CHAPTER

16

THE HARNESS PLATFORM: A HARDWARE- AND NETWORK-ENHANCED SOFTWARE SYSTEM FOR CLOUD COMPUTING

Jose G.F. Coutinho[*], Mark Stillwell[*], Katerina Argyraki[†], George Ioannidis[†], Anca Iordache[‡], Christoph Kleineweber[§], Alexandros Koliousis[*], John McGlone[‖], Guillaume Pierre[‡], Carmelo Ragusa[¶], Peter Sanders[**], Thorsten Schütt[§], Teng Yu[*], Alexander Wolf[††]

[*]Imperial College London, UK [†]EPFL, Switzerland [‡]Université de Rennes 1, France [§]Zuse Institute Berlin, Germany
[¶]SAP, UK [‖]SAP Labs, USA [**]Maxeler Technologies, UK [††]University of California, Santa Cruz, USA

16.1 INTRODUCTION

Modern cloud computing technologies and interfaces, as demonstrated by the web-service industry, can vastly improve the flexibility and ease-of-use of distributed systems, while simultaneously simplifying administration and reducing downtimes and maintenance costs. However, current data center infrastructures are built primarily to service distributed N-tier web-based applications that run on commodity hardware, leaving out many scientific and engineering applications that have complex task communication interdependencies, and may rely on a variety of heterogeneous accelerator technologies (e.g., GPGPUs, ASICs, FPGAs) to achieve the desired performance. Furthermore, web service applications are engineered to scale horizontally in response to demand, while for many other types of applications there may be a need to select the most appropriate configuration. This involves not only identifying the number and type of resources, but also providing a more complex description of resource requirements, such as the network bandwidth between resources, or the desired FPGA interconnect topology.

To better service workloads stemming from application domains currently not supported by cloud providers, we have designed and implemented an *enhanced cloud platform stack*, HARNESS, that fully embraces heterogeneity. HARNESS handles all types of cloud resources as *first-class entities*, breaking away from the VM-centric model employed by most cloud platforms used today. Moreover, HARNESS is designed to be resilient to different types of heterogeneous resources with its multitier architecture composed of agnostic and cognizant resource managers, and a novel API that provides a common interface to these managers. New types of cloud resources can be integrated into HARNESS, including complex state-of-the-art FPGA accelerator clusters, such as the MPC-X developed and marketed by Maxeler [8]; as well as more abstract resources, such as QoS-guaranteed network links. Our approach provides full control over which resource features and attributes are exposed to cloud tenants. This way,

Software Architecture for Big Data and the Cloud. DOI: 10.1016/B978-0-12-805467-3.00016-8

cloud tenants are able to submit fine-grained allocation requests that are tailored to the requirements of their application workloads, instead of relying on a set of predefined configurations (flavors) defined by cloud providers.

This chapter is structured as follows. Section 16.2 describes related work. Section 16.3 provides an overview of the key features of the HARNESS approach. Section 16.4 focuses on how heterogeneous cloud resources are managed within HARNESS. Section 16.5 covers the implementation of a HARNESS cloud prototype based on components developed by the project partners. The evaluation of the HARNESS cloud platform and infrastructure is reported in Section 16.6, and Section 16.7 concludes this chapter.

16.2 RELATED WORK

The HARNESS cloud platform provides two autonomous, yet fully integrated, cloud layers:

1. The platform layer, commonly known as Platform-as-a-Service (PaaS), manages the life-cycle of applications deployed in a cloud infrastructure. There are several commercial PaaS systems currently available, including Cloud Foundry [19], Google App Engine [5], Microsoft Azure [9], and OpenShift [32];
2. The infrastructure layer, commonly known as Infrastructure-as-a-Service (IaaS), exposes cloud resources and services that allow applications to run. Notable IaaS systems include Amazon AWS [1], Google Compute Engine [6], and Rackspace Open Cloud [37].

In addition to commercial cloud platforms, there have been a number of EU research projects that focus on solving specific cloud computing problems. For instance, Venus-C [13] is targeting the development, test and deployment of a highly-scalable cloud infrastructure; PaaSage [11] is covering the intelligent and autonomic management of cloud resources that included elastic scalability; CELAR [2] is focusing on the dynamic provisioning of cloud resources; LEADS [7] is working on the automatic management of resources across multiple clouds; BigFoot [12] is designing a scalable system for processing and interacting with large volumes of data; CloudSpaces [4] is focusing on cloud data concerns such as consistency and replication over heterogeneous repositories; and CloudLightning [3] is working on the problem of provisioning heterogeneous cloud resources using a service description language.

An important part of a cloud platform is the **API**, which provides an interface for interacting with its components. Attempts to standardize cloud interfaces have been spearheaded by a number of organizations, most notably Organization for the Advancement of Structured Information Standards (OASIS), Distributed Management Task Force (DMTF) and Open Grid Forum (OGF). The current state of the major available open cloud interface standards is summarized in Table 16.1, and described next:

- The **Topology and Orchestration Specification for Cloud Applications (TOSCA) [34]** standard from OASIS describes how to specify the topology of applications, their components, and the relationships and processes than manage them. This allows the cloud consumer to provide a detailed, but abstract, description of how their application or service functions, while leaving implementation details up to the cloud provider. With TOSCA, cloud users can describe how their distributed application should be deployed (in terms of which services need to communicate with each other), as well as how management actions should be implemented (e.g., start the database before the web service, but after the file server);

Table 16.1 Current open cloud interface standards

Specification	Standards org.	Version	Focus
TOSCA [34]	OASIS	1.0	orchestration
CAMP [33]	OASIS	1.1	deployment
CIMI [18]	DMTF	2.0	infrastructure
OCCI [31]	OGF	1.1	infrastructure

- The **Cloud Application Management for Platforms (CAMP)** [33] standard from OASIS targets PaaS cloud providers. This standard defines how applications can be bundled into a Platform Deployment Package (PDP), and in turn how these PDPs can be deployed and managed through a REST over HTTP interface. It includes descriptions of *platforms*, *platform components* (individual services offered by a platform), *application components* (individual parts an application that run in isolation) and *assemblies* (collections of possibly-communicating application components). Users can specify how application components interact with each other and components of the platform, but infrastructural details are left abstract. For example, a user would not know if their components were executing within shared services, containers or individual virtual machines;
- The **Cloud Management Initiative (CMI)** [18] standard was developed by the Cloud Management Working Group of DMTF. As with most cloud standards it relies on REST over HTTP. While XML is the preferred data transfer format due to the ability to easily verify that data conforms to the appropriate schema, JSON is also allowed. CIMI has predefined resource types defined in the standard, notably networks, volumes, machines, and systems (collections of networks, volumes, and machines), and the actions that may be performed upon a resource are constrained by its type (e.g., machines can be created, destroyed, started, stopped, or rebooted);
- The **Open Cloud Computing Interface (OCCI)** [31] standard comes from OGF, an organization that was previously very active in defining grid computing standards. It is divided into three sections: core, infrastructure, and HTTP rendering. The core specification defines standard data types for describing different types of resources, their capabilities, and how they may be linked together, while the infrastructure specification explicitly defines resource types for compute, network, and storage. The HTTP rendering section defines how to implement OCCI using REST over HTTP. Taken together, these three sections allow OCCI to present similar functionality as found in other IaaS-layer interfaces, such as Amazon EC2 [1], OpenStack Nova [10], and CIMI. However, OCCI is designed to be flexible, and the core specification can be used with extension standards to define new resource types.

16.3 OVERVIEW

A key distinguishing feature of HARNESS, when compared to current state-of-the-art and state-of-the-practice cloud computing platforms, is that it *fully embraces heterogeneity*. More specifically, HARNESS handles resources that are not only different in terms of size and nominal capacity, but also that are intrinsically different from each other, including FPGAs, GPGPUs, middleboxes, hybrid

switches, and SSDs, in cloud data center infrastructures. Such support exacerbates the complexity of the management process for both the platform and the infrastructure layers.

While the availability of heterogeneous resources in cloud data centers is not a new concept, the current approach to supporting heterogeneity involves the use of the VM-centric model, where specialized resources are viewed as mere attributes of VMs, e.g., a VM coupled with a GPU or an FPGA. One can also envision a cloud computing platform that handles a closed set of heterogeneous resources; however, it is less obvious how to create a platform that is truly open to new and arbitrary types of cloud resources without having to redesign its underlying architecture and management algorithms. In this context, we have introduced novel aspects to the cloud computing platform design:

- HARNESS supports the allocation of different types of cloud resources, from conceptual (e.g., bandwidth) to complex physical devices (e.g., cluster of FPGAs). In contrast, existing IaaS systems, such as OpenStack [10], expose only one type of compute resource to tenants: virtual machines (VMs) or physical machines (baremetal). On the other hand, database and communication facilities are exposed as high-level services. HARNESS goes beyond the VM-centric model by treating all types of resources as first-class entities, thus cloud tenants can request virtual FPGAs and virtual network links between pairs of resources in the same way VMs are allocated in today's cloud platforms;
- HARNESS allows the side-by-side deployment of commodity and specialized resources, thus dramatically increasing the number of possible resource configurations in which an application can be deployed. In other words, an application may be deployed in many ways by varying the types, the number, and the attributes of resources, each option having its own cost, performance, and utilization footprint. In this context, applications express their wants and needs to the HARNESS platform, as well as the price they are prepared to pay for various levels of service. This expression of wants and needs builds upon what can be expressed through today's simple counts of virtual machines or amounts of storage, to encompass the specific characteristics of specialized technologies;
- HARNESS supports the automatic generation of performance models that guide the selection of well-chosen sets of resources to meet application requirements and service-level objectives. We developed several techniques to reduce the profiling effort of generic applications, including the use of monitoring resource utilization to generate higher-quality performance models at a fraction of time [25], as well as extrapolating production-size inputs using smaller sized datasets;
- HARNESS is designed to be resilient to heterogeneity. We developed a multitier infrastructure system, such that the top level management can perform operations with different levels of agnosticism, so that introducing new types of resources and tailoring a cloud platform to target specialized hardware devices does not lead to a complete redesign of the software architecture and/or its top-level management algorithms;
- The various resource managers that make up the HARNESS infrastructure are governed by a single API specification that handles all types of resources uniformly. Thus, a new type of resource can be incorporated into HARNESS by integrating a resource manager that implements the HARNESS API. Furthermore, cross-cutting functionality such as monitoring, debugging, pricing and security features can be introduced by extending the HARNESS API specification, thus covering all cloud resources types managed by HARNESS.

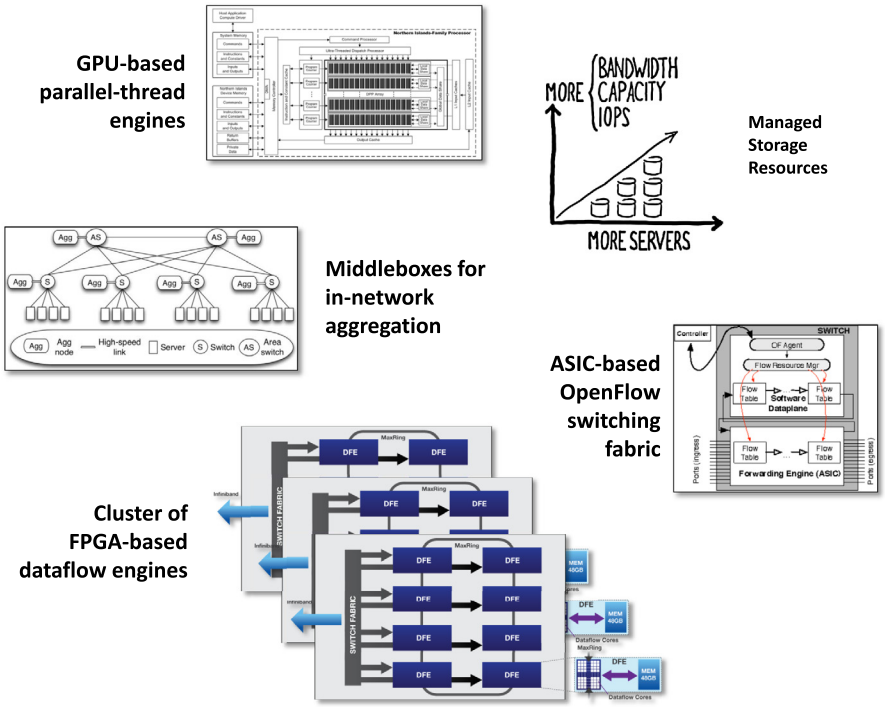


FIGURE 16.1

The HARNESS resource set consists of a group of compute, communication and storage resources. Some of these resources include (clockwise from top): (a) GPU accelerators, (b) managed storage volumes with performance guarantees, (c) hybrid switches that can handle millions of access control rules, (d) a cluster of FPGA-based devices that can be accessed via network, and (e) general-purpose middleboxes that allow application-specific network functions to be performed along network paths

16.4 MANAGING HETEROGENEITY

One of the key problems addressed by the HARNESS project is how to seamlessly incorporate and manage different types of heterogeneous resources in the context of a cloud computing platform. This includes not only supporting resources targeted by the HARNESS consortium (Fig. 16.1), but also generalizing this approach beyond the HARNESS resource set.

There are many challenges in handling heterogeneity in the context of a cloud computing platform. First, in contrast with commodity CPU-based servers, specialized resources are largely invisible to operating systems. Moreover, established cloud infrastructure management software packages, including OpenStack, provide very limited support for these types of resources. Instead, specialized resources must be directly managed by the application programmer, including not just execution of code but also in many cases tasks that are traditionally supported by both hardware and software such as allocation, de-allocation, load balancing, context switching and virtualization. A second challenge in handling heterogeneity is developing a system that does not require redesigning the architecture or

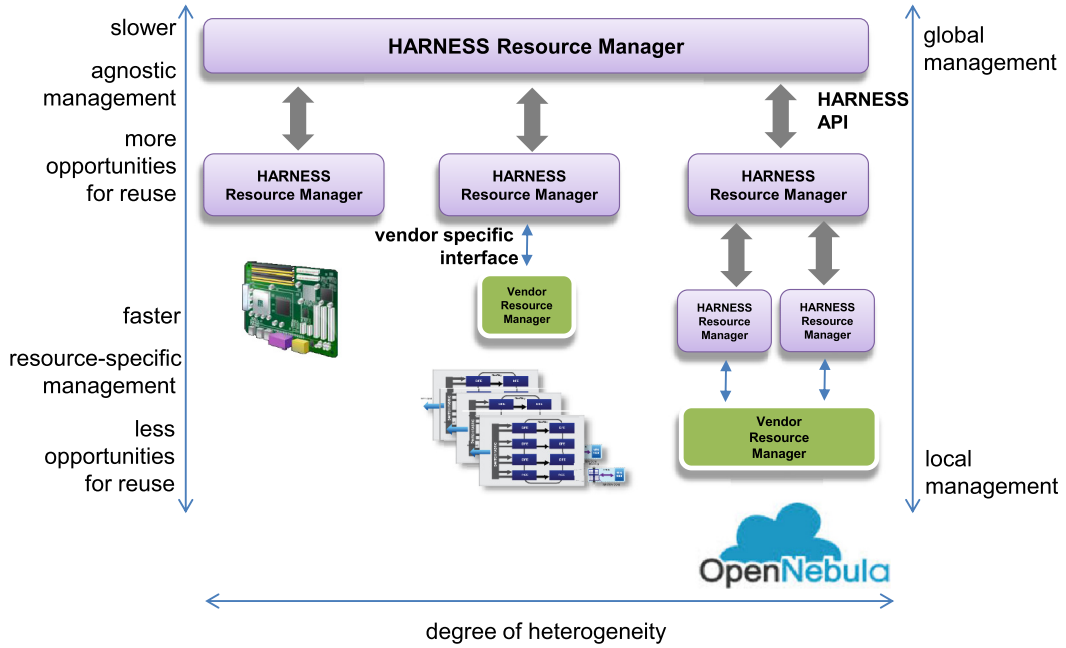


FIGURE 16.2

A heterogeneous cloud computing platform, supporting various types of resources, can be built by composing resource managers that implement the HARNESS API in a hierarchical system

rewriting the allocation algorithms when new types of resources are introduced, considering that each type of resource exposes different control mechanisms, as well as interfaces for acquiring feedback about availability and monitoring. Furthermore, each type of resource exhibits different semantics for expressing capacity and allocation requests.

16.4.1 HIERARCHICAL RESOURCE MANAGEMENT

We have addressed these challenges by designing a cloud computing architecture where runtime resource managers can be combined hierarchically, as illustrated in Fig. 16.2. In this organization, the top levels of the hierarchy are HARNESS resource managers which service requests using the HARNESS API. At the lower levels of the hierarchy we find resource managers, provided by third-party vendors, that handle specific devices using proprietary interfaces. One of the responsibilities of the HARNESS resource managers is to translate agnostic requests defined by the HARNESS API into vendor-specific requests. Supporting multiple hierarchical levels allows a system integrator to design an architecture using separation of concerns, such that each manager can deal with specific types of resources. A typical HARNESS platform deployment has a top-level manager with complete knowledge about all the resources that are available in the system, but very little understanding of how to deal with any of these resources specifically. Any management request (such as reservation or monitoring feedback) are del-

egated to child managers that have a more direct understanding of the resource(s) in question. A child manager can handle a resource type directly or can delegate the request further down to a more specific resource manager.

Hence, the top levels of the hierarchy have a more agnostic and globalized view of available resources. This view can be acquired dynamically by querying the lower-level resource managers at regular time intervals. As we go down the hierarchy, we find a more localized and cognizant management. The question remains, how does one build an agnostic resource manager that can make allocation decisions without understanding the specific semantics of each type of resource? This is the topic of the next section.

16.4.2 AGNOSTIC RESOURCE MANAGEMENT

The scheduling research community has been aware of and interested in the problem of allocating resources and scheduling tasks on large-scale parallel distributed systems with some degree of heterogeneity for some time. While work in this area generally acknowledges the underlying variations in capabilities between classes of resources, and much has been made of the differences in how time- and space- shared resources (e.g., CPUs vs memory) are partitioned between users and tasks, these works usually assume *uniformity in semantics* presented by the interfaces used to allocate these resources. For example, the authors of [39] assume that a multiresource allocation can be mapped to a normalized Euclidean vector, and that any combination of resource allocation requests can be reasonably serviced by a compute node so long as the vector sum of the allocations assigned to a node does not exceed the vector representing that node's capacity in any dimension.

What we observe, however, is that heterogeneous resources frequently expose complicated semantics that cannot be captured by a simple single- or multidimensional availability metric. This is generally due to internal constraints that disallow certain allocations that would otherwise fall within the total "amount" of resource presented by the device. For an illustrative example of why this is so, consider the MPC-X cluster presented in Fig. 16.3. This type of cluster contains one or more MPC-X nodes, where each node harbors a number of dataflow engine (DFE) devices [35] that are physically interconnected via a ring topology. A DFE is a specialized computation resource employing an FPGA to support reconfigurable designs and 48 GB (or more) RAM for bulk storage. With this setup, applications running on CPU-based machines dispatch computationally intensive tasks to single or multiple DFEs across an Infiniband network. In the context of a cloud platform, we wish to support allocation requests that specify not only the number of DFEs, but also whether these DFEs must be interconnected (RING) or not (GROUP). The choice of topology for allocated DFEs depends on the application. For instance, stencil-based applications can make use of interconnected DFEs (RING) to increase throughput [28]. On the other hand, a GROUP allocation allows the creation of a worker pool, to which tasks are sent to, and are serviced by available DFE workers. In this case, once a DFE allocation is satisfied, the residual capacity must be computed, and that requires understanding how GROUP and RING allocations affect the capacity of the MPC-X cluster.

Also in Fig. 16.3, we consider another type of resource, the XtreemFS storage, which allows users to request storage volumes with specific capacity and performance. As with the MPC-X cluster, to compute the residual capacity of an XtreemFS storage resource requires understanding the inner works of the XtreemFS management algorithms, the physical storage devices used and the corresponding performance models. These performance models indicate, for instance, how performance degrades with

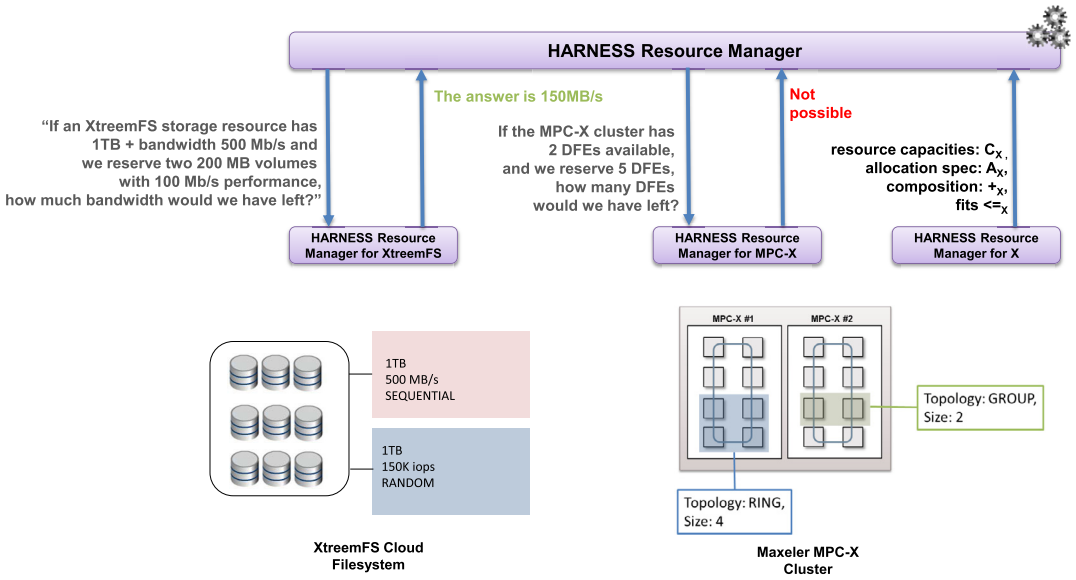


FIGURE 16.3

The top-level manager can allocate abstract resources with the support of child resource managers. To support higher-level management, each child resource manager handling a resource x exposes abstract operators ($+_x$, $\leq_x$), the allocation specification (A_x) which defines the space of all valid allocation requests, and the capacities of all available resources of type x (C_x). XtreemFS is a cloud filesystem which manages heterogeneous storage devices. The MPC-X cluster provides a pool of Maxeler dataflow engines (DFEs). A dataflow engine is a physical compute resource which contains an FPGA as the computation fabric and RAM for bulk storage, and can be accessed by one or more CPU-based machines

the number of concurrent users. Additional examples of resources with complex semantics include: a (not necessarily homogeneous) collection of machines capable of running some mix of programs or virtual machines (e.g., a cluster), and a virtual network with embedded processing elements.

To support a central allocation process that has no specific understanding about the type of resources it handles, we use child resource managers, as shown in Fig. 16.3, which oversee the process of translating agnostic management requests from the top-level manager to requests that are specific to a particular type of resource. Each child resource manager is responsible for reporting available resources, servicing allocation and deallocation requests, as well as reporting monitoring information (e.g., resource utilization).

More importantly, child managers report the capacity of all resources to the top-level managers. As previously discussed, an agnostic top-level manager cannot simply look at the descriptions of available resources and reason about their nominal and residual capacities, including understanding whether a sequence of allocation requests can "fit" on or in a resource, as such a capability would require an intimate knowledge about how to interpret the capacity of each type of resource. Moreover, while every resource considered by a resource allocation algorithm has some finite *capacity*, for some resources

it may not be possible to fully represent this capacity as a singleton or vector of numerical values, requiring instead a more complex representation.

Rather than ignoring this limitation and modeling finite resource capacities as integers or vectors, as in most works, we assume a more abstract representation, wherein the top-level manager is supplied the following information from the child manager overseeing resources of type x:

(i) The allocation space specification (A_x), e.g., the set of all valid allocation requests for resource of type x;

(ii) An operation of composition ($+_x$) that groups allocation requests;

(iii) A partial-ordering relation ($\leq_x$) on the set of basic requests that fulfills the following property: to say that an allocation request $a \in A_x$ is "smaller" than $b \in A_x$ means that servicing a requires less of a resource's capacity than servicing b. More formally, $a \leq_x b$ means that if the residual capacity of resource x is sufficient to service b then it must be sufficient to service a if a was substituted for b.

In Fig. 16.3 we illustrate the interactions between the top-level and child resource managers during the allocation process. The implementation of each abstract operator, including the ordering of both capacities and allocation requests, is specific to each type of resource. Fig. 16.5 presents an example of the ordering of MPC-X allocation requests.

This level of abstraction is necessary to allow the platform to incorporate resources with arbitrary resource allocation semantics, but it can be difficult to adapt standard resource allocation algorithms. In particular, traditional resource allocation approaches have some notion of a numerical quantity that can represent the size of either an item to be placed or the residual capacity of a resource. For example, consider the best-fit strategy commonly used for bin-packing and memory allocation, which simply follows the heuristic that the largest items remaining to be packed will be the ones with the least flexibility as to where they can be placed, and so these should be handled first, and they should be assigned to the resources where there is a minimum of left-over space, so as to minimize underutilized resources. In the next section we will discuss a strategy for ranking resource allocation requests that can be used to adapt conventional allocation algorithms to work with arbitrary heterogeneous resources in an agnostic manner.

16.4.3 RANKING ALLOCATION REQUESTS

The aim of this section is to describe a general procedure for computing a ranking function that can serve as a proxy for the size or the residual capacity of arbitrary resource types [40]. This ranking function can be used as an input to an allocation heuristic. While in this section we focus primarily on the MPC-X device, our technique can be applied deterministically to generate ranking functions for resources with arbitrary resource capacity and allocation representations, extending beyond a simple scalar or vector representation. We demonstrate this process through an illustrative example and argue that the same approach can be applied more broadly to any device for which a HARNESS resource manager can supply the functional interface described in the previous section.

To apply a standard and well established resource allocation algorithm, such as *first-fit* or *best-fit*, to an extended domain of heterogeneous resources, we need some way to compare the relative "size" of requests that may not be related under the inequality defined for the resource type. The standard approach is to base the size of the request on the amount of resource consumed (e.g., number of CPU cores or megabytes of memory). However, we note that for resource allocation among competing

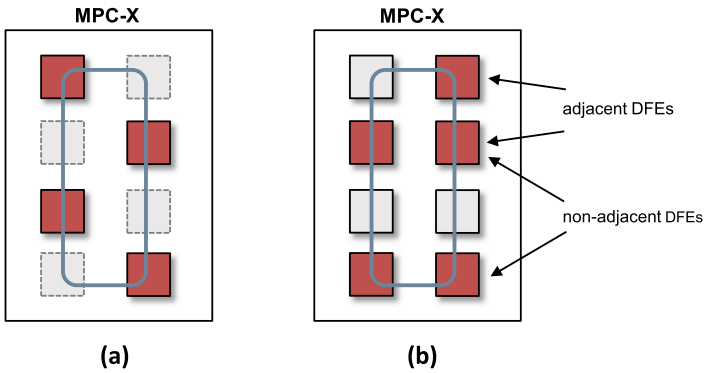


FIGURE 16.4

To understand the relationship $R_2 \leq G_5$, consider a G_5 allocation corresponding to the selection of five arbitrary DFEs inside an MPC-X box. As can be seen in **(a)**, selecting the fifth DFE implies that at least two of the selected DFEs in a G_5 allocation must be adjacent, i.e., R_2. While this example may lead us to conclude that $R_3 \leq G_5$, note that **(b)** depicts the case of a G_5 allocation that does not hold three adjacent DFEs (R_3)

users what matters is the degree to which satisfying a request limits the possible future uses of the corresponding resource.

Recall that the MPC-X device supports two kinds of resource allocation requests: GROUP and RING requests. A GROUP request contains the number of desired DFE devices inside an MPC-X box, while a RING request requires that all of the allocated devices be interconnected (see Fig. 16.3). We use G_n $(n = 1, \ldots, 8)$ to represent GROUP requests and R_n $(n = 1, \ldots, 8)$ to represent RING requests. The index of a request represents the number of DFE devices required.

Given our previous definition of inequality, we can see that the following must hold:

$$\forall n \in [1, 7], G_n \leq G_{n+1}, \tag{16.1}$$

$$\forall n \in [1, 7], R_n \leq R_{n+1}, \tag{16.2}$$

$$\forall n \in [1, 8], G_n \leq R_n. \tag{16.3}$$

Furthermore, when considering how DFE devices must be selected in an actual resource allocation for the MPC-X, we can derive the following relationships:

$$G_1 = R_1, \tag{16.4}$$

$$G_7 = R_7, \tag{16.5}$$

$$G_8 = R_8, \tag{16.6}$$

$$R_2 \leq G_5 \tag{16.7}$$

$$R_3 \leq G_6. \tag{16.8}$$

Fig. 16.4 explains relationship (16.7), and the same argument can be extended to relationship (16.8).

In addition, we can include a more constrained relation defined as: $a \preceq b$ iff a is less than b and there is no third allocation request c, distinct from a and b, such that $a \leq c \leq b$. In this case we say that

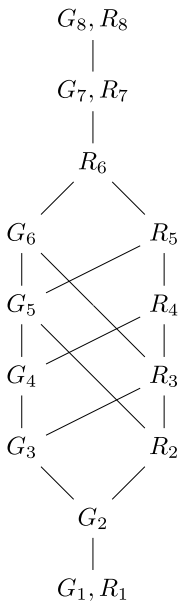


FIGURE 16.5

A lattice-topology for MPC-X resource allocation requests

b covers a, and thus:

$$\forall n \in \{1, 2, 3, 4, 5, 7\}, G_n \preceq G_{n+1}, \tag{16.9}$$

$$\forall n \in [2, 7], R_n \preceq R_{n+1}, \tag{16.10}$$

$$\forall n \in [1, 7], G_n \preceq R_n. \tag{16.11}$$

It should be noted that it is *not* the case that $R_1 \preceq R_2$, as we have $G_1 = R_1$ and therefore $R_1 < G_2 < R_2$. Similarly, G_7 does not cover G_6, as $G_7 = R_7$, and therefore we have $G_6 < R_6 < G_7$.

These covering relationships can be used to form an initial lattice of basic resource allocation requests for the MPC-X device as depicted in Fig. 16.5. As this structure places "larger" requests (e.g., those that either take up more space or put greater constraint on future use of the resource) higher in the hierarchy, this suggests that height might be a reasonable proxy for size. Unfortunately, the relationships between the basic allocation requests for the MPC-X device prevent the easy definition of a consistent height function. For example, we cannot define the height value of G_5 because two nodes immediately under it, G_4 and R_2, are at different levels in the hierarchy.

While the initial lattice structure does not admit a well-defined height function, that is, it is not *modular*, we can use the reverse of *Birkhoff's representation theorem* [17] to extend the nonmodular partially-ordered topology to a modular *downset* lattice (Fig. 16.6). The downset of a basic resource allocation request is a set containing that request and all smaller requests (e.g., the downset of G_3, denoted $G_3\downarrow$, is the set $\{G_2, G_3\}$). The downset lattice preserves the inequality relationship, but also

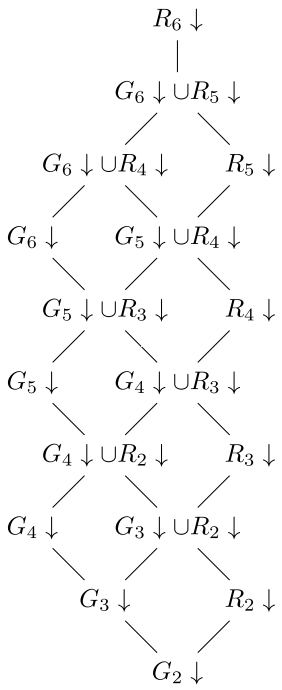


FIGURE 16.6

A *modular* lattice-topology for MPC-X resource allocation requests, which has been derived from the non-modular lattice presented in Fig. 16.5

includes additional nodes labeled with unions of the downsets of basic requests. By constructing a modular lattice, we can define a ranking function for the lattice that obeys the *valuation law*: for any two nodes in the lattice, the sum of their ranks will equal the sum of the ranks of the smallest element larger than both and the largest element smaller than both. For a finite lattice, this is equivalent to stating that for any node the length of every path from that node to the bottom will be the same, and so it is possible to define a consistent ranking function based on the *height*. Note that if requests have the same *height*, then it is up to the allocation algorithm to break ties, for instance, giving priority to the requests that arrived first.

In practice, with the above approach, we can develop an agnostic top-level manager which generates a modular lattice that captures the total ordering of allocation requests and resource capacities, as long as the lower-level manager implements the API described in Section 16.4.4. In this context, this lower-level manager needs only to define the partial ordering between any two resources (i.e., $\preceq_x$).

16.4.4 HARNESS API

The HARNESS API provides a uniform interface for resources managers to support hierarchical resource management and agnostic resource allocation as described in Sections 16.4.1 and 16.4.2. Re-

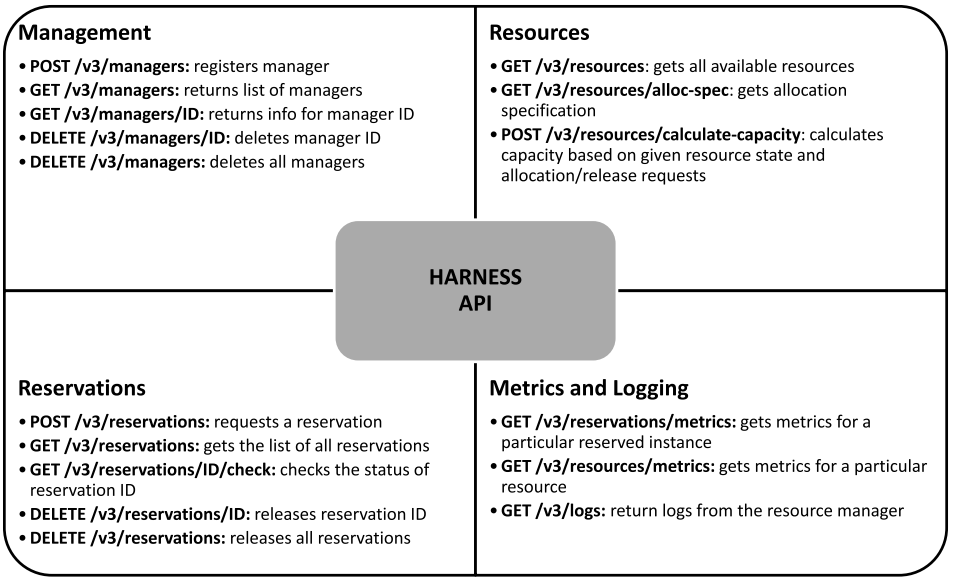


FIGURE 16.7

The HARNESS API provides a REST-based interface for top-level resource managers to handle heterogeneous resources

source managers implementing the HARNESS API (referred to as HARNESS resource managers) can be combined hierarchically taking into account separation of concerns, such that each resource manager handles specific classes of resources at different levels of abstraction. The HARNESS API follows the RESTful style, where interactions are handled through HTTP requests to provide an Infrastructure-as-a-Service (IaaS) platform.

We have successfully employed the HARNESS API to support and manage a wide range of resources that span from physical clusters of FPGAs to conceptual resources such as virtual links. Each HARNESS resource manager exposes a set of resources of a specific type, including their corresponding residual capacities, allowing virtualized instances of these resources to be allocated by defining the required capacity.

While the implementation of the API is specific to a type of resource, the API makes a small number of assumptions about its nature:

(i) A resource has a specific capacity which can be finite or infinite;
(ii) A resource operates on a specific allocation space;
(iii) The availability of a function which computes (or estimates) changes in capacity based on an allocation request;
(iv) Instances of any resource type (e.g., virtual machine) can be created and destroyed;
(v) Resources can be monitored with feedback provided on specific metrics.

We group the HARNESS API functions in four categories (Fig. 16.7):

- **Management** category allows resource managers to register to other HARNESS resource managers in a hierarchical organization;
- **Resources** category provides information about the state of available resources (nominal or residual capacities – C_x), the allocation space specification (A_x), and a method that returns whether the aggregation ($+_x$) of a given set of allocation requests can be serviced ($\leq_x$) by a given resource capacity;
- **Reservations** category allows allocation requests to be submitted, the status of resource instances to be queried, and resource instances to be destroyed;
- **Metrