activis: Visual exploration of industry-scale deep neural network models", *IEEE Trans. Vis. Comput. Graph.*, vol. 24, no. 1, pp. 88–97, 2018. doi: 10.1109/TVCG.2017.2744718.

[188] S. Dutta, "An overview on the evolution and adoption of deep learning applications used in the industry", *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 8, no. 4, 2018.

[189] L. Deng, G. E. Hinton and B. Kingsbury, "New types of deep neural network learning for speech recognition and related applications: An overview", in *IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2013, Vancouver, BC, Canada, May 26-31, 2013*, IEEE, 2013, pp. 8599–8603. doi: 10.1109/ICASSP.2013.6639344.

[190] F. Richardson, D. A. Reynolds and N. Dehak, "Deep neural network approaches to speaker and language recognition", *IEEE Signal Processing Letters*, vol. 22, no. 10, pp. 1671–1675, 2015. doi: 10.1109/LSP.2015.2420092.

[191] A. Lozano-Diez, R. Zazo, D. T. Toledano and J. Gonzalez-Rodriguez, "An analysis of the influence of deep neural network (DNN) topology in bottleneck feature based language recognition", *PloS one*, vol. 12, no. 8, 2017.

[192] D. Erhan, C. Szegedy, A. Toshev and D. Anguelov, "Scalable object detection using deep neural networks", in *2014 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2014, Columbus, OH, USA, June 23-28, 2014*, IEEE Computer Society, 2014, pp. 2155–2162. doi: 10.1109/CVPR.2014.276.

[193] B. Huval, T. Wang, S. Tandon, J. Kiske, W. Song, J. Pazhayampallil, M. Andriluka, P. Rajpurkar, T. Migimatsu, R. Cheng-Yue, F. A. Mujica, A. Coates and A. Y. Ng, "An empirical evaluation of deep learning on highway driving", *CoRR*, vol. abs/1504.01716, 2015. arXiv: 1504.01716.

[194] D. Strigl, K. Kofler and S. Podlipnig, "Performance and scalability of gpu-based convolutional neural networks", in *Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-based Processing, PDP 2010, Pisa, Italy, February 17-19, 2010*, M. Danelutto, J. Bourgeois and T. Gross, Eds., IEEE Computer Society, 2010, pp. 317–324. doi: 10.1109/PDP.2010.43.

[195] Y. Sun, D. Liang, X. Wang and X. Tang, "Deepid3: Face recognition with very deep neural networks", *CoRR*, vol. abs/1502.00873, 2015. arXiv: 1502.00873.

[196] H. Y. Xiong, B. Alipanahi, L. J. Lee, H. Bretschneider, D. Merico, R. K. Yuen, Y. Hua, S. Gueroussov, H. S. Najafabadi, T. R. Hughes *et al.*, "The human splicing code reveals new insights into the genetic determinants of disease", *Science*, vol. 347, no. 6218, p. 1 254 806, 2015.

[197] L. Deng and Y. Liu, *Deep learning in natural language processing*. Springer, 2018.

[198] D. C. Ciresan, A. Giusti, L. M. Gambardella and J. Schmidhuber, "Mitosis detection in breast cancer histology images with deep neural networks", in *Medical Image Computing and Computer-Assisted Intervention - MICCAI 2013 - 16th International Conference, Nagoya, Japan, September 22-26, 2013, Proceedings, Part II*, K. Mori, I. Sakuma, Y. Sato, C. Barillot and N. Navab, Eds., ser. Lecture Notes in Computer Science, vol. 8150, Springer, 2013, pp. 411–418. doi: 10.1007/978-3-642-40763-5_51.

[199] *Tensorflow convnets on a budget with bayesian optimization*, https://sigopt.com/blog/tensorflow-convnets-on-a-budget-with-bayesian-optimization/, 2020.

[200] B. Zoph, V. Vasudevan, J. Shlens and Q. V. Le, "Learning transferable architectures for scalable image recognition", in *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, IEEE Computer Society, 2018, pp. 8697–8710. doi: 10.1109/CVPR.2018.00907.

