Performance-Cost Trade-Offs in Heterogeneous Clouds

Anca Iordache

ANNÉE 2016

UNIVERSITÉ DE RENNES 1

UNIVERSITE BRETAGNE LOIRE

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Bretagne Loire

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

Ecole doctorale MATISSE

présentée par

Anca Iordache

préparée à l'unité de recherche n^o 6074 - IRISA
Institut de Recherche en Informatique et Systèmes Aléatoires
ISTIC

Performance-Cost Trade-Offs in Heterogeneous Clouds

Thèse soutenue à Rennes
le 9 septembre 2016

devant le jury composé de :

Alexander Wolf
Professeur, Imperial College London / rapporteur

Thilo Kielmann
Professeur associé, VU University Amsterdam / rapporteur

Lionel Seinturier
Professeur, Université de Lille 1 / examinateur

François Taïani
Professeur, Université de Rennes 1 / examinateur

Christine Morin
Directrice de recherche, INRIA / examinateur

Guillaume Pierre
Professeur, Université de Rennes 1 / directeur de thèse

Publications

Our contributions are published or under submission in peer-reviewed conferences, books and workshops.

International Conferences

- *Heterogeneous Resource Selection for Arbitrary HPC Applications in the Cloud.* Anca Iordache, Eliya Buyukkaya and Guillaume Pierre. In Proceedings of the 10th International Federated Conference on Distributed Computing Techniques (DAIS 2015), Grenoble, France, June 2015.

- *High Performance in the Cloud with FPGA Groups.* Anca Iordache, Peter Sanders, Jose Gabriel de Figueiredo Coutinho, Mark Stillwell and Guillaume Pierre. In Proceedings of the 9th IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2016), Shanghai, China, December 2016.

Book Chapters

- *The HARNESS Platform: A Hardware- and Network-Enhanced Software System for Cloud Computing.* Jose Gabriel de Figueiredo Coutinho, Mark Stillwell, Katerina Argyraki, George Ioannidis, Anca Iordache, Christoph Kleineweber, John McGlone, Guillaume Pierre, Carmelo Ragusa, Peter Sanders, and Thorsten Schütt. Book chapter in "Software Architecture for Big Data and the Cloud", Elsevier. To appear in 2017.

Posters

- *Accelerating Clouds with FPGA Virtualization.* Anca Iordache, Peter Sanders, Jose Gabriel de Figueiredo Coutinho, Mark Stillwell and Guillaume Pierre. EIT Digital Symposium on Future Cloud Computing, Rennes, France, 2015.

Résumé

La soumission d'une application sur une infrastructure de cloud, est précédée par les besoins des utilisateurs en ce qui concerne le coût et la durée de son exécution. Selon les circonstances, un utilisateur peut choisir l'exécution la plus rapide, la plus chère, ou un compromis entre les deux. De plus, les besoins des utilisateurs peuvent varier dans le temps. Faire des compromis entre coût et performance demande à avoir controle sur les propriétés de l'application qui influent le coût et la performance de son exécution.

L'exécution d'une application est largement influencée par la configuration des ressources de calcul, stockage et réseaux qu'elle utilise. La configuration des ressources est répresentée par un ensemble des paramétrés qui définissent des propriétés caractérisant les ressources utilises. Ces propriétés peuvent etre des types de ressources (calcul, stockage, etc.), des détails internes des ressources, le nombre de ressources d'un certain type, la connexion entre les differentes ressources dans la configuration, etc. La variation des paramétrés genére un grand nombre de configurations ayant des différents niveaux de performance. L'objectif de cette thèse est d'etudier des techniques pour controler l'execution des applications en choisissant prudemment les ressources qu'elles utilisent.

Dans les environnements de calcul traditionels, pour modifier la configuration des ressources, il faut realiser des opérations complexes comme l'aquisition des nouveaux équipements, leur installation, etc. Cependant, grâce aux environnements informatiques de type cloud on a l'opportunité de varier les réssources des applications plus facilement. Les infrastructures de cloud fournissent de larges capacités de calcul sur demande basées sur le modle de tarification à l'usage (pay-as-you-go). Depuis quelques annees, ces infrastructures sont constituees de groupes de serveurs homogenes. De plus l'adoption rapide des technologies du cloud par les utilisateurs du domaine de HPC, a conduit a une grande diversification des équipements informatiques disponibles. Ces utilisateurs ont généralement besoin d'une grande capacité de calcul. Aujourd'hui, ces utilisateurs peuvent choisir entre une grande diversité d'instances virtuelles optimisées pour différents cas: des instances avec une faible taille de memoire/CPU/disque pour le developpement des applications et l'exécution des applications moins exigentes; des instances optimisées stockage SSD- et HDD pour des applications données-intensives; des instances optimisées pour des applications HPC (calcul, memoire, GPU), etc. On peut dire que cette tendance à diversifier les ressources du cloud va continuer avec l'intégration des accélérateurs de type FPGA. Les industriels ont fortement investit dans cette demarche, comme c'est le cas de l'aquisition d'Altera par Intel et l'integration des FPGA dans les datacenters de Microsoft. Cette diversification des ressources du cloud donne une grande flexibilité pour les utilisateurs à choisir les ressources selon leurs besoins et notamment faire des compromis entre la performance et le coût de l'exécution de leurs applications.

Le choix des ressources de cloud pour exécuter une application en respectant un compromis coût-peformance est très difficile. D'abord, les utilisateurs ont des difficultés à estimer avec précision la performance de leurs applications dans certaines circonstances. Différentes types des ressources fournissent différents degrées de performance, quelquefois contre-intuitifs. Par exemple, une ressource à bas prix execute une application lentement et génère un coût total d'exécution élevé alors qu'une ressource apparemment plus chere est plus rapide et génère un coût total plus bas. De plus, varier le nombre de ressources utilisées pour l'exécution peut aussi fournir des compromis coût-performance interesants. Face à cette grande diversité de configurations, le choix du nombre et du type des ressources à utiliser pour obtenir le compromis coût-performance que les utilisateurs exigent constitue un défi majeur.

Dans cette thèse, nous soutenons que le choix optimal des ressources est trop complexe pour être accompli de façon réaliste par des utilisateurs. Au contraire, cette tâche difficile doit être automatisée avec des systemes informatiques. Ces systemés recoivent des applications arbitraires ainsi que les attentes des utilisateurs et choisissent la configuration des ressources satisfaissant la demande. L'implémentation d'un tel système peut radicalement simplifier l'usage des systemes de cloud pour les futurs utilisateurs.

Objectifs

Le choix des ressources pour exécuter une application conforme aux besoins des utilisateurs doit prendre en compte deux aspects. Premièrement, il est important d'optimiser l'utilisation des ressources allouées. Cela implique d'utiliser les ressources au maximum pour justifier leur coût. Deuxièmement, le choix des ressources satisfaissant un certain compromis coût-performance demandé par l'utilisateur doit être fait de manière efficace.

Objectif #1 : Bon usage des ressources : maximiser leur utilisation

Pour obtenir des excellents compromis coût-performance, nous devons d'abord utiliser les ressources attribuées à une application au maximum de leur capacité. Maximiser l'utilisation de ces ressources implique qu'on pourrait faire plus pour le même coût. Toutefois, le développement rapide du matériel informatique rend plus difficile pour les applications d'utiliser pleinement les ressources informatiques disponibles. Cette remarque est particulièrement vraie pour des accélérateurs comme les FPGAs et les GPUs. La sous-utilisation de ces ressources produit toutes sortes de problèmes comme du gaspillage d'énergie, faible disponibilité des ressources, coûts de gestion augmenté, etc.

Une technique courante que les clouds emploient pour maximiser l'utilisation de ressources est de partager ces ressources entre plusieurs utilisateurs. Cela est réalisé par virtualisation qui consiste partager une ressource matérielle en exposant les ressources virtuelles comme des composants autonomes de cette machine. La virtualisation permet à plusieurs applications de partager la même machine physique en même temps, ce qui facilite l'utilisation des ressources au maximum.

Les technologies de virtualisation ont été essentielles pour le développement des techniques de calcul dans le cloud. De nombreux efforts sont entrepris pour rechercher et developper des technologies de virtualisation pour partager les ressources des serveurs (processeur, mémoire, disque), du réseau et de stockage. Cependant, les exigences croissantes pour la capacité de calcul ont entraîné l'integration des accélérateurs comme des GPUs et des FPGAs dans les infrastructures des clouds [1, 2]. Des nombreux travaux de recherche ont été orientés en particulier sur la virtualisation des GPUs, alors que les FPGAs n'ont commencé que récemment d'être integré dans le cloud. Par example, dans le projet Catapult, Microsoft a utilisé des FPGAs pour accélérer des operations pour leur moteur de recherche Bing. Ceci a eu pour résultat l'accroissement de la performance de 95% avec seulement 10% de consommation énergétique supplémentaire [3]. En même temps, Intel a lancé un proceseur Xeon avec un FPGA intégrée, suivi de l'acquisition d'Altera, le deuxième grand fabricant des FPGAs. De plus, Intel affirme que pour 2020, 30% des serveurs des centres de données contiendront la technologie FPGA[4]. La virtualisation de ce type de ressource est un réel défi à cause de son architecture qui n'a pas été conçue pour cela. C'est à présent un sujet de recherche important pour l'avenir des centres de données. L'intégration des FPGAs dans les infrastructures de cloud permet un accès grand public à cette technologie qui est devenue de plus en plus populaire ; et offrira des possibilités pour maximiser leur utilisation en les partageant entre plusieurs utilisateurs.

Nous considérons que la maximisation de l'utilisation des FPGAs est la responsabilité de la plateforme de cloud et leur virtualisation doit être gérée automatiquement par la plateforme. Nous proposons comme contribution de cette thèse, une méthode de virtualisation pour groupes des FPGAs dans l'objectif de maximiser leur utilisation.

Objectif #2 : Bon choix des ressources

La seconde technique qui permet aux utilisateurs de contrôller les compromis côut-performance est le choix soigné de la configuration des ressources. La sélection des ressources sur lesquelles une application est exécutée conformément aux objectifs des utilisateurs est très difficile. Cela est dû au fait que, dans les clouds hétérogènes, l'espace de toutes les configurations des ressources possibles est très grand. La recherche des configurations optimales consiste à modeler la performance d'une application et comprendre comment les différentes configurations des ressources peuvent influencer son comportement pendant l'exécution.

Pour illustrer cela, Amazon EC2 propose aux utilisatuers d'essayer plusieurs types des ressources de façon empirique et choisir celle qui leur convient le mieux [5]. Chercher les configurations des ressources optimales selon la manière décrite par Amazon est très difficile au cause du grand espace des configurations possible. De plus, ce processus est long et côuteux. Les utilisateurs devraient analyser chaque exécution et sélectionner la configuration suivante pour être testée. Il est donc impossible pour les utilisateurs de gèrer efficacement ce processus de recherche.

Nous proposons que les plateformes de cloud gèrent automatiquement la sélection des ressources pour exécuter des applications conformément aux objectifs des utilisateurs. Pour cela, une plateforme doit analyser le comportement d'une application en l'exécutant plusieurs fois avec des configurations différentes. Après l'analyse de chaque exécution, la plateforme doit trouver les paramètres qui ont un impact sur la performance et le côut de l'exécution. La variation de ces paramètres peut générer des configurations de ressources intéressantes qui fournissent des bons compromis côut-performance.

En déléguant le choix de ressources aux platformes de cloud, l'effort de l'utilisateur pour exécuter des applications est donc réduit. Cela peut rendre les plateformes de cloud plus attractives pour les utilisateurs qui pourront consacrer leur temps et efforts pour améliorer le fonctionnement de leurs applications.

Nous proposons dans une seconde contribution de cette thèse, des méthodes de profilage pour la modélisation de la demande en ressources des applications. L'objectif de ces méthodes est de fournir des bons compromis coût-performance.

Contributions

Les contributions principales des cette thèse sont les suivantes:

Contribution #1 : Améliorer l'utilisation des FPGA par la virtualisation

Nous proposons une architecture de cloud pour l'intégration des accélérateurs de type FPGA dans des environnement multi-utilisateurs (partagées). Au lieu de considerer les FPGAs comme des dispositifs attachés à un seul serveur, nous placons un certain nombre des FPGA dans un serveur dédié qui est accessible via un réseau á faible latence. Ce placement augmente la disponibilité en permettant l'accès à un grand nombre d'utilisateurs a des FPGA. Cela permet à plusieurs FPGAs d'être utilisés par des applications executées sur des autres serveurs et aussi d'avoir des FPGA partagés entre plusieurs applications. Nous définissons un *FPGA virtuel* comme un groupe de FPGAs implémentant le même modèle hardware. Ce modèle hardware représente une description d'un circuit électronique implementé dans le FPGA. Le FPGA virtuel est consideré comme une ressource de cloud

de première classe plutôt qu'une propriété de base d'une machine virtuelle. Cela permet l'addition/suppression facile des FPGAs dans un groupe pour controler la performance d'une application. L'élasticité du FPGA virtuel est essentiele pour augmenter son utilisation. En conséquence, il nous est necessaire de varier automatiquement la taille des FPGAs virtuels conformément à leur charges de travail. Pour cela, nous proposons un algorithme de partage de temps pour implementer des migrations des FPGA physiques entre les FPGA virtuels. Automatiser l'adaptation à de differentes charges de travail a des nombreux avantages. Premierement, il augmente l'utilisation des FPGAs réduissant ainsi le temps de réponse des tâches soumises par les applications. Deuxièmement, la libération des ressources inutilisées d'un groupe reduit le coût du FPGA virtuel. Troisiemement, les FPGA inutilisés peuvent être migré dans des FPGA virtuels nouvellement créés, augmentant ainsi le nombre des applications qui ont acces à des FPGA.

Contribution #2: Automatiser le choix des ressources hétérogènes

Pour automatiser le choix des ressources de cloud pour exécuter des applications arbitraires et non-intéractives, nous proposons un nombre des méthodes de profilage qui ont l'objectif de trouver les configurations des ressources fournissant des bons compromis coût-performance. Le profilage represente l'analyse de l'exécution d'une application sur differentes configurations de ressources. Après chaque execution, le profileur actualise le modèle courant et choisit la configuration suivante à tester. La première méthode que nous proposons est basée sur un profilage "boîte noire" qui utilise un algorithme d'otimisation globale pour effectuer une recherche des configurations optimales de ressources. L'avantage de cette méthode est qu'elle gère des applications et des ressources sans avoir des détails sur leur fonctionalités. Pour cette raison, cette méthode est adéquate pour des applications arbitraires executées dans des environnments de cloud hétérogènes. Le noyau de la méthode consiste à explorer l'espace des toutes les configurations possibles en prenant uniquement en considération les configurations qui semblent fournir un bon compromis coût-performance.

Cependant, malgré la flexibilité à gèrer des ressources hétérogènes, la méthode a des limitations. Quand on gère des ressources sans avoir de détailes sur leurs fonctionalités, on perd l'opportunité d'optimiser le choix des ressources. Par exemple, une ressource de stockage peut fournir des informations sur l'espace maximal qu'une application utilise pendant son exécution. Cet information nous permettrait de réserver l'espace exact de stockage que l'application requiert et, en conséquence, reduire le coût de l'exécution. Pour surmonter ces limitations, nous proposons une extension avec une méthode "boîte blanche" qui exploite des informations sur l'utilisation des ressources pour améliorer le choix des configurations. Cette méthode va accelerer la recherche en se concentrant sur les parametres des ressources sous- et/ou sur-utilisées. En comparaison avec la méthode boîte noire, cette méthode diminue le nombre des itérations effectuées pour identifier des bonnes configurations des ressources intéressantes. Toutefois, dans le cas d'une application traitant de grandes quantités de données avec un temps detreécution considérable, le coût et la durée du profilage sont encore trop élevés.

Pour aborder ce problème, nous proposons une méthode complémentaire appellée "profilage extrapolé". Elle exploite des données d'entrée de dimension réduite pour diminuer le nombre d'itérations. Cette méthode suppose que la performance de l'application à des caractéristiques qui peuvent être observées en exécutant une fraction des données d'entrée. Elle consiste à effectuer une recherche "blackbox+whitebox" sur des données d'entrée de dimension réduite pour identifier rapidement des configurations de ressources intéressantes. Ensuite, parmi ces configurations nous en sélectionnons un certain nombre pour être testées avec les données d'entrée entières. Nous utilisons la corrélation entre

les temps d'exécution de l'application sur les deux jeux de données pour prédire la performance des configurations inconnues sur des larges données d'entrée. Cette methode réduit la durée et le côut du profilage et, selon la complexité de l'application, fournit des predictions raisonnables.

Contribution #3 : L'intégration des prototypes dans une platforme de Cloud hétérogènes

Nous avons conçu et implémenté un prototype pour chacune des contributions. Ils ont ensuite été intégrées dans une platforme de Cloud hétérogène developé dans le cadre du projet européen HARNESS[1]. Cette intégration valide nos contributions et nous en montre un exemple d'usage.

[1] www.harness-project.eu

Contents

Chapter 1

Introduction

Contents

This thesis addresses the problem of making good usage and choice of resources in heterogeneous clouds for executing arbitrary applications according to users' expectations. It focuses mainly on maximizing resource utilization and efficient resource selection to enable performance-cost trade-offs in cloud platforms.

1.1 Problem

Every user who launches a computation on a cloud infrastructure has implicit or explicit expectations regarding the performance and cost of its execution. Depending on the circumstances, a user may want the fastest possible execution, the cheapest, or any trade-off between the two. Moreover, users expectations may also vary over time. For example, Reverse Time Migration (RTM) is a compute-intensive algorithm used for creating 3D models of underground geological structures from data collected during measurement campaigns in specific areas. It is often used for oil exploration in deep-water areas. Geologists typically first execute RTM over data collected across a very large zone. Having to deal initially with large data sets, geologists may arguably be willing to trade-off execution time in favor of reducing the cost of the processing. However, once they identify a potentially interesting structure, they typically start to refine the geological model by running RTM repetitively on that particular area with various choice of configuration parameters. This time, reducing execution time becomes important because they need to wait for one execution to complete before they can choose the next set of configuration parameters. Users may therefore prefer to minimize the processing time of the data set representing the target area in order to quickly proceed with the geological model analysis.

Enabling performance-cost trade-offs requires control over the properties of an application which impact the performance and the cost of execution. The performance and the cost of a specific application typically depends on the following elements: input datasets, application implementation, and resource configuration. First, different input datasets may produce different performance by exercising different paths/branches of the application or augmenting the number of similar operations to perform [6][7]. Manipulating the properties of the input data would therefore allow one to change the runtime behaviour of an application. However, as the input dataset is given by the user, modifying its properties would change the purpose of the application execution. We therefore exclude this possibility from the discussion. Second, the performance and cost of application execution is often influenced by the implementation of the application. An application can be implemented in various ways to target different software and hardware platforms while providing the same functionality. For example, the aforementioned RTM application can be implemented using different algorithms which trade computation for storage [8]. Aside from CPU-based implementations, many algorithms can be implemented to offload core computation parts on accelerators such as field-programmable gate arrays (FPGAs) or general-purpose graphics processing units (GPGPUs). Making these implementation decisions has a strong impact on the performance and cost of the execution, and optimizing application implementations to exhaust a given set of resources is actively addressed in the area of high performance computing (HPC). However, optimizing an application is a laborious process which takes time. Every time the underlying hardware is upgraded, the application must be again optimized to exploit it efficiently. This becomes a burden for users when having to deal with a high number of applications. Because of this, in this thesis, we discard the approach of optimizing the implementation and rely on users to provide proper application code for a target resource type.

Finally, the runtime behaviour of an application is driven by the configuration of compute, storage and network resources used for execution. A resource configuration is defined by a set of parameters representing different properties of the involved resources. Such properties are the nature of the resources (compute, storage), internal details of resources, the number/size of resources of a specific type, the interconnection between different resources in the configuration etc. Varying these parameters generates a large

number of configurations which exhibit disparate performance levels. Thus, the purpose of this thesis is to study how one may control the runtime behaviour of arbitrary applications by carefully choosing and managing the resources used for their execution.

In traditional computing environments, varying resource configurations implies complex operations which require hardware purchases, installation, etc. However, thanks to cloud environments, we now have the opportunity to easily vary the resource configurations of applications. Cloud infrastructures provide on-demand large computing capacities based on a pay-as-you-go pricing model. A few years ago, they used to consist of large sets of homogeneous commodity servers. But, the quick adoption of cloud technologies by end-users from HPC domains having high compute requirements, has lead to a diversification of available hardware. Nowadays, users can select between a large number of virtual instances optimized to fit different use cases: general purpose instances with low memory/CPU/disk sizes for application development and running light workload use cases; SSD- and HDD-storage-optimized instances for data-intensive applications; compute-, memory-optimized and GPU instances for HPC applications, etc [9]. Arguably this trend is going to continue with the integration of FPGAs as cloud computing resources. Industry is heavily investing in this as exemplified by Intel's purchase of Altera and Microsoft's effort in integrating FPGAs in its datacenters [10][3]. This unprecedented level of flexibility creates great opportunities for cloud users to fine-tune their resource usage according to the needs of their applications and the performance-cost trade-offs they want to achieve.

Moreover, selecting the appropriate cloud resources to execute an application such that it respects a given performance-cost trade-off is difficult. First, users often find it very difficult to accurately estimate the performance of their applications in given circumstances. In HPC systems where the choice of resources is more limited than cloud environments, users often overestimate their computing requirements to avoid having their job killed before its completion [11] [12] [13]. This performance estimation problem escalates in cloud environments due to the huge diversity of resource configurations the user has to choose from. Cloud resources of different instance types deliver distinct performance-cost trade-offs which are sometimes counter-intuitive. A cheap resource may, for example, perform so slow that it generates a high total execution cost whereas an apparently more expensive resource may perform faster and end up with a lower execution cost. Moreover, varying the number of resources for a given execution also provides interesting trade-offs. When the runtime behaviour of an application can be varied by horizontal scaling, the addition and removal of resources generates many new possible configurations with different impact on the performance and cost of the execution. Because of the very large variety of possible configurations, deciding which and how many resources are needed in order to achieve an expected level of performance and/or execution costs is extremely challenging.

We claim in this thesis that the choice of optimal resource configurations is too complex to be realistically performed by end-users. Instead, this difficult task should be handled by generic automated systems which can take arbitrary applications and user requirements as input, and automatically optimize the resource configurations which should be used. Designing and implementing such a system may radically simplify the usage of cloud technologies for many current and future cloud tenants.

Choosing the right resource configuration to execute an application according to the expectations of the user must consider two aspects. First, it is important to optimize the use of allocated resources. This implies utilizing the resources at their maximum capacity to get as much work as possible in return for the cost of the resources. Second, we need to make a good selection of resources that satisfies the performance-cost constraints given by the user. We discuss these two aspect in the following sections.

1.1.1 Making Good Use of Resources : Maximizing Utilisation

The first way to obtain excellent performance-cost trade-offs is to make the best possible usage of the resources given to an application. Maximizing the utilization of resources implies that more work is done at the same cost. However, the rapid advances in hardware development often make it difficult for applications to generate constant work to keep the resources fully utilized. This is particularly true for accelerator devices such as FPGAs and GPUs. The underutilization of such resources creates issues as wasted energy, limited resource availability, increased management costs etc.

The standard technique by which clouds maximize their resource utilization is by sharing the hardware resources between multiple tenants. This is achieved using virtualization, which consists in partitioning a physical resource and exposing virtual resources as standalone encapsulations of the underlying machine. Virtualization allows many applications to share a same physical host at the same time, and therefore to use physical resources at their full capacity.

Virtualization technologies became a main focus point with the emergence of cloud computing. Many efforts have been made on researching and developing virtualization technologies for server resources (CPU, memory, disk), network and storage. However, the increasing computation requirements now lead to the integration of accelerators such as GPUs and FPGAs in cloud infrastructures [1][2]. In particular, while many research works focused on GPU virtualization, FPGAs have only recently started to draw more attention as promising cloud resources for high-performance computations. For example, Microsoft's Catapult project makes use of FPGAs to speed-up Bing search. This results in a gain of 95% more throughput with an increase of 10% in energy consumption [3]. At the same time, Intel released a Xeon chip with an integrated FPGA followed recently by the acquisition of Altera, second main FPGA manufacturer. Intel predicts that, by 2020, up to 30% of the servers in a datacenter will host an FPGA[4]. Virtualizing this kind of device constitutes a challenge as by design FPGAs were not intended for virtualization. Therefore, virtualization of FPGAs is becoming a hot research topic for the future datacenters. Integrating FPGAs in clouds can enable public access to this increasingly popular technology and will offer opportunities to increase their utilization by sharing them between tenants, thereby reducing their total utilization cost for the cloud tenants.

We consider that maximizing FPGA utilization is a responsability of the cloud platform and that its virtualization should be managed automatically by the cloud platform. We propose as a contribution of this thesis a method to virtualize pools of FPGAs with the goal of maximizing their utilization.

1.1.2 Making Good Choice of Resources

The second technique to allow cloud tenants to control their performance/cost trade-offs is to carefully select resource configurations. Selecting the right resource configurations for specific applications and user requirements is a difficult task because of the very large space of possible resource configurations that heterogeneous clouds provide. The search for optimal configurations consists in modelling the performance of an application and understanding how various resource configurations influence its runtime behaviour. To illustrate this simple idea, Amazon EC2 proposes to empirically try a variety of instance types and choose the one which works best [5].

> *"Because you can launch and terminate instances as desired, profiling and load testing across a variety of instance types is simple and cost effective. Unlike*

> *a traditional environment where you are locked in to a particular hardware configuration for an extended period of time, you can easily change instance types as your needs change. You can even profile multiple instance types as part of your Continuous Integration process and use a different set of instance types for each minor release."*
>
> (Jeff Barr. *Choosing the Right EC2 Instance Type for Your Application*)

However, the main difficulty when searching optimal configurations in the way recommended by Amazon, is the extremely large number of possible resource combinations we can use. For example, let us consider a IaaS cloud providing 30 predefined instance types. An application using five instances can choose among a total of 30^5=24.3 millions possible resource configurations. A more complicated scenario consists in using clouds providing dynamic configurations within the limits of their servers' capacity. In this case, the number of all possible resource configurations is much larger. Moreover, the search of optimal configurations is a long and costly process. Users are burdened with the analysis of every execution and the selection of the next configurations to be tested. Therefore, it is impossible for users to manage efficiently the search process.

We claim that cloud platforms should automatically manage the choice of the right resource configuration according to predefined user objectives. For this, the cloud platform must analyse the runtime behaviour of an application by executing it several times using different resource configurations. After the analysis of each execution, the cloud platform must derive the configuration parameters with an impact on the performance and cost of the execution. The variation of these parameters may generate new interesting configurations with good trade-offs. For example, let's consider an application which was executed on a configuration with 10 CPU cores and 100 GB of storage. The analysis of the runtime shows that the application, while intensively using the allocated cores, it did not use more than 50 GB of the allocated storage. This implies that a resource configuration with only 50 GB of allocated storage instead of 100 GB, may offer the same performance yet at a reduced cost. Furthermore, another configuration with a higher number or cores may also provide an interesting performance-cost trade-off as it may speed up the execution.

Putting the performance modelling process out of the hands of the user will therefore reduce the effort to manage the execution of applications on cloud platforms. This may make cloud platforms more attractive to users who will be able to dedicate their time and effort on application-level concerns rather than trying to understand the resource requirements of their applications.

The second main contribution of this thesis proposes methodologies for modelling application resource requirements in order to enable cost-performance trade-offs.

1.2 Contributions

The main contributions of this thesis are as follows:

Contribution #1: Improving FPGA utilization by means of virtualization. We propose a cloud architecture which supports FPGAs in a multi-tenant environment. Instead of considering FPGAs as local accelerator devices attached to a single server, we co-locate a number of FPGAs in dedicated servers accessed through a low-latency network. This setup increases availability by enabling a full many-to-many mapping between FPGAs and clients accessing them. This allows FPGAs to be used by applications located in different servers and even be shared between multiple applications.

We define a *virtual FPGA* as a group of FPGAs configured with the same hardware design. A hardware design is a description of the digital circuit to be implemented in the FPGA. The virtual FPGA is considered a first-class cloud resource rather than an attribute of a virtual machine. This allows to easily add/remove FPGAs from a group in order to control the performance of an application. The elasticity of virtual FPGAs is essential in maximizing their utilization. We therefore propose to automatically scale the virtual FPGAs based on their workload. For this, we introduce a time-sharing algorithm to implement physical FPGAs trade-offs between virtual FPGAs. Automating the adaptation to different workload demands provides several benefits. First, it increases the utilization of FPGAs which means lower response times for tasks submitted by applications. Second, removing unused physical FPGAs from a virtual FPGA decreases the cost of the virtual FPGA supported by users. Third, it maximizes the number of applications having access to FPGAs as running virtual FPGAs may trade their unused physical FPGAs to the newly created ones.

Contribution #2: Automating heterogeneous resource selection. To automate the selection of cloud resources for the execution of arbitrary non-interactive applications, we propose a number of profiling methodologies to find optimal resource configurations providing good performance-cost trade-offs. Profiling consists in investigating and analyzing the runtime behaviour of an application being executed on different configurations. After each execution, the profiler updates its current model and decides on the next configuration to test.

The basic method is based on *blackbox profiling* which uses a global optimization algorithm to perform a resource-agnostic search of optimal configurations to execute an application. Its main advantage is that it operates on applications and resources without having any knowledge on their inner workings. Therefore, this is a suitable method to apply for arbitrary applications executed in heterogeneous cloud environments. The core functionality of the method relies on strategically exploring large configurations spaces by focusing on configurations which appear to provide good performance-cost trade-offs. However, in spite of the flexibility in managing heterogeneous resources, the resource agnosticity brings some limitations. When handling resources without specific knowledge on their internal workings, we miss the opportunity to optimize the resource selection based on resource-specific knowledge. For example, a storage resource may provide an upper bound for the disk space the profiler should not exceed when making a storage request; a cpu-based resource may provide help in determining if the application is multi-threaded or not etc.

To overcome these limitations, we propose to extend the blackbox approach with a *whitebox approach* which uses resource utilisation to help selecting configurations which have a better chance than others to provide interesting performance-cost trade-offs. This speeds up the profiling process by focusing on the resource parameters that are considered to be over- or under-utilized. In comparison with blackbox profiling, the *blackbox + whitebox* approach reduces the number of iterations to be performed until good configurations are reached. However, it still induces significant profiling delays and costs overhead when dealing with applications processing large amounts of data where one run may take long time and, consequently, having a high cost.

To address this issue, we propose a complementary method named *extrapolated profiling* which exploits smaller input datasets and minimizes the number of runs with large datasets. This method assumes that application performance characteristics may be observed using input datasets representing a fraction of the ones used in production. We call these reduced inputs *benchmarking input datasets*. The extrapolated profiling approach

consists in running blackbox + whitebox profiling to parse the resource configuration space using a benchmarking input dataset. We then select a number of identified configurations to run the production input dataset; and use the correlation between the performance on the two datasets as a basis to predict the performance of the large input dataset on untested configurations. This approach minimizes profiling cost and runtime and, based on application's complexity, provides acceptable predictions for the tested applications.

Contribution #3: Prototype integration in a heterogeneous cloud platform. The integration of the prototypes in a cloud platform in the context of the European project HARNESS [14] validates our contributions. Moreover, it demonstrates how the methodologies we propose for maximizing resource utilization and making good choice of resources can be realistically integrated in future heterogeneous cloud platforms. We developed code prototypes for both previous contributions: an IaaS component called *Autoscaler* implementing the FPGA virtualization method; and a PaaS component called *Application Manager (AM)* implementing the profiling methodologies and resource selection based on user objectives.

1.3 Organization

This thesis is organized in five main chapters.

Chapter 2 discusses the background of this work. We start by providing an overview on virtualization technologies of resources and cloud resource heterogeneity. We discuss the role of accelerators in cloud environments and the current limitations for their integration as first-class resources. Finally, we introduce the application and resource management mechanisms implemented in current PaaS platforms and how user objectives are expressed and enforced on/by the PaaS platforms.

Chapter 3 of this documents introduces our first contribution: the FPGA virtualization technology. We start by presenting FPGA specific approaches to virtualization followed by a cloud setup enabling the approach we propose. Then, we show the benefits of the approach by presenting evaluation results.

Chapter 4 focuses on automating the resource selection to enable performance-cost trade-offs. We start by describing blackbox profiling and evaluate several search strategies of optimal configurations it can implement. Further we discuss the limitations of blackbox profiling and the approaches to overcome them: whitebox and extrapolation profiling followed by their evaluation.

Chapter 5 presents technical details of our two prototypes. We provide the generic cloud platform architecture designed in the context of the HARNESS project and the placement of our prototypes in it. We also discuss the requirements future PaaS must satisfy in order to integrate our prototypes.

Chapter 6 summarizes this thesis and provides future work directions.

Chapter 2

Background

Contents

Traditionally, a large part of the IT investment capital of an organization has been into the acquisition of hardware resources and in the employment of engineers to make it operational. Meanwhile, the time and effort of running applications concentrating the core business is much smaller than the time and effort spent for these operating processes [15, 16]. The *cloud computing* paradigm provides opportunities for organizations to cut these operating expenses and focus on the core business processes which are the ones essential in generating revenue. One of the most important economic benefits coming along with the use of clouds is the cost of running applications, as organizations now have the ability to provision cloud resources and services when they require them and release them when they do not need them anymore. Moreover, the cost of running applications can be easily calculated in advance due to the transparent cost model for cloud resources and resource billing based on a predefined time-unit. The support to run applications is implemented by the two fundamental cloud models: *Infrastructure-as-a-Service* (IaaS) and *Platform-as-a-Service* (PaaS).

IaaS clouds offer on-demand access to large varieties of virtual resources running on commodity hardware. The pay-per-use pricing model applied to cloud resources provides significant benefits for applications where users do not need constant accessibility to the resources. The key benefits such as instant availability and scalability of virtual resources make IaaS cloud attractive to users from different computing domains. However, there are several challenges for current cloud infrastructures to support the execution of workloads which are outside their original scope. Primarily, cloud infrastructures were built to host multi-tier web applications whose performance could be easily controlled by scaling out or up the resource configurations. Following the same principle, embarrassingly parallel applications such as MapReduce and Bag-of-Tasks applications are also suitable to be executed on cloud infrastructures because there is little or no dependency between their parallel tasks. This leaves out applications which have more complex resource demands and high connection inter-dependencies such as applications from the domain of high-performance computing (HPC).

HPC applications often require specialized compute devices, high-performance storage and fast interconnects. Satisfying these requirements is very challenging in a cloud environment where there are no guarantees of the level of performance of virtual instances and low-latency communication [17, 18]. HPC systems often encompass specialized devices as GPUs and FPGAs [19, 20] while communication mostly relies on high-speed interconnects as Infiniband [21].

However, current public clouds mostly rely on commodity hardware and offer limited access to GPU-enabled virtual instances with 10Gbps Ethernet as the fastest interconnect [22] while, for the time being, FPGAs are not yet provided as cloud resources. Because of this, cloud systems are not a great competitor to HPC systems for performance-critical applications. Therefore, incorporating not only specialized computing resources but also low-latency interconnects as cloud resources is a main requirement for current and future clouds to support new types of application and, consequently, attract new users with demanding applications. Nevertheless, the integration of new types of resources must take into consideration methods to virtualize them and maximize their utilization. These are important challenges cloud providers have to deal with in order to broaden the use of their infrastructures and to maximize their profit by exploiting them more efficiently. We address the problem of integrating FPGA-based accelerators in cloud infrastructures as the first contribution of this thesis.

PaaS clouds provide on-demand access to configured environments and services to automatically manage the execution of applications on virtual resource configurations pro-

vided by IaaS clouds. This cloud level enables the optimization of different metrics through the control of different attributes of application execution. Although there are many metrics users may be concerned with (e.g. response time, carbon footprint, resource utilization, cost), we choose to focus only on the performance (execution time) and the cost of executing an application.

Enabling performance-cost trade-offs is a very difficult task because of the huge diversity of applications having complex resource requirements and the increasing diversification of IaaS cloud resources. The main cause for application diversity is the diversification of IaaS cloud resources which provides flexibility in implementing applications as users would have access to new types of resources. Consequently, this diversity offers a richer context to make trade-offs for application execution.

The main motivation for enabling performance-cost trade-offs is to facilitate the selection of IaaS cloud resources to use. We consider it is necessary for PaaS platforms to enable trade-offs between these metrics to allow the optimization of application execution according to user expectations. We propose as the second contribution of this thesis, several approaches to resource selection in heterogeneous clouds.

2.1 Maximizing Resource Utilization in IaaS Clouds

Low utilization rates of computing resources constitute a major problem that IaaS providers constantly have to deal with. The main cause for low utilisation is that applications are not always able to produce sustained loads to exploit the underlying resources at their maximum capacity. Research studies have reported that most of the servers in a datacenter have an average utilization rate below 25% [23]. The utilization of the underlying infrastructure has an indirect impact on the pricing of the virtual resources provided by an IaaS cloud. For example, Amazon seeks to reduce their datacenter power consumption, hardware and operational cost which results in passing savings back to users in the form of lower pricing [24].

Underutilized resources create several issues an IaaS cloud provider must work through. First, idle resources consume a significant amount of energy without any productive purpose. A typical datacenter has been shown to consume 30 times the energy required to perform actual computations [25]. This means the maintenance cost greatly exceeds the computing cost of user services which are the ones generating profit for the datacenter providers. Second, another cause for underutilization is the resource fragmentation and the scattered distribution of workloads in the datacenter [26]. This raises difficulties in making an efficient capacity planning to accommodate future workloads. Furthermore, users often over-provision their applications to ensure a desired performance level is met. This excess of resources inflicts a cost overhead for the execution of applications with no return benefits [27].

Important progress is being made in cloud technologies in order to address these issues, mostly based on virtualization technologies. Virtualization at its most basic level provides multiple isolated views of the same physical resource. This allows multiple applications, previously each requiring one physical resource, to make use of virtual resources sharing the same physical host.

However, the placement policies of virtual resources on the physical hosts is very important in making efficient usage. Many approaches to consolidation of resources were therefore developed in order to maximize resource utilization and minimize energy consumption [28, 29]. Consolidation refers to the process of placing/migrating virtual resources on a minimum number of physical hosts [30, 31]. A particular case of server

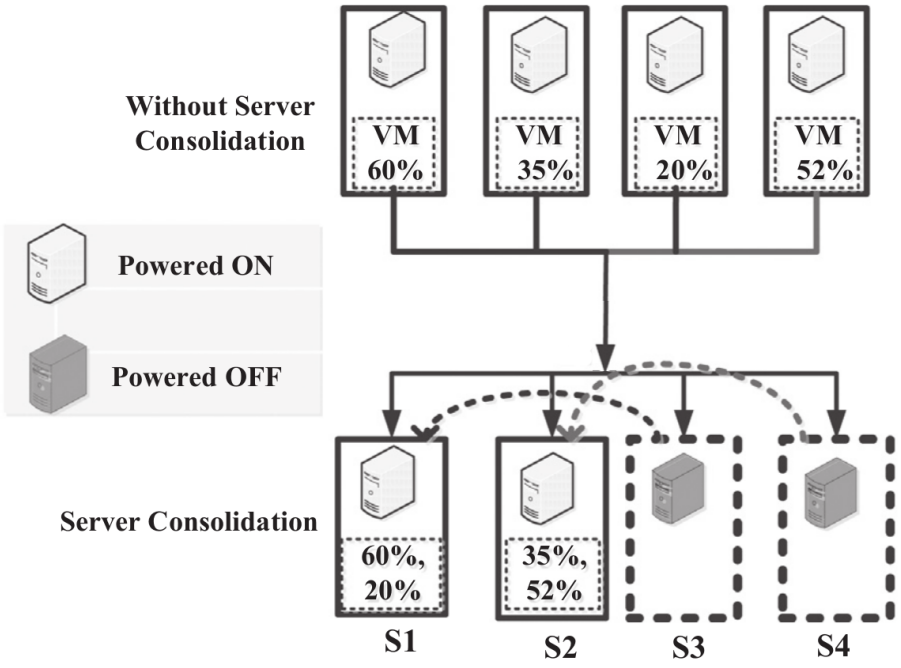


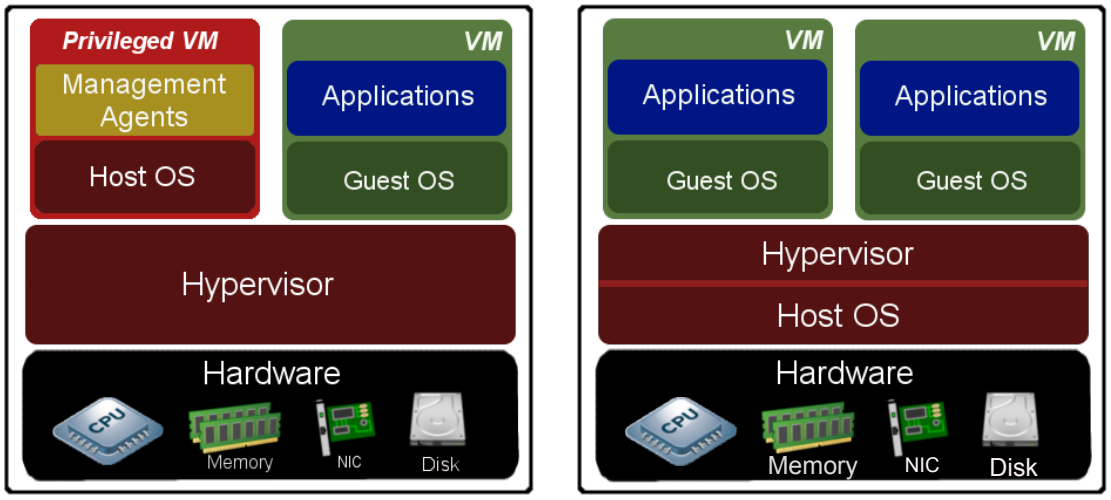
Figure 2.1: Server consolidation. *Adapted from [28]*

consolidation is depicted in Figure 2.1. The main goal of resource consolidation is to increase the utilization of active physical resources while allowing the powering off of the idle ones. Having a smaller number of active resources provides several benefits such as a more efficient management of virtual resources, lower energy consumption and operating costs. Furthermore, packing up the virtual resources on less physical resources also increases the resource availability for future demands.

Nevertheless, resource consolidation may not always result in effective usage of a cloud infrastructure as there are many issues affecting the consolidation process (e.g. application resource utilization patterns, workload variations, restrictions on the location of application components). These issues may make the consolidation process too long and inefficient from a power consumption and running cost perspective. Moreover, there is still the issue of the interference between co-located virtual resources [32]. Although over time, the performance overhead of virtual resources was reduced, they are not yet noise free.

2.1.1 Server Virtualization Technologies

A first barrier to overcome underutilization of resources is to enable multiple applications to use the same resource simultaneously. However, placing applications belonging to different tenants on the same machine creates numerous security issues. For this, virtualization emerged as a method to share a resource between different applications which are nevertheless totally isolated from each other. Virtualization organizes the partitioning of physical resources and exposes as virtual resources stand-alone representations of the underlying ones. The concept of virtualization first appeared in 1960 when IBM developed



(a) Type 1 Hypervisor (b) Type 2 Hypervisor

Figure 2.2: Popular cloud server virtualization technologies.

the idea of a *virtual machine (VM)* in order to give applications concurrent access to a mainframe computer [33].

The development of virtualization technologies has accelerated with the emergence of cloud computing whose foundation primarily relies on the virtualization of the resources in underlying datacenters. *Infrastructure-as-a-Service (IaaS)* clouds provide large pools of compute, storage and network resources and rely in particular on virtualization of server machines with the purpose to maximize the utilization of their infrastructure and increase their revenue.

The modern concept of virtualization refers to an abstract virtual layer with the main goal of hiding physical resources from virtual resources through the use of different virtual interfaces. Instructions on the virtual interfaces are translated by the virtualization layer into instructions on the physical resource. We focus further on the two most popular virtualization technologies implemented in a cloud infrastructure to share the resources of server machines.

Hardware virtualization

The most popular type of virtualization technologies employed in IaaS clouds relies on hardware virtualization (system-level virtualization) which provides virtual resources called *virtual machines (VMs)* as representations of full servers with the same hardware components as the physical ones. This technology enables a server(*host*) to run simultaneously other operating systems (*guest*) on top of its own.

The virtualization layer, *Hypervisor* also called *Virtual Machine Monitor (VMM)* controls the flow of instructions between the guest OSs and the physical hardware. As depicted in Figure 2.2, a hypervisor can be implemented in different layers of the virtualization software stack:

- *Type 1 Hypervisor*, running directly on the bare-metal machine, has complete control over the physical server. It is used to coordinate the low-level interactions between virtual machines and physical hardware. A *controlling domain (Dom0)* runs as a privileged VM to manage the host server and to control the other guest VMs. Ex-

amples of type 1 hypervisors are Xen [34], VMware vSphere ESXi [35] and Microsoft Hyper-V [36].

- *Type 2 Hypervisor* does not run directly on the hardware but on top of the operating system of the host and uses the interfaces provided by the host operating system to interact with the hardware. Examples of type 2 hypervisors are KVM [37] and VirtualBox [38].

Hypervisors implement different techniques for virtualizing the resources of a x86-based server machine such as CPU, storage, memory, and I/O devices. There are three main approaches for this.

First, the *full virtualization* approach delivers VMs which are complete representations of the underlying hardware running unmodified OSes and applications. Thus, decoupling the software from the underlying hardware allows to run VMs with proprietary OSes such as MacOS and Microsoft Windows. As the most reliable approach currently available, full virtualization provides the best isolation and security for VMs. However, it may incur large performance overheads because of the usage of *binary translation (BT)* which consists in replacing the code that the kernel of the guest OS wants to execute with a "safe" and slightly longer translated version [39]. Hence, user applications running on top of the guest OS can be executed straight on CPU without being translated. A full virtualization solution is implemented in the Linux KVM hypervisor.

Second, the *paravirtualization* approach aims to minimize virtualization performance overhead by modifying the guest OS to replace non-virtualizable instructions with *hypercalls* that communicate directly with the hypervisor [34]. It also provides hypercalls for critical kernel operations such as memory management, interrupt handling and time keeping. Therefore, the hypervisor and the guest OS work together more efficiently and deliver higher performance for some application than fully virtualization system. As an example, this approach is implemented in the hypervisor researched by Xen open-source project which virtualizes the CPU and memory using a modified Linux kernel and the I/O devices using custom guest OS device drivers [40]. However, paravirtualized hypervisors do not support guest VMs with unmodified OSes such as Windows 2000/XP, which limits compatibility and portability. Thus, the performance gain must be weighted against the maintenance and support cost for running modified guest OSes.

Third, the *hardware-assisted* approach relies on CPU features offered by newer generation processors, with Intel Virtualization Technology (Intel VT) or AMD Virtualization (AMD-V) technology, targeting privileged instructions with a new CPU execution mode feature that allows the hypervisor to run in a new root mode below the one with the highest priority in the x86 CPU ring architecture. A VMware's early research study on the first generation of processors offering this feature, showed that its BT-based virtualization outperforms the hardware-assisted virtualization [41]. The overhead incurred by the hardware-assisted virtualization is associated with the mechanisms employed by the hypervisor to map virtual memory addresses to physical memory addresses.

However, with the afterward introduction of hardware support for memory virtualization, address mapping is done through Extended Page Table (Intel) [42] or Rapid Virtualization Indexing (AMD) [43] support built into the Memory Management Unit (MMU). VMware studies show that this hardware-assisted memory mapping provides a significant performance boost compared to doing memory translation in software [44, 45]. Later adjacent research works focused on the optimization of memory paging in hardware-assisted virtualized systems [46, 47].

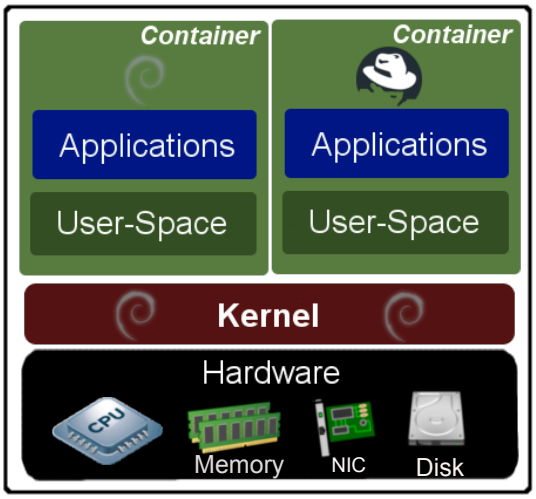


Figure 2.3: OS-level virtualization.

Most popular hypervisors such as KVM, Xen, VMware vSphere, Microsoft Hyper-V contain hardware-assisted virtualization extensions (Intel VT or AMD-V). Some research studies analyzed the performance of these hypervisors and show performance variations determined by the application type and requirements [48, 49]. They conclude that there is no ideal hypervisor to match all applications' needs.

Operating System-level Virtualization

The second increasingly popular type of virtualization technologies based on *operating system-level virtualization* aim to avoid the overhead incurred by the hypervisor-based approaches. Considering an OS having generally two layers: the kernel managing the hardware and the user space running processes, the OS-level virtualization refers to the virtualization of the kernel interface of the OS. This type of virtualization provides an alternative to VMs commonly known as *containers* which are virtual environments isolated with kernel Control Groups (cgroups) and kernel Namespaces [50]. The layout of OS-level virtualization is presented in Figure 2.3 where multiple user-spaces run in parallel on the same host and share the same kernel. However, the user-space layer must be compatible with the underlying kernel. For instance, it is not possible to run Microsoft Windows/MacOS user-spaces on top of the Linux kernel.

Although limited to running Linux-based guests and with a weaker isolation in comparison with a hypervisor-based virtualization, OS-virtualization provides some compelling advantages. One important advantage is the minimal performance overhead. Containers provide nearly native performance as the user-space applications and kernel run directly on the physical hardware. Consequently, resource requirements are also reduced. Another advantage is the real-time resource allocation as it does not require to load a full kernel for each new container.

Examples of container technologies are LXC [51], Docker containers [52], OpenVZ [53], Linux VServer [54] and BSD Jails [55].

Containers are becoming increasingly popular due to the rapid adoption of platforms as Docker which allows to easily build and manage Linux-based containers [52]. Although in the beginning Docker used LXC containers, more recent versions make use of *libcontainer* to integrate low-level Kernel namespace and cgroups features directly. The key difference between LXC and Docker containers is that LXC creates a container with mul-

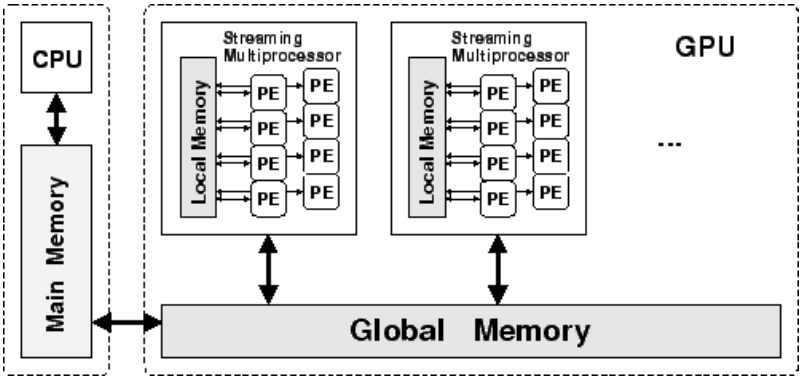


Figure 2.4: GPU architecture.

tiple processes while Docker reduces it to a single process. Various software systems have been developed to facilitate the scheduling, management and orchestration of distributed deployments of Docker containers. Examples of such systems are Swarm [56], Fleet [57] and Kubernetes [58].

Several research works investigated the overhead incurred for different use cases by several container-based technologies, some of them in comparison with hypervisor-based technologies [59, 60, 61, 62].

2.1.2 Accelerator Virtualization

An interesting category of I/O devices is represented by computational *accelerators* such as *general-purpose graphics processing units* (GPGPUs) and *field-programmable gate arrays* (FPGAs). Many compute intensive algorithms may benefit from using these accelerators as they offer orders of magnitude in performance improvement while being more energy efficient [63, 64].

However, only certain tasks of an application can be off-loaded to accelerators [65]. As a consequence, accelerators may be underutilized during the runtime of CPU sections of the application. To address this issue, virtualization techniques for accelerators are intensively researched in order to maximize their utilization.

GPGPUs

GPGPU processors are a popular compute device for accelerating computationally intense functions in many domains. An overview of the GPGPU architecture is presented in Figure 2.4. A GPGPU consists in a number of streaming multi-processors that contain a number of small processing elements and shared resources. This architecture allows to efficiently run thousands of threads to operate over a single dataset. Threads are run in groups which, in turn, are independently executed by the streaming multi-processors. Therefore, synchronization between groups is not possible during execution.

Code for GPGPU can be developed using C-based languages such as CUDA and OpenCL. Developers create special functions, called kernels, that are run on the GPGPU.

Client programs running on traditional machines typically use a GPGPU as a co-processor device. Applications access the GPGPUs via local library functions designed to issue kernel executions via the PCIe bus.

Integrating GPGPUs in a virtualized environment requires one to rethink the communication between the client program which runs within a virtual machine, and the

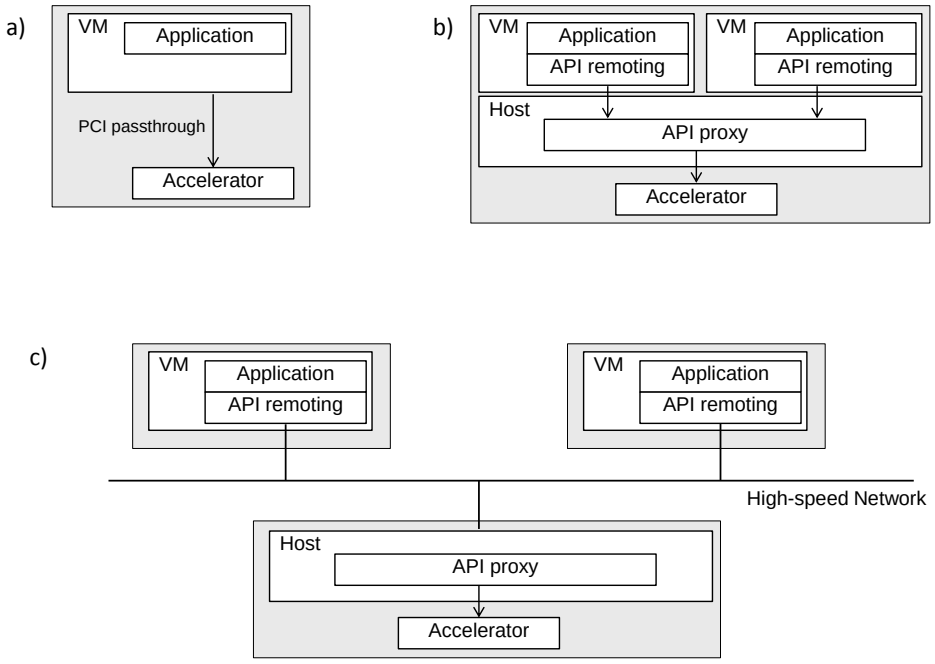


Figure 2.5: Local and remote virtualization of accelerators; (a) using PCI passthrough; (b) using API remoting locally; and (c) using API remoting remotely.

GPGPU. Ideally, this communication should exhibit as little overhead as possible so as not to reduce the interest of using accelerators. It should also allow any VM to make use of any accelerator device, regardless of the physical location of the VM and the accelerator. There exist three main techniques for this, which have been applied to support access to GPGPUs.

I/O passthrough consists of exposing the accelerator to a virtual machine running on the same server using the 'passthrough' feature supported by all modern hypervisors [66]. This technique, illustrated in Figure 2.5(a), is for example used by Amazon EC2 to create GPU-enabled virtual machine instances [67]. I/O passthrough delivers performance levels similar to the native capacity of the PCI bus. However, it restricts the usage of an accelerator only to the VMs running on the same host. In addition, it currently does not allow multiple co-located VMs to share the same physical GPGPU.

Paravirtualization allows multiple VMs to share a physical accelerator by providing a device model that is backed by the hypervisor, as well as the guest operating system and the device driver. Examples of approaches utilizing paravirtualization include GPUvm [68] and pvFPGA [69] supporting GPGPUs and FPGAs, respectively. However, this technique suffers from performance overheads and remains limited to sharing the accelerator between VMs running on the same server machine.

API remoting, shown in Figure 2.5(b), allows multiple applications to use a common accelerator API such as OpenCL or CUDA. Calls to the API are intercepted in the VM and passed through to the host OS on which the accelerator is accessible. Each

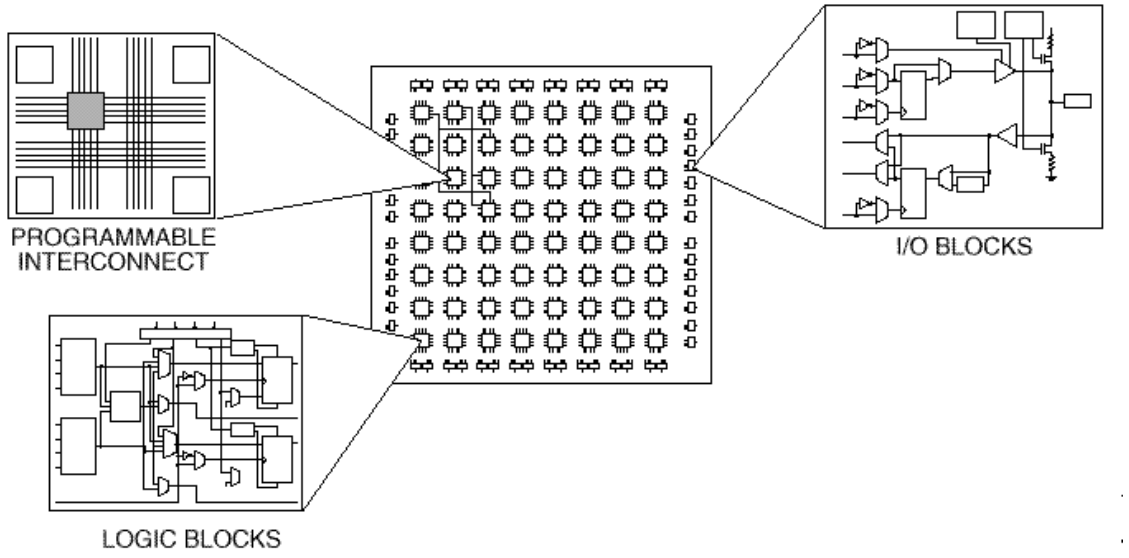


Figure 2.6: FPGA Architecture.

VM appears to have exclusive access to the accelerator and software on the host to resolves any contention. A number of technologies employ API remoting in a tightly coupled systems including vCUDA [70], gVirtuS [71] and LoGV [72]. In such a case the issue of sharing becomes one of scheduling and the applications obtain a fraction of the accelerator dependent on the other users and the scheduling algorithm employed.

API remoting also allows accessing a (possibly shared) accelerator remotely, as shown in Figure 2.5(c). In this case, the choice of network technology becomes very important as it can become the dominating performance factor.

FPGAs

FPGAs (Field-Programmable Gate Arrays) have the potential of complementing the current landscape of accelerator devices for high-performance computations. When applied to suitable problems, they deliver excellent performance, computational density and energy consumption.

Figure 2.6 presents an overview of the architecture of an FPGA. FPGAs are semiconductor devices organized as two-dimensional or three-dimensional arrays of Configurable Logic Blocks (CLBs) [73]. Each CLB is an elementary circuit which can be programmatically configured to realize a variety of simple logical functions, such as AND/OR/NOT digital gates, flip-flops and 1-bit full adders. The interconnection between CLBs is highly reconfigurable as well, allowing one to produce any digital electronic circuit — within the size limits of the gate array — by proper reconfiguration of the CLBs and their interconnection matrix. FPGA boards are often also equipped with external memory and a PCI Express® interface for high-performance communication with their host machine.

Although they are more difficult to program than a GPU accelerator, FPGAs provide the flexibility in what they can be programmed to do. FPGAs are well suited for repetitive digital processing tasks which need consistent timing: digital filters, signal processing, video/image processing etc.

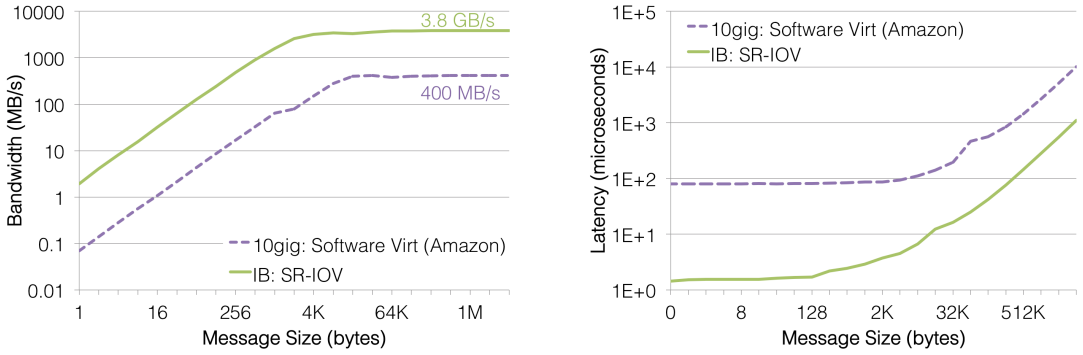


Figure 2.7: Difference in MPI bandwidth and latency for virtualized Infiniband and Amazon's 10GigE. Adapted from Glen K. Lockwood's blog [79].

FPGA circuit designs are usually implemented using hardware description languages such as Verilog and VHDL, which operate at a very low level of abstraction using logic elements such as digital gates, registers and multiplexors. These descriptions can also be automatically derived from higher-level programming languages similar to C, Java or OpenCL [74, 75] through a process called high-level synthesis (HLS). However, to obtain the best possible performance, FPGA programmers must still have a deep understanding of the mapping between a high-level program and its low-level translation, and annotate their code with appropriate optimization guidelines [76]. Efficient programming of FPGAs therefore requires specific training and experience, which effectively creates a barrier to entry for new developers to exploit the full potential of FPGAs. For this reason, FPGA manufacturers propose off-the-shelf highly-optimized library implementations for specific computations. Although many of these implementations are domain-specific, they can also provide generic functions such as various types of data analysis computations, multimedia processing, and machine learning [77, 78].

FPGAs can outperform CPU-based compute resources by at least one order of magnitude while running at considerably lower clock frequencies due to the fact that the physical organization of a hardware design can closely match the dataflow logic of the implemented algorithm and be realized by a deep pipelined architecture. This allows multiple function executions to operate in parallel over different parts of the gate array, providing high-throughput performance. A typical usage pattern therefore consists for a client program to issue large *batches* of task execution requests which can then be efficiently pipelined by the FPGA.

The issues and solutions for FPGA virtualization are very similar to the ones of GPGPU devices. However, FPGA virtualization has only recently started to be intensively studied because of the interest of major IT companies as Microsoft and Intel in exploiting these accelerators [3, 4].

2.1.3 Network Virtualization

Despite the advancements in virtualization technologies, I/O device virtualization still remains the major source for performance degradation. This problem aggravates in particular when dealing with network-intensive applications. Virtualization of network devices is an important concern especially when sharing remote accelerators.

Current virtualization technologies for network devices can be classified as software-based and hardware-based approaches. In software-based approaches, the network device cannot be accessed directly by the guest VMs/containers and each I/O operation is virtualized by several software layers : guest VM, hypervisor, privileged VM (xen). This solutions incur significant performance overheads. In the hardware-based approaches, the guest VM has direct hardware access. These approaches are relying on I/O virtualization support in the PCI Express adapters implementing the Single-Root I/O Virtualization (SR-IOV) standard [80, 81]. According to this standard specification, a PCIe device is presented as multiple separated devices called virtual functions. Virtual functions are lightweight representations of physical PCIe functions. [82] studies the performance of SR-IOV and software-based virtualization for 10Gb Ethernet and InfiniBand interconnects and draws the conclusion that SR-IOV provides substantial improvement in latency and negligible bandwidth overheads when compared with the software-based approach.

InfiniBand

The primary challenge towards cloud adoption for HPC is the virtualization of high-speed networking devices. HPC system are deployed widely with interconnects as InfiniBand, while cloud systems typically offer 10GigE as the fastest interconnect. Furthermore, Infiniband offers features as Remote Direct Memory Access(RDMA) which allows a host to write data directly into the memory of another host; and IP-over-Infiniband which enables TCP/IP over the Infiniband network. For these reasons, Infiniband is increasingly being deployed in clouds.

Virtualized Infiniband with SR-IOV provides comparable performance to the native Infiniband [83]. Figure 2.7 shows a comparison between the performance of a SR-IOV virtualized Infiniband and Amazon's 10GigE with software-based virtualization.

The usage of low-latency interconnects such as Infiniband in a cloud infrastructure is the dominant performance factors to efficiently virtualize remote accelerators [84].

2.1.4 Conclusion

Virtualization technologies for server and network resources made significant progress in reducing the performance overhead. GPU virtualization has developed dramatically over the last few years fueled by their integration in virtualized environments. However, FPGA-based accelerators only recently have started to be considered for the integration in virtualized environments. Very few reasearch works propose methods for FPGA virtualization and integration in cloud infrastructures which lack certain properties that FPGAs require in order to be considered first-class cloud resources [2, 85, 86]. Therefore, we propose as our first contribution a solution for virtualizing FPGAs and integrating them in cloud infrastructures with the purpose to increase their utilization. This first contribution is presented in Chapter 3.

2.2 Performance-Cost Trade-Offs in PaaS Clouds

IaaS clouds provide their tenants with a large diversity of on-demand virtual resources in order to satisfy various application requirements. The virtualization methodologies and the management of physical resources are hidden behind APIs which greatly facilitate the reservation and release of virtual heterogeneous resources [87, 88]. However, the deployment and execution of applications may be a burden for most users as they need to choose the right set of virtual resources for their applications, reserve them and then prepare

the environment to execute their applications. It becomes even more complicated when applications need to adapt at runtime to certain events such as varying workload, resource failures, crashes of child processes etc.

To address these issues, the PaaS model offers to automate these laborious tasks by providing configured virtual execution environments ready to manage the execution of user applications. Current PaaS systems offer several specialized virtual execution platforms. These execution platforms are designed to support applications implemented using a specific programming language (e.g., Python, Ruby and Java) and PaaS-specific tools and APIs for service management.

PaaS systems may rely on resources provided by IaaS clouds employing different deployment models: private, public and different combinations of public and private. Popular public IaaS providers also offer various PaaS services running on their infrastructures. For example, Amazon provides services as Amazon Elastic MapReduce [89] for data analytics applications running on top of the Hadoop framework; Amazon Dynamo DB and Amazon Relational Database System for database systems; and Amazon Beanstalk [90] for hosting web applications. Google's AppEngine [91], Microsoft's Windows Azure Web Services [92] and Rackspace's Cloud Sites [93] also provide support for hosting web applications on their infrastructure. Other PaaS providers such as Heroku [94] may not have private computing resources to run services and rather choose to rely on resources provided by public clouds such as Amazon and Salesforce. Open-source PaaS solutions follow the same approach. Examples of open-source platforms are ConPaaS [95], CloudFoundry [96] and AppScale [97].

The interactions between tenants and PaaS systems are in general governed by Service-Level Agreements (SLAs) establishing the criteria a PaaS should guarantee for the proper functioning of user applications. However, SLAs from popular PaaS providers focus mostly on resource availability and uptime while no guarantee is provided to optimize the performance of user applications [98, 99].

Despite the flexibility provided by the current PaaS systems in deploying and managing the execution of an application, there are still a number of issues to be considered when choosing a platform to use. Most PaaS systems provide only limited help in choosing the set of virtual resources whose performance-cost trade-off meet users' expectations. This is difficult because of the lack of performance guarantees for virtual resources which creates uncertainty in predicting the runtime behaviour of applications [100]. The choice of the type and number of virtual resources to be used is therefore delegated to the users when requesting the creation of a virtual execution platform [101, 102, 103]. However, users notoriously find it difficult to accurately estimate the exact needs of their applications [11, 12, 13]. This leads to higher execution costs than what could be achieved if the resources were selected with adequate knowledge.

2.2.1 Requirements

Enabling performance-cost trade-offs allows cloud users to have more control over the execution of their applications. We identify several criteria a platform should ideally meet in order to provide good performance-cost trade-offs and facilitate cloud usage.

- *Application agnosticity.* Many cloud platforms are limited to the management of specific types of applications such as Web and MapReduce applications. This leaves out important application families such as scientific applications, which may have complex implementations and resource requirements. These applications also could benefit from having access to the large variety of heterogeneous resources that clouds

provide to exploit a mix of traditional VMs and accelerators. Therefore, a good cloud platform should be able to handle a wide range of applications in a generic manner.

- *Heterogeneous resources support.* Cloud heterogeneity is not limited to a discrete set of configurations of server-based virtual resources. Specialized devices such as GPUs and FPGAs are also gaining traction in cloud infrastructures. This diversification of resources allows the execution of applications with different performance and cost levels. Therefore, it is important for PaaS systems to be able to manage heterogeneous resources and offer to include them in their performance models.

- *Minimum human effort.* Enabling performance-cost trade-offs for arbitrary applications requires a careful analysis of their resource requirements. Because of the large variety of heterogeneous resources that cloud provide, modelling the performance of applications may consider a large number of resource parameters which makes the process very complicated and difficult to be performed by users alone. Therefore, a good PaaS system should reduce the time and effort users would have to spend when analysing the behaviour of applications.

- *Modelling time and cost.* Analysing the behaviour of an application may require one to execute it repeatedly in different conditions (e.g., different resource configurations, input data) until the system can get a good understanding of its requirements. This can produce a significant cost and delay that users may not always be willing to accept. Therefore, performing the analysis with a low cost and time overhead is an important requirement for PaaS system in order to increase its attractiveness for users.

- *Support for variable input sizes.* Many applications are optimized to perform high-volume, repetitive tasks where successive executions process inputs with the same size and runtime behaviour. Targeting these applications first allows one to focus on exploring the large space of resource configurations to use for their execution. However, a good platform should also be able to manage the execution of applications with variable input sizes which can have a different impact on their runtime behaviour.

A cloud platform satisfying all the aforementioned requirements would facilitate the execution of applications in heterogeneous clouds. This may lead to attract more users having applications which can exploit the resource heterogeneity or encourage current users to adapt their applications to make use of different types of resources offering interesting performance-cost trade-off.

2.2.2 Utilization-based Approaches

Seeking to maximize resource utilization may not always be the best approach towards identifying different performance-cost trade-offs for the execution of an application. This is determined by the way the application makes use of resources.

Applications with complex resource requirements often exhibit a complicated resource utilization pattern for which it is very difficult to choose better resource configurations. For instance, Figure 2.8 presents a way a hypothetical application may make use of a set of resources (i.e., memory, CPU and FPGA). We can see several utilization peak switches between different types of resources (i.e., CPU and FPGA). Deciding which resource to add or remove in order to maximize utilization is challenging in this case because there

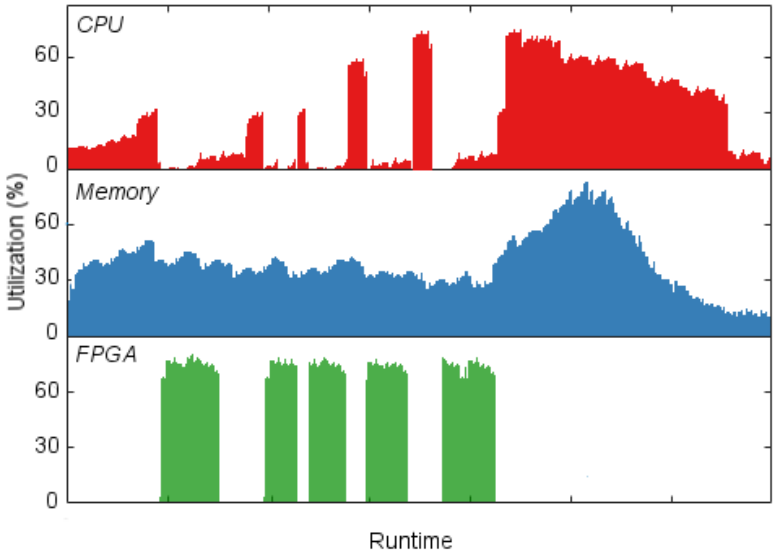


Figure 2.8: Example of a hypothetical application's resource utilization pattern.

is no single dominant resource determining the runtime behaviour. In such conditions, deciding which resources to scale in order to vary the runtime behaviour of the application is not trivial.

Based on their resource requirements, applications can be classified as *static* and *dynamic*. Static applications require a fixed set of resources which cannot be changed during their execution. Therefore, one must choose a good set of resources before launching them. Most batch applications (e.g., MPI applications) have this static nature. On the other hand, dynamic applications can adapt to changes in the resource set at runtime (e.g., MapReduce and Web applications). To model the behaviour of an application, one must carefully consider its static or dynamic nature. The performance of static applications is determined by the total runtime whether for dynamic applications is the response time for user requests.

Dynamic applications such as multi-tier web systems are popular cloud workloads exploiting features such as on-demand provisioning to maintain a certain level of quality of service. Most popular PaaS systems provide mechanisms to automate the adaptation of dynamic applications to changes in their workload. These mechanisms imply monitoring and autoscaling the resource configuration according to predefined elasticity rules.

Amazon Auto Scaling [104]. It is an Amazon PaaS service used to scale horizontally a resource configuration based on Amazon CloudWatch metrics, or predictably according to a schedule that users define. It gives users control over the resource configuration running their application. For example, users may set conditions to add more instances when the average utilization of the resource configuration is high. Similarly, when resources are underused, they may set conditions to remove instances. It makes use of CloudWatch alarms which are objects that watch over single metrics (e.g, the average CPU utilization of the EC2 instances in a resource set) over a specified time period. When the values of the metrics breach the thresholds that users defined, the alarms fire up Auto Scaling to make

a scaling decision based on the user conditions. Auto Scaling is well suited for dynamic applications where the resource demand is a function of resource utilization.

Google Compute Engine (GCE) Autoscaler [105]. This service is similar to Amazon Auto Scaling; it horizontally scales a group of homogeneous resources based on user predefined thresholds for certain metrics. Autoscaler works with metrics such as CPU utilization and requests/second/instance.

Rackspace Cloud Autoscale [106]. It offers similar horizontal scaling capabilities to Amazon's and GCE's systems. However, this system is noteworthy for its support for vertical scaling. For example, a user can create two sets of resources of two different flavour instances: small-flavour and high-flavour. Vertical scaling relies on switching between these two groups in certain conditions. When demand is low, the group with small-flavour instances may be horizontally scaled to couple with the demand. For high demands, the high-flavour group can scale up while the small-flavour group shrinks to zero. This way, it switches from using a small-flavour-based resource configuration to a high-flavour-based resource configuration. However, it does not allow the use of mixed-flavoured groups of instances.

2.2.3 Performance Modelling Approaches

Performance modelling is a complex process of analysing the runtime of an application in order to create a model to be used in predicting its behaviour in a given scenario. The accuracy of the prediction greatly depends on the time and effort invested in building a model. There are three main approaches to model the performance of an application.

- ***Analytical modelling***. This approach requires developers to provide models of their applications. These analytical models are potentially very accurate, but building them is labor-intensive and difficult to automate. Moreover, user estimates of application runtimes are often highly inaccurate [12, 11, 13]. Several research studies implemented analytical models employing different techniques such as queueing networks and statistics to model the behaviour of multi-tiered applications [107, 108, 109, 110, 111, 112]. Furthermore, the analytical models can be refined to capture burstiness and utilization variability [113].

- ***Code analysis***. This approach relies on tools which automatically analyse application code to create a model of execution paths. An example of a cloud system employing code analysis is [114].

- ***Profiling***. This approach relies on profiling methodologies which gather information from experimenting and analysing executions with different application and resource parameters. Profiling methodologies often use machine learning techniques to extract utilization patterns from historical traces. Examples of research works employing profiling are [115], [116] and [117]. The techniques proposed in this dissertation belong to this profiling family.

The most popular class of applications executed in clouds consists of online applications which are interactive systems deployed permanently whose processing demands vary over time according to user access patterns. Representatives of this class are the multi-tiered web applications. Web applications are distributed systems containing web servers, application servers, database systems and load balancers. As the dominant cloud workload, they are engineered to scale horizontally in response to demand. The metrics considered for scaling are the response time, throughput and cost. Modelling the

performance of web applications is done through different techniques relying on analytical models based on queueing networks and probability theory; and machine learning techniques to extract user access patterns. The models aim to derive good resource allocations to satisfy the current demand and ensure low response times while minimizing the cost. Research studies propose models using queueing network [107], machine learning and statistics methods [116, 118, 111] to achieve a required quality of service (QoS) while optimizing running costs. Nikravesh et al. [117] propose to improve the accuracy of predictive autoscaling systems by choosing appropriate time-series prediction algorithms based on incoming workload patterns. Vasic et al. [116] propose a cost-saving resource allocation methodology based on the analysis of past past workloads and the performance of the allocated resources using machine learning techniques such as clustering and classification. Fernandez et al. [118] propose to scale web applications in heterogeneous cloud infrastructures based on different performance-cost trade-offs implemented in a number of predefined QoS levels.

Another interesting class of applications being executed in clouds consists of batch, non-interactive applications having a limited runtime. Research studies focused on performance modelling specific types of batch applications such as bag-of-tasks [119] and MapReduce [120, 121] applications. They automatically derive based on the performance of a subset of their tasks, a resource scaling strategy to satisfy certain time and cost constraints. Oprescu et al. [119] propose a scheduler for executing bag-of-tasks applications in heterogeneous clouds within a user-given budget and without apriori knowledge on the task runtimes. It however assumes there is a form of task runtime distribution and uses stochastic methods to determine it at runtime.

Table 2.1 briefly presents several recent research works for modelling the performance of the two classes of applications according to the criteria defined in 2.2.1. Many of them rely on profiling instance types to identify their performance capabilities and use these profiles to make a schedule that meets constraints as minimizing execution cost within a given deadline, minimize cost while considering a specific level of QoS or minimize runtime within a given budget. Current performance modelling studies focus on specific classes of applications using heterogeneous resources limited to a small number of instance types. Moreover, the parameters of the performance model usually consists of an aggregation of properties of the programming model the application implements and parameters of the instance type's profile.

In conclusion, we have not found a single method that fully satisfies all the requirements defined in 2.2.1 and, therefore, we need better techniques such as the ones presented in this dissertation.

2.2.4 Conclusion

Current PaaS systems do not offer any support in making a good choice of heterogeneous resources when executing applications. This is an important issue becoming more prominent with the increasing heterogeneity of cloud infrastructures which also drives out the diversification of applications. Enabling control over performance-cost trade-offs is a desirable PaaS functionality which would simplify and assist in making a good resource selection which satisfies user expectations.

Therefore, we propose as our second contribution a solution for selecting heterogeneous resources to enable cost-performance trade-offs to run arbitrary applications. This contribution is presented in Chapter 4.

Table 2.1: A summary of related work.

Research Works	Applications	Heterogeneous resources	User effort	Modelling overhead	Variable input sizes	Modelling Objective
[104], [105]	web	no	high	no	yes (time-varying workload)	(user-defined metric)-based scaling
[106]	web	very limited	high	no	yes (time-varying workload)	(user-defined metric)-based scaling
DejaVu [116]	web	no	high	fair	yes (time-varying workload)	Minimize provisioning cost
BaTs [119]	Bag-of-Tasks	yes (predefined instance types)	very low	no	yes	Budget constrained execution
[118]	web	yes (mix of predefined instance types)	-	-	yes (time-varying workload)	Satisfy selected level of QoS requirements
[121]	MapReduce, Monte Carlo simulations	yes (different instance types)	-	fair	yes	Cost-Performance Trade-offs
[122]	MapReduce	yes (mix of different instance types)	-	-	yes	Cost-Performance Trade-offs
SOMPI[123], [124]	MPI	no (Spot Instances);	-	-	no	Minimize execution cost with deadline constraints

Chapter 3

FPGA Virtualization

Contents

The work described in this chapter was done during a 6-months internship at Maxeler Technologies under the guidance of Peter Sanders and has been submitted for publication.

Many efforts have been made on researching and developing virtualization technologies for different resources such as CPU, memory, disk, network and storage in order to maximize their utilization. However, the increasing computation requirements now lead to the integration in cloud infrastructures of new resources such as GPUs and FPGAs [1][2]. In particular, while many research works focused on GPU virtualization, FPGAs have only recently started to draw more attention as promising cloud resources for high-performance computations. They can offer invaluable computational performance for many compute-intensive algorithms. In particular, FPGAs have become increasingly popular within the high-performance computing community for their excellent computation density, performance/price and performance/energy ratios [125]. For instance, FPGAs are 40 times faster than CPUs at processing some of Microsoft Bing's algorithms [126]. FPGAs are commonly used in domains a diverse as financial market data processing [127], signal processing [128], and DNA sequence alignment [129].

Virtualization of FPGAs is becoming a hot research topic for the future datacenters as Intel claims that in the next few years up to 30% of the servers in the datacenters may host an FPGA. Therefore, integrating FPGAs in clouds can enable public access to this increasingly popular technology and will offer opportunities to increase their utilization by sharing them between tenants. This may justify the relatively high purchase and administration costs of such devices. However, maximizing utilization may be difficult for many applications whose computation needs are well below the capacity of one FPGA, or whose workload intensity significantly varies over time. The owners of such applications are therefore likely to ignore the benefits of FPGAs and prefer less efficient but more flexible solutions.

We claim that making FPGAs available in a cloud environment would lower the barrier and make them attractive to new classes of applications. For example, FPGAs can be programmed to execute the AdPredictor click-through rate prediction algorithm [130] orders of magnitude faster than its counterpart implementations based on CPUs [131]. A personalized advertisement service using this algorithm could exploit this performance to process incoming new facts in real-time to continuously adapt its recommendations to any change in user behavior. However, simply providing entire physical FPGAs attached to a virtual machine instance (similar to the GPU-enabled instance types proposed by Amazon Web Services) would not be sufficient, as the workload of a personalized advertisement service may vary considerably over time. Maximizing the FPGA utilization therefore requires one to deliver elastic processing capacities ranging from fractions of a single device's capability to that of multiple devices merged together.

Turning complex FPGA devices into easy-to-use virtual cloud resources requires one to address two main issues. First, programming FPGAs requires skills and expertise. FPGAs are essentially a set of logical gates (AND, OR, NOT) which can be dynamically wired programmatically. The best performance is obtained when the FPGA design closely matches the data-flow logic of the program itself, following a pipelined architecture. Such circuit designs are typically compiled from high-level programming languages, but this process still requires specific skills and experience [74]. While such difficulties are not a major issue for large organizations willing to invest massively in FPGA technologies, they can be a significant hurdle for other users.

The second issue is the lack of satisfactory techniques for virtualizing FPGAs. Current solutions are based either on statically partitioning the gate array between multiple applications (i.e., sharing the FPGA in space), or on naive context switching (i.e., sharing the FPGA in time). As extensively discussed in Section 3.1, both techniques exhibit significant problems: sharing in space implies that each application must be implemented with

a smaller number of digital gates, thereby negatively impacting performance. Conversely, naive time sharing incurs prohibitive context-switching costs, as reconfiguring an FPGA from one circuit design to another may take in the order of a couple of seconds.

To become fully integrated as regular cloud resources, virtual FPGAs should exhibit the following properties:

Management. Physical FPGAs should expose an abstract interface that allows them to be actively managed by the cloud platform, including tasks that deal with reservation and deallocation, deployment and execution, as well as providing monitoring information about their utilization.

Programmability. Once FPGAs have been provisioned to cloud tenants, they should be programmable to tailor their application needs, similar to CPU compute resources.

Sharing. Like other types of cloud resources, FPGAs should be virtualized, allowing multiple instances of the same physical device to be used as isolated resources, in order to maximize resource utilization and the availability of resources.

Accessibility. To facilitate sharing, FPGAs should not only be made available to virtual machines executing on the same host. Rather, they should be organized as a pool of resources accessible from any host.

Performance. To retain the high performance/price ratio of physical FPGAs, the performance overhead of FPGA virtualization should remain minimal.

High utilization. When multiple virtual FPGAs compete for a limited set of physical resources, the processing capacity of physical FPGAs should be dynamically assigned to the virtual FPGAs which need it the most.

Isolation. Sharing resources requires that each resource instance is completely isolated from each other, not allowing tenants to access each other's data through the shared device.

We propose a new virtualization technique for FPGAs called *FPGA groups*. An FPGA group is composed of one or more physical FPGAs which are configured with the exact same circuit design. By load-balancing incoming execution requests between its members, an FPGA group can be considered by its clients as a *virtual FPGA* with aggregates the computational capacity of multiple physical FPGAs. FPGA groups are elastic, as one can easily add or remove physical devices to/from a group. An FPGA group may, for example, be created by a cloud tenant whose computational needs exceed the capacity of a single FPGA.

FPGA groups may also be *shared* among multiple tenants who wish to use the same circuit design. Although this condition is very unlikely in the case where tenants compile their own circuit designs from custom code, we claim it is realistic in the case of circuit designs chosen from a standard library. Such a library would typically contain highly-optimized circuits for common types of functions in domains such as data analysis (with functions such as regression, correlation and clustering), multimedia (with functions such as video encoding and fast Fourier transform), and machine learning (with functions for Bayesian and neural networks).

Finally, multiple FPGA groups may also *compete* for the use of a limited set of physical devices. In this case, we present an autoscaling algorithm which dynamically assigns FPGAs to FPGA groups. This algorithm maximizes FPGA utilization (which improves

Table 3.1: A summary of related work.

Approach	Accelerator type	Sharing method	Accessibility	Sharing type
Amazon EC2 [67]	GPGPU	None (PCI passthrough)	Host	Many-to-one
GPUvm [68]	GPGPU	Paravirtualization	Host	Many-to-one
pvFPGA [69]	FPGA	Paravirtualization	Host	Many-to-one
vCUDA [70], gVirtuS [71]	GPGPU	API remoting	Host	Many-to-one
rCUDA [132], DS-CUDA [133]	GPGPU	API remoting	Network	Many-to-one
Byma *et al* [85], Chen *et al* [134]	FPGA	Partial reconfiguration	Host	Many-to-one
Shared FPGA groups	*FPGA*	*Time-Sharing*	*Network*	*Many-to-many*

the cloud provider's revenues), while reducing individual task execution times in most cases.

Our experiments show that FPGA groups incur a low overhead in the order of 0.09 ms per submitted task, while effectively aggregating the processing capacity of multiple FPGAs. When faced with a challenging workload, our autoscaling algorithm increases resource utilization from 52% to 61% compared to a static resource allocation, while reducing the average task execution latency by 61%.

This chapter is organized as follows. Section 3.1 introduces the background and related work. Section 3.2 presents the design of FPGA groups, and Section 3.3 discusses elasticity and autoscaling. Finally, Section 4.5 evaluates our work and Section 3.5 concludes.

3.1 State of the Art

Allowing cloud tenants to reserve fractions of an FPGA's processing capacity requires a cloud operator to share FPGA devices among multiple tenants. Most of the research on accelerator virtualization is focused on the many-to-one solution where several applications share a single accelerator, since the accelerator is the expensive device that needs to be shared.

Sharing in space consists of running multiple independent FPGA designs (possibly belonging to different tenants) next to each other in the gate array. This is made possible by a technique called *partial reconfiguration* where the FPGA area is divided in multiple regions, allowing each region to be reconfigured with a particular circuit design [85, 134]. This approach effectively parallelizes the execution of the different designs, and increases the device's utilization. However, space sharing reduces the area that is made available to host an FPGA design, which can have a considerable performance impact because it limits the number of functional units that can work in parallel. Sharing in space also requires some switching logic in the FPGA to route incoming requests to the appropriate design, which can add additional overhead.

Sharing in time consists of executing a single FPGA design at any point of time, but of switching the FPGA usage from tenant to tenant over time (similarly to operating system process-level context switching). This approach is often overlooked in the FPGA community due to the very high reconfiguration costs from one design to another: a naive implementation would impose prohibitive context switching costs in the order of a couple of seconds.

The *shared FPGA groups* we propose belong to the sharing-in-time category. In order to avoid issuing costly FPGA reconfigurations each time a new task is submitted, shared FPGA groups retain the same configuration across large numbers of task submissions,

and incur reconfiguration costs only when an FPGA needs to removed from one group and added to another. Apart from this infrequent operation, tasks submitted by multiple tenants can therefore execute with no reconfiguration delay.

A summary of related work described in this section is presented in Table 3.1. These approaches focus on many-to-one scenarios allowing multiple VMs to share a single FPGA. To our best knowledge, our approach is the first virtualization method which also considers the one-to-many and the many-to-many situations where an application uses what it considers to be a single (virtual) accelerator which is backed by multiple physical FPGAs.

3.2 FPGA Virtualization

A client application using FPGAs usually executes on a host that has an FPGA board connected via a dedicated bus, such as PCI Express®. The application must first reconfigure the FPGA with a specific design which provides the appropriate function, and then perform a sequence of I/O operations on the FPGA, including initializing static data on the FPGA, streaming input data to it, and streaming results from it. The computation is performed on the data as it flows through the logic of the FPGA design.

However, in this architecture the FPGAs are accessible only to applications running on the host machine, which limits the number of applications that can share the FPGAs. To support full many-to-many mapping between FPGAs and the VMs accessing them, our solution is to co-locate a number of FPGAs in specific "*FPGA-Server*" appliances which can be communicated with using the API remoting technique over an Infiniband network. This solution allows the client applications to run on commercial servers and use FPGAs where necessary on the FPGA-Server appliance.

3.2.1 The FPGA-Server Appliance

Figure 3.1 provides an overview of an FPGA-Server. In this setup, tasks and results are sent to the appliance across an Infiniband network. This network technology is particularly suitable for such usage thanks to its low packet latency, and RDMA support which avoids unnecessary data copying within the FPGA-Server.

The switch fabric of the FPGA-Server manages the arrival and execution of tasks on the appropriate FPGA. Although it would seem natural to implement it in software, and to execute it in a standard processor within the FPGA-Server device, such design is likely to incur in significant performance overheads without RDMA support [135, 136]. Instead, we decided to implement the switch fabric directly in hardware. As the FPGA-Server is only concerned with the efficient execution of tasks on FPGAs, it can be designed solely for this purpose so its performance is not be impacted by the side effects of running the application's CPU code.

We envisage that a cloud infrastructure supporting FPGAs in a multi-tenant environment will consist of multiple FPGA-Server appliances interconnected with (some of) the CPU machines using Infiniband networks. Each FPGA-Server provides a number of FPGA devices accessible to any CPU client machine via the Infiniband network.

Our FPGA-Server appliances include eight FPGAs — each equipped with 48 GB of RAM — and two Infiniband interfaces to overcome any bandwidth limitation of the Infiniband connection.

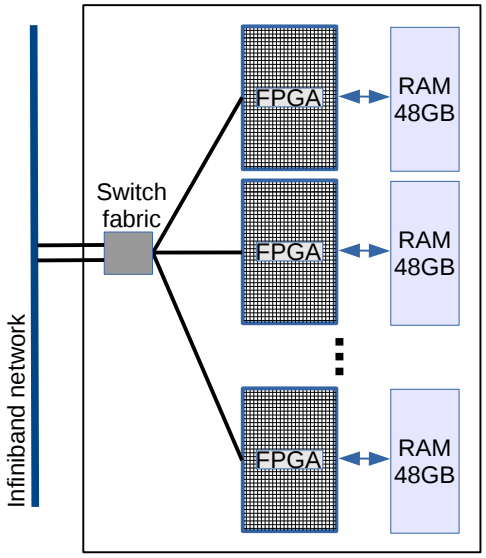


Figure 3.1: An FPGA-Server with eight FPGAs and two Infiniband interfaces.

3.2.2 Resource Management

FPGA-Servers and physical FPGAs are passive hardware devices. They have no operating system nor embedded software that can handle even basic resource management operations such as allocating/deallocating FPGAs, sharing them between applications, and establishing a connection with authorized client programs.

These operations are handled by a software component called the *Orchestrator*, as shown in Figure 3.2. The Orchestrator runs in a regular server machine connected to the same Infiniband network as the FPGA-Servers and the client machines. Similar to cloud resource manager services such as OpenStack Nova, the Orchestrator is in charge of maintaining the reservation state of each FPGA, including whether it is currently reserved and by which tenant. When a client reserves one or more (physical) FPGAs, the Orchestrator chooses available FPGAs, updates their reservation state, and returns a handle to the client containing the address of the FPGA-Server it belongs to and a local identifier. The Orchestrator is also in charge of managing the FPGA groups, as we discuss next.

3.2.3 FPGA Groups

A client application which requires the combined processing capacity of multiple FPGAs can obviously reserve the devices it needs, configure all FPGAs with the same design, and load-balance its task execution requests across them. However, this introduces complexity in the application as it would need to decide how many FPGAs to request and then load balance the tasks across those FPGAs. Sharing these FPGAs across multiple client machines is even more challenging.

We propose a virtualization infrastructure using so-called *FPGA groups*. An FPGA group presents itself to an application as a single virtualized FPGA. However this virtual computational resource can be backed by a pool of physical FPGAs that perform the actual computation. All the FPGAs within the group are configured with the same hardware design. The client application submits tasks to the virtual FPGA in exactly the same way it would of a single physical FPGA.

As shown in Figure 3.2, in our current implementation, an FPGA group resides entirely within a single FPGA-Server. When the Orchestrator receives a request to create a new

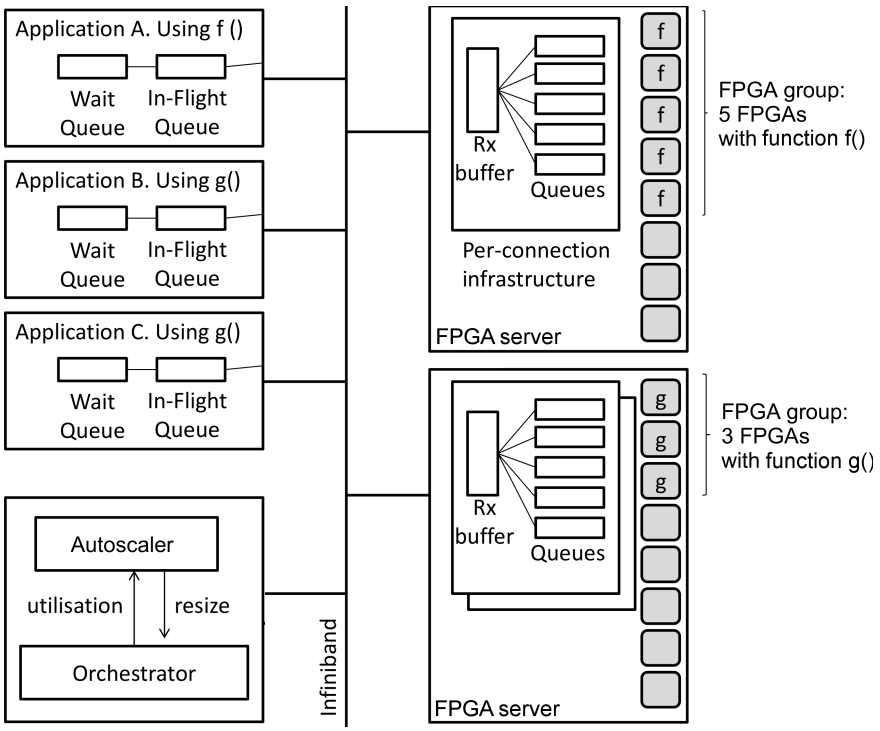


Figure 3.2: A simplified system architecture with two FPGA-Server appliances on the right-hand side and three application VMs on the left-hand side. Application A is using the function $f()$ provided by the an FPGA group composed of five physical FPGAs. Applications B and C are sharing the FPGA group supplying the function $g()$, which is composed of three physical FPGAs. The figure also shows the Orchestrator and Autoscaler management components. All the components are connected with an Infiniband network.

group, it allocates the required number of FPGAs within one FPGA-Server. It then sets up a new receive (Rx) buffer to hold the arriving tasks requests and a number of queues equal to the number of physical FPGAs in the FPGA-Server (regardless of the number of FPGAs assigned to this group). A new set of Rx buffer and queues is created each time a new client application wants to use the FPGA group, and deleted after the client application terminates. For example, in Figure 3.2, the FPGA group holding function $g()$ is shared between application B and application C; each application access its own Rx buffer and queues in the FPGA server.

On the application side the group is represented as a fixed-size *in-flight queue* which holds knowledge of all the tasks that are being processed on the FPGA-Server. There is also a *wait queue* which stores the waiting tasks and is not bounded in size.

We carefully designed this organization to minimize the task-submission overhead. Each application can submit its own tasks from the client-side wait queue to the group-side queues with no interference with other clients. When a task arrives in the FPGA-Server's Rx buffer, the FPGA-Server places the tasks into the task queue with the shortest length. The request is then placed into the task queue with the shortest length.

Creating as many task queues as there are FPGAs in the FPGA-Server implies that the number of task queues is always greater or equal to the number of FPGAs in a group. This allows (as we discuss in the next section) to minimize the time taken to dynamically scale the group at runtime. When a queue has at least one task to execute, it attempts to lock an FPGA from the pool of FPGAs in the group. When a lock has been obtained,

the task is executed on the corresponding FPGA. After execution, the FPGA is unlocked and the task result is sent back to the application. Thus, each queue has a simple and sequential behavior. Note that tasks in the same task queue do not necessarily execute on the same FPGA.

An additional benefit of creating a separate Rx buffer per client application is that it makes it easy to revoke an application's access to the FPGA group, for example if the tenant decides to stop using a shared FPGA group.

3.2.4 Discussion

We now discuss a number of specific issues about virtual FPGAs.

Memory isolation. One important issue for virtual FPGAs is memory isolation between multiple independent tasks running simultaneously in the same FPGA group, but potentially belonging to different tenants. FPGAs do not have Memory Management Units (MMUs) nor kernel/userspace separation, which makes it impossible for them to page memory the same way CPU-based servers do. When an FPGA group is used by a single user (e.g., to serve tasks issued by an elastic set of VMs), the FPGA group will consider all incoming tasks as if they had been issued by a single VM. In multi-tenant scenarios, our system currently does not address isolation issues. As a consequence, we restrict this multi-tenant scenario to the use of stateless FPGA designs such as signal processing and video encoding.

Inter-task dependencies. We assumed so far that tasks are independent from each other: this allows us to execute incoming tasks in a worker thread-pool pattern. However, one may for example want to stream large amounts of data and to perform statistical analysis across multiple tasks. To address these issues, a client can send a pseudo-task to the FPGA group which performs a 'lock" function. This then reserves a queue and its associated FPGA exclusively for this client until the 'unlock" function is sent. Obviously, we can lock only a finite number of concurrent times – i.e., the number of FPGAs.

Orchestrator scalability. Although we represented the Orchestrator (and the Autoscaler introduced in the next section) as a single server in our architecture, they maintain very simple state which may be stored in a fault-tolerant key-value store: for each FPGA group they essentially store a copy of the circuit design (which represents about 40 MB per design) and the list of FPGAs which are currently part of the group. This allows for easy replication and/or partitioning across the data-center. In addition, any temporary unavailability of the Orchestrator would only impair the ability to start/delete/resize FPGA groups, without affecting the availability of the existing FPGA groups, or their functionality.

Scaling an FPGA group beyond a single FPGA-Server. There is no fundamental reason why FPGA groups cannot span more than one FPGA-Server. This is essentially a limitation of our current implementation: spreading FPGA groups across multiple servers would simply require an additional load-balancing mechanism at the client side, which implies that the client-side library must receive notifications upon every update in the list of FPGA-Servers belonging to a group. This feature is currently under development.

3.3 Elasticity and Autoscaling

What has been described so far uses a fixed number of physical FPGAs in each FPGA group. However this virtualization structure allows the easy addition and removal of FPGAs from a group without any disruption to executing tasks or any interaction with the client application. We discuss FPGA group elasticity first, then present our algorithms for automatically controlling FPGA group sizes to maximize resource utilization.

3.3.1 Elasticity of Virtual FPGAs

The design of FPGA groups makes it easy to add or remove FPGAs to/from an FPGA group located in the same FPGA-Server. All that has to be done is reconfigure the FPGA with the appropriate design and update the load-balancing information in the task queues at the server side. The resized group still presents itself to an application as a single virtualized FPGA, however this virtual computational resource has a varying number of physical FPGAs that will perform the actual computation.

This elasticity allows a cloud provider to place multiple FPGA groups on the same FPGA-Server and dynamically reassign physical FPGAs to groups according to the demand each group is experiencing, similar to the way hypervisors can dynamically reallocate the CPU shares granted to virtual CPUs. This automatic elasticity is managed by a software component called the *Autoscaler*, as shown in Figure 3.2.

3.3.2 Autoscaling of Virtual FPGAs

The workload incurred by FPGA groups may significantly vary over time. Such variations may be caused by fluctuations in the workload experienced by the groups' client applications themselves, as well as the arrival or departure of client applications making use of each group. Increasing the number of applications using the same group raises the contention of the FPGAs in the group and consequently it affects the performance of individual applications. The resource utilization of each group is monitored by the Autoscaler which takes FPGAs from under-utilized groups and attaches them to groups with a higher utilization.

The aim of the autoscaling algorithm is to maximize the utilization of the infrastructure while improving the overall application completion time. Therefore, more applications get access to the FPGAs, which in turn is also beneficial for the cloud provider.

The Autoscaler periodically computes the average task runtime for each FPGA group (defined as the sum of the queuing time and the processing time). It then resizes the groups to minimize the average runtime across all groups which share the same FPGA-Server.

Note that resizing FPGA groups is an expensive operation, as it requires to reconfigure one or more FPGAs with a new hardware design. To avoid spending more time reconfiguring FPGAs than using them for processing incoming tasks, the periodicity (currently set by the adminstrators) at which group size is updated should be at least an order of magnitude greater than the reconfiguration time, which we have observed to be in the order of a few seconds in our appliances (see Section 4.5).

When the Autoscaler is triggered, it first computes the total queue length per client application and per FPGA group. A group with empty task queues is scaled down to the minimum working size of one FPGA, allowing other groups with higher workloads to utilize the idle FPGAs. When not enough FPGAs are available to satisfy all the demands, the Autoscaler balances the groups by assigning FPGAs in proportion to their workloads.

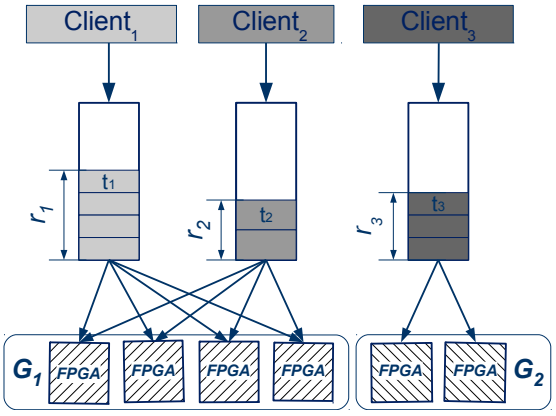


Figure 3.3: Two FPGA groups receiving tasks. Group G_1 contains four FPGAs and is servicing tasks from two client applications. Group G_2 contains two FPGAs and services tasks from one application. The runtime of all outstanding tasks is r_{Gi} and the task time for each client application is t_i.

Figure 3.3 shows the metrics of interest in FPGA group scaling. Tasks from applications can have a different runtime t_i. We define T as the total runtime of all tasks in an application's queue Q corresponding to a group G:

$$T = t \times size(Q) \tag{3.1}$$

The total processing time for all tasks submitted to a group G is the sum of the total processing time of all the queues in the group.

$$R_G = \sum T_i \tag{3.2}$$

The objective of the autoscaler is to balance the overall completion times for the n groups sharing FPGAs of the same FPGA-Server. This is achieved by minimizing the mean of absolute values of R_{G_i} differences:

$$minimize \quad \frac{\sum_{i=1}^{n}\sum_{j=1}^{n} \mid R_{G_i} - R_{G_j} \mid}{n^2} \tag{3.3}$$

Solving this optimization problem requires knowing the size of the queues using a group and the runtime t of every task. The information concerning the number of tasks in the queue can be retrieved periodically from the Orchestrator. On the other hand, the runtime of a task is unknown in the system. In order to determine it we rely on the periodical measurements the Autoscaler is retrieving from the Orchestrator. These measurements express how many tasks of each type have been executed in the measuring interval. Aggregating this information for several measurements helps in building a linear equation system where the unknown variables are the runtimes of tasks corresponding to the queues connected to a FPGA group and the coefficients are the number of tasks executed in one measurement interval.

The resulting system for one group is the following:

Algorithm 1 Autoscaling Algorithm

Input: Groups $G = \{G_1, G_2, \ldots, G_n\}$
Output: Group sizes S_{new}
1: I = scaling interval (e.g. $10s$)
2: $S \leftarrow \{S_1, S_2, \ldots, S_n\}$ where S_i is the current size of $G_i \in G$
3: initialize $S_{new} = \{0, 0, \cdots, 0\}$
4: **for** $i = 1$ to $|G|$ **do**
5: $A \leftarrow \{A_1, A_2, \ldots, A_n\}$ number of executed tasks for each queue of group G_i in the last measurement interval I
6: $T \leftarrow \{T_1, T_2, \ldots, T_n\}$ total runtime estimates for A_i on G_i of size S_i using NNLS
7: $t \leftarrow \{T_j / S_i\}$ for $j \in \{1 \ldots n\}$ runtime estimate of T_j on single FPGA
8: $T_W \leftarrow$ tasks waiting to be executed on G_i
9: $R_i \leftarrow \sum T_W \times t$ required processing time for G_i
10: $S_{new} \leftarrow \{\frac{R_i \times \sum_{j=0}^{n} S_j}{\sum_{j=0}^{n} R_j}\}$ for $i \in \{1 \ldots n\}$
11: return S_{new}

$$\begin{cases} A_{11} \times t_1 + A_{12} \times t_2 + \cdots + A_{1n} \times t_n = I \\ A_{21} \times t_1 + A_{22} \times t_2 + \cdots + A_{2n} \times t_n = I \\ \quad \cdots \\ A_{m1} \times t_1 + A_{m2} \times t_2 + \cdots + A_{mn} \times t_n = I \end{cases} \tag{3.4}$$

where:
n = number of applications using the group;
m = number of past measurements of task execution;
I = measuring interval;
t_i = task runtime corresponding to queue i;
A_{ij} = number of executed tasks of type j in the interval i

Note that the Autoscaler builds a system for each group. In order to solve the system and calculate the runtimes t_i we apply a non-negative least-squares (NNLS) algorithm [137]: given an $m \times n$ matrix A and an m-vector I, the algorithm computes an n-vector T that solves the least squares problem $A \times T = I$, subject to $T \geq 0$.

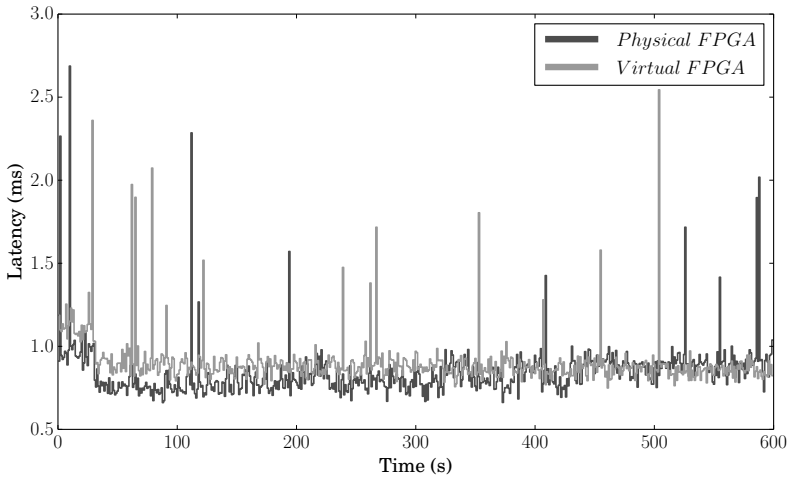
The solution of the linear system provides an approximation of the task runtimes which are then used in calculating the total processing time required of a group. Using this information the Autoscaler assigns an FPGA share proportional to the demand on each group. A simplified view of the scaling algorithm is presented in Algorithm 1.

This algorithm tries to treat all applications as fairly as possible in order to minimize the overall runtime of the current workload. An extension to this work might rather prioritize certain applications that are considered more valuable.

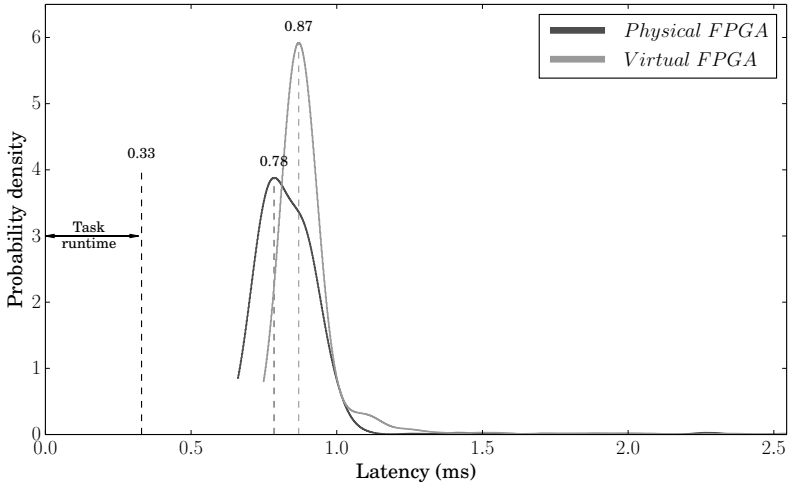
3.4 Evaluation

We now evaluate the performance of FPGA groups considering two perspectives: (i) the cloud provider's interest to maximize the utilization of its infrastructure and to accommodate more client applications in order to increase revenue; and (ii) client applications aiming to minimize the runtime and cost of the jobs by using elastic virtual FPGAs.

Our experiments are based on a single FPGA-Server equipped with eight FPGAs and two Infiniband interfaces. In our experiments, the average time to configure the FPGA when resizing a group is $3.78\,s \pm 0.13$, but the actual time for moving an FPGA from one



(a) Task execution delays



(b) Statistical distribution of task execution delays

Figure 3.4: FPGA virtualization overhead.

group to another is slightly larger because of the need to wait until currently-running tasks have finished executing. The Autoscaler is configured to re-evaluate the FPGA group sizes every 10 seconds. An additional server machine submits tasks to the FPGAs.

3.4.1 Virtualization Overhead

We first evaluate the performance overhead due to virtualization. To do this, we submit a simple task workload to a single FPGA. We chose the workload so that it does not overload the FPGAs: the client issues one task requiring 0.33 ms every second. We compare the task execution latency when the application addresses a single non-virtualized FPGA (without FPGA groups) and a virtualized FPGA group composed of a single FPGA.

The results are shown in Figure 3.4. We can see that, even in the fastest case, the client-side task submission and network transfers between the client machine and the FPGA add a total execution latency in the order of 0.45 ms. This is the reason why FPGA programmers typically submit large batches of task execution requests together:

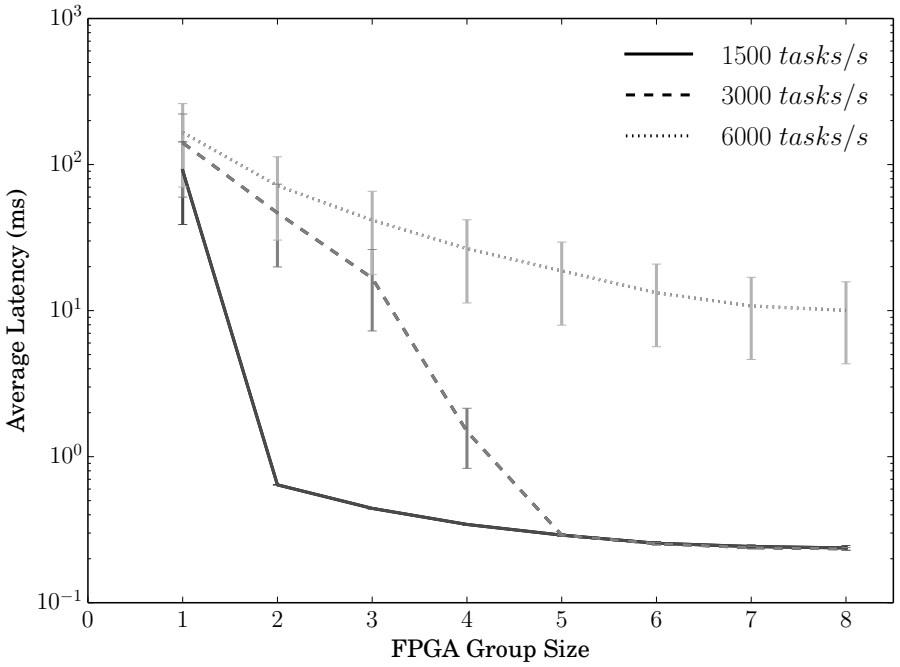


Figure 3.5: Effects of FPGA group elasticity.

this incurs the network overhead only once for the whole batch rather than once for every task.

We can also see the performance overhead due to virtualization: the latency of tasks submitted to the virtual FPGA is on average 0.09 ms greater than when submitted to the physical FPGA. This difference is due to the additional queues introduced by FPGA groups. We can however note that virtualization does not increase the variability of execution latencies compared to a non-virtualized scenario.

3.4.2 FPGA Group Elasticity

We now turn to evaluating the effectiveness of FGPA group elasticity to handle large task execution workloads. Here, we issued a constant load of 1500 tasks/second, 3000 tasks/second or 6000 tasks/second to FPGA groups of various sizes. Each task has an execution time on the FPGA of 1 ms. Each workload lasted 300 seconds.

Figure 3.5 shows the effect of varying the FPGA group sizes when handling these workloads. The smaller workload (1500 tasks/second) can in principle be handled using only two FPGAs. We can see that, when using only one FPGA, task execution requests pile up in the queues. As a result, execution latencies are very high, and also show very high variability. However, with a group of size 2, the task execution latency decreases to roughly 0.64 ms, with almost no variability. When increasing the group size further, the average task execution latency decreases even further down to 0.23 ms.

We can observe similar behavior from the more challenging workloads: whenever the FPGA group is too small to handle the incoming stream of task execution requests, the task execution latency is very high, with very large standard deviation. As soon as the group becomes sufficiently large to handle the tasks, latencies decrease drastically with very low standard deviation.

This demonstrates that FPGA groups can actually aggregate the processing capacity of multiple physical FPGAs. Varying the group size effectively allows one to control the

capacity of the virtual FPGA, while keeping these operations totally invisible for the client application.

3.4.3 FPGA Group Autoscaling

We now turn to evaluating the FPGA group autoscaling algorithm. To this extent, we defined three synthetic applications that can be executed using FPGA groups which compete for resource usage:

Application A consists of tasks which require 0.33 ms of execution time. An example of such a task is a Fast Fourier Transform operated over a buffer of 262,144 double-precision values. To maximize the execution efficiency, tasks are submitted in batches where each batch contains 300 tasks. For simplicity, we implemented Application A as a simple sleep function.

Application B consists of longer tasks which require 1 ms of execution time. Here as well, tasks are sent by batches of 300 tasks.

Application C consists of long-running tasks which require 100 ms of execution time. Tasks are sent by batches of 3 tasks.

Figures 3.6 and 3.7 compare the behavior and performance of static-size and autoscaled FPGA groups based on two workload scenarios. For both figures the static-size groups were provisioned with two FPGAs for application A, four for application B, and two for application C. We chose these numbers such that the group sizes would be proportional to the total workload execution time of each application.

Figure 3.6 shows a scenario with a relatively light workload where the FPGA server has more than enough resources to process all incoming requests, despite workload variations in applications A, B and C (depicted in the top part of the figure). In this situation, the static FPGA allocation reaches an average resource utilization of 32%, while keeping low individual batch execution latencies (on average 0.43 s per batch).

In this scenario where no group resizing is necessary, the autoscaling system incurs the overhead of FPGA group reconfiguration while having nothing to gain from this operation. It would be very easy to adjust Algorithm 1 so that it refrains from any resizing operation when all current queue lengths are small enough. We however decided to let the Autoscaler overreact to workload variations to highlight the overhead of FPGA group reconfiguration.

The bottom half of Figure 3.6 shows the resizing decisions taken by the Autoscaler, as well as the consequences of these decisions on resource utilization and batch execution latency. We clearly see a spike in execution latency each time an significant resizing operation takes place. The reason for this is threefold: first, upon any reconfiguration there is a period of several seconds during which less FPGAs are available to process batch execution requests; second, the FPGA group which gets shrunk may not have sufficient resources to process its workload, so requests queue up until the next resizing operation where the system has an opportunity to increase the group size again; and, finally, in this particular experiment these unavoidable costs are not compensated by any benefits from the useless rescaling operation.

Figure 3.7 shows the same system in a scenario with twice as much workload as previously. Here, the static FPGA groups cannot efficiently process all incoming requests. In particular, application A does not have enough resources to handle its peak workload, so batch execution latencies for application A grow substantially, up to 22 s per batch. On the other hand, during peak activity of application A, application B underutilizes its

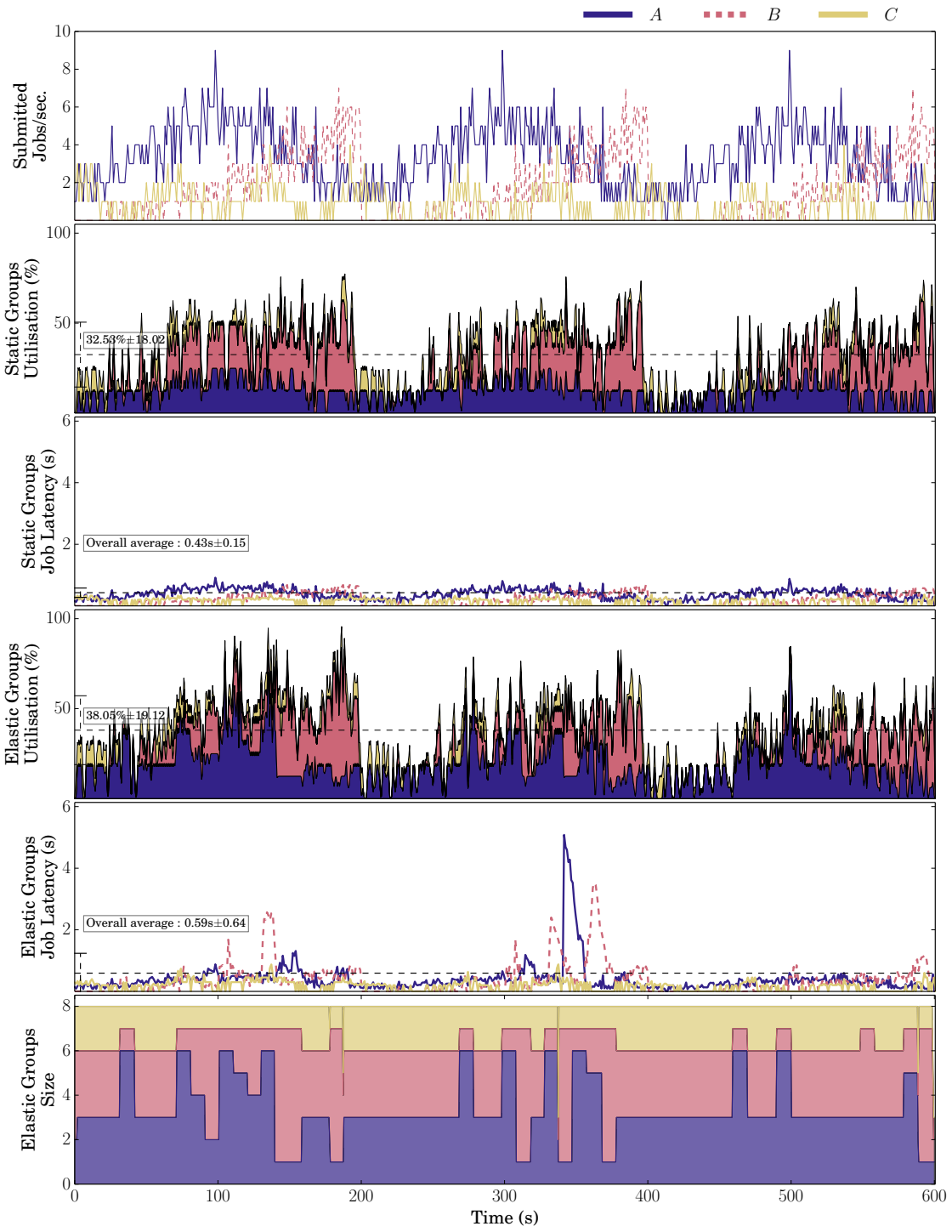


Figure 3.6: FPGA group autoscaling (low workload).

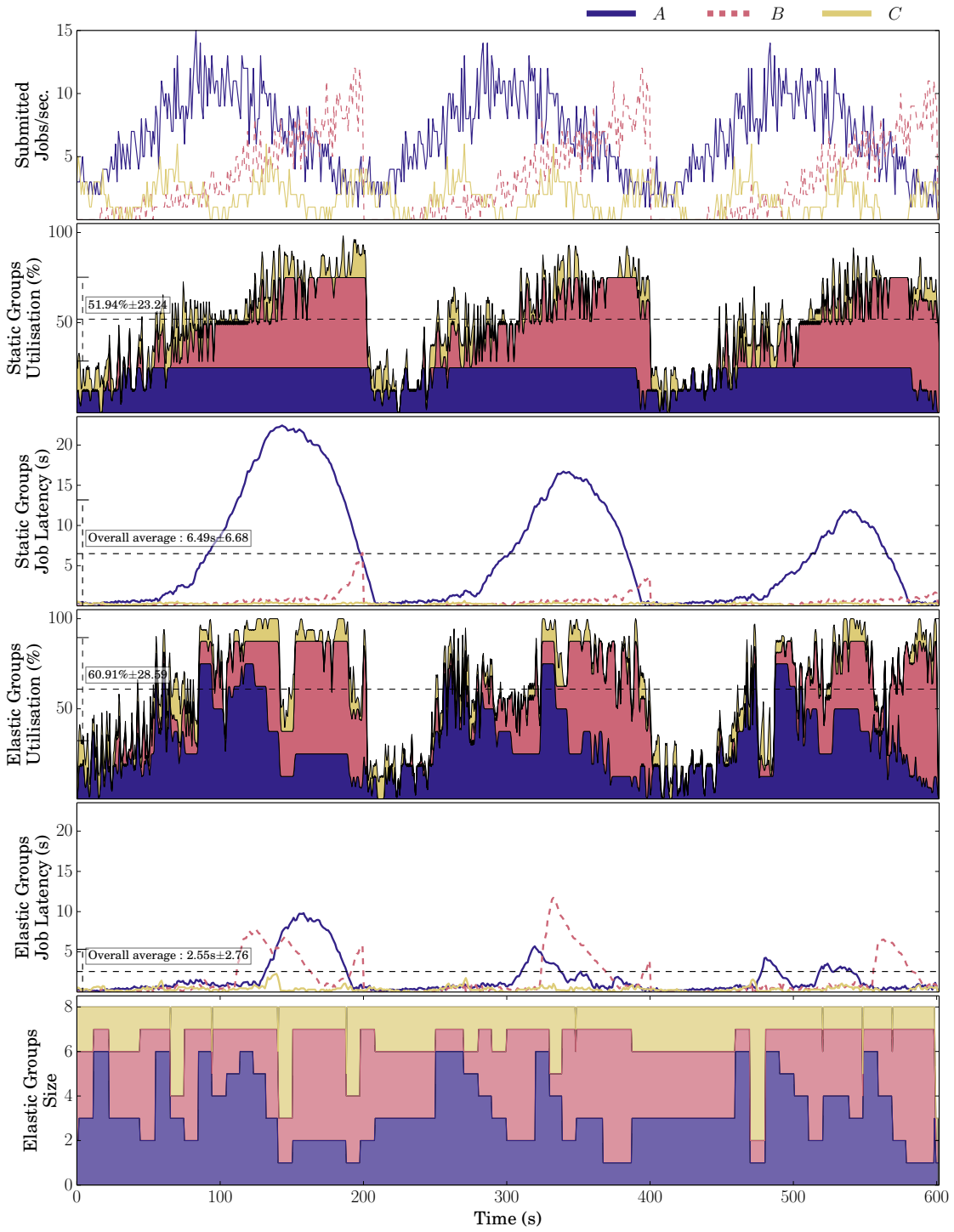


Figure 3.7: FPGA group autoscaling (high workload).

own resources so the obvious solution is to temporarily reassign FPGAs from application B to application A. This is exactly what the Autoscaler does in this case. As a result, application A now shows much better performance, at the expense of slightly slowing down application B.

Another interesting aspect can be seen in Figure 3.7 between times 150 s and 200 s. Here, both applications A and B experience high workloads, which once put together exceed the total processing capacity of the system. In this situation, the Autoscaler maintains a reasonable fairness between all applications.

These experiments highlight the interest of dynamically resizing FPGA groups in situations where multiple applications compete for limited resources. On the one hand, the cloud provider's interest is to maximize the utilization of its infrastructure and to accommodate more client-applications in order to increase revenue. We can see that autoscaling increases the average resource utilization from 52% to 61%, and manages to handle all three workloads simultaneously using only one FPGA server. Using static groups only, the administrator of application A would probably rather decide to reserve additional FPGAs to handle high-workload scenarios, thereby reducing the average resource utilization.

On the other hand, cloud tenants aim to minimize the runtime of their batches, while reducing their consumption of expensive FPGA resources. The Autoscaler reduces the average batch execution latency by 61%, from 6.49 s to 2.55 s while using the same total number of resources, thanks to a better usage of these resources.

3.5 Conclusion

FPGAs have the potential of complementing the current landscape of accelerator devices for high-performance computations. When applied to suitable problems, they deliver excellent performance, computational density and energy consumption. However, without virtualization techniques, FPGAs remain limited to scenarios where a single entire device is attached to one client application. Maximizing FPGA utilization in such conditions is not always easy.

As the first contribution of this thesis, we proposed to virtualize FPGAs as a way to increase their usage flexibility. With FPGA groups, one can create virtual FPGAs which aggregate the processing capacity of one or more physical FPGAs. FPGA groups can be resized manually or automatically to maximize device utilization and further improve user's experience. FPGA groups can be used by a single tenant, or shared between multiple tenants. However, to isolate tenants from each other, this second scenario should be limited to stateless FPGA designs, i.e., designs that do not keep state from one job to another.

Enjoying the performance benefits of virtual FPGAs using a pay-as-you-go model, and without having to master their complex programming model, would arguably help democratize FPGA-based high performance in the cloud.

Chapter 4

Performance Modelling

Contents

Parts of this chapter were previously reported in [138].

Having access to a large variety of resource configurations allows one to select the exact amount and type of resources an application needs to run according to specific objectives. However, selecting the right resource configurations for specific applications and user requirements is a difficult task because of the very large space of possible resource configurations that heterogeneous clouds provide. Moreover, users often find it very hard to accurately estimate the resource requirements of complex applications [12].

Cloud platforms offer very little support to help users choose appropriate resources. Autoscaling systems such as Amazon AutoScale allow users to define their own rules, but without guiding them in writing efficient rules. Other platforms designed for online applications usually choose a single instance type and vary only the number of instances when the load changes. Such techniques are however not well suited for batch applications which often cannot adjust their choice of resources during execution.

Selecting the "right" set of resources for a batch application requires a fine-grained understanding of the relationship between a set of resources and the performance that the application will have using these resources. This is challenging because the space of all possible resource configurations one may choose from is usually extremely large. For example, Amazon EC2 currently proposes 40 different instance types. An application requiring just five nodes must therefore in principle choose one out of $40^5 = 102,400,000$ possible configurations.

In a cloud computing environment, achieving the lowest possible execution time for a batch application is not always desirable, as the fastest execution often requires using expensive resources. Depending on the circumstances, a user may want to choose the fastest configuration, the cheapest, or any configuration implementing a trade-off between these two extremes.

We propose to automate the choice of resources that should be assigned to arbitrary non-interactive applications that get executed repeatedly. Upon the first few executions of the application, the system tries a different resource configuration for each execution. It then uses the resulting execution times and costs to build a custom performance model for the concerned application. Once this model has been built, cloud users can simply specify the execution time or the financial cost they are ready to tolerate for each execution, and let the system automatically choose the resource configuration which best satisfies this constraint.

The main assumption facilitating this resource selection is that the execution time and cost are deterministic with respect to a given resource configuration and input size. Although slightly limiting, this assumption is met in many HPC applications which are optimized to perform high-volume, repetitive tasks where successive executions process inputs with the same size and runtime behaviour. This is the case in particular of the two real-world applications we use in our evaluations (one in the domain of oil exploration, the other in the domain of high-performance database maintenance). Furthermore, we propose to address the question of applications which process inputs of varying sizes by studying the correlation between the execution times of the different input sizes on a resource configuration.

Allowing the automatic selection of computing resources for arbitrary batch applications requires one to address a number of challenges. First, we need to describe arbitrary applications in such a way that a generic application manager can automate the choice of resources that the application may use. Second, we need efficient search methods to quickly identify the resource configurations that should be tested. Finally, we need to generate performance models that easily allow one to choose resources according to the performance/cost expectations of the users.

We address these challenges in the following sections as follows. In Section 4.1 we discuss the related work. In Section 4.2, we show how to describe different types of applications in an abstract manner such that they are easily handled by a generic system. This is followed by a description of the general architecture of such a system. In Section 4.3 we present our general profiling approach for automating the selection of resources in heterogeneous clouds. In Section 4.4 we propose three complementary profiling methodologies to automate the resource selection: *Blackbox, Blackbox+Whitebox and Extrapolated* profiling. These methodologies aim to identify good performance-cost trade-offs for executing applications. We evaluate how they perform in Section 4.5. Finally, in Section 4.6 we conclude this chapter which represents the second contribution of this thesis.

4.1 State of the Art

Many efforts have been dedicated to performance modeling in HPC or cloud environments. HPC aims at executing applications as efficiently as possible, in order to optimize a variety of metrics such as the makespan of a set of jobs, high throughput, and low average stretch. In consequence, performance modeling has always been a priority concern in this area [139]. HPC environments usually consist of large supercomputers where users have direct access to the bare-metal machine. This is useful for getting the best possible performance, and it also helps performance modeling because the computing resources often expose their detailed hardware configuration.

HPC modeling techniques can be classified into analytical predictive methods, code analysis and profiling [139, 140]. Analytical methods require developers to provide a model of their application. They are potentially very accurate, but building new analytical models is labor-intensive and difficult to automate. Moreover, user estimates of application runtimes are often highly inaccurate [12]. Code analysis automates this process, but it usually restricts itself to coarse-grained decisions such as the choice of the best acceleration device for optimizing performance [114]. Finally, Ipek et al. [141] proposed an automatic profiling-based approach which consists of training artificial neural networks using performance results from application executions on a target HPC platform. However, they use this technique to study the impact of varying the input size on a fixed configuration of resources. Moreover, it requires a very large number of executions in order to predict application performance within 5%-7% errors. The work described in this chapter is complementary to this one by studying the impact of varying the resource configuration in order to provide different performance-cost trade-offs. Furthermore, we also make use of the performance correlation between different input sizes to predict the behaviour on a specific configuration.

Web Applications

In cloud environments, performance modeling was studied for a number of specific types of applications such as Web applications. Besides the numerous techniques which dynamically vary the number of identically-configured resources to follow the request workload, one can study historical traces in order to define horizontal and vertical scaling rules to handle various types of workloads.

Vasic et al. [116] propose to reuse optimized VM allocations in order to meet SLOs more efficiently when workload changes, and lower the cost for the cloud provider. This work relies on the assumption that changes in the workload of network services follow a repeating time-based pattern. They cache the results of past VM allocation decisions

in order to quickly reuse them when facing similar workloads. This paper reports cost savings of up to 60% when scaling horizontally the resource configuration. They also address vertical scaling limited to the use of only two distinct types of VM instances.

Watson et al. [142] propose a probabilistic approach for modelling the performance of three-tier Web applications by exploiting the relationship between CPU allocation and contention and application response time. The generated probabilistic model is used to estimate response time distributions with a less than 6% mean absolute error. Their study is however focused on CPU resources only while claiming the methodology to be generally applicable to other non-CPU virtualized resources.

Jiang et al. [143] address the problem of dynamically provisioning heterogeneous resources for multi-tier web applications. They model the web application as a queueing network where each tier represents a separate queue. Then, they benchmark the performance of newly allocated VM instances and generate a performance profile. This profile is used to select the tier the resource should be assigned to or to adjust load-balancing to get homogeneous performance. This work focuses on efficiently exploiting newly allocated heterogeneous resources in order to achieve a target response time. However, they do not specify on which basis they select the new instance type to be provisioned which is the actual focus of our work.

Fernandez et al. [118] propose an autoscaling system implementing different scaling policies for web applications where one may dynamically choose the best resource type based on short-term traffic predictions. They enable users to make performance-cost trade-offs through a number of predefined QoS levels. Each QoS level is expressed using a metal classification scheme (gold, silver, and bronze users) where each class is backed up by a scaling strategy operating over a number of different instance types. Therefore, the heterogeneity of the resource configuration to be used is limited by the scaling strategy backing up each QoS class. This makes the exploitation of a high diversity of resource configurations difficult as it would require one to define many different QoS classes to express the cost-performance trade-offs that different combinations of resources provide.

Bags-of-Tasks Applications

Performance modeling has been also addressed for specific types of scientific applications. For bags-of-tasks applications, Oprescu et al. [119] propose a scheduling method for large bags of tasks to be executed on cloud resources with different CPU performance and cost. They rely on the fact that one can observe the statistical distribution of task execution times, and automatically derive task scheduling strategies to execute the bag under certain time and budget constraints. They address resource heterogeneity through the use of machine clusters of different instance types. Contrary to this work, we do not make assumptions on the distribution of execution times of individual tasks and consider a single task is to be executed under user constraints. Therefore, the problem we address becomes a resource selection problem rather than a scheduling one.

MapReduce Applications

Similar work has been done for MapReduce applications. Verma et al. [120] propose a framework consisting of a MapReduce job profiler, a MapReduce job model and a scheduler. Its main purpose is to allocate the appropriate amount of resources to execute the MapReduce job within a required deadline. Tian et al. [121] propose a regression-based model to predict the performance of MapReduce jobs. The model parameters are extracted from test runs. Similarly to [120], the model allows the selection of the appropriate amount

of resources to execute MapReduce jobs under different time or cost constraints. However, these works focus mainly on determining the amount of identical resources to be used in order to perform according with a imposed budget or deadline.

Arbitrary Applications

For batch applications which do not fit the MapReduce or the bags-of-tasks models, the only solution currently proposed by Amazon EC2 is to empirically try a variety of instance types [5], evaluate the most important performance metrics for their application, and, choose the instance type which works best.

> *"Because you can launch and terminate instances as desired, profiling and load testing across a variety of instance types is simple and cost effective. Unlike a traditional environment where you are locked in to a particular hardware configuration for an extended period of time, you can easily change instance types as your needs change. You can even profile multiple instance types as part of your Continuous Integration process and use a different set of instance types for each minor release."*
>
> (Jeff Barr. *Choosing the Right EC2 Instance Type for Your Application*)

This is a clear incitation for cloud tenants to profile their applications using a variety of instance types. However, Amazon does not give any hint about the way such profiling should be organized. As discussed earlier, this is not a trivial exercise.

The purpose of this chapter is to define best practices and automated tools to facilitate the profiling process for arbitrary cloud tenants and applications.

4.2 Handling Arbitrary Applications

Batch applications can be extremely different from each other. They may require different sets of libraries, parameters, input/output files, etc. This is also true for the selection of resources they may execute on: some applications expect specific type of hardware, or constrain the number of machines they use. For example, some applications may require the number of machines to be a power of two or they may require mutually dependent CPU numbers and memory sizes. In order to handle such applications, a generic application manager must provide a flexible way to describe any application with unique characteristics and resource requirements.

To address this issue, we rely on two specification files. The *Application Manifest* describes the application's structure and constraints about the resource types it needs. It is typically written by the application developer. On the other hand, the *Service-Level Objective* describes a user's expectations about acceptable execution times or costs, and is thus typically written by the application user.

4.2.1 Describing Arbitrary Applications with Application Manifests

A manifest file is the specification of an application's structure and the type of resources it depends on to execute correctly. It describes the application input parameters, the types of resources supported by the application, the constraints between input parameters and resources, and the way the application can be deployed and executed on each resource type. Figure 4.1 shows the most important parts of a manifest file represented by the description

```
ApplicationName: HelloWorld

Parameters {
   Parameter_1 (
      Name:    Parameter_1
      Type:    Integer
      Values:  {v_1,v_2,...,v_n}
      Default: v_1
   ), ...
}

Resources {
  Resource_1 (
    Type:     Virtual Machine
    Number:   1
    Configuration: {
      Cores:  {1..16}
      Memory: {2,4,6,8,12,16,24,48,64,96,124}
    }
    Role:     Master
  ), ...
}
```

Figure 4.1: Application Manifest Example

```
ManifestUrl:          www.cloud.org/../manifest
ExecutionParameters: {v_1, ...}
Objective:           {
                        Constraints: [ cost <= 100 ]
                        Optimize: execution_time
                     }
```

Figure 4.2: Service-Level Objective Example

of input parameters and resource requirements. A complete version of a manifest template file is also available in Appendix A.

The manifest provides the information a generic application manager needs in order to deploy an application and execute it on a resource configuration. Thanks to this, the application manager can be application- and resource-agnostic.

Input Parameters

The important factors influencing the behaviour of an application on a resource configuration can be expressed as input parameters. These imput parameters may impose different execution paths inside the application structure or may point to different datasets the application should process. For example, an application performing image compression may get as input parameters the path to the data volume it has to proces, the compression algorithm to apply, the quality of the compression etc.

In order to model the performance of such an application, it is important to identify these performance-critical input parameters. This has to be done by an application expert (typically its developer). Each parameter is modeled as a *(type, set-of-acceptable-values, default-value)* tuple.

Resources

The Application Manifest describes the types of resources the application needs, with their number, configuration and role. For any type of required resource, a manifest may specify either a fixed value, or a set of acceptable values to choose from. For instance, a sequential application needs a single VM in order to run while a distributed application may make use of a variable number of VMs. The *Configuration* attribute describes the properties that resources may have. A computing resource may thus for example specify a number of cores and memory size, while a storage resource may describe properties such as the disk size and supported IOPS (Input/Output Operations Per Second).

Depending on supported functionalities of the underlying IaaS cloud, the manifest may also specify constraints such as the network capacity between different resources. This feature is, for example, used in the HARNESS cloud to allocate resources in locations which match these requirements (see Chapter 5). However, as most of the IaaS clouds do not provide yet support to reserve virtual link capacity, we focus mostly here on the configuration of the virtual machine instances.

Finally, one may assign a *Role* to a resource. This allows us to describe applications with multiple components potentially having specific requirements. For example, a master/slave application may separately describe *Master* and *Slave* resources.

4.2.2 Specifying User's Expectations with Service-Level Objectives

A Service-Level Objective (SLO) file describes a user's request for executing an application. As shown in Figure 4.2, it contains a reference to the application's manifest, the list of parameters that should be passed to the application, and the user's expectations in terms of execution time and/or cost.

Users' expectations can take two different forms. In Figure 4.2 the user imposes a maximum execution cost, with a secondary goal to execute the application as fast as possible. Alternatively one could rather impose a maximum execution time, while trying to spend as little as possible under this constraint.

The application manager uses the information provided by the SLO file to filter and select the appropriate resource configuration for the execution of the application.

4.2.3 System Architecture

Our system architecture is depicted in Figure 4.3. A user triggers an execution of the application by submitting an SLO and a manifest file to the application manager. The application manager is a generic element which does not need any application-specific information besides the application manifest and the SLO. It is in charge of choosing and provisioning resources, deploying of the application onto these resources, launching the execution of the application, and of measuring the execution time and implied costs.

Initially, the application manager has no knowledge about the types of resources it should choose for a newly-submitted application. After loading the manifest and SLOs files, in case no performance model is specified, the *Controller* forwards the application

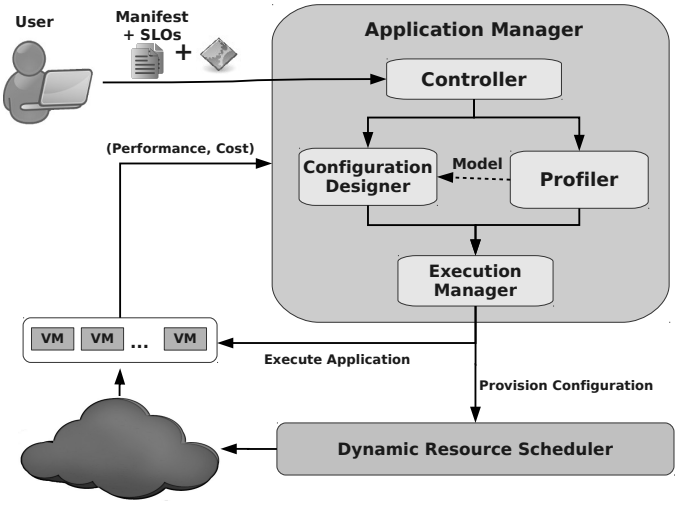


Figure 4.3: System overview

to the *Profiler* which executes the application repeatedly using a different resource configuration every time. This profiling process continues until either a predefined number of executions has been performed or a profiling budget has been exceeded.

The result of these executions is used to build a performance model which is sent to the *Configuration Designer*. If a performance model was already specified in the manifest, the *Controller* skips the *Profiler* and sends the application and its model directly to the *Configuration Designer*. Based on this model, the *Configuration Designer* then selects a configuration that satisfies the SLOs and launches the execution.

In both cases, the execution is handled by the *Execution Manager* which provisions the configuration through a *Dynamic Resource Scheduler* and finally executes the application on it.

Each time an application execution is triggered, the system monitors its total execution time and cost, and learns the relation between the choices made for this execution and the observed result:

$$(ExecTime, Cost) = Run_{app}(Parameters, Resources)$$

The results generated after several executions with various resource configurations can be plotted as shown in Figure 4.4. In this figure, each point represents the execution time and cost that are incurred by one particular resource configuration. The figure shows the result of an exhaustive exploration of a search space with 176 possible configurations. In a more challenging scenario the number of configurations would be much greater, and this type of exhaustive exploration would be practically unfeasible.

4.2.4 Cloud Model

In this work we assume a cloud infrastructure capable of dynamically creating virtual resources based on a fine-grained description of its properties such as number of cores, memory, core frequency etc. Such fine-grained configuration specifications are for example supported by the OCCI standard [87].

Note that this model creates considerably larger configuration search spaces than traditional clouds which offer a handful of fixed instance types. It is therefore trivial to adapt this work to EC2-like clouds, by simply disabling resource configurations which do not match one of the predefined instance types.

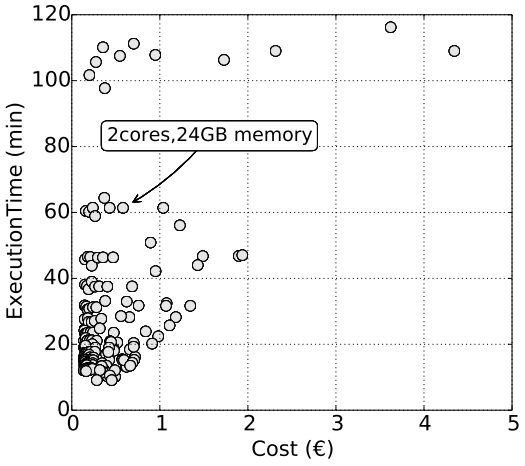


Figure 4.4: Exhaustive exploration of the resource configuration space

We also assume that the cloud can dynamically generate a financial cost for any resource type that can be requested. Our system makes no assumption about the form that this cost model takes, and can therefore adjust to the specificities of any cloud's pricing model. In our experiments, however, we use a pricing model which charges resources on a per-minute basis according to a linear model:

$$Cost(R) = R_1 * Cost_{unit_{R1}} + R_2 * Cost_{unit_{R2}} + ...$$

As an example, for a virtual machine (VM) with the configuration { N Cores, M GB of Memory }, the cost per time unit may be calculated as:

$$Cost(VM) = N * Cost_{1core} + M * Cost_{1GB}$$

where $Cost_{1core}$ may vary based on the frequency, amount of cache, etc.

4.3 Profiling Principles

The main issue when building the performance model of an application is that the space of all possible configurations is usually much too large to allow an exhaustive exploration. We therefore need to carefully choose which configurations should be tested, such that we identify the optimal configurations as quickly as possible.

4.3.1 Assumptions

The input of an application and the resource configuration it runs on are the main factors influencing its execution. However, as both can have a large variety of values, exploring the search space created by aggregating them would be extremely difficult. Therefore, we chose to focus primarily on modelling applications whose performance is independent from the input. This allows us to model the impact of resource variability on the their performance. For example, an application can be executed periodically on inputs with different content but of the same size. This is the case of the RTM application that we target for evaluation.

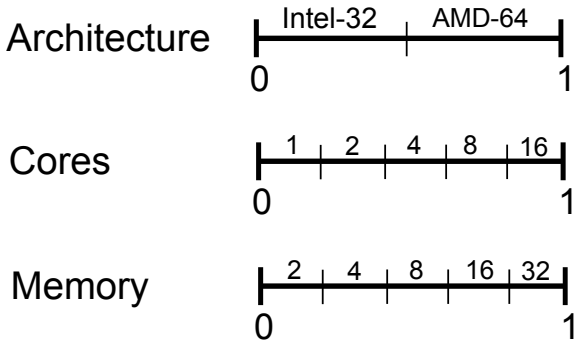


Figure 4.5: Uniform distribution on the continuous interval $[0, 1]$ for the sets of discrete values that VM properties such as CPU architecture, number of CPU cores and memory size may take.

However, in Section 4.4.3, we relax this initial assumption and study how to model applications whose performance may be expressed as a function of the input data size and the resource configuration it runs on.

4.3.2 Search Space

The search space of resource configurations to explore for an application is generated using the application manifest. Each resource parameter which should be chosen by the platform constitutes one dimension of the space. The number of possible configurations therefore increases exponentially as new dimensions are added, an issue often referred to as the curse of dimensionality.

In the example from Figure 4.1, the search space of the application has 2 dimensions (corresponding to 16 possible numbers of cores and 11 possible memory sizes). This creates a total of $16 \times 11 = 176$ possible configurations. Within these 176 configurations, only a subset of them may offer interesting trade-offs between performance and cost.

4.3.3 Mapping Discrete Parameters

As shown in the manifest presented in the Figure 4.1, resource parameters take values from a discrete set. This ensures an efficient exploration of the search space. For some resource properties, assigning values that are very close from each other (e.g., by exploring available memory in steps of 1 MB) would provide no benefit as the performance would be largely the same.

While describing resource parameters using discrete values works well in many cases, exploring the search space defined by "truly discrete" dimensions is more difficult. For example, one dimension may represent the choice of a particular CPU architecture among a list (Intel-32, AMD-64, etc.). To tackle this problem and to be able to simplify the search process, we map all the dimensions of the search space on the continuous interval $[0, 1]$. For each dimension, we generate a step unit function by placing an equal weight on each discrete value as observed in Figure 4.5. This mapping allows us to apply agnostically different algorithms for function minimization without having to worry about the type of the parameters.

An important aspect to be noticed is that this mapping allows us to work also on configurations spaces consisting of predefined instance types as the ones offered by Amazon

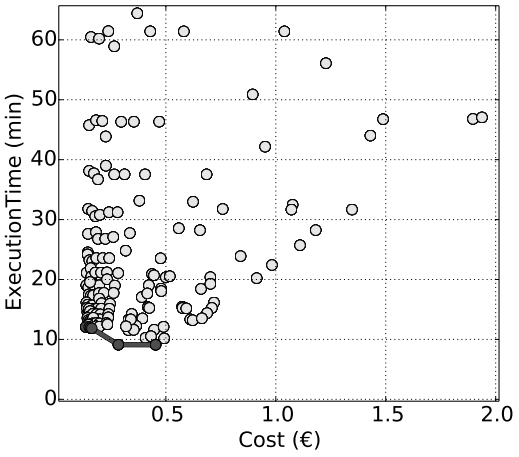


Figure 4.6: Resource configuration space of a real application. The set of Pareto-optimal configurations is shown in black.

EC2. In this scenario, the search space would contain only one dimension where each predefined instance type is represented by a discrete value.

4.3.4 Identifying Optimal Configurations

It is interesting to notice that not all configurations provide interesting properties. Regardless of the application, a user is always interested in minimizing the execution time, the financial cost, or in finding a sweet spot between these two[2].

Figure 4.6 presents the search space of a real application used later in the evaluation. Configurations which appear at the top-right of the figure are both slow and expensive. Such configurations can be discarded as soon as we discover another configuration which is both faster and cheaper. The remaining configurations form the *Pareto frontier* of the explored search space. The figure highlights the set of Pareto-optimal points of this application: they all implement interesting trade-offs between performance and cost: points on the top-left represent inexpensive-but-slow configurations, while points on the bottom-right represent fast-but-expensive configurations.

The Pareto frontier (and the set of configurations leading to these points) forms the performance model that the application manager uses to choose configurations satisfying the user's SLOs. If an SLO imposes a maximum execution time, the system discards the Pareto configurations which are too slow, and selects the cheapest remaining one. Conversely, if the SLO imposes a maximum cost, it discards the Pareto configurations which are too expensive, and selects the fastest remaining one.

4.3.5 Profiling Policies

Profiling an application requires one to execute it a number of times in order to measure its performance and cost under various resource configurations. This process may be handled in two different ways, depending on the user's preferences:

[2]An interesting extension of this work would be to consider additional evaluation metrics such as energy consumption. This can be easily done as long as the relevant metrics are designed such that a lower value indicates a better evaluation.

1. The *offline approach* triggers artificial executions of the application whose only purpose is to generate a performance model. In this case, the output of executions is simply discarded.

2. The *online approach* opportunistically uses the first actual executions requested by the user to try various resource configurations and lazily build a performance model.

Choosing one of these approaches requires the user to make a simple trade-off. In offline profiling, the user will incur delays and costs of the profiling executions before a performance model has been built. On the other hand, all the subsequent executions will benefit from a complete performance model. In online profiling, the user should be aware of the fact that the first executions may not fulfill their SLO until a performance model has been built. On the other hand, the overall marginal cost and delays caused by the profiling will be reduced. We evaluate the impact of this choice in Section 4.5.1.

4.4 Profiling Methods

The purpose of the profiling process is to perform a search through the space of possible configurations and to quickly identify configurations which implement interesting performance/cost trade-offs, without having to explore every configuration from the search space. Notably, it does not only aim to find the fastest nor the cheapest configuration, but it also aims to identify as many configurations as possible which offer interesting trade-offs between these two extremes. However, because the search space is too large for us to test all the resource configurations, we need heuristics that identify promising resource configurations in a time- and cost-efficient manner.

We propose three profiling methods for modelling arbitrary applications in heterogeneous clouds without having prior knowledge about them. The first two methods are designed for applications whose performance is independent of their input while the last one addresses applications whose dependency between different input sizes is reflected in its performance on similar resource configurations.

- *Blackbox*: this generic method is input-independent and relies only on the performance and the cost of an execution to issue new resource configurations to test. Being application- and resource-agnostic, it automatically profiles arbitrary cloud applications written using any programming language or framework, and requiring any set of cloud resources. On the other hand, it may require a large number of iterations until good performance-cost trade-offs are discovered which makes the profiling process long running and costly. Moreover, when a failure occurs because of unsufficient amount of resources, it marks it as "bad" and may continue to test other similar configurations which are very likely to fail as well. This induces an additional cost and runtime overhead.

- *Blackbox+Whitebox*: this method is input-independent and aims to minimize the number of iterations performed by blackbox profiling in order to reach faster to good performance-cost trade-offs. It relies on the feedback from a "whitebox" plugin model to drive the search towards better resource configurations. A whitebox plugin is a specialized component that may provide useful recommendations to the blackbox model in order to optimize the selection of resource configurations and avoid bad configurations. As a consequence of minimizing the number of iterations and avoiding failures, it reduces the total runtime and cost of the profiling process.

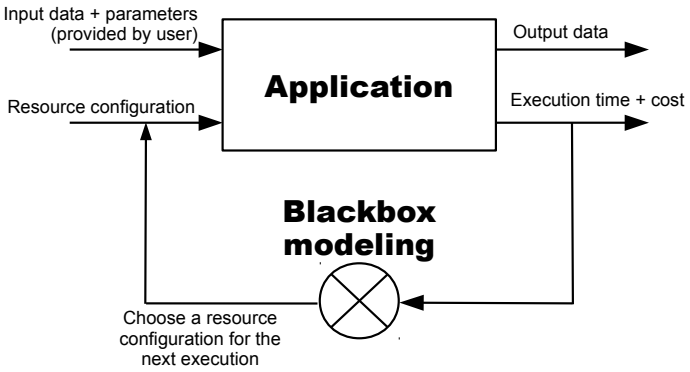


Figure 4.7: Blackbox profiling.

- *Extrapolated profiling*: this input-dependent method relies on application input size dependency to find good configurations. HPC applications regularly run for a long period of time. This makes Blackbox and Blackbox+Whitebox profiling to have a very long runtime and, consequently, induce a high cost overhead. However, in many cases, it may not be necessary to let the application run for hours to produce good insights in its performance profile. Some good insights may sometimes be obtained using smaller inputs which produce shorter executions. Extrapolated profiling aims to radically reduce the profiling time and cost overhead for long running applications by making use of the correlation between a short-running benchmarking input size and the long-running production input size that an application is executed with. It explores the search space using the benchmarking input size which is less costly and time-consuming and uses the correlation to predict the performance of the applications on the production input size. Besides reducing the total cost and runtime of the profiling, it also may provide acceptable performance-cost trade-offs although they may not be better than the ones identified with Blackbox and Blackbox+Whitebox profiling.

We discuss each profiling methodologies in the following sections.

4.4.1 Blackbox profiling

The first input-independent profiling method we propose is entitled "Blackbox profiling" and implements application- and resource-agnostic search strategies to identify resource configurations providing good performance-cost trade-offs. Figure 4.7 presents the basic workflow of this method. Its main feature is being application- and resource-agnostic and, therefore, it operates effectively on arbitrary applications and heterogeneous clouds.

We define two generic search algorithms that can be implemented by a Blackbox profiling process to explore a configuration space without any specific knowledge besides the execution time and cost:

Uniform Search

This algorithm explores stepwise points in the resource search space to select a configuration for the profiling process. As shown in Algorithm 2, the application is executed for all combinations of stepwise resource values (lines 2-5). When executed with low exploration steps, it may cover very well the configuration search space and eventually identify very good performance-cost trade-offs. On the other hand, it may waste time exploring large

Algorithm 2 Uniform Search

Input: Application A, Resources $R = \{R_1, R_2, ..., R_n\}$
Output: Set of configurations, their execution time and cost $S_{r,t,c}$

```
1: S_{r,t,c} ← ∅
2: for r_1 = min_1 to max_1 by step_1 do
3:     for r_2 = min_2 to max_2 by step_2 do
4:         ...
5:         for r_n = min_n to max_n by step_n do
6:             r ← {r_1, r_2, ..., r_n}
7:             (t, c) ← execution time and cost of running A on r
8:             S_{r,t,c} ← S_{r,t,c} ∪ {(r, t, c)}
9:         ...
```

areas which are unlikely to deliver interesting performance/cost trade-offs. Therefore, low exploration step values result in high complexity, while using high step values (to reduce the complexity) may skip relevant configurations.

Simulated Annealing (SA)

Simulated Annealing is a well-known generic algorithm for global optimization problems [144]. It initially tries a wide variety of configurations, then gradually focuses its search around configurations which were already found to be interesting. It relies on a global time-varying parameter called the temperature to decrease the probability to accept bad configurations. Accepting bad configurations is fundamental in avoiding to get stuck in local minima.

Algorithm 3 shows the SA algorithm applied to the resource configurations. The algorithm starts with a random resource configuration (line 1), and explores new configurations in the neighborhood of the current configuration (line 5). The *neighbor()* function determines a new configuration by drawing random values around the current configuration using a normal distribution determined by the temperature. $rate_{learn}$ is a scale constant for adjusting updates and *upper* and *lower* are the parameter r's interval bounds. The temperature decreases gradually (line 10), which means that the algorithm accepts new configurations to explore with slowly decreasing probability (lines 8-10). Due to its convergence to optimal solution in a fixed amount of time, simulated annealing quickly explores the search space, focusing most of its efforts in the "interesting" parts of the search space.

In order for the algorithm to explore configurations that are both cost- and performance-efficient, we evaluate each configuration based on the product between the cost and the execution time it generates:

$$utility = ExecTime \times Cost$$

The minimization of the product requires the minimization of at least one of them, and preferably both. Using this utility function the algorithm explores the entire Pareto frontier, instead of focusing on optimizing only the execution time or the cost.

Contrary to Uniform Search, this algorithm avoids exploring areas with uninteresting performance-cost trade-offs and reaches good configurations considerably faster.

It is important to remark there are many other optimization algorithms which might be applied similarly such as ant colony optimization, particle swarm optimization and genetic algorithms. It would be an interesting research direction to study and evaluate which optimization algorithm provides the best results when applied to resource selection.

Algorithm 3 SA

Input: Application A, Resources R, Temperatures $T_{cooling}$ and $T_{current}$
Output: Set of configurations, their execution time and cost $S_{r,t,c}$

$r \leftarrow \{r_1, r_2, ..., r_n\}$, r_i is random value of resource $R_i \in R$
$(t, c) \leftarrow$ execution time and cost of running A with resource configuration r
$S_{r,t,c} \leftarrow \{(r, t, c)\}$
while $T_{current} > T_{cooling}$ **do**
    $r_{new} \leftarrow neighbor(r, T_{current})$
    $(t_{new}, c_{new}) \leftarrow$ execution time and cost of running A with resource configuration r_{new}
    $S_{r,t,c} \leftarrow S_{r,t,c} \cup \{(r_{new}, t_{new}, c_{new})\}$
    if $Probability_{Acceptance}((t, c), (t_{new}, c_{new}), T_{current}) > random()$ **then**
        $r, t, c \leftarrow r_{new}, t_{new}, c_{new}$
    decrease $T_{current}$

$neighbor(r, T_{current})$

$\sigma \leftarrow min(sqrt(T_{current}), (upper - lower)/(3 * rate_{learn}))$
$updates \leftarrow random.Normal(0, \sigma, size(r))$
$r_{new} \leftarrow r + updates * rate_{learn}$
return r_{new}

However, we rather decided to focus our efforts on improving profiling methods based on resource utilization information, as we discuss next.

4.4.2 Blackbox+Whitebox profiling

Although Blackbox profiling performs without any specific knowledge, it is often a long-running and consequently expensive process. This is caused by the large number of iterations we have to perform until good resource configurations are identified. However, the IaaS clouds may provide ways to monitor the actual utilization of a resource configuration during its reservation. This allows one to retrieve the utilization of resources after every application execution and therefore make use of this information without breaking the agnosticity of the blackbox model. For example, if a resource (CPU, memory) had a high utilization for a significant fraction of the execution time, then scaling up this resource is likely to reduce the execution time. Similarly, if the resource has been mostly under-utilized, then scaling it down may reduce the execution cost without significantly impacting the execution time.

We therefore propose to complement the blackbox modeling method with one or more "whitebox" plugins. Unlike the generic blackbox method, a *whitebox plugin* can be specialized for a certain type of application or resource. Each whitebox plugin aims to detect well-defined characteristics of application executions, and to provide useful "hints" when possible to the blackbox method. It does not need to always provide useful information, but when it does provide a hint, the requirement is that this hint should always be correct. The integration between blackbox and whitebox methods is presented in Figure 4.8.

We propose two additional search algorithms which make use of the feedback from a whitebox model which monitors the utilization of a resource configuration and points out a direction to drive the search towards possibly better one. The implementation of the whitebox method is presented in Algorithm 4. The algorithm receives the specification of a resource configuration and, based on its utilisation data it acquires during execution, it issues a "increase" or "decrease" action for each particular resource in the configuration. If no action can be derived from the utilization data, the whitebox algorithm returns "none".

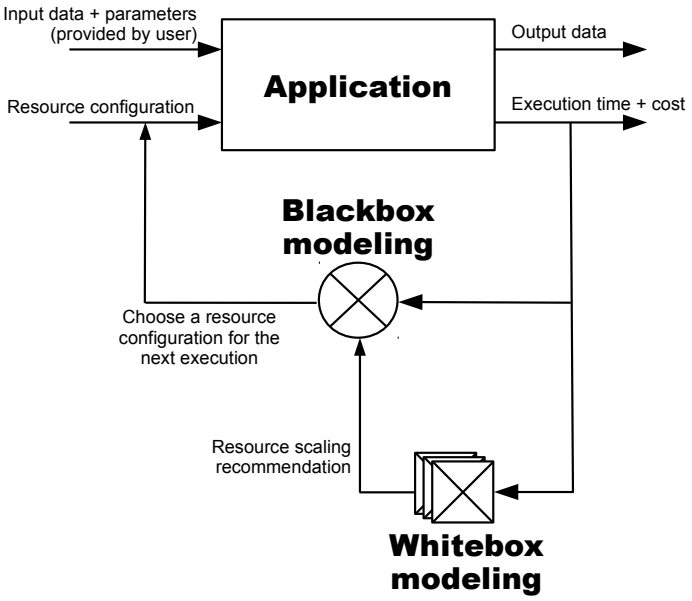


Figure 4.8: Integrated Blackbox+Whitebox Modeling Method.

Algorithm 4 WhiteboxModel

Input: Resources $r = \{r_1, r_2, ..., r_n\}$
Output: Resources $r' = \{r'_1, r'_2, ..., r'_n\}$
1: $r' \leftarrow \{r_1, r_2, ..., r_n\}$
2: **for** $i = 1$ to $|r|$ **do**
3: **if** r_i is over- **or** underutilized **then**
4: **if** r_i is overutilized **then**
5: $r'_i \leftarrow$ "increase"
6: **else if** r_i is underutilized **then**
7: $r'_i \leftarrow$ "decrease"
8: **else**
9: $r'_i \leftarrow$ "none"

Utilization-Driven

This algorithm simulates the utilization-based autoscaling systems. It iteratively refines an initial resource configuration by monitoring the resource utilization generated by the application. As shown in Algorithm 5, the algorithm starts with a random resource configuration (lines 1-2), and following each execution it retrieves the actions the whitebox plugin recommends to be performed (lines 8). If the whitebox plugin issues an "increase" action, the algorithm then allocates a higher amount of this resource in the hope of delivering better performance (lines 10-11). On the other hand, if the whitebox plugin issues a "decrease" action, the algorithm then reduces this resource amount in the hope of reducing resource costs (lines 12-13). Otherwise, it stops its exploration once there is no useful feedback from the whitebox plugin.

This algorithm is simple and intuitive and it converges quickly to good configurations. However, because it is highly dependant on the whitebox feedback, it may stop prematurely whenever the whitebox plugin cannot provide any hints for improvements. This may be interpreted as getting stuck in a local minima as the algorithm has no options to further extend the search.

Algorithm 5 Utilization-Driven

Input: Application A, Resources $R = \{R_1, R_2, ..., R_n\}$
Output: Set of configurations, their execution time and cost $S_{r,t,c}$

```
1: r ← {r_1, r_2, ..., r_n} where r_i is random value of resource R_i ∈ R
2: Q ← {r}
3: S_{r,t,c} ← ∅
4: while Q ≠ ∅ do
5:     r ← dequeue(Q)
6:     (t, c) ← execution time and cost of running A with resource configuration r
7:     S_{r,t,c} ← S_{r,t,c} ∪ {(r, t, c)}
8:     r' ← WhiteboxModel(r)
9:     for i = 1 to |r'| do
10:        if r'_i is "increase" then
11:            r''_i ← next value of R_i (value after r_i)
12:        else if r'_i is "decrease" then
13:            r''_i ← previous value of R_i (value before r_i)
14:        else
15:            continue
16:        enqueue(Q, {r_1, r_2, ..., r''_i, ..., r_n})
```

Directed Simulated Annealing

This is a variant of the simulated annealing algorithm presented previously. As shown in Algorithm 6, the difference lies in the implementation of the *neighbor()* function: instead of choosing configurations randomly around the current best one, Directed Simulated Annealing uses resource utilization information provided by a whitebox plugin to drive the search towards better configurations. The whitebox plugin issues hints in the form of a request to increase or decrease a resource amount. This allows to focus the search of optimal configurations towards resource configurations of which the application can make good use. If a resource is under-utilized (resp. over-utilized), Directed SA increases (resp. decreases) this resource value by a random amount. Otherwise, if the whitebox plugin cannot offer any direction to drive the search, Directed SA updates the resource value in any direction. This algorithm can therefore be seen as a combination of the Utilization-Driven and the Simulated Annealing algorithms which may benefit from the quick convergence of the Utilization-Driven and the SA capability to avoid local minima.

4.4.3 Extrapolated profiling

The integrated blackbox+whitebox profiling performs well in reducing the runtime/cost of the profiling process. However, many HPC applications executed with production-input sizes may have a very long indivdual execution runtime. Therefore, executing them multiple times would generate signigicant cost and runtime overheads even when applying the blackbox+whitebox approach. Another difficulty is handling applications whose input size may vary from one run to the next, since in principle the entire blackbox and blackbox+whitebox profiling process should be redone for every new input size.

To overcome these limitations, we designed a method named "extrapolated profiling" which relies on the assumption that the important application performance characteristics may be observed using "benchmarking" input sizes representing a fraction of the ones used in production. This approach performs most of the profiling phase using relatively small benchmarking input sizes (which speeds up this part of the profiling), and then uses only a handful of the identified resource configurations to run the production input sizes and

Algorithm 6 $NeighborDirectedSA(r, T_{current})$

```
1: if Probability_directed < random() then
2:     r' ← WhiteboxModel(r)
3:     for i = 1 to |r'| do
4:         if r'_i is "increase" then
5:             σ ← 1 − r_i
6:             r_new_i ← r_i + random.Normal(0, σ, 1)
7:         else if r'_i is "decrease" then
8:             σ ← r_i
9:             r_new_i ← r_i + random.Normal(0, σ, 1)
10:        else
11:            r_new_i ← r_i
12:    if no update has been done then
13:        r_new ← neighbor(r, T_current)
14: else
15:    r_new ← neighbor(r, T_current)
16: return r_new
```

generate a correlation between the runtimes. Then, it uses this correlation to extrapolate the benchmarking model to a larger scale. An overview of its workflow is presented in Figure 4.9.

The extrapolated profiling process consists of two main phases:

The discovery phase makes use of the blackbox+whitebox profiling method presented above to discover a number of resource configurations by executing the application using a (small) *benchmarking* input set. The aim is to characterize the general application behavior as quickly and efficiently as possible.

The extrapolating phase aims to extrapolate the first model to production-sized input data. It selects a small number of configurations in order to run the production input size. The runtimes observed on the selected configurations are used to generate a performance model. This performance model captures the relationship between the impact of the benchmarking input and the production input on a configuration.

Once the relationship between the application execution times of benchmarking-sized and production-sized inputs is known, we can efficiently test any new configuration using the small benchmarking-sized input, and extrapolate the result to production-sized inputs. The full algorithm of the Extrapolated Profiling method is presented in Algorithm 7.

Although this idea is very simple, it works surprisingly well in many scenarios. However, to turn it into a practical profiling technique, we need to face two main issues:

1. Estimating the relationship between execution times using the benchmarking and production input sizes sometimes requires a large number of data points. When the application's computational complexity versus its input size is known a priori (for example, the RTM application has perfectly linear complexity with respect to the input size) then a small number of executions is sufficient to accurately characterize the relationship. However, when the computational complexity is complex or unknown, establishing the extrapolation function may require a large number of executions, which defeats the purpose of extrapolated profiling. For this reason, we restrict the use of extrapolated profiling to scenarios where the general form of the extrapolation function is known (e.g., linear, quadratic, exponential).

Algorithm 7 Extrapolation Modeling

Input: Application A, Resources R, Benchmark Input I_B, Production Input I
Output: Performance Model f

$confs_{benchmark} \leftarrow Blackbox + Whitebox(I_B)$
$confs_{test} \leftarrow$ select 7 elements from $confs_{benchmark}$
$confs_I \leftarrow \emptyset$
$confs_{I_B} \leftarrow \emptyset$
$constraints \leftarrow \emptyset$
while $confs_{test} \neq \emptyset$ **do**
    $C \leftarrow confs_{test}.dequeue()$
    $success, runtime, monitor \leftarrow execute(A, I, C)$
    if *success* **then**
        $confs_{train}.enqueue(C, runtime)$
    else
        $bottlenecks \leftarrow get_bottlenecks_from monitor$
        $parameters \leftarrow get_variable_properties_of C$
        if *bottlenecks* $\neq \emptyset$ **then**
            $C_{new} \leftarrow copy(C)$
            for p in *bottlenecks* **do**
                $C_{new}[p] \leftarrow nextvalue(Interval(p))$
            $success, runtime \leftarrow execute(A, I, C_{new})$
            if *success* **then**
                $confs_I.enqueue(C_{new}, runtime)$
                $constraints.enqueue(p \geq C_{new}[p])$
            else
                go to 16
        else
            for p in *parameters* **do**
                $C_{new} \leftarrow copy(C)$
                $C_{new}[p] \leftarrow maxvalue(Interval(p))$
                $success, runtime, monitor \leftarrow execute(A, I, C_{new})$
                if *success* **then**
                    $confs_I.enqueue(C_{new}, runtime)$
                    $constraints.enqueue(p > C[p])$
                    break
                else
                    continue
for C in $confs_I$ **do**
    if $C \notin confs_{benchmarked}$ **then**
        $runtime \leftarrow execute(A, I_B, C)$
        $confs_{I_B}.enqueue(C, runtime)$
    else
        $confs_{I_B}.enqueue(C, get\_runtime(C, confs_{benchmarked}))$
$f = fit(confs_{I_B}, confs_I)$
return f

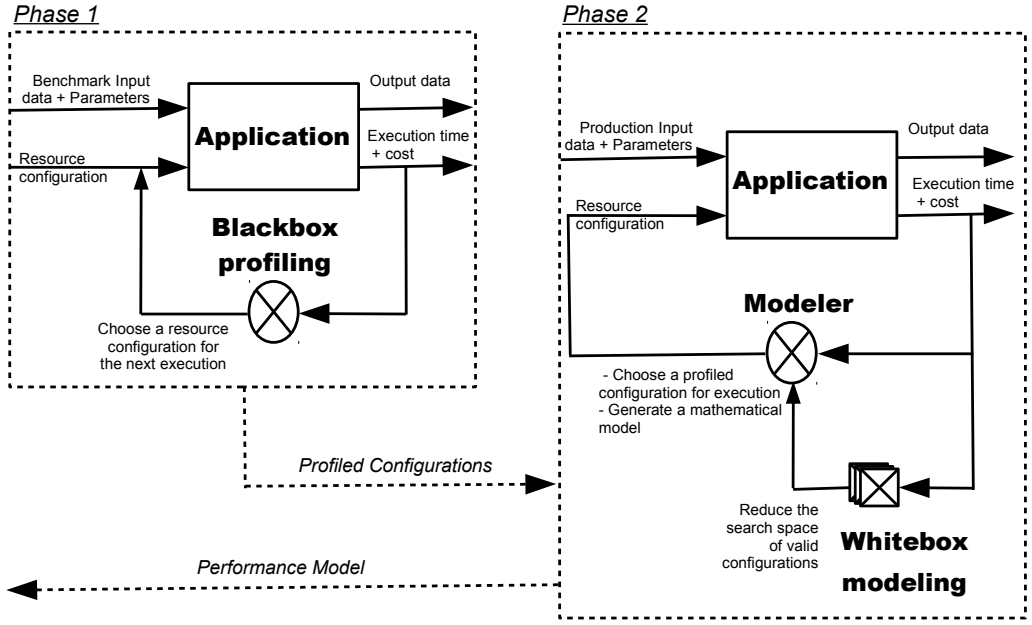


Figure 4.9: Extrapolated modeling method.

2. Depending on the application, in some cases a resource configuration which works well for the benchmarking-sized input may not be sufficient to process the production-sized input and generate a crash. This is common for example for applications whose memory usage depends on the input data size. In such situations, we use the relevant whitebox plugins to identify the reason for the crash and subsequently exclude all the other configurations which are likely to exerience the same insufficient resource problem. This issue is further discussed in the next section.

Bottleneck Detection

When dealing with variable input sizes, many of the configurations that run successfully with the benchmarking dataset may fail when running the production-sized one. For example, an application which loads the entire input data into memory in order to process it, would require only a very small memory size to run the benchmarking input. However, this memory size would obviously be insufficient when running the production input, and may cause the execution to fail. Each such failed execution generates additional cost and runtime to the profiling process. Therefore, once an application crashes, it is important to detect the resource which may have caused it and avoid testing other configurations with similar defficiencies.

Extrapolated profiling addresses this issue by inspecting the utilisation feedback from resource managers to identify resource bottlenecks that crash the application. Once the bottleneck resource is identified, this profiling method avoids testing configurations that may trigger the same problem. By doing this, we minimise the number of failures and consequently the additional runtime and cost they induce. More specifically, the Extrapolated profiling selects a number of optimal and non-optimal configurations discovered using Blackbox profiling targeting a benchmarking dataset. Then, the selected configurations are tested with the production-sized dataset.

The resource utilization information that can be retrieved from the IaaS cloud can provide useful hints for detecting the resources that may cause applications to crash. The

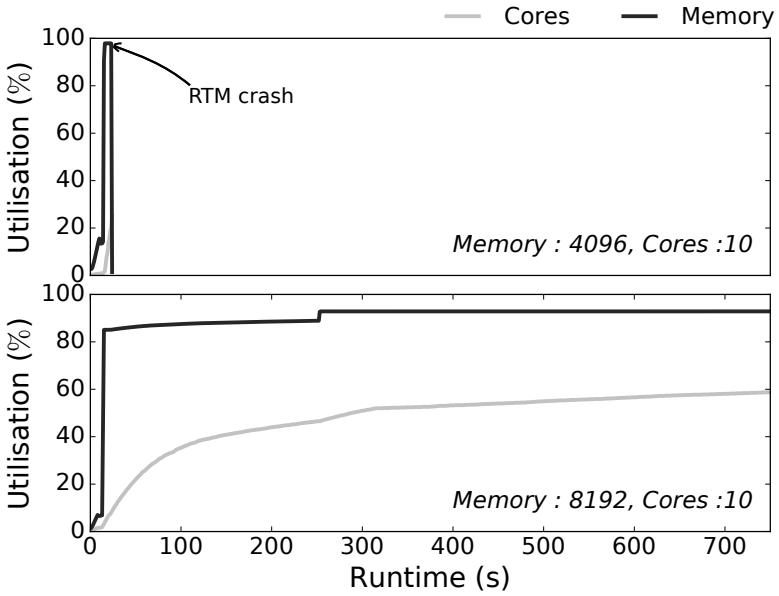


Figure 4.10: Bottleneck detection mechanism.

reason for such failures is usually the allocation of insufficient capacities than what the application requires to run. By analysing the resource utilization of a failed application, we may derive the resource whose capacity must be increased in order to have a successful execution. An example of such a scenario is presented in Figure 4.10. The figure on top presents the utilization feedback for a failed execution of a real application on a configuration with 4 GB of memory and 10 cores. The utilization feedback shows a memory usage growing close to 100% before the application crashes. Hence, to validate this assumption we generate a configuration with 8 GB of memory to execute the application. As the execution is successful, the memory proves to be the bottleneck-resource (bottom figure).

4.5 Evaluation

This evaluation of the different profiling methods is organized in two parts. First, we evaluate and compare the search methods for applications whose runtimes are independent of their input data. Second, we evaluate the benefits provided by the Extrapolated profiling for applications with predictable input-determined runtime. We compare Extrapolated profiling with the Blackbox and the Blackbox+Whitebox profiling and outline the different trade-offs we can make for executing applications.

We base our evaluations on two real HPC applications:

- *Reverse Time Migration* (RTM) is a computationally intensive algorithm used in the domain of computational seismography for creating 3D models of underground geological structures [145]. It is typically used by oil exploration companies to repeatedly analyze the geology of fixed-sized areas. We use a single-node, multithreaded implementation of this application.

- *Delta Merge* (DM) is a re-implementation of an important maintenance process in the SAP HANA in-memory database [146]. This operation is used to merge a table snapshot with subsequent update operations (which are kept separately) in

order to generate a new snapshot. For consistency reasons the database table must remain locked during the entire operation. It is therefore important to minimize the execution time of Delta-Merge as much as possible.

4.5.1 Input-Independent Methods

This section evaluates the search strategies for Blackbox and Blackbox-Whitebox profiling. We focus on three evaluation criteria: (i) the convergence speed of different search strategies towards identifying the full set of Pareto-optimal configurations; (ii) the quality of configurations we can derive from these results when facing various SLO requirements; and (iii) the costs and delays imposed by offline vs. online profiling.

The manifests for our two applications define resource configurations between 1 and 16 CPU cores and 11 discrete values between 2 and 124 GB of memory. We simplify the RTM case by setting a static CPU frequency of 2.2 GHz while for DM we assign 4 possible values. This creates a relatively small search space with 172 configurations for RTM and a much larger one for DM with 704 configurations. Figure 4.4 shows the result of this exhaustive evaluation for RTM.

We perform all experiments using the Grid'5000 experimentation testbed [147]. For RTM, we use machines from the "paranoia" cluster equipped with 2 Intel CPUs with 10 cores each running at 2.2 GHz, 128 GB of RAM and a 10 Gbps Ethernet connectivity. Additional machines with different CPU frequency, number of cores and amount of memory are used for executing the DM application.

All machines run a 64-bit Debian Squeeze 6.0 operating system (OS) with the Linux-2.6.32-5-amd64 kernel. We use QEMU/KVM version 0.12.5 as the hypervisor. We deploy the OpenNebula cloud infrastructure in these machines so our application manager can request any VM configuration via the OCCI interface. We repeated all experiments three times, and kept the average values for execution time.

Although we can use Grid'5000 at no cost, we defined a simple cost model to emulate the situation of a commercial cloud. Our applications typically run within tens of minutes so we based our pricing scheme on a one-minute pricing granularity. Longer-running applications would probably use a more classical one-hour granularity. Similarly, our model charges the user for each resource separately. The parameters of this model are derived from a linear regression over the price of cloud resources at Amazon EC2:

$$Cost_{min} = 0.0396 * N_{Cores} + 0.0186 * N_{Memory(GB)} + 0.0417$$

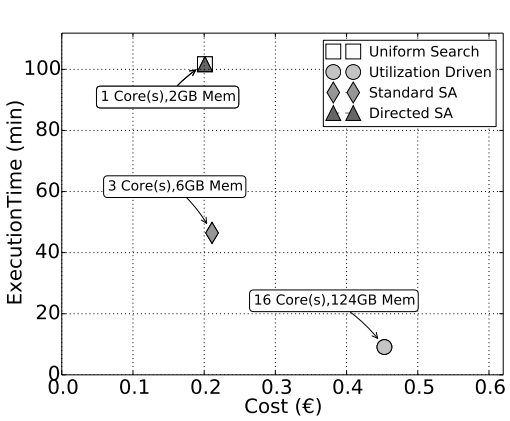
When using cores of different frequency, their cost is scaled accordingly.

Note that our system does not rely on this particular cost model. It is general enough to accept any other function capable of giving a cost for any VM configuration.

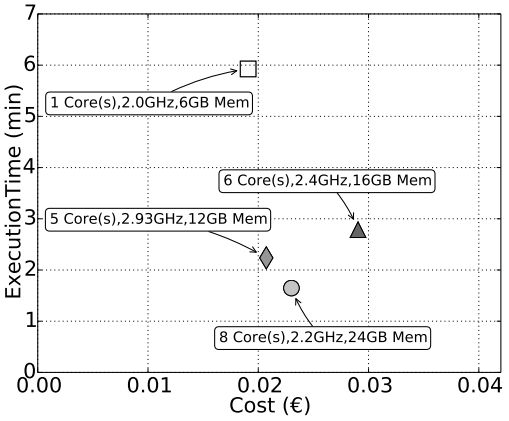
Convergence speed

To understand which search algorithm identifies efficient configurations faster, we compute the Pareto frontiers produced by different algorithms after 10 and 20 executions. This helps us to observe which algorithm converges fastest to the real Pareto frontier. The results are presented in Figure 4.11.

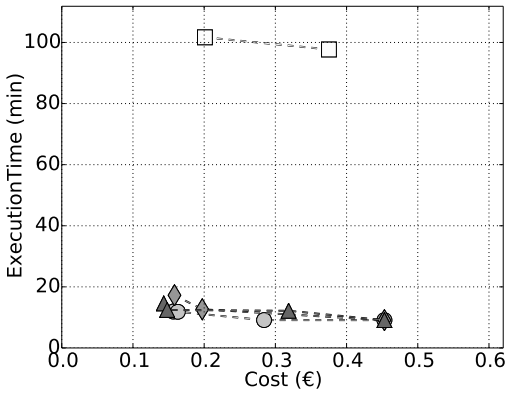
In the case of Uniform Search, we use a step equal to the unit for each dimension of the search space. It therefore actually completes an exhaustive search of the configuration space. We can observe that this algorithm converges very slowly. It eventually finds the full Pareto frontier, but only after it completes its exhaustive space exploration.



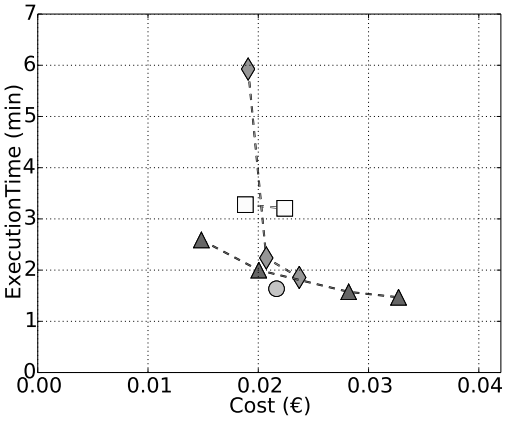
(a) RTM after 1 execution



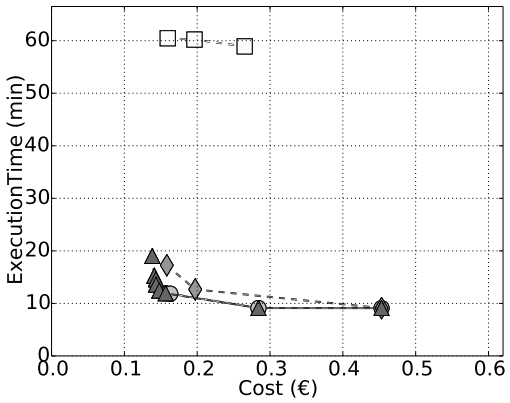
(b) DM after 1 execution



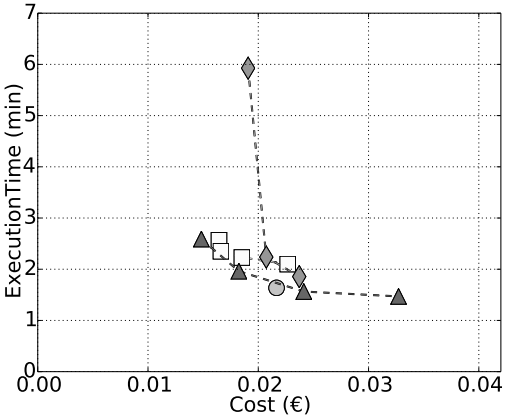
(c) RTM after 10 executions



(d) DM after 10 executions



(e) RTM after 20 executions



(f) DM after 20 executions

Figure 4.11: Pareto frontiers for RTM (a,c,e) and DM (b,d,f).

The Utilization-Driven algorithm starts from a randomly generated configuration in the search space. This randomly-chosen starting point creates a different search path for each run of this algorithm. In the worst case, this algorithm starts with a configuration which neither over- nor underutilizes its resources, so the search stops after a single run. In the best case, the algorithm starts from a configuration already very close to the Pareto frontier, in which case it actually identifies a number of good configurations. We show here an average case (neither the best nor the worst we have observed): it quickly identifies a few interesting configurations but then stops prematurely so it does not identify the entire frontier.

Finally, Standard SA and Directed SA also start from randomly generated configurations. We can however observe that they converge faster than the others towards the actual Pareto frontier. For both applications, after just 10 iterations they already identified many interesting configurations. We can note that Directed SA converges faster than Standard SA, which confirms the usefulness of using resource utilization information to optimize the profiling methods.

SLO Satisfaction Ratio

An interesting perspective from which to evaluate the search results is the range of SLO requirements it can fulfill, and the quality of the configurations that will be chosen by the platform under these SLOs. We now compare the quality of solutions proposed by the different search algorithms after having had the opportunity to issue just 10 profiling executions.

Table 4.1 presents the execution times that would be observed with the RTM application if the SLO imposed various values of maximum cost. Several search algorithm rely on random behavior so we compute the average and standard deviations of 100 runs of each profiling technique. We also show the number of runs where the algorithm failed to propose a configuration for a given SLO. Conversely, Table 4.2 shows the costs that would be obtained with the RTM application after defining a maximum execution time. For both tables we also show the performance that would result from an exhaustive search of the entire space. Tables 4.3 and 4.4 show similar results for the DM application.

It is clear from all the tables that Directed SA provides better configurations for the majority of the SLOs. With its good approximation of the entire Pareto frontier, it can handle all SLOs from the table. However, when facing challenging SLOs, Utilization-Driven may sometimes happen to provide better configurations than Directed SA, but with a high failure rate. This means than in most of the cases it fails to provide a configuration satisfying the SLO. The other algorithms have only a partial or sub-optimal frontier and cannot find configurations for demanding SLOs. At the same time, when several algorithms can propose solutions that match the SLO constraint, the overall solutions found by Directed Simulated Annealing are almost always better, with a lower standard deviation and failure rate.

Profiling Costs

An important aspect to evaluate the efficiency of a search algorithm is the time and cost incurred by the profiling process. The iterations performed until Pareto-optimal configurations are reached, incur an additional cost and duration which can be minimized based on user's choice on profiling approach: offline or online.

Table 4.5 presents the cost and duration overhead of offline profiling for the RTM application using 20 experiments. Offline profiling using the utilization-driven algorithm

SLO / Algorithm	C < 0.15€	C < 0.25€	C <0.35€
Uniform Search	Fail = 100%	T = 60.21 min ± 0.00, Fail = 0%	T = 58.91 min ± 0.00, Fail = 0%
Utilization-driven	T = 16.82 min ± 4.50, Fail = 83%	T = 18.56 min ± 7.44, Fail = 1%	T = 18.62 min ± 7.77, Fail = 0%
Standard SA	T = 13.01 min ± 2.17, Fail = 15%	T = 13.12 min ± 5.74, Fail = 2%	T = 11.46 min ± 3.78, Fail = 1%
Directed SA	T = 12.67 min ± 1.58, Fail = 0%	T = 12.00 min ± 0.24, Fail = 0%	T = 11.34 min ± 1.21, Fail = 0%
Exhaustive search	*T = 12.07 min*	*T = 11.84 min*	*T = 9.12 min*

Table 4.1: Performance after 10 executions of RTM under cost (C) constraints. The values correspond to the average and standard deviation of 100 runs of the search techniques.

SLO / Algorithm	T < 10.00 min	T < 20.00 min	T< 30.00 min
Uniform Search	Fail = 100%	Fail = 100%	Fail = 100%
Utilization-driven	C = 0.28€ ± 0.00, Fail = 98%	C = 0.16€ ± 0.01, Fail = 33%	C = 0.17€ ± 0.05, Fail = 6%
Standard SA	C = 0.35€ ± 0.08, Fail = 36%	C = 0.16€ ± 0.05, Fail = 0%	C = 0.15€ ± 0.05, Fail = 0%
Directed SA	C = 0.40€ ± 0.08, Fail = 30%	C = 0.14€ ± 0.00, Fail = 0%	C = 0.14€ ± 0.00, Fail = 0%
Exhaustive search	*C = 0.28€*	*C = 0.13€*	*C = 0.13€*

Table 4.2: Performance after 10 executions of RTM under time (T) constraints. The values correspond to the average and standard deviation of 100 runs of the search techniques.

SLO / Algorithm	C < 0.02€	C < 0.04€	C <0.06€
Uniform Search	T = 2.23 min ± 0.00, Fail = 0%	T = 2.10 min ± 0.00, Fail = 0%	T = 2.10 min ± 0.00, Fail = 0%
Utilization-driven	T = 2.11 min ± 0.25, Fail = 74%	T = 2.14 min ± 0.64, Fail = 22%	T = 2.20 min ± 0.91, Fail = 12%
Standard SA	T = 3.47 min ± 1.42, Fail = 26%	T = 2.14 min ± 0.92, Fail = 5%	T = 1.97 min ± 0.63, Fail = 3%
Directed SA	T = 2.62 min ± 1.10, Fail = 7%	T = 1.66 min ± 0.18, Fail = 0%	T = 1.60 min ± 0.16, Fail = 0%
Exhaustive search	*T = 1.81 min*	*T = 1.46 min*	*T = 1.46 min*

Table 4.3: Performance after 10 executions of DM under cost (C) constraints. The values correspond to the average and standard deviation of 100 runs of the search techniques.

SLO / Algorithm	T < 2.00 min	T < 3.00 min	T< 4.00 min
Uniform Search	Fail = 100%	C = 0.02€ ± 0.00, Fail = 0%	C = 0.02€ ± 0.00, Fail = 0%
Utilization-driven	C = 0.03€ ± 0.02, Fail = 49%	C = 0.03€ ± 0.02, Fail = 10%	C = 0.03€ ± 0.02, Fail = 4%
Standard SA	C = 0.04€ ± 0.02, Fail = 28%	C = 0.02€ ± 0.01, Fail = 6%	C = 0.02€ ± 0.01, Fail = 1%
Directed SA	C = 0.02€ ± 0.01, Fail = 1%	C = 0.02€ ± 0.00, Fail = 0%	C = 0.02€ ± 0.00, Fail = 0%
Exhaustive search	*C = 0.01€*	*C = 0.01€*	*C = 0.01€*

Table 4.4: Performance after 10 executions of DM under time (T) constraints. The values correspond to the average and standard deviation of 100 runs of the search techniques.

appears to be both cheap and fast. However, this is only due to the fact that this algorithm stops long before having identified the full Pareto frontier. Uniform Search starts its exploration from the slowest available resource types which may incur long execution times, and therefore become overall very expensive.

Standard SA is slightly cheaper and faster than Directed SA. This is due to the fact that we decided to keep the simulated annealing implementation standard (using the implementation from SciPy [148]). In our experience, the initial temperature chosen by the Standard SA implementation is sometimes too low, which means that the algorithm converges too quickly afterward. Occasionally the algorithm stops before issuing 20 executions, which explains its relatively low cost and profiling time.

Directed SA does not have this limitation as it does not rely always on the temperature to choose a next configuration. This algorithm therefore explores more configurations, thus having a higher total cost and execution time than Standard SA. On the other hand, it identifies more optimal configurations.

Figure 4.12 shows the execution times and costs incurred by the user using the Directed Simulated Annealing algorithm in conjunction with online profiling. In this case,

Algorithm	Total cost	Duration
Uniform Search	19.92 €	1727.93 min
Utilization-Driven	2.63 €	234.51 min
Standard SA	7.09 €	426.41 min
Directed SA	9.38 €	635.39 min

Table 4.5: Total cost and duration overhead for an offline profiling of RTM limited to 20 executions. The values represent the average of 100 profiling processes with each search technique.

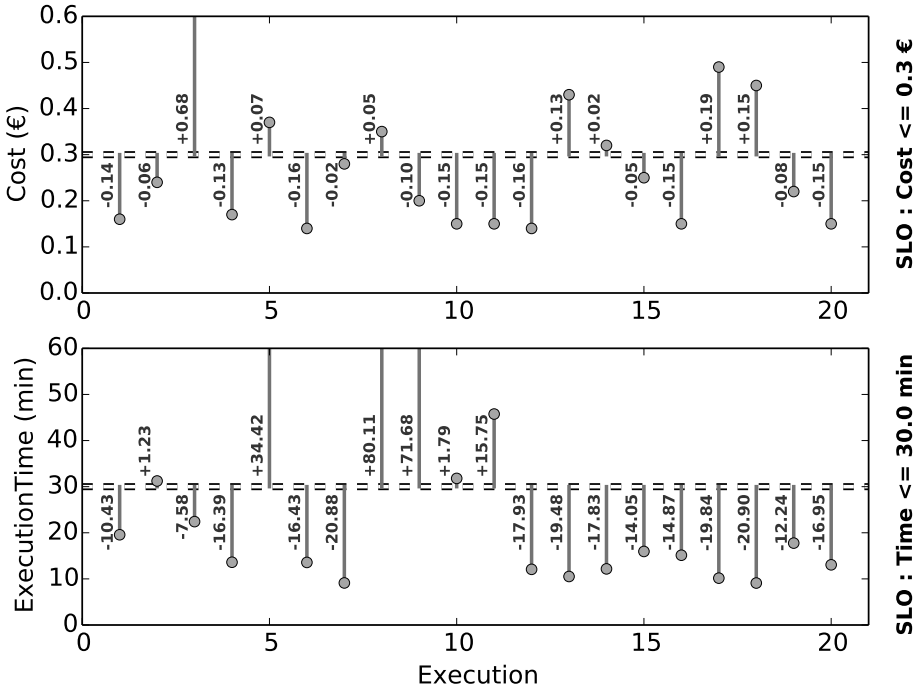


Figure 4.12: Cost and Execution Time fluctuation in an online profiling of RTM limited to 20 executions.

no artificial execution is generated. On the other hand, as we can see in the figure, many executions violate an arbitrary SLO of 0.30 €. However, it is interesting to notice that the overall group of execution remains within its aggregated budget (with a cost saving of 0.21 €). Similarly, when applying an arbitrary SLO of 30 minutes of execution time, numerous individual executions violate the SLO but overall they perform faster than the aggregated execution time tolerated (20.82 minutes).

For this first part of the evaluation, we conclude that the Blackbox+Whitebox method based on the Directed Simulated Annealing algorithm shows the fastest convergence to optimal configurations and provides a better satisfaction for the majority of the SLOs. It generates good configurations to be used when creating an application profile in a smaller number of executions.

For users willing to tolerate SLO violations on individual executions, the online profiling method provides obvious benefits: it remains within the aggregate time or budget of the overall profiling phase, and therefore offers fast and cost-effective generation of a full performance model. On the other hand users unwilling or unable to tolerate individual SLO violations can revert to the offline method, at the expense of artificial executions which consume both time and money.

4.5.2 Input-Dependent Search Methods

In order to show the benefits provided by exploiting dependencies between application input data, we now compare the Extrapolated Profiling with the Blackbox and Blackbox+Whitebox profiling based on several criteria: (i) profiling performance when we impose different stopping conditions; (ii) the time and cost overhead when we limit the profiling to a certain number of iterations; (iii) the impact of failures on the profiling process; and (iv) the quality of configurations we can derive when facing SLO requirements.

These evaluations are based on the previous two use-case applications: RTM and DM. The application manifests define resource configurations between 2 and 16 CPU cores and 5 discrete values between 2 and 32 GB of memory. Note that we ignore the application deployment time since it is mostly constant, and only consider the execution time of the application. We apply the same cost model derived from Amazon's pricing model presented in Section 4.5.1.

Duration- and Budget-Constrained Profiling

The benefits of Extrapolated profiling can be studied from different perspectives. One such perspective relies on evaluating how well Extrapolated profiling can perform given a time or cost budget dedicated to the profiling phase. For each search algorithm, we analyze how many resource configurations for running production input sizes it can identify until the time or budget limit is reached.

Figure 4.13 presents the averaged results for the RTM application of 100 profiling processes performed with each search algorithm under different cost and time budgets. The top figure shows successively the total profiling runtime, cost and number of working configurations versus the number of failed configurations identified by each profiling process when the stopping criteria was set to different budget limits (i.e. 1€, 2€ and 3€). As we can observe, Extrapolated profiling identifies the greatest number of working configurations under all budgets. Directed SA and Uniform Search make use entirely of the given budget. However, while Directed SA identifies a reasonable number of working configurations and avoids failing configurations thanks to the whitebox feedback, Uniform Search makes bad use of the budget by incurring significant number of failed configurations. Standard SA and Utilization-Driven algorithms stop before spending the given budget. As these methods do not implement any bottleneck detection mechanism, they incurr a large number of failures and, consequently, get trapped in a local minima. The bottom figure presents similar results for the profiling performed under different time limits (i.e. 400min, 600min and 800 min).

Similarly to RTM, Figure 4.14 presents the results for DM. As concluded in the first part of this evaluation, Blackbox+Whitebox implementing Directed SA provides the best results when compared with the other input-independent strategies. Therefore, we choose to focus the rest of the evaluation on Blackbox+Whitebox and Extrapolation profiling.

Cost and Time Overhead

Extrapolated profiling aims at reducing the cost and time overhead of profiling applications. In order to evaluate the cost and time overhead it produces, we set the profiling process to stop after a predefined number of iterations has been performed.

Table 4.6 presents a comparison of the total cost and runtime of profiling RTM and DM with Blackbox+Whitebox and Extrapolated profiling. The values correspond to the average of 100 profiling processes.

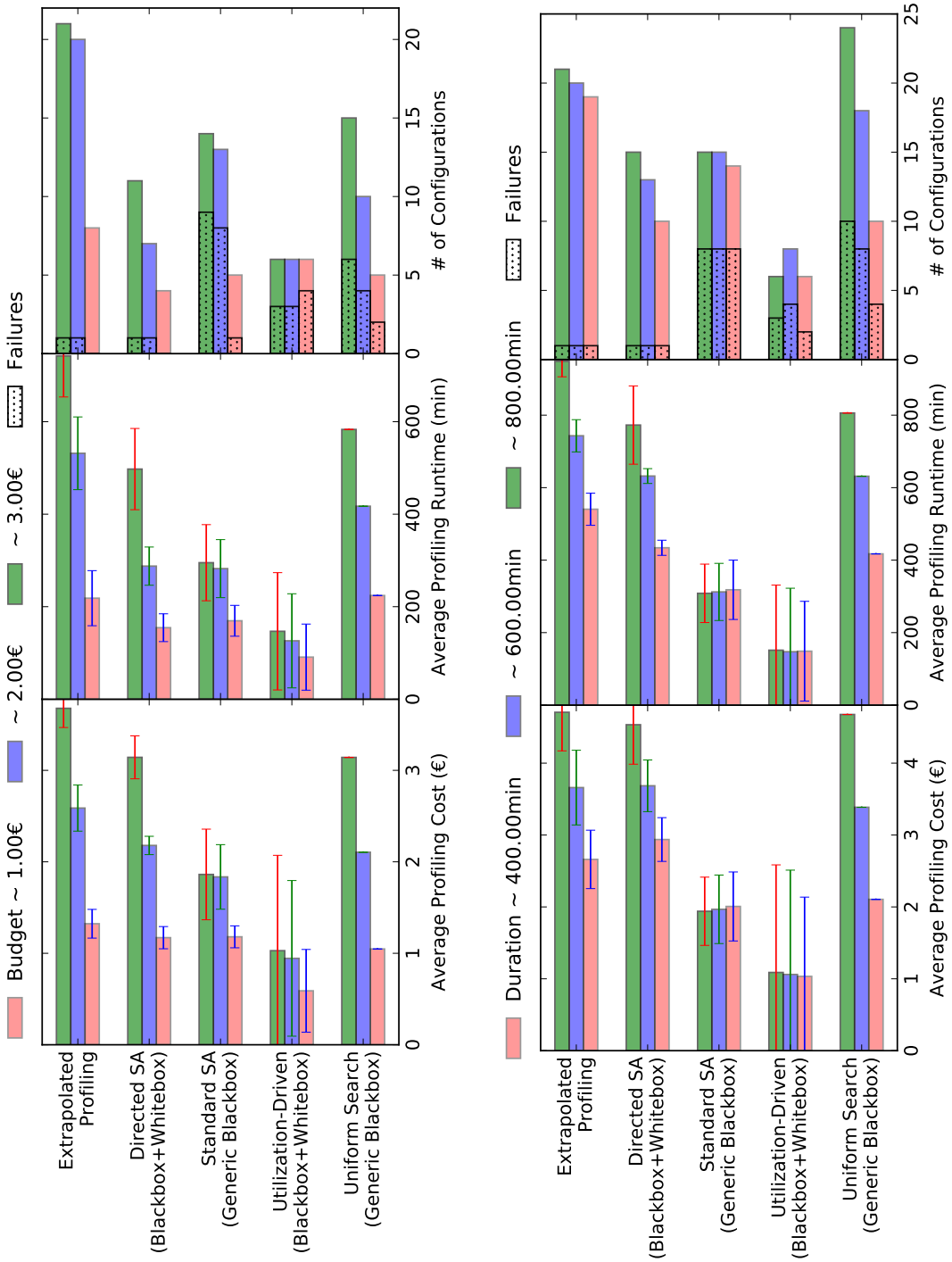


Figure 4.13: Average total cost and runtime of RTM profiling performed within user-defined budget and duration limits.

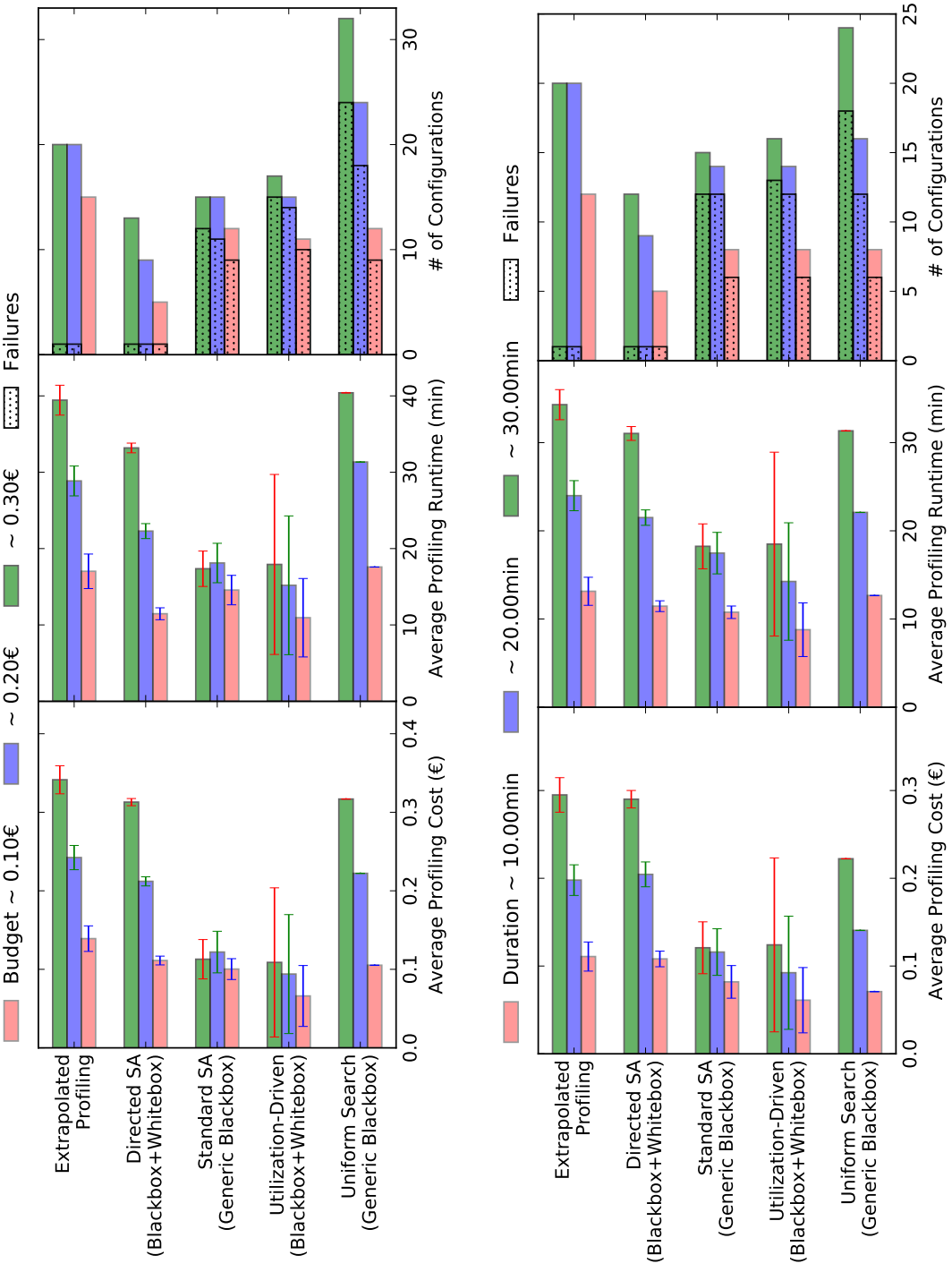


Figure 4.14: Average total cost and runtime of DM profiling performed within user-defined budget and duration limits.

Application	Profiling method	Runtime (minutes)	Cost (€)	# of failed executions
RTM	*Blackbox+Whitebox*	818.57	4.68	1
	Extrapolated	693.71	3.32	1
DM	*Blackbox+Whitebox*	40.61	0.39	1
	Extrapolated	31.72	0.27	2

Table 4.6: Total cost and duration of profiling. The values correspond to the average values of 100 profiling processes. Blackbox+Whitebox stops when reaching 15 iterations while Extrapolated profiling stops after 15 iterations with benchmarking input and 7 with the production input. An iteration corresponds to a successful execution. As expected, Extrapolated profiling runs faster and cheaper because of its use of the benchmarking input to explore the search space.

For both applications, the extrapolated profiling requires 10-20% less profiling time than the Blackbox+Whitebox method, and 30-35% less financial costs. This latter result is largely due to the fact that the extrapolated method issues long and expensive profiling executions only for configurations which are likely to deliver interesting performance. We can therefore conclude that Extrapolated profiling reached its objective of reducing the total cost and runtime of the profiling process. We study the quality of the identified configurations in the next section.

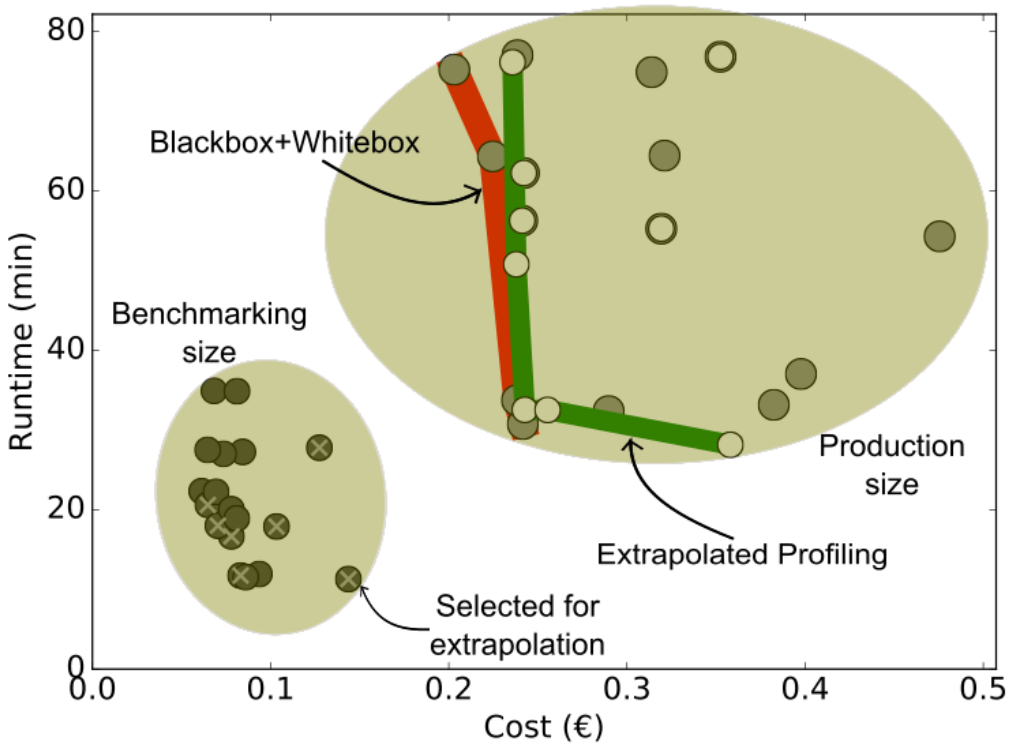
Results Quality

However, when aiming to reduce the overhead of profiling, we need to consider that Extrapolated profiling may fail to provide better performance-cost trade-offs when compared with the Blackbox+Whitebox approach. We focus further on this aspect by evaluating the Extrapolate profiling from a Pareto Frontier perspective. We study a particular scenario based on the two profiling methods where we compare the Pareto frontiers generated.

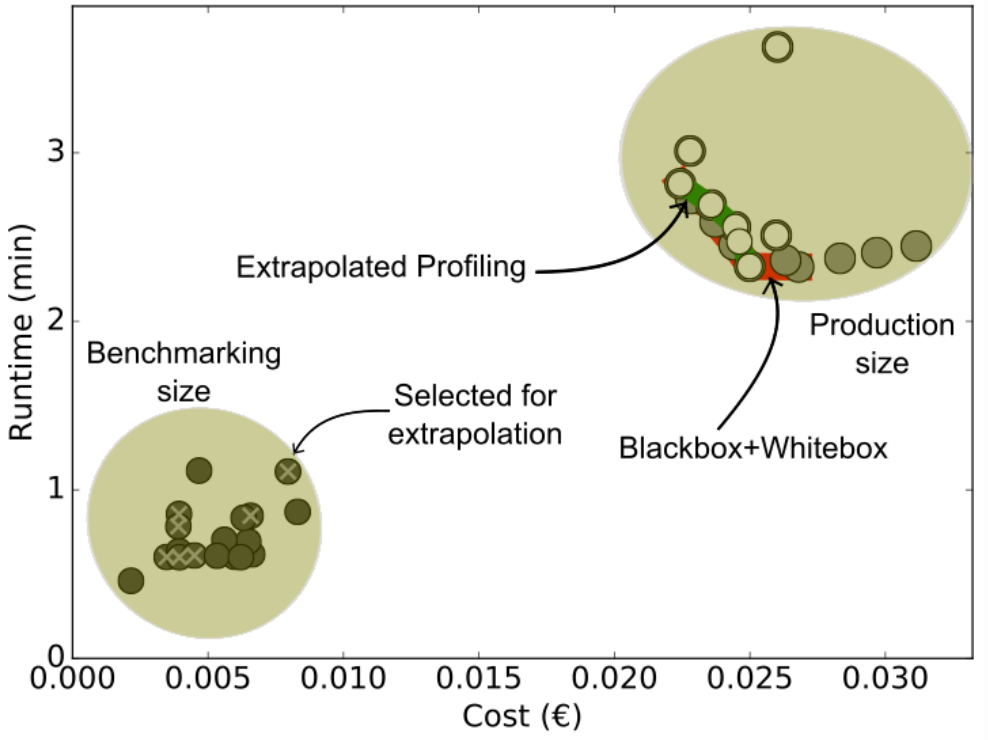
Figure 4.15 presents the configurations identified when profiling RTM and DM with these two approaches. The extrapolated approach shows two sets of points: one set represents the benchmarking-sized inputs, and the other set represents the production-sized inputs. We can observe that both methods produce close Pareto frontiers for both RTM and DM which means that they may produce similar results after the profiling phase, and therefore exhibit similar levels of SLO satisfaction.

Failures Overhead

In order to better understand the impact of the failures on the profiling process, we evaluate the bottleneck detection mechanism we implemented in the Extrapolated profiling based on three criteria: failure runtime ratio to the total runtime of the profiling process; failure cost ratio to the total cost of the profiling process; and failed iterations ratio to the total iterations of the profiling process. For this, we consider the average cost and runtime induced by bottlenecks during the extrapolation phase. Therefore, we compare two scenarios: extrapolation without bottleneck detection mechanisms, and extrapolation with an utilisation-based bottleneck detection mechanism. Each scenario consists in profiling RTM and DM 100 times and compute the average number of failures with the assigned cost and runtime.

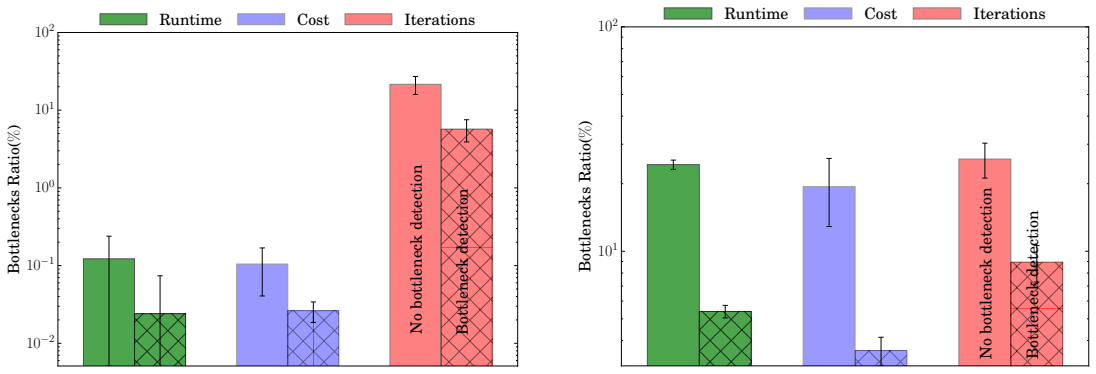


(a) RTM application



(b) DM application

Figure 4.15: Optimal configurations identified by the Blackbox+Whitebox and extrapolated profiling methods.



(a) RTM (b) DM

Figure 4.16: The bottleneck ratio in the total profiling runtime and cost. The later the application crashes, the higher the bottleneck overhead is. The utilisation-based detection mechanism allows to reduce the bottleneck overhead for both RTM and DM (lower, the better).

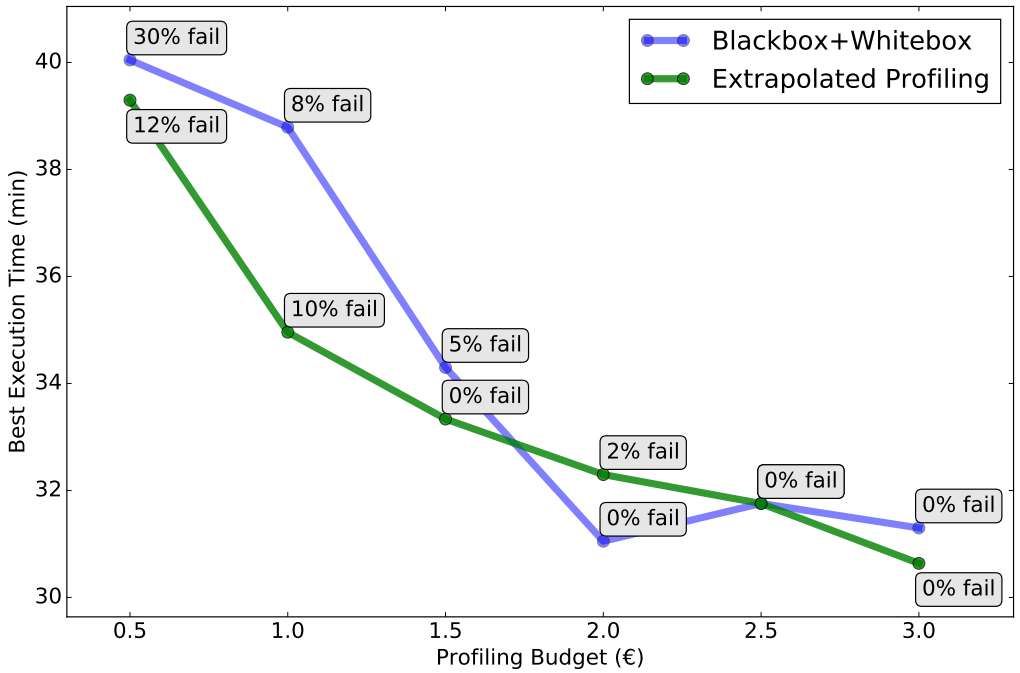
Figure 4.16 presents the results for the RTM and DM applications. In the case of RTM, the total runtime and cost overhead induced by the bottlenecks is negligible. This is because RTM fails right in the beginning of the execution when there is no sufficient memory allocated to load data; the failure runtime is less than 0.2% of the total time. In the case of DM, on average, 26% of the total number of configurations failed to execute the production-sized dataset. The runtime and cost of the failed configurations represented 24% and 19% of the total. On the other hand, the extrapolation employing the utilisation-based bottleneck detection mechanism generated 15% less failures with a minimization of 19% for runtime and 15% for cost. This significant overhead is caused by DM failing long after the beginning of the execution.

In both cases, extrapolating with utilisation-based bottleneck detection lowers the number of failed executions and, consequently, minimizes the runtime and cost of the profiling process.

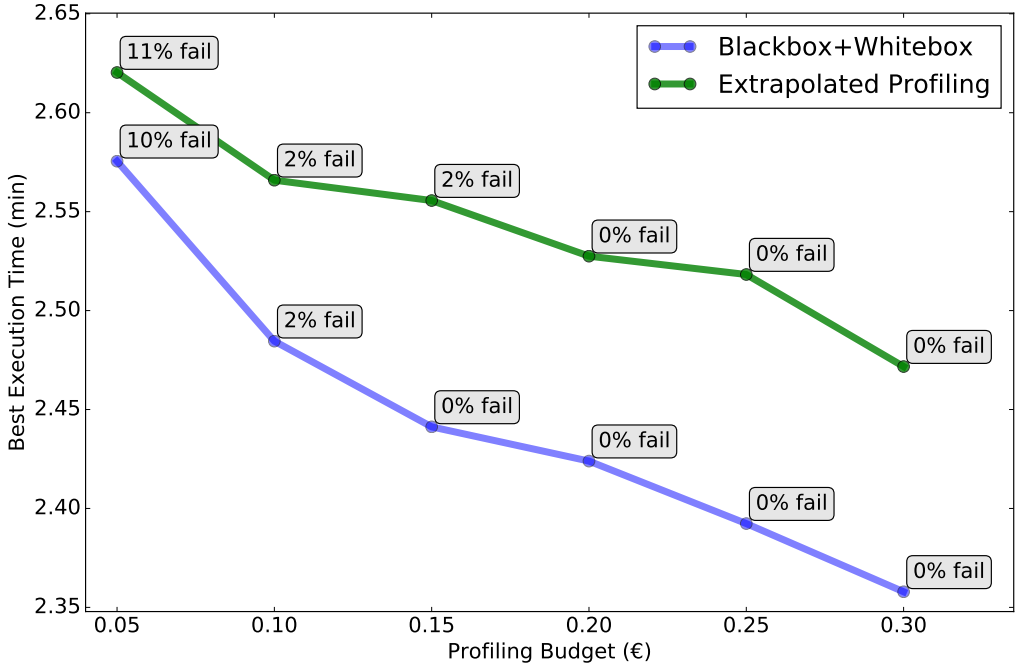
SLO Satisfaction

An interesting perspective from which to study the profiling is the control over the quality of configurations that we can derive. For this, we profile RTM and DM under a range of profiling budgets and inspect the improvement in the proposed configurations satisfying an SLO.

Figure 4.17 (a) presents the results for RTM. The SLO imposed requires the selection of a configuration with a cost smaller than 0.35 €. As expected, increasing the budget of profiling impacts the quality of the proposed configurations. With a higher budget allocated to the profiling process, the system chooses faster configurations that satisfy the SLO and therefore, makes a better performance-cost trade-off. As we can observe, the failure rate to satisfy the SLO is minimized with the budget increase. Another aspect to notice in this case is that Extrapolated profiling provides better configurations than Blackbox+Whitebox when limited to a low profiling budget. This is due to the fairly linear complexity of the RTM application which allows Extrapolated profiling to focus on good performance-cost trade-offs.



(a) RTM application. As expected, increasing the profiling budget allows the discovery of faster resource configurations performing under 0.35 €(imposed SLO).



(b) DM application. Similarly to RTM, setting a higher profiling budget provides faster resource configurations performing under 0.026 €(imposed SLO for DM).

Figure 4.17: Performance of best configurations identified by the Blackbox+Whitebox and Extrapolated profiling methods under cost constraints.The values correspond to the average of 100 profiling processes for each method.

Figure 4.17 (b) presents similar results for DM when we impose an slo requiring a configuration having a cost smaller than 0.026 €. However, in this case, the quality of the configurations proposed by Extrapolated profiling is slightly worse than in the case of Blackbox+Whitebox. This is because Extrapolated profiling cannot derive a good correlation between the performance of the benchmarking and the production input on the same configuration. The main cause for this is that there is no fairly predictable relationship between the performance of different input sizes of DM on the same resource configuration.

To conclude this evaluation, Extrapolated profiling provides good performance and SLO satisfaction when applied to applications where the relationship between the performance of different input sizes can be determined from a very small number of tests. An example of such a "good" behaving application is RTM. On the other hand, for applications as DM where the relationship is not easily determined and it may depend on settings in the input, Blackbox+Whitebox outperforms the Extrapolated profiling.

4.6 Conclusion

Making a good resource selection in order to enable different performance-cost trade-offs for the execution of arbitrary applications is a difficult problem. In this chapter, we have presented several approaches to enable application performance-cost trade-offs through profiling. Our approaches range from input-independent search strategies making use only of minimial execution information such as runtime and cost to approaches making use of specific knowledge such as resource utilization, application input dependencies etc.

Chapter 5

Integration in a heterogeneous IaaS-PaaS Cloud System

Contents

This work was done in collaboration with partners from the HARNESS consortium. It is reported in the HARNESS white paper[149] and in a book chapter submitted for publication.

5.1 The HARNESS Heterogeneous Cloud

This thesis was conducted in the context of the HARNESS European project [14]. HARNESS aimed to incorporate heterogeneous hardware and network technologies in cloud platforms and to facilitate the execution of arbitrary applications with complex resource requirements according to user expectations. We therefore developed the first two contributions of this thesis in the HARNESS prototype, demonstrating how these technologies may be employed and integrated in future public or private heterogeneous clouds. We first focus on the overall architecture of the HARNESS platform. The next sections discuss the integration of the FPGA virtualization and performance modelling technologies in this platform.

A simplified overview of the HARNESS platform's architecture is presented in Figure 5.1. It consists in three main parts: (1) a Platform-as-a-Service layer in charge of managing applications; (2) an Infrastructure-as-a-Service layer for resource management; and (3) a virtual execution layer where the applications actually run. This architecture includes the minimum set of components providing management of physical resources, creation of virtual execution platforms and deployment of generic applications on the virtual execution platforms. The red-framed components incorporate our prototypes. The techniques for FPGA virtualization and autoscaling are implemented in the Orchestrator and Autoscaler, coupled with Maxeler's MPC-X and its FPGA-based Dataflow Engine Cards (DFE) as the FPGA-server. The profiling methodologies enabling the performance-cost trade-offs are implemented in the platform layer and additional components running in the virtual execution platform.

5.1.1 The Infrastructure-as-a-Service Layer

The IaaS layer of a cloud is responsible for the management of physical resources such as servers, storage, network devices, accelerators (e.g., GPU, FPGA), and the on-demand allocation of virtual resources based on them. Current cloud offerings of virtual resources are based on instance types with predefined configurations and limited placement options which are charged at a set price. However, a tenant's application may have different resource requirements and may not fully exploit a predefined instance configuration. This means that a part of the resources coming with the predefined configuration may be underutilized although the tenant supports its cost. Cloud interfaces such as OCCI have been proposed to enable the allocation of custom instance types [87]. However, they rely on built-in models for their resource types which makes it difficult to add new heterogeneous resources on-the-fly.

The HARNESS IaaS layer addresses this issue by enabling the allocation of custom configurations of resources through the use of flexible models for virtual resources. A custom configuration of resources consists of a set of virtual heterogeneous resources and network dependencies between them. The characteristics of a resource such as type, number, internal configuration can be fully specified by the tenants according to their needs. Clearly, these characteristics must fit within the capacity limit of cloud's hardware resources. Custom heterogeneous configurations may allow to reduce the cost of executing applications by avoiding over-provisioning resources required by their applications. But they potentially increase resource fragmentation.

The key components of the HARNESS IaaS cloud managing the allocation of custom resource configurations are as follows:

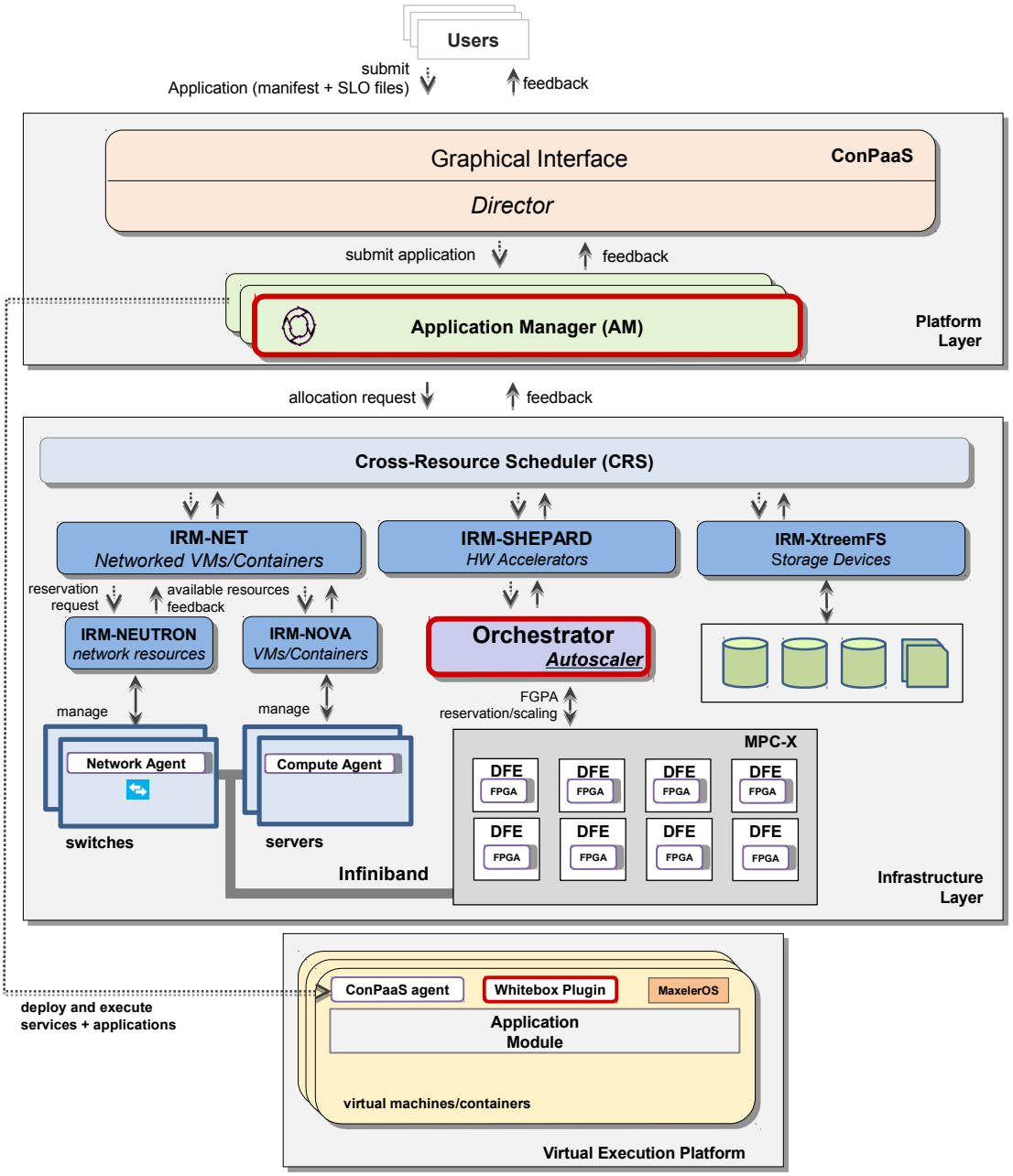


Figure 5.1: The architecture of the HARNESS platform with the outlined components implementing the contributions of this dissertation.

- ***Infrastructure Resource Managers (IRMs)*** are ***resource-specific*** components in charge of the allocation and release of virtual resources; an IRM operates over a single specific class of resources only, such as servers, storage, network and accelerators. IRMs allow heterogeneous and specialized hardware such as general-purpose graphics processing units (GPGPUs), FPGA-based cards, XtreemFS storage devices and software-defined networking (SDN) switches to be exposed in a generic manner as first-class cloud resources. All IRMs implement the same API dedicated to the discovery, specification, reservation and feedback of resources. Some IRMs may make use of resource-specific solutions such as OpenStack components (Nova, Neutron) to implement these standard functions. Moreover, IRMs can be hierarchically combined in order to manage complex resource requirements. For instance, in Figure 5.1, the composition of IRM-NOVA, IRM-NEUTRON and IRM-NET is organized in a hierarchical fashion in order to be able to manage groups of VMs/containers.

- ***The Cross-Resource Scheduler (CRS)*** is a ***resource-agnostic*** component acting as the entry point of the HARNESS IaaS cloud. The CRS has complete knowledge on resource availability in the system and is in charge of scheduling dynamic sets of heterogeneous virtual resources without having specific knowledge on the resource types. The scheduling process consists in selecting the physical host from the available resources on which to allocate the virtual resources. Once the CRS chooses a schedule, it delegates the resource-specific virtual resource creation requests to IRMs which have better understanding on how to allocate and release the requested type of resources.

  For example, an application may send a request to the CRS to reserve a group of resources consisting of one VM, one FPGA and a virtual network link with a bounded latency between them. The CRS first interogates a network IRM (IRM-NET) on the proximity measurements such as latency and available bandwidth between the physical hosts for VMs and FPGAs. The CRS filters the available physical resources satisfying the network constraint specified in the reservation request, and selects the physical resources on which to allocate the VM and virtual FPGA. Once the physical resources have been selected, the CRS delegates the requests to provision the VM and virtual FPGA to the corresponding IRMs (IRM-NOVA and IRM-SHEPARD).

This flexible design allows the HARNESS platform to be customized to manage not only existing heterogeneous hardware devices but also to be resilient to new forms of heterogeneity which may become available in the future.

5.1.2 The Platform-as-a-Service Layer

The PaaS layer of HARNESS handles the management of arbitrary applications on custom heterogeneous resource configurations. This layer is implemented by ConPaaS, an open-source runtime environment for hosting applications in cloud infrastructures [95, 150]. ConPaaS serves as the interface between the HARNESS platform and the end-users. The main components of the platform are as follows:

- ***a Graphical User Interface*** is provided through a web server for tenants to submit applications and their specifications.

- ***the Director*** is a service in charge of user authentication and deployment of application managers when an application is submitted through the graphical interface.

Any user request concerning the management of an application is forwarded by the Director to the Application Manager in charge of its execution.

- ***Application Managers (AMs)*** are in charge of controlling the lifecycle of applications. An AM is a generic and application-agnostic component in order to support any class of applications. It operates within a VM/container provisioned using the HARNESS IaaS layer. It takes as input a manifest describing the structure of the application with its resource requirements and an SLO file describing the expectation of the user for its execution, as previously discussed in Chapter 4. The functionality of the AM consists in interpreting the manifest and SLO files, building a performance model for the assigned application, choosing the type and number of resources to run the application according to the SLO, deploying the application in the provisioned resources and finally collecting resource utilization feedback. For every submitted application, the Director instantiates one dedicated AM to manage its execution. The AM is therefore the component where performance profiling is integrated, as further discussed in Section 5.3.

Any resource provisioning request issued either by the Director or an AM, is made through requests to the CRS.

5.1.3 The Virtual Execution Layer

The virtual execution platform is the layer where applications are actually executed. It consists of a group of heterogeneous virtual resources provisioned through the CRS. An example of a virtual execution layer is presented in Figure 5.2.

The configuration of the virtual resources is chosen by the AM according to the specifications in the application manifest. Aside the application's code itself and drivers for different types of resources, the virtual execution platform contains also ***ConPaaS agents*** running on CPU-based virtual instances (VMs or containers). These agents execute commands on behalf of the AM to set up the runtime environment for the application, start/stop the application, monitor its resource utilization, etc.

The interactions between different types of resources encompassed in the virtual platform are also shown in Figure 5.2. Usually, a virtual platform requires at least one CPU-based VM/container as the main point to control the processing and data movement. For example, an application cannot be executed on a virtual platform consisting of one storage volume and one virtual FPGA as it needs at least one CPU to coordinate data movements between the two.

5.2 Integration of Virtual FPGA Resources

FPGAs have only recently started to draw more attention as promising cloud resources for high-performance computations. Although few methods have been proposed for integrating FPGAs in cloud systems, we still lack an efficient method to virtualize and expose them as first-class resources. We addressed this issue in Chapter 3 of this dissertation where we proposed an approach to share FPGAs in a multi-tenant environment. Figure 5.3 provides a magnified view on the integration of the FPGA-based servers and the two components managing the creation and scaling of virtual FPGAs in the HARNESS IaaS layer. As we can observe, we make use of an FPGA-specific IRM, entitled *IRM-SHEPARD*, in charge of managing the resource-agnostic requests issued by the CRS. These requests are translated into requests for virtual FPGAs and forwarded to the *Orchestrator* managing the

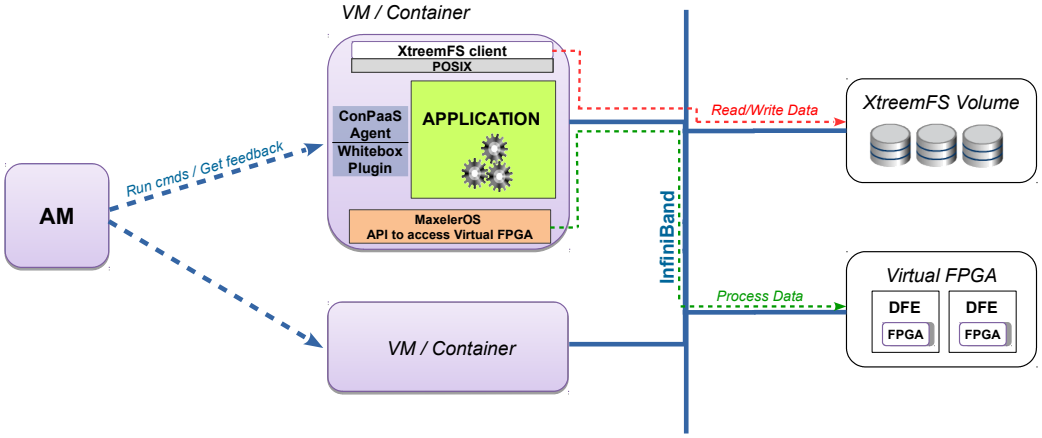


Figure 5.2: Example of a virtual execution platform consisting of two VMs, one XtreemFS storage volume and one virtual FPGA backed up by two physical devices.

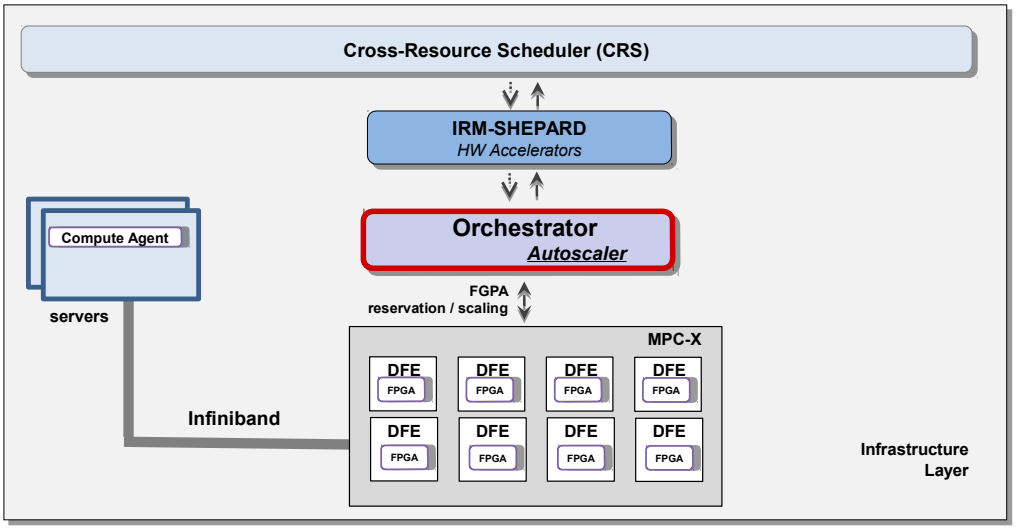


Figure 5.3: Integration of FPGAs in the HARNESS IaaS layer.

FPGA-servers. The IRM-SHEPARD delegates some of its functionality. As any other IRM, IRM-SHEPARD implements the generic HARNESS API.

Running side-by-side with the Orchestrator, the *Autoscaler* component periodically monitors the workload of the virtual FPGAs with elastic properties and triggers their autoscaling according to the workload demand. We described the Orchestrator and Autoscaler and their functionalities in Chapter 3.

In order to send tasks to a virtual FPGA, an application uses the API implemented in the *Maxeler OS* drivers which should be pre-installed on the virtual execution platform before launching the application.

5.3 Enabling Performance-Cost Trade-Offs

Making a good choice of cloud resources to execute arbitrary applications according to user expectations is very difficult because of the large space of heterogeneous resource configurations an application may use. To address this problem, we proposed several

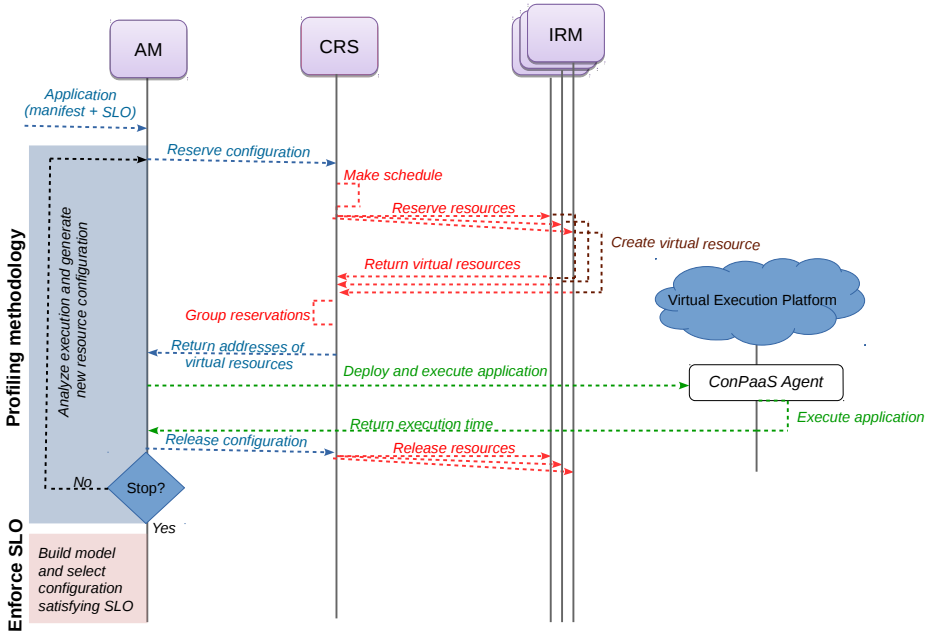


Figure 5.4: Application management.

approaches to enable performance-cost trade-offs through performance profiling. These approaches for profiling arbitrary applications are presented in Chapter 4 of this thesis.

To demonstrate how these approaches may be employed in cloud platforms, we built profilers implementing the *Blackbox* and *Extrapolated profiling* methods, and integrated them in the AM component of the HARNESS platform. The requirements and the internal architecture of the AM are discussed in Chapter 4. We therefore focus here on describing the life-cycle of arbitrary applications and interaction between the components of the HARNESS platform.

The life-cycle of an arbitrary application consists in the following steps depicted in Figure 5.4:

1. An users submits an application to the HARNESS platform using the Frontend services, and provides a manifest file and the SLO. The manifest describes the structure of the application and the resources it requires to run. For example, the manifest can specify the type of resources, their number and internal characteristics that can satisfy the needs of the application. The SLO describes the required execution time or budget for its execution. The manifest and SLO are submitted to the Director, which creates an AM instance in charge of handling this particular application.

2. The AM runs the profiling process in order to identify the optimal configurations providing good performance-cost tradeoffs. It does so by issuing several runs of the application on different resource configurations selected according to the search strategy implemented and reserved through the CRS.

3. The CRS, upon receiving a reservation request from the application manager, activates the resource scheduling process and searches for a good mapping of the requested configuration to available resources. It then contacts the relevant IRMs to create the virtual resources. The response returned to the AM consists of a list of addresses for each virtual resource requested in the configuration. These virtual resources constitute the virtual execution platform used to run the application.

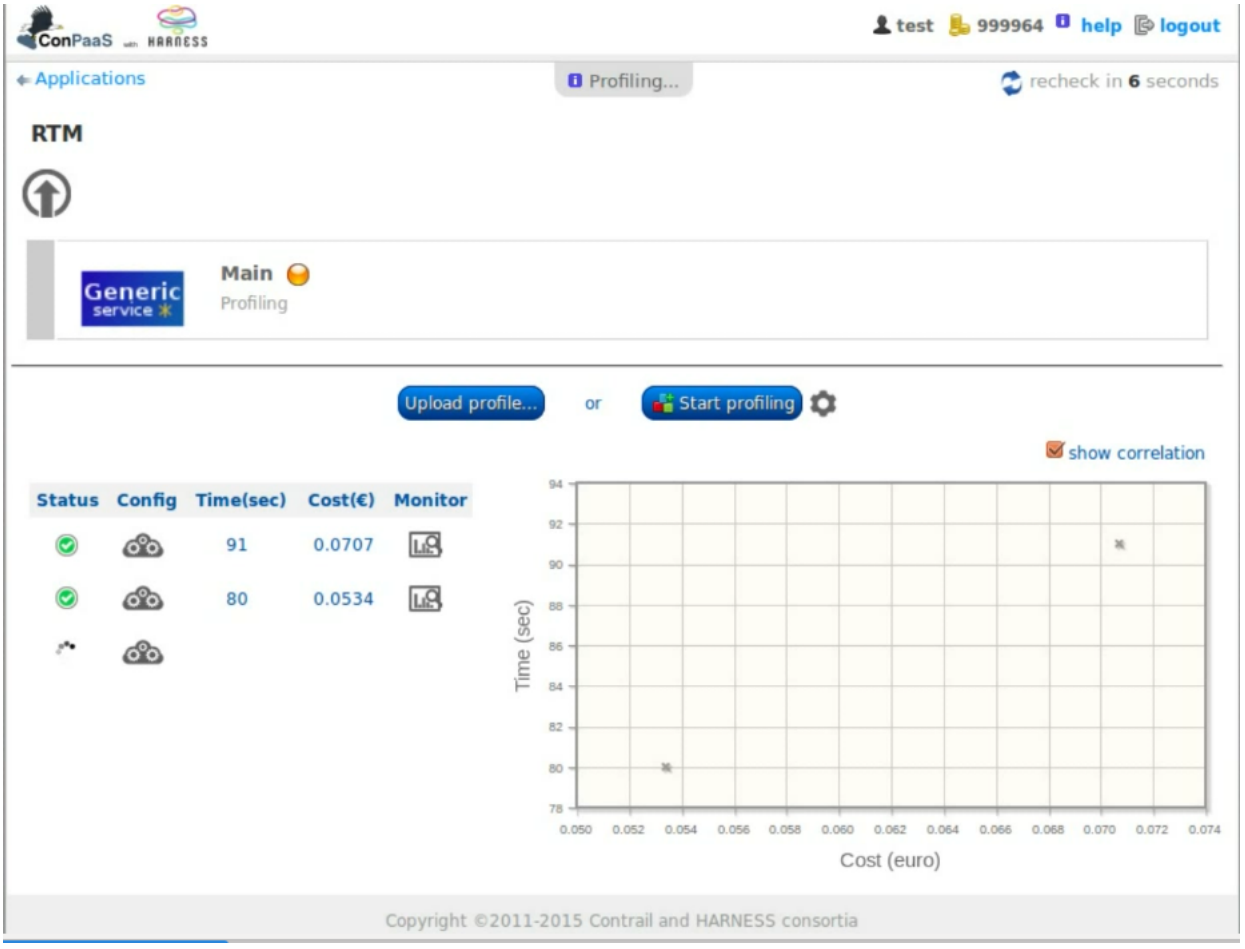


Figure 5.5: Profiling RTM using the HARNESS cloud.

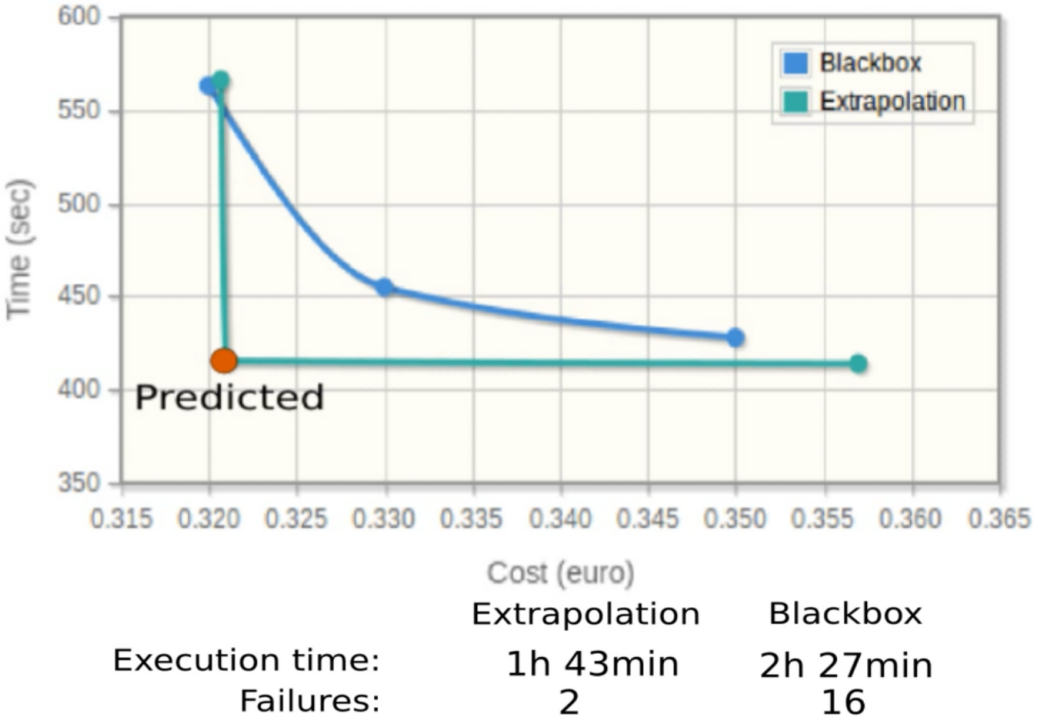


Figure 5.6: Comparison of the Extrapolated and Blackbox profiling results for the RTM application.

4. The AM sets up the environment and deploys the application through commands issued to a ConPaaS agent running in the VM/container instances in the virtual execution platform. Figure 5.5 provides a visualization of the profiling process for the RTM use-case.

5. Once the application has been profiled, based on the resulting performance model, the AM chooses the resource configuration which best satisfies the SLO and issues one request to the CRS for this resource configuration to be provisioned.

Figure 5.6 provides the results of an experiment run on the HARNESS platform. The experiment consisted in running in parallel the blackbox and extrapolated profiling in order to build a performance model for the RTM application having as input a medium sized dataset. The extrapolated profiling uses as a benchmarking dataset a dataset representing 15% of the medium sized dataset. We set blackbox profiling to stop after 15 iterations while extrapolated profiling runs 15 iterations on the benchmarking dataset and 7 iterations on the medium dataset. In this particular experiment, extrapolated profiling discovers better performance-cost trade-offs than the blackbox profiling with roughly 30% decrease in profiling time. This is due to the fairly linear complexity of RTM which provides good predictions based on the correlation between different input datasets. Moreover, bottleneck detection provides also an advantage for extrapolated profiling by enabling it to avoid most of bad configurations.

5.4 Discussion

In this chapter we have demonstrated how our contributions can be integrated in the heterogeneous HARNESS cloud platform. We hope that this implementation may serve as a blueprint for integrating these techniques in other cloud platforms such as Amazon Web Services (AWS). We discuss further several approaches to integrate them in a public cloud platform.

5.4.1 Virtual FPGAs

Amazon may choose to deliver virtual FPGAs to their users in two different forms. First, Amazon may augment their compute resource offering by providing virtual FPGAs as a first-level resource type managed by a service similar to Amazon EC2 (for VMs) or Amazon EBS (for volumes). A very important requirement for the integration of FPGA-based servers in Amazon's infrastructure is to have a low latency interconnect between the VMs' hosts and FPGAs. The Orchestrator would provide the interface implementing the reservation and release of virtual FPGAs. A tenant may therefore reserve a virtual FPGA as a standalone resource and, later, link it to its VMs. She may also specify it as an property of a VM through the EC2 interface in a similar way to an EBS volume. For instance, an EBS volume may be specified in addition to the root device volume when creating a VM in Amazon EC2. Virtual FPGAs can be charged based on the number of physical devices backing it up and their configuration.

Second, a PaaS service similar to the Amazon DynamoDB [151] or Amazon Machine Learning [90] may use shared FPGA groups to offer efficient high-level services for standard algorithms such as Fast Fourier transform, machine learning and video compression. Users would issue regular API calls to this service without necessarily realizing that FPGAs are being used by the service. The Autoscaler would then be in charge with maintaining the quality of service by autoscaling the FPGA groups according to their workloads. This service can be charged on an API call-basis.

5.4.2 Performance Modelling

The integration of the AM implementing our profiling methodologies in a profiling service of Amazon's PaaS would enable the support for performance-cost trade-offs and facilitate the virtual resource selection. The profiling service would function similarly to Amazon Elastic Beanstalk configured with Auto Scaling[152]. The user would submit an application together with its description and expectations, and the profiling service would manage it without further involvement of the user.

The AM would require little modification in order to handle the predefined types exposed by Amazon EC2. However, searching for performance-cost trade-offs implies to be able to evaluate these two metrics on Amazon EC2. While the performance (execution time) can be independently measured, the cost cannot be easily estimated on resource release without support from Amazon EC2. Therefore, the integration of the AM would require for Amazon EC2 to provide through its API on VM release the total cost incurred. Alternatively, another metric being of user-interest which could be measured independently may be selected to replace the cost.

The profiling service can be charged similarly to Amazon Elastic MapReduce (EMR). Amazon EMR incurs additional hourly fees varying with the price of the resources used.

Chapter 6

Conclusions and Perspectives

Contents

Cloud infrastructures provide on-demand access to a large variety of computing devices with different performance and cost. This creates many opportunities for cloud users to run applications having complex resource requirements, starting from large numbers of servers with low-latency interconnects, to specialized devices such as GPUs and FPGAs. However, selecting the right amount and type of resources to run applications according to user expectations is very challenging. First, it is important to optimize the usage of the allocated resources by maximizing their utilization. In particular accelerator-based resources often have low utilization rates because of their usage as CPU co-processors. In cloud environments, the access of applications to accelerators is also limited by the location of accelerators and the available interconnects between CPUs and accelerators. Second, making a good selection of resources is difficult because of the huge search space created by all the possible combinations of resources capable of running an application.

We proposed in this dissertation to enable performance-cost trade-offs for application execution in heterogeneous clouds. This would facilitate the use of the heterogeneous cloud platforms by allowing users to have more control over the execution of their applications as they would be able to specify an expected cost or runtime of their execution. As for the cloud providers, making good use of resources implies reducing the underutilisation of their cloud infrastructures and consequently their operating cost. This has a great impact on the providers' revenue and the availability of resources for future reservations.

The work described in this thesis addresses the challenge of enabling performance-cost trade-offs by focusing on the following objectives:

- Improving the utilisation of FPGA-based accelerators by means of virtualization;
- Automating the selection of heterogeneous resources for executing applications under user-provided SLOs;
- Demonstrating how these technologies can be implemented in heterogeneous cloud platforms.

6.1 Contributions

The contributions of this dissertation are as follows:

Improving FPGA utilization by means of virtualization. Field-Programmable Gate Arrays are integrated circuits that can be reprogrammed at runtime to implement any given circuit design. They are increasingly getting used as accelerators in cloud infrastructures to speed up specific computations. However, they are not yet put at the disposal of cloud users which could also benefit from them. An important challenge in making FPGA first-class cloud resources is to share them among multiple tenants and to maximize their utilization. To address this challenge, we propose a cloud infrastructure supporting FPGAs in a multi-tenant environment. Instead of considering FPGAs as local accelerator devices attached to a single server, we co-locate a number of FPGAs in dedicated machines accessed through a low-latency network. This setup increases availability by enabling a full many-to-many mapping between FPGAs and the clients accessing them. This allows FPGAs to be used by applications located in different servers and even to be shared between different applications. We define a *virtual FPGA* as an elastic group of physical FPGAs configured with the same circuit design. A virtual FPGA is exposed as a first-class cloud resource instead of being specified as a property of a virtual machine/container. Our experiments showed that a virtual FPGA resource incurs a low overhead in the order of 0.09 ms per submitted task. Addition and removal of FPGAs

from the group backing up the virtual FPGA is done seamlessly, without disturbing the computation. To maximize the utilization of an FPGA-based cloud infrastructure, we propose an algorithm which automatically resizes virtual FPGAs to maximise resource utilisation and reduce user-perceived computation latencies. When dealing with challenging workloads, our autoscaling algorithm increases resource utilisation compared to a static resource allocation while reducing the average task execution latency. In our experiments, the autoscaling increased the average resource utilization from 52% to 61% and reduced the average batch execution latency by 61% while using the same total number of resources.

Automating heterogeneous resource selection. Selecting the right number and type of resources to use to execute arbitrary applications according to user-specific requirements is very difficult. The reason why is that heterogeneous clouds are offering large and diverse pools of resources which generate a huge number of possible resource configurations to be used. To address this issue, we propose a number of profiling methodologies to find optimal resource configurations providing good performance-cost trade-offs. The basic approach, entitled "blackbox profiling", uses a global optimization algorithm to search for optimal resource configurations. It is a very general method, since it can automatically profile arbitrary cloud applications written using any programming language or framework, and requiring any set of cloud resources. On the other hand, it may require a significant number of profiling executions before discovering configurations providing good performance-cost trade-offs.

Therefore, using the blackbox approach to profile complex applications where the space of all possible resource configurations is very large may induce significant time and cost overheads. To address this, we propose to extend the blackbox profiling with a "whitebox approach" which makes use of resource utilization data to reduce the number of iterations performed until good configurations are reached. Although this approach provides improvements over the pure blackbox approach, running applications having a long runtime may still induce significant time and cost overheads.

To further reduce these overheads, we propose a complementary profiling approach named "extrapolated profiling." This approach relies on exploiting small input datasets with shorter runtime to explore the resource configuration space and reuse selected configurations for processing larger input datasets. We then use the correlation between the performance on the two datasets as a basis to predict the performance of the large input dataset on untested configurations. This approach minimizes profiling cost and runtime by performing less iterations on large datasets and by avoiding most of the resource configurations that may fail to run successfully.

An extensive evaluation of the profiling approaches based on two HPC applications demonstrates the benefits provided of each approach. We studied different aspects such as the convergence speed to the optimal resource configurations, the satisfaction rate of different predefined SLOs, the runtime and cost overhead of the profiling process; and the detection of failing configurations (bottlenecks). In our experiments, the blackbox+whitebox profiling performed better than the agnostic blackbox approach with a maximum failure rate of 7% - 30% to satisfy an user-given service-level objective. A further comparison between the blackbox-whitebox and extrapolated profiling approach showed that for applications with an estimable correlation of their complexity, extrapolated profiling performs better with a 10-20% decrease of the profiling time, a 30-35% decrease in financial cost time and 15-20% less failures.

Prototype integration in a heterogeneous cloud platform. The solutions we propose for FPGA virtualization and heterogeneous resource selection are designed to be easily implemented in heterogeneous cloud platforms. We demonstrate how this can be

done by integrating the prototypes of our solutions in a heterogeneous cloud platform developed in the context of HARNESS European project. We address also the problem of their integration in a public cloud system such as Amazon Web Services.

6.2 Perspectives

Our contributions present several interesting research directions for future work. We describe further the short- and long-term research perspectives of our contributions.

6.2.1 Short-term Perspectives

Impact of interconnect virtualization on data-intensive accelerated applications. The virtual FPGAs implemented as elastic groups of FPGAs enable accelerated applications to have better control over their performance. However, it would be very interesting to extend our evaluation of virtual FPGAs to study the impact of the network virtualization on the task runtime of data-intensive accelerated applications. This would allow us to identify the processing limits of the FPGA-server when taking into consideration the network performance in a multi-tenant scenario. Slow data transfer in comparison with fast FPGA processing rates would be the main challenge in any cloud infrastructure encompassing FPGA-servers.

Application profiling with predefined instance types. The profiling methodologies we proposed assume that IaaS clouds can provide custom resource configurations within the limits of their infrastructure. This is an important aspect in optimizing the performance, cost and resource utilization. However, current IaaS clouds provide a limited set of predefined instance types. It would be interesting to evaluate the impact on the profiling profiling process when using predefined instance types. We need to choose instance types to test such that we minimize their underutilization during application execution. The difficulty lies in the limited vertical scaling which does not allow the addition or removal of specific amounts of resources from a virtual instance determined by their utilization.

Exploring input datasets inter-dependencies. We proposed extrapolated profiling as a methodology to capture the relationship between a benchmarking and a production dataset. This allows us to test new resource configurations using the small benchmarking dataset and extrapolate the obtained results for the production dataset. It would be an interesting research direction to extend the extrapolated methodology to provide performance models for new datasets without having to run them on any configuration. This requires to determine the relationship between different datasets and use this relationship to extrapolate the performance of new datasets. This strategy would provide benefits such as a high decrease in cost and runtime profiling overheads.

6.2.2 Long-term Perspectives

PaaS system for developing accelerated applications. FPGA circuit design is usually done using standard hardware design languages as VHDL and Verilog. Developers are usually limited in exploiting FPGAs due to the need to think like a hardware engineer. However, tools and frameworks are being developed to support implementation of circuit description in higher-level programming languages, compilation to bitfiles, generating the interface between CPU host and the FPGA. In order to make use of the functions implemented in an FPGA, an application must include in its code the interface generated after

the compilation to bitfiles. We consider it would be interesting to further democratize the use of FPGAs by offering PaaS systems to support the development and compilation of accelerated applications.

Resource utilisation profiles for application runtime prediction. Analysing the pattern of resource utilization from one execution to another may provide new means to refine the resource selection. For example, an application may spend the first 30% of its total runtime to load data in memory, then 60% to process it and the rest to write the results to disk. Automatically identifying this utilisation pattern and correlating the runtime ratio for each phase may be an interesting approach to predict the remaining runtime after the first phase of the execution. Based on this, we could optimize resource utilization and the running costs by allocating resources only when they are needed during the execution of an application.

Appendix A

Application Manifest

A.1 RTM Manifest

```
{
  "Name": "RTM",
  "Author": "User",
  "Modules": [
    {
      "ModuleName": "Main",
      "Parameters": [
        {
          "Var": "%arg1",
          "Name": "input",
          "Values": [ "rtm_parameters_medium"]
        }
      ],
      "Implementations": [
        {
          "ImplementationName": "RTM-CPU",

          "Resources": {
            "Groups": [
              {
                "GroupID": "id0",
                "Role": "MASTER",
                "Type": "Machine",
                "NumInstances": {
                  "Value": 1
                },
                "Attributes": {
                  "Memory": {
                    "Var": "%master_mem",
                    "Values": [
                                        2048,
                                        4096,
                                        6144,
                                        8192,
```

```
                                    16384,
                                    32768
                  ]
                },
                "Cores": {
                  "Var": "%master_cores",
                  "Range": [
                    2,
                    16
                  ]
                }
              }
            }
          ],
          "Constraints": []
        },
        "GlobalConstraints": [],
        "EnvironmentVars": { "OMP_NUM_THREADS": "%master_cores" },
        "Tarball": "http://cloud.fr/apps/rtm/archive.tar.gz",
        "DeploymentArgs": "",
        "ExecutionArgs": "%arg1"
      }
    ]
  }
 ]
}
```

A.2 DeltaMerge Manifest

```
{
  "Name": "DM",
  "Author": "User",
  "Modules": [
    {
      "ModuleName": "Main",
      "ModuleType": "generic",
      "Online": 0,
      "Parameters": [
        {
          "Var": "%arg1",
          "Name": "input",
          "Values": [ "http://cloud.fr/apps/deltamerge/tabel-size-0.tar.gz"]
       }
      ],
      "Implementations": [
        {
          "ImplementationName": "DM",

          "Resources": {
```

```
            "Groups": [
              {
                "GroupID": "id0",
                "Role": "MASTER",
                "Type": "Machine",
                "NumInstances": {
                  "Value": 1
                },
                "Attributes": {
                  "Memory": {
                    "Var": "%master_mem",
                    "Values": [
                            6144,
                            8192,
                            16384,
                            32768
                    ]
                  },
                  "Cores": {
                    "Var": "%master_cores",
                    "Range": [
                      1,
                      16
                    ]
                  }
                }
              }
            ],
            "Constraints": []
          },
          "GlobalConstraints": [],
          "EnvironmentVars": {},
          "Tarball": "http://cloud.fr/apps/deltamerge/archive.tar.gz",
          "DeploymentArgs": "%arg1",
          "ExecutionArgs": ""
        }
      ]
    }
  ]
}
```

A.3 Manifest Template

```
{
 /* Application Name */
    "Name": "Hello World",
    "Author": "User",

 /* The application consists of one or more modules
```

```
(services that are required to run the application) */
"Modules": [  {

    "ModuleName": "Main",
  /* The "Parameters" field contains the list of input parameters
  use the AM to evaluate their impact at runtime */
     "Parameters": [{
        "Var": "%arg1",
        "Name": "<parameter name>",
        "Type": "STR",
        "Values": ["<parameter value1>", ...]
     }],
  /* List of implementations that can realise the module. Only one
  is chosen at a given time */
    "Implementations": [{
            "ImplementationName": "ImplementationName",

            /* The "Resources" field specifies groups of resources associated
            to a particular role, and placement constraints */
            "Resources": {
                "Groups": [{
                    "GroupID": "id0",

                  /* Role is a label associated to one or more groups of
                  resources, which can be referenced during the deployment
                  phase */
                    "Role": "MASTER",

                  /* Resource Type */
                    "Type": "Machine",

                  /* Number of resource instances required. For a static number
                  of instances use the field { Value : <number>}, otherwise, for
                  a dynamic number of instances, use
                  { Var : <var_id>, Values/Range :[...]} */
                    "NumInstances": {
                        "Value": 1
                    },

                    /* Attributes used for allocation requests.
                       For each attribute, a value or range of values can be
                       specified.
                    */
                    "Attributes": {
                        "Memory": {
                            "Var": "%master_ram",
                            "Values": [1024, 2048, 4096, 8192]
                        },
                        "Cores": {
```

```
                        "Var": "%master_cores",
                        "Range": [1, 7]
                    }
                }
            }, {
                "GroupID": "id1",
                "Role": "SLAVE",
                "Type": "Machine",
                "NumInstances": {
                    "Value": 1
                },
                "Attributes": {
                    "Memory": {
                        "Var": "%master_ram",
                        "Values": [1024, 2048, 4096, 8192]
                    },
                    "Cores": {
                        "Var": "%master_cores",
                        "Range": [1, 7]
                    }
                }
            }, {
                "GroupID": "id2",
                "Role": "None",
                "Type": "ResourceX",
                "NumInstances": {
                    "Value": 1
                },
                "Attributes": {
                    /* ... */
                }
            }],

            /* Placement constraints between groups of resources  */
            "Distances": [{
                "Source": "id0",
                "Target": "id1",
                "Constraints": ["hops < 1"]
            }]
        },

        /* Specifies global dependencies between groups of resources,
         input parameters, etc.*/
          "GlobalConstraints": ["%master_ram > 1024 * %master_cores", ...],

        /* Environment variables to be set when deploying and launching
         this implementation */
          "EnvironmentVars": {
            "MASTER_IP": "%master_address",
```

```
                    ...
                },

                /* URL where the application tarball can be downloaded from
                 (Mandatory) */
                "Tarball": "<tarball URL>",

                /* Input arguments for deployment scripts */
                "DeploymentArgs": "<%arg1 %arg2 static_arg ...>",

                /* Input arguments for execution scripts */
                "ExecutionArgs": "<%arg1  %arg2 static_arg ...>"
            }
            ...
        ]
    }]
}
```

Bibliography

[1] S. Crago, K. Dunn, P. Eads, L. Hochstein, D.-I. Kang, M. Kang, D. Modium, K. Singh, J. Suh, and J. Walters, "Heterogeneous Cloud Computing," in *Proceedings of the 2011 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 378–385, Sept 2011.

[2] F. Chen, Y. Shan, Y. Zhang, Y. Wang, H. Franke, X. Chang, and K. Wang, "Enabling FPGAs in the Cloud," in *Proceedings of the 11th ACM Conference on Computing Frontiers*, CF '14, (New York, NY, USA), pp. 3:1–3:10, ACM, 2014.

[3] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, (Piscataway, NJ, USA), pp. 13–24, IEEE Press, 2014.

[4] FPGA in future datacenters, "Intel's acquisition of Altera." `http://intelacquiresaltera.transactionannouncement.com/wp-content/uploads/2015/06/Investor-Deck.pdf`. Accessed: 2016-03-30.

[5] Jeff Barr, "Choosing the Right EC2 Instance Type for Your Application." `http://aws.amazon.com/blogs/aws/choosing-the-right-ec2-instance-type-for-your-application/`. Accessed: 2016-03-30.

[6] L. Eeckhout, H. Vandierendonck, and K. D. Bosschere, "Quantifying the Impact of Input Data Sets on Program Behavior and its Applications," *Journal of Instruction-Level Parallelism*, vol. 5, pp. 1–33, 2003.

[7] W. C. Hsu, H. Chen, P. C. Yew, and D.-Y. Chen, "On the Predictability of Program Behavior Using Different Input Data Sets," in *Proceedings of the Sixth Annual Workshop on Interaction Between Compilers and Computer Architectures*, INTERACT '02, (Washington, DC, USA), pp. 45–53, IEEE Computer Society, 2002.

[8] RTM, "RTM implementations." `http://www.harness-project.eu/wp-content/uploads/2014/02/HARNESS-D2.3.pdf`. Accessed: 2016-03-30.

[9] Amazon GPU, "Amazon Cluster GPU Instance." `https://aws.amazon.com/ec2/instance-types/`. Accessed: 2016-03-30.

[10] Intel, "Intel's acquisition of Altera." `http://newsroom.intel.com/community/intel_newsroom/blog/2015/12/28/intel-completes-acquisition-of-altera`. Accessed: 2016-03-30.

[11] C. Bailey Lee, Y. Schwartzman, J. Hardy, and A. Snavely, "Are User Runtime Estimates Inherently Inaccurate?," in *Job Scheduling Strategies for Parallel Processing* (D. Feitelson, L. Rudolph, and U. Schwiegelshohn, eds.), vol. 3277 of *Lecture Notes in Computer Science*, pp. 253–263, Springer Berlin Heidelberg, 2005.

[12] W. Tang, N. Desai, D. Buettner, and Z. Lan, "Job scheduling with adjusted runtime estimates on production supercomputers," *Journal of Parallel and Distributed Computing*, vol. 73, no. 7, pp. 926 – 938, 2013. Best Papers: International Parallel and Distributed Processing Symposium (IPDPS) 2010, 2011 and 2012.

[13] A. W. Mu'alem and D. G. Feitelson, "Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, pp. 529–543, June 2001.

[14] HARNESS, "HARNESS." `http://www.harness-project.eu//`. Accessed: 2016-03-30.

[15] Microsoft, "The Economics of the Cloud." `http://news.microsoft.com/download/archived/presskits/cloud/docs/the-economics-of-the-cloud.pdf`. Accessed: 2016-03-30.

[16] Rackspace, "The Economics of Cloud Comuputing." `http://broadcast.rackspace.com/hosting_knowledge/whitepapers/Cloudonomics-The_Economics_of_Cloud_Computing.pdf`. Accessed: 2016-03-30.

[17] J. Dejun, G. Pierre, and C.-H. Chi, "EC2 Performance Analysis for Resource Provisioning of Service-oriented Applications," in *Proceedings of the 2009 International Conference on Service-oriented Computing*, ICSOC/ServiceWave'09, (Berlin, Heidelberg), pp. 197–207, Springer-Verlag, 2009.

[18] S. Ostermann, A. Iosup, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema, *Cloud Computing: First International Conference, CloudComp 2009 Munich, Germany, October 19–21, 2009 Revised Selected Papers*, ch. A Performance Analysis of EC2 Cloud Computing Services for Scientific Computing, pp. 115–131. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010.

[19] Titan, "Titan." `https://www.olcf.ornl.gov/titan/`. Accessed: 2016-03-30.

[20] Janus2, "JanusII." `http://www.janus-computer.com/janusII`. Accessed: 2016-03-30.

[21] IB, "InfiniBand." `http://www.infinibandta.org/content/pages.php?pg=technology_overview`. Accessed: 2016-03-30.

[22] P. Zaspel and M. Griebel, "Massively Parallel Fluid Simulations on Amazon's HPC Cloud," in *Proceedings of the 2011 First International Symposium on Network Cloud Computing and Applications (NCCA)*, Nov 2011.

[23] M. Pawlish, A. S. Varde, and S. A. Robila, "Analyzing utilization rates in data centers for optimizing energy management," in *Proceedings of the 2012 International Green Computing Conference (IGCC)*, pp. 1–6, June 2012.

[24] Amazon White Paper, "How AWS Pricing Works." `https://media.amazonwebservices.com/AWS_Pricing_Overview.pdf`, 2015. Accessed: 2016-03-30.

[25] M. Aggar, "The it energy efficiency imperative.." White paper, 2011.

[26] K. Sunil Rao and P. Santhi Thilagam, "Heuristics Based Server Consolidation with Residual Resource Defragmentation in Cloud Data Centers," *Future Gener. Comput. Syst.*, vol. 50, pp. 87–98, Sept. 2015.

[27] Sharon Wagner, "The Great Hope of Cloud Economics and the Over-provisioning Epidemic." `https://www.cloudyn.com/wp-content/uploads/2013/06/The_Great_Hope_of_Cloud_Economics.pdf`. Accessed: 2016-03-30.

[28] R. W. Ahmad, A. Gani, S. H. A. Hamid, M. Shiraz, A. Yousafzai, and F. Xia, "A Survey on Virtual Machine Migration and Server Consolidation Frameworks for Cloud Data Centers," *Journal of Network and Computer Applications*, vol. 52, pp. 11–25, June 2015.

[29] R. Zhang, R. Routray, D. M. Eyers, D. Chambliss, P. Sarkar, D. Willcocks, and P. Pietzuch, "IO Tetris: Deep Storage Consolidation for the Cloud via Fine-Grained Workload Analysis," in *Proceedings of the 2011 IEEE International Conference on Cloud Computing (CLOUD)*, pp. 700–707, July 2011.

[30] F. Hermenier, X. Lorca, J.-M. Menaud, G. Muller, and J. Lawall, "Entropy: A Consolidation Manager for Clusters," in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '09, (New York, NY, USA), pp. 41–50, ACM, 2009.

[31] S. Srikantaiah, A. Kansal, and F. Zhao, "Energy Aware Consolidation for Cloud Computing," in *Proceedings of the 2008 Conference on Power Aware Computing and Systems*, HotPower'08, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2008.

[32] Y. Amannejad, D. Krishnamurthy, and B. Far, "Detecting performance interference in cloud-based web services," in *Proceedings of the 2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pp. 423–431, May 2015.

[33] Bill Bitner, Susan Greenlee , "z/VM A Brief Review of Its 40 Year History ." `http://www.vm.ibm.com/vm40hist.pdf`. Accessed: 2016-03-30.

[34] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, SOSP '03, (New York, NY, USA), pp. 164–177, ACM, 2003.

[35] VMware, "vSphere Hypervisor." Available at `https://www.vmware.com/products/vsphere-hypervisor`. Accessed: 2016-03-30.

[36] Microsoft, "Hyper-V Hypervisor." `https://technet.microsoft.com/library/hh831531.aspx`. Accessed: 2016-03-30.

[37] KVM, "Kernel-based Virtual Machine." `http://www.linux-kvm.org/`.

[38] Oracle, "VirtualBox." Available at `https://www.virtualbox.org/`. Accessed: 2016-03-30.

[39] Geert Jansen, "Binary Translation." `https://www.ravellosystems.com/blog/nested-virtualization-with-binary-translation/`. Accessed: 2016-03-30.

[40] Xen Open Source Project, "Paravirtualization." http://wiki.xen.org/wiki/Paravirtualization_(PV). Accessed: 2016-03-30.

[41] K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, (New York, NY, USA), pp. 2–13, ACM, 2006.

[42] Matthew Gillespie, "Intel EPTand VT-d." https://software.intel.com/en-us/articles/best-practices-for-paravirtualization-enhancements-from-intel-virtualization- Accessed: 2016-03-30.

[43] Jason (blog), "AMD Rapid Virtualization Indexing (RVI)." http://www.boche.net/blog/index.php/2009/03/08/rapid-virtualization-indexing-rvi/. Accessed: 2016-03-30.

[44] VMware, "Intel EPT." http://www.vmware.com/pdf/Perf_ESX_Intel-EPT-eval.pdf. Accessed: 2016-03-30.

[45] VMware, "AMD RVI." http://www.cse.iitd.ernet.in/~sbansal/csl862-virt/2010/readings/RVI_performance.pdf. Accessed: 2016-03-30.

[46] J. Ahn, S. Jin, and J. Huh, "Revisiting hardware-assisted page walks for virtualized systems," in *Proceedings of the 2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pp. 476–487, June 2012.

[47] X. Wang, J. Zang, Z. Wang, Y. Luo, and X. Li, "Selective hardware/software memory virtualization," in *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '11, (New York, NY, USA), pp. 217–226, ACM, 2011.

[48] J. Li, Q. Wang, D. Jayasinghe, J. Park, T. Zhu, and C. Pu, "Performance overhead among three hypervisors: An experimental study using hadoop benchmarks," in *Proceedings of the 2013 IEEE International Congress on Big Data (BigData Congress)*, pp. 9–16, June 2013.

[49] J. Hwang, S. Zeng, F. y. Wu, and T. Wood, "A component-based performance comparison of four hypervisors," in *Proceedings of the 2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*, pp. 269–276, May 2013.

[50] Namespaces, "Namespaces." https://lwn.net/Articles/531114/. Accessed: 2016-03-30.

[51] Linux containers, "Linux LXC." https://linuxcontainers.org/. Accessed: 2016-03-30.

[52] Docker, "Docker." https://www.docker.com/. Accessed: 2016-03-30.

[53] Virtuozzo, "OpenVZ." https://openvz.org/Main_Page. Accessed: 2016-03-30.

[54] Linux VServer, "Linux VServer." http://linux-vserver.org/Welcome_to_Linux-VServer.org. Accessed: 2016-03-30.

[55] Matteo Riondato, "BSD Jails." https://www.freebsd.org/doc/handbook/jails.html. Accessed: 2016-03-30.

[56] Docker, "Docker Swarm." `https://docs.docker.com/swarm/`. Accessed: 2016-03-30.

[57] CoreOS, "Fleet." `https://coreos.com/fleet/`. Accessed: 2016-03-30.

[58] Kubernetes, "Kubernetes." `http://kubernetes.io/`. Accessed: 2016-03-30.

[59] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *Proceedings of the 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 171–172, March 2015.

[60] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. F. D. Rose, "Performance Evaluation of Container-Based Virtualization for High Performance Computing Environments," in *21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2013, Belfast, United Kingdom, February 27 - March 1, 2013*, pp. 233–240, 2013.

[61] M. Gomes Xavier, M. Veiga Neves, and C. Fonticielha de Rose, "A Performance Comparison of Container-Based Virtualization Systems for MapReduce Clusters," in *Proceedings of the 2014 22nd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP),*, pp. 299–306, Feb 2014.

[62] R. Morabito, J. Kjallman, and M. Komu, "Hypervisors vs. lightweight virtualization: A performance comparison," in *Proceedings of the 2015 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 386–393, March 2015.

[63] B. Van Essen, C. Macaraeg, M. Gokhale, and R. Prenger, "Accelerating a Random Forest Classifier: Multi-Core, GP-GPU, or FPGA?," in *Proceedings of the 2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, FCCM '12, (Washington, DC, USA), pp. 232–239, IEEE Computer Society, 2012.

[64] S. Mittal and J. S. Vetter, "A Survey of Methods for Analyzing and Improving GPU Energy Efficiency," *ACM Comput. Surv.*, vol. 47, pp. 19:1–19:23, Aug. 2014.

[65] V. Kindratenko, J. Enos, G. Shi, M. Showerman, G. Arnold, J. Stone, J. Phillips, and W. M. Hwu, "GPU clusters for high-performance computing," in *Proceedings of the 2009 IEEE International Conference on Cluster Computing and Workshops (CLUSTER)*, pp. 1–8, Aug 2009.

[66] J. P. Walters, A. J. Younge, D.-I. Kang, K.-T. Yao, M. Kang, S. P. Crago, and G. C. Fox, "GPU-Passthrough Performance: A Comparison of KVM, Xen, VMWare ESXi, and LXC for CUDA and OpenCL Applications," in *Proceedings of the 7th IEEE International Conference on Cloud Computing (CLOUD)*, June 2014.

[67] Amazon Web Services, "EC2: Elastic Compute Cloud." `http://aws.amazon.com/ec2/`. Accessed: 2016-03-30.

[68] Y. Suzuki, S. Kato, H. Yamada, and K. Kono, "GPUvm: why not virtualizing GPUs at the hypervisor?," in *Proceedings of the USENIX Annual Technical Conference*, June 2014.

[69] W. Wang, M. Bolic, and J. Parri, "pvFPGA: Accessing an FPGA-based Hardware Accelerator in a Paravirtualized Environment," in *Proceedings of the 9th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '13, (Piscataway, NJ, USA), pp. 10:1–10:9, IEEE Press, 2013.

[70] L. Shi, H. Chen, J. Sun, and K. Li, "vCUDA: GPU-accelerated high-performance computing in virtual machines," *IEEE Transactions on Computers*, vol. 61, no. 6, pp. 804–816, 2012.

[71] G. Giunta, R. Montella, G. Agrillo, and G. Coviello, "A GPGPU transparent virtualization component for high performance computing clouds," in *Proceedings of the International Euro-Par Conference on Parallel Processing*, 2010.

[72] M. Gottschlag, M. Hillenbrand, J. Kehne, J. Stoess, and F. Bellosa, "LoGV: Low-overhead GPGPU Virtualization," in *Proceedings of the International Workshop on Frontiers of Heterogeneous Computing*, Nov. 2013.

[73] D. W. Page, "Dynamic data re-programmable PLA." `http://www.google.com/patents/US4524430`, June 1985. U.S. patent US 4524430 A.

[74] D. F. Bacon, R. Rabbah, and S. Shukla, "FPGA Programming for the Masses," *ACM Queue*, vol. 11, Feb. 2013.

[75] J. M. Cardoso and P. C. Diniz, *Compilation Techniques for Reconfigurable Architectures*. Springer, 2009.

[76] P. Grigoraş, X. Niu, J. G. Coutinho, W. Luk, J. Bower, and O. Pell, "Aspect Driven Compilation for Dataflow Designs," in *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, June 2013.

[77] Maxeler Technologies, "Maxeler AppGallery." `http://appgallery.maxeler.com/`. Accessed: 2016-03-30.

[78] Xilinx Inc., "Applications." `http://www.xilinx.com/applications.html`. Accessed: 2016-03-30.

[79] Glen K. Lockwood, "InfiniBand SR-IOV Evaluation." `http://glennklockwood.blogspot.fr/2013/12/high-performance-virtualization-sr-iov_14.html`. Accessed: 2016-03-30.

[80] Intel, "PCI-SIG SR-IOV Primer: An Introduction to SR-IOV Technology." `http://www.intel.com/content/www/us/en/pci-express/pci-sig-sr-iov-primer-sr-iov-technology-paper.html`. Accessed: 2016-03-30.

[81] IOV, "IOV." `https://pcisig.com/specifications/iov`. Accessed: 2016-03-30.

[82] Lockwood, Glenn K. and Tatineni, Mahidhar and Wagner, Rick, "SR-IOV: Performance Benefits for Virtualized Interconnects." `https://www.sdsc.edu/assets/docs/SR-IOV-XSEDE14.pdf`. Accessed: 2016-03-30.

[83] J. Jose, M. Li, X. Lu, K. Kandalla, M. Arnold, and D. Panda, "SR-IOV Support for Virtualization on InfiniBand Clusters: Early Experience," in *Proceedings of the 2013 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2013.

[84] C. Reaño, F. Silla, G. Shainer, and S. Schultz, "Local and Remote GPUs Perform Similar with EDR 100G InfiniBand," in *Proceedings of the Industrial Track of the 16th International Middleware Conference*, Middleware Industry '15, (New York, NY, USA), pp. 4:1–4:7, ACM, 2015.

[85] S. Byma, J. G. Steffan, H. Bannazadeh, A. L. Garcia, and P. Chow, "FPGAs in the Cloud: Booting Virtualized Hardware Accelerators with OpenStack," in *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines*, May 2014.

[86] J. Weerasinghe, F. Abel, C. Hagleitner, and A. Herkersdorf, "Enabling FPGAs in Hyperscale Data Centers," in *Proceedings of the 2015 IEEE International Conference on Cloud and Big Data Computing (CBDCom)*, 2015.

[87] occi-wg.org, "OCCI." `http://occi-wg.org/`. Accessed: 2016-03-30.

[88] Amazon EC2 API, "Amazon EC2 API." `http://aws.amazon.com/documentation/ec2/`. Accessed: 2016-03-30.

[89] A. W. Services, "Amazon elastic mapreduce." `http://aws.amazon.com/elasticmapreduce/`. Accessed: 2016-03-30.

[90] Amazon Web Services, "Amazon Machine Learning." `http://aws.amazon.com/aml/`. Accessed: 2016-03-30.

[91] Google AppEngine, "Google AppEngine." `https://cloud.google.com/appengine/docs`. Accessed: 2016-03-30.

[92] Microsoft, "Windows azure web services." `http://azure.microsoft.com/en-us/services/`. Accessed: 2016-03-30.

[93] Rackspace, "Cloud Sites." `https://www.rackspace.com/cloud/sitessss`. Accessed: 2016-03-30.

[94] Heroku, "Heroku." `http://docs.cloudfoundry.org/buildpacks/`. Accessed: 2016-03-30.

[95] G. Pierre and C. Stratan, "ConPaaS: A Platform for Hosting Elastic Cloud Applications," *IEEE Internet Computing*, vol. 16, no. 5, pp. 88–92, 2012.

[96] CloudFoundry, "CloudFoundry buildpacks." `http://docs.cloudfoundry.org/buildpacks/`. Accessed: 2016-03-30.

[97] AppScale, "AppScale." `https://github.com/AppScale/appscale`. Accessed: 2016-03-30.

[98] Amazon EC2, "Amazon EC2 SLA." `http://aws.amazon.com/ec2/sla/`. Accessed: 2016-03-30.

[99] Rackspace, "Rackspace SLA." https://www.rackspace.com/information/legal/cloud/sla. Accessed: 2016-03-30.

[100] A. V. Papadopoulos, "Design and performance guarantees in cloud computing: challenges and opportunities," in *10th International Workshop on Feedback Computing*, 2015.

[101] Amazon EMR Documentation, "Amazon EMR API." http://docs.aws.amazon.com/ElasticMapReduce/latest/API/API_RunJobFlow.html. Accessed: 2016-03-30.

[102] Amazon Elastic BeanStalk Documentation, "Amazon Elastic BeanStalk API." http://docs.aws.amazon.com/elasticbeanstalk/latest/api/API_EnvironmentResourceDescription.html. Accessed: 2016-03-30.

[103] AppEngine Documentation, "AppEngine Applications." https://cloud.google.com/appengine/docs/python/scaling#introduction_to_instances. Accessed: 2016-03-30.

[104] Amazon AWS, "Auto Scaling." https://aws.amazon.com/autoscaling/. Accessed: 2016-03-30.

[105] Google AppEngine, "Autoscaler." https://cloud.google.com/compute/docs/autoscaler/. Accessed: 2016-03-30.

[106] Rackspace, "Rackspace Auto Scale tips and how-to's." https://support.rackspace.com/how-to/rackspace-auto-scale-tips-and-how-tos/. Accessed: 2016-03-30.

[107] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi, "An Analytical Model for Multi-tier Internet Services and Its Applications," in *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '05, (New York, NY, USA), pp. 291–302, ACM, 2005.

[108] Q. Zhang, L. Cherkasova, and E. Smirni, "A Regression-Based Analytic Model for Dynamic Resource Provisioning of Multi-Tier Applications," in *Proceedings of the Fourth International Conference on Autonomic Computing*, ICAC '07, (Washington, DC, USA), pp. 27–, IEEE Computer Society, 2007.

[109] J. Bi, Z. Zhu, R. Tian, and Q. Wang, "Dynamic Provisioning Modeling for Virtualized Multi-tier Applications in Cloud Data Center," in *Proceedings of the 2010 IEEE 3rd International Conference on Cloud Computing (CLOUD)*, pp. 370–377, July 2010.

[110] L. Lu, X. Zhu, R. Griffith, P. Padala, A. Parikh, P. Shah, and E. Smirni, "Application-driven dynamic vertical scaling of virtual machines in resource pools," in *Proceedings of the 2014 IEEE Network Operations and Management Symposium (NOMS)*, pp. 1–9, May 2014.

[111] B. J. Watson, M. Marwah, D. Gmach, Y. Chen, M. Arlitt, and Z. Wang, "Probabilistic Performance Modeling of Virtualized Resource Allocation," in *Proceedings of the 7th International Conference on Autonomic Computing*, ICAC '10, (New York, NY, USA), pp. 99–108, ACM, 2010.

[112] M. Turowski and A. Lenk, *Service-Oriented Computing - ICSOC 2014 Workshops: WESOA; SeMaPS, RMSOC, KASA, ISC, FOR-MOVES, CCSA and Satellite Events, Paris, France, November 3-6, 2014, Revised Selected Papers*, ch. Vertical Scaling Capability of OpenStack, pp. 351–362. Cham: Springer International Publishing, 2015.

[113] N. Mi, G. Casale, L. Cherkasova, and E. Smirni, "Burstiness in Multi-tier Applications: Symptoms, Causes, and New Models," in *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '08, (New York, NY, USA), pp. 265–286, Springer-Verlag New York, Inc., 2008.

[114] T. H. Beach, O. F. Rana, and N. J. Avis, "Integrating Acceleration Devices Using CometCloud," in *Proceedings of the 1st ACM Workshop on Optimization Techniques for Resources Management in Clouds*, ORMaCloud '13, (New York, NY, USA), pp. 17–24, ACM, 2013.

[115] A. Li, X. Yang, S. Kandula, and M. Zhang, "Cloudcmp: Shopping for a cloud made easy," in *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, (Berkeley, CA, USA), pp. 5–5, USENIX Association, 2010.

[116] N. Vasić, D. Novaković, S. Miučin, D. Kostić, and R. Bianchini, "DejaVu: Accelerating Resource Allocation in Virtualized Environments," in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, (New York, NY, USA), pp. 423–436, ACM, 2012.

[117] A. Y. Nikravesh, S. A. Ajila, and C.-H. Lung, "Towards an autonomic auto-scaling prediction system for cloud resource provisioning," in *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '15, (Piscataway, NJ, USA), pp. 35–45, IEEE Press, 2015.

[118] H. Fernandez, G. Pierre, and T. Kielmann, "Autoscaling Web Applications in Heterogeneous Cloud Infrastructures," in *Proceedings of the 2014 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 195–204, March 2014.

[119] A.-M. Oprescu, T. Kielmann, and H. Leahu, "Budget estimation and control for bag-of-tasks scheduling in clouds," *Parallel Processing Letters*, vol. 21, June 2011.

[120] A. Verma, L. Cherkasova, and R. H. Campbell, "ARIA: Automatic Resource Inference and Allocation for Mapreduce Environments," in *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ICAC '11, (New York, NY, USA), pp. 235–244, ACM, 2011.

[121] F. Tian and K. Chen, "Towards Optimal Resource Provisioning for Running MapReduce Programs in Public Clouds," in *Proceedings of the 2011 IEEE 4th International Conference on Cloud Computing*, CLOUD '11, (Washington, DC, USA), pp. 155–162, IEEE Computer Society, 2011.

[122] A. Ruiz-Alvarez, I. K. Kim, and M. Humphrey, "Toward Optimal Resource Provisioning for Cloud MapReduce and Hybrid Cloud Applications," in *Proceedings of the 2015 IEEE 8th International Conference on Cloud Computing (CLOUD)*, pp. 669–677, June 2015.

[123] Y. Gong, B. He, and A. C. Zhou, "Monetary Cost Optimizations for MPI-based HPC Applications on Amazon Clouds: Checkpoints and Replicated Execution," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, (New York, NY, USA), pp. 32:1–32:12, ACM, 2015.

[124] A. Marathe, R. Harris, D. Lowenthal, B. R. de Supinski, B. Rountree, and M. Schulz, "Exploiting Redundancy for Cost-effective, Time-constrained Execution of HPC Applications on Amazon EC2," in *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, (New York, NY, USA), pp. 279–290, ACM, 2014.

[125] S. Sirowy and A. Forin, "Wheres the Beef? Why FPGAs Are So Fast," Tech. Rep. MSR-TR-2008-130, Microsoft Research, Sept. 2008.

[126] R. McMillan, "Microsoft supercharges Bing search with programmable chips." `http://www.wired.com/2014/06/microsoft-fpga/`, july 2014. Wired, Accessed: 2016-03-30.

[127] S. Parsons, D. E. Taylor, D. V. Schuehler, M. A. Franklin, and R. D. Chamberlain, "High speed processing of financial information using FPGA devices." `https://www.google.com/patents/US7921046`, April 2011. U.S. patent US7921046 B2.

[128] R. Woods, J. McAllister, Y. Yi, and G. Lightbody, *FPGA-based Implementation of Signal Processing Systems*. Wiley, 2008.

[129] J. Arram, W. Luk, and P. Jiang, "Ramethy: Reconfigurable Acceleration of Bisulfite Sequence Alignment," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Feb. 2015.

[130] T. Graepel, J. Q. Candela, T. Borchert, and R. Herbrich, "Web-Scale Bayesian Click-Through Rate Prediction for Sponsored Search Advertising in Microsofts Bing Search Engine," in *Proceedings of the 27th International Conference on Machine Learning (ICML)*, June 2010. Invited Applications Track.

[131] J. G. Coutinho, O. Pell, E. ONeill, P. Sanders, J. McGlone, P. Grigoras, W. Luk, and C. Ragusa, "HARNESS Project: Managing Heterogeneous Computing Resources for a Cloud Platform," in *Reconfigurable Computing: Architectures, Tools, and Applications*, vol. 8405 of *Lecture Notes in Computer Science*, Springer, 2014.

[132] C. Reaño, R. Mayo, E. S. Quintana-Orti, F. Silla, J. Duato, and A. J. Peña, "Influence of InfiniBand FDR on the Performance of Remote GPU Virtualization," in *Proceedings of the International Conference on Cluster Computing*, Sept. 2013.

[133] A. Kawai, K. Yasuoka, K. Yoshikawa, and T. Narumi, "Distributed-Shared CUDA: Virtualization of Large-Scale GPU Systems for Programmability and Reliability," *Proceedings of the International Conference on Future Computational Technologies and Applications*, July 2012.

[134] F. Chen, Y. Shan, Y. Zhang, Y. Wang, H. Franke, X. Chang, and K. Wang, "Enabling FPGAs in the Cloud," in *Proceedings of the 11th ACM Conference on Computing Frontiers*, May 2014.

[135] Maxeler Technologies, "New Maxeler MPC-X series: Maximum Performance Computing for Big Data applications." `http://bit.ly/1Mk7UxO`. Accessed: 2016-03-30.

[136] S. Potluri, K. Hamidouche, A. Venkatesh, D. Bureddy, and D. Panda, "Efficient inter-node MPI communication using GPUDirect RDMA for InfiniBand clusters with NVIDIA GPUs," in *Proceedings of the 42nd International Conference on Parallel Processing (ICPP)*, Oct. 2013.

[137] Turku PET Centre, "libtpcmodel." `http://www.turkupetcentre.net/software/libdoc/libtpcmodel/nnls_8c_source.html`. Accessed: 2016-03-30.

[138] A. Iordache, E. Buyukkaya, and G. Pierre, *Distributed Applications and Interoperable Systems: 15th IFIP WG 6.1 International Conference, DAIS 2015, Held as Part of the 10th International Federated Conference on Distributed Computing Techniques, DisCoTec 2015, Grenoble, France, June 2-4, 2015, Proceedings*, ch. Heterogeneous Resource Selection for Arbitrary HPC Applications in the Cloud, pp. 108–123. Cham: Springer International Publishing, 2015.

[139] R. Allan, "Survey of HPC Performance Modelling and Prediction Tools," Tech. Rep. DL-TR-2010-006, Science and Technology Facilities Council, July 2009.

[140] S. Pllana, I. Brandic, and S. Benkner, "Performance Modeling and Prediction of Parallel and Distributed Computing Systems: A Survey of the State of the Art," in *Proceedings of the 2007 1st International Conference on Complex, Intelligent and Software Intensive Systems(CISIS)*, pp. 279–284, April 2007.

[141] E. Ipek, B. R. de Supinski, M. Schulz, and S. A. McKee, "An Approach to Performance Prediction for Parallel Applications," in *Proceedings of the 11th International Euro-Par Conference on Parallel Processing*, Euro-Par'05, pp. 196–205, Berlin, Heidelberg: Springer-Verlag, 2005.

[142] B. J. Watson, M. Marwah, D. Gmach, Y. Chen, M. Arlitt, and Z. Wang, "Probabilistic Performance Modeling of Virtualized Resource Allocation," in *Proceedings of the 7th International Conference on Autonomic Computing*, ICAC '10, (New York, NY, USA), pp. 99–108, ACM, 2010.

[143] J. Dejun, G. Pierre, and C.-H. Chi, "Resource Provisioning of Web Applications in Heterogeneous Clouds," in *Proceedings of the 2nd USENIX Conference on Web Application Development*, WebApps'11, (Berkeley, CA, USA), pp. 5–5, USENIX Association, 2011.

[144] I. O. Bohachevsky, M. E. Johnson, and M. L. Stein, "Generalized Simulated Annealing for Function Optimization," *Technometrics*, vol. 28, pp. 209–217, Aug. 1986.

[145] CGC, "Reverse time migration." `http://www.cgg.com/default.aspx?cid=2358`. Accessed: 2016-03-30.

[146] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd, "Efficient Transaction Processing in SAP HANA Database: The End of a Column Store Myth," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, (New York, NY, USA), pp. 731–742, ACM, 2012.

[147] Grid'5000. http://www.grid5000.fr/. Accessed: 2016-03-30.

[148] SciPy, "Scipy." http://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.anneal.html#scipy.optimize.anneal. Accessed: 2016-03-30.

[149] HARNESS consortium, "HARNESS White Paper." http://www.harness-project.eu/wp-content/uploads/2015/12/harness-white-paper.pdf. Accessed: 2016-03-30.

[150] ConPaaS, "ConPaaS." http://www.conpaas.eu. Accessed: 2016-03-30.

[151] Amazon AWS, "Amazon DynamoDB." https://aws.amazon.com/dynamodb/. Accessed: 2016-03-30.

[152] Amazon AWS, "Amazon Elastic Beanstalk." http://docs.aws.amazon.com/elasticbeanstalk/latest/dg/using-features.managing.as.html. Accessed: 2016-03-30.