[201] J. Snow, "Amazon's face recognition falsely matched 28 members of congress with mugshots", *American Civil Liberties Union*, vol. 28, 2018.

[202] K. Hao, "Police across the us are training crimepredicting ais on falsified data", *MIT Technology Review.*, vol. 13, 2019.

[203] R. Pascanu, T. Mikolov and Y. Bengio, "On the difficulty of training recurrent neural networks", in *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*, ser. JMLR Workshop and Conference Proceedings, vol. 28, JMLR.org, 2013, pp. 1310–1318.

[204] M. P. Ranjit, G. Ganapathy, K. Sridhar and V. Arumugham, "Efficient deep learning hyperparameter tuning using cloud infrastructure: Intelligent distributed hyperparameter tuning with bayesian optimization in the cloud", in *12th IEEE International Conference on Cloud Computing, CLOUD 2019, Milan, Italy, July 8-13, 2019*, E. Bertino, C. K. Chang, P. Chen, E. Damiani, M. Goul and K. Oyama, Eds., IEEE, 2019, pp. 520–522. doi: 10.1109/CLOUD.2019.00097.

[205] O. Agmon Ben-Yehuda, M. Ben-Yehuda, A. Schuster and D. Tsafrir, "Deconstructing amazon EC2 spot instance pricing", *ACM Trans. Economics and Comput.*, vol. 1, no. 3, 16:1–16:20, 2013. doi: 10.1145/2509413.2509416.

[206] *Perform Automatic Model Tuning*, https://docs.aws.amazon.com/sagemaker/latest/dg/automatic-model-tuning.html, Last accessed on: 02-13-2022.

[207] G. E. Hinton, "A practical guide to training restricted boltzmann machines", in *Neural Networks: Tricks of the Trade - Second Edition*, ser. Lecture Notes in Computer Science, G. Montavon, G. B. Orr and K. Müller, Eds., vol. 7700, Springer, 2012, pp. 599–619. doi: 10.1007/978-3-642-35289-8_32.

[208] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams and N. de Freitas, "Taking the human out of the loop: A review of bayesian optimization", *Proceedings of the IEEE*, vol. 104, no. 1, pp. 148–175, 2016. doi: 10.1109/JPROC.2015.2494218.

[209] J. Snoek, H. Larochelle and R. P. Adams, "Practical bayesian optimization of machine learning algorithms", in *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*, P. L. Bartlett, F. C. N. Pereira, C. J. C. Burges, L. Bottou and K. Q. Weinberger, Eds., 2012, pp. 2960–2968.

[210] S. R. Young, D. C. Rose, T. P. Karnowski, S. Lim and R. M. Patton, “Optimizing deep learning hyper-parameters through an evolutionary algorithm”, in *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments, MLHPC 2015, Austin, Texas, USA, November 15, 2015*, ACM, 2015, 4:1–4:5. doi: 10.1145/2834892.2834896.

[211] M. Suganuma, S. Shirakawa and T. Nagao, “A genetic programming approach to designing convolutional neural network architectures”, in *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, J. Lang, Ed., ijcai.org, 2018, pp. 5369–5373. doi: 10.24963/ijcai.2018/755.

[212] M. Jaderberg, V. Dalibard, S. Osindero, W. M. Czarnecki, J. Donahue, A. Razavi, O. Vinyals, T. Green, I. Dunning, K. Simonyan, C. Fernando and K. Kavukcuoglu, “Population based training of neural networks”, *CoRR*, vol. abs/1711.09846, 2017. arXiv: 1711.09846.

[213] Y. Bengio, “Gradient-based optimization of hyperparameters”, *Neural Computation*, vol. 12, no. 8, pp. 1889–1900, 2000. doi: 10.1162/089976600300015187.

[214] *Linux kernel profiling with perf*, https://perf.wiki.kernel.org/index.php/Tutorial, Last accessed on: 02-13-2022.

[215] K. Wagstaff, C. Cardie, S. Rogers and S. Schrödl, “Constrained k-means clustering with background knowledge”, in *Proceedings of the Eighteenth International Conference on Machine Learning (ICML 2001), Williams College, Williamstown, MA, USA, June 28 - July 1, 2001*, C. E. Brodley and A. P. Danyluk, Eds., Morgan Kaufmann, 2001, pp. 577–584.

[216] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. VanderPlas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot and E. Duchesnay, “Scikit-learn: Machine learning in python”, *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, 2011.

[217] Y. Li, “Deep reinforcement learning: An overview”, *CoRR*, vol. abs/1701.07274, 2017. arXiv: 1701.07274.

[218] T. Mikolov, M. Karafiát, L. Burget, J. Cernocký and S. Khudanpur, “Recurrent neural network based language model”, in *INTERSPEECH 2010, 11th Annual Conference of the International Speech Communication Association, Makuhari, Chiba, Japan, September 26-30, 2010*, T. Kobayashi, K. Hirose and S. Nakamura, Eds., ISCA, 2010, pp. 1045–1048.

[219] F. A. Gers, J. Schmidhuber and F. A. Cummins, “Learning to forget: Continual prediction with LSTM”, *Neural Comput.*, vol. 12, no. 10, pp. 2451–2471, 2000. doi: 10.1162/089976600300015015.

[220] H. Xiao, K. Rasul and R. Vollgraf, “Fashion-mnist: A novel image dataset for benchmarking machine learning algorithms”, *CoRR*, vol. abs/1708.07747, 2017. arXiv: 1708.07747.

[221] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee and K. Skadron, “Rodinia: A benchmark suite for heterogeneous computing”, in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization, IISWC 2009, October 4-6, 2009, Austin, TX, USA*, IEEE Computer Society, 2009, pp. 44–54. doi: 10.1109/IISWC.2009.5306797.

[222] D. Molchanov, A. Ashukha and D. P. Vetrov, “Variational dropout sparsifies deep neural networks”, in *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, D. Precup and Y. W. Teh, Eds., ser. Proceedings of Machine Learning Research, vol. 70, PMLR, 2017, pp. 2498–2507.

[223] A. Aghaebrahimian and M. Cieliebak, “Hyperparameter tuning for deep learning in natural language processing”, in *Proceedings of the 4th edition of the Swiss Text Analytics Conference, SwissText 2019, Winterthur, Switzerland, June 18-19, 2019*, M. Cieliebak, D. Tuggener and F. Benites, Eds., ser. CEUR Workshop Proceedings, vol. 2458, CEUR-WS.org, 2019.

[224] G. Zhu and R. Zhu, “Accelerating hyperparameter optimization of deep neural network via progressive multi-fidelity evaluation”, *Advances in Knowledge Discovery and Data Mining*, vol. 12084, p. 752, 2020.

[225] S. F. Crone, S. Lessmann and R. Stahlbock, “The impact of preprocessing on data mining: An evaluation of classifier sensitivity in direct marketing”, *European Journal of Operational Research*, vol. 173, no. 3, pp. 781–800, 2006.

[226] J. J. Davis and A. J. Clark, "Data preprocessing for anomaly based network intrusion detection: A review", *computers & security*, vol. 30, no. 6-7, pp. 353–375, 2011.

[227] S. Falkner, A. Klein and F. Hutter, "Bohb: Robust and efficient hyperparameter optimization at scale", in *International Conference on Machine Learning*, PMLR, 2018, pp. 1437–1446.

[228] I. Rocha, N. Morris, L. Y. Chen, P. Felber, R. Birke and V. Schiavoni, "Pipetune: Pipeline parallelism of hyper and system parameters tuning for deep learning clusters", in *Proceedings of the 21st International Middleware Conference*, 2020, pp. 89–104.

[229] S. Dev, "Multitune: Dynamic budget allocation for hyperparameter tuning", 2020.

[230] J. Sanders and E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[231] *Nvidia graphic cards*, https://www.nvidia.com/en-gb/graphics-cards/, 2021.

[232] P. Warden, "Speech commands: A dataset for limited-vocabulary speech recognition", *arXiv preprint arXiv:1804.03209*, 2018.

[233] *Ag news*, http://groups.di.unipi.it/~gulli/AG_corpus_of_news_articles.html, 2021.

[234] J. Zhang, C. Zong *et al.*, "Deep neural networks in machine translation: An overview", *IEEE Intell. Syst.*, vol. 30, no. 5, pp. 16–25, 2015.

[235] I. Rocha, N. Morris, L. Y. Chen, P. Felber, R. Birke and V. Schiavoni, "Pipetune: Pipeline parallelism of hyper and system parameters tuning for deep learning clusters", *CoRR*, vol. abs/2010.00501, 2020.

[236] D. L. Quoc, F. Gregor, S. Arnautov, R. Kunkel, P. Bhatotia and C. Fetzer, "SecureTF: A Secure TensorFlow Framework", in *Proceedings of the 21st International Middleware Conference*, ser. Middleware '20, Delft, Netherlands: Association for Computing Machinery, 2020, 44–59.

[237] B. Moons, K. Goetschalckx, N. Van Berckelaer and M. Verhelst, "Minimum energy quantized neural networks", in *2017 51st Asilomar Conference on Signals, Systems, and Computers*, IEEE, 2017, pp. 1921–1925.

[238] W. Harvey, D. Kalp, M. Tambe, D. McKeown and A. Newell, "The effectiveness of task-level parallelism for production systems", *Journal of Parallel and Distributed Computing*, vol. 13, no. 4, pp. 395–411, 1991.

[239] R. Hempel, "The mpi standard for message passing", in *International Conference on High-Performance Computing and Networking*, Springer, 1994, pp. 247–252.

[240] L. Dagum and R. Menon, "Openmp: An industry standard api for shared-memory programming", *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[241] L. V. Kale and S. Krishnan, "Charm++ a portable concurrent object oriented system based on c++", in *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, 1993, pp. 91–108.

[242] C. Göttel, R. Pires, I. Rocha, S. Vaucher, P. Felber, M. Pasin and V. Schiavoni, "Security, Performance and Energy Trade-Offs of Hardware-Assisted Memory Protection Mechanisms", in *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*, 2018, pp. 133–142.

[243] D. Kaplan, J. Powell and T. Woller, "Amd memory encryption", *White paper*, 2016.

[244] V. Costan and S. Devadas, "Intel sgx explained.", *IACR Cryptol. ePrint Arch.*, vol. 2016, no. 86, pp. 1–118, 2016.

[245] *Intel Software Guard Extensions (Intel SGX)*, https://download.01.org/intel-sgx/sgx-linux/2.13/docs/Intel_SGX_Developer_Guide.pdf, Last accessed on: 02-13-2022.

[246] O. Port and Y. Etsion, "Dfiant: A dataflow hardware description language", in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, IEEE, 2017, pp. 1–4.

[247] *Maxcompiler*, https://www.maxeler.com/products/software/maxcompiler, Last accessed on: 02-13-2022.

[248] NVIDIA, *Cuda, release: 10.2.89*, https://developer.nvidia.com/cuda-toolkit, Last accessed on: 02-13-2022.

[249] *Hpc challenge benchmark*, http://icl.cs.utk.edu/hpcc/, Last accessed on: 02-13-2022.

[250] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement", *arXiv preprint arXiv:1804.02767*, 2018.

[251] M. Everingham, L Van Gool, C. Williams, J. Winn and A. Zisserman, *The pascal visual object classes homepage*, http://host.robots.ox.ac.uk:8080/pascal/VOC, Last accessed on: 02-13-2022.

[252] Y. LeCun, L. Bottou, Y. Bengio and P. Haffner, "Gradient-based learning applied to document recognition", *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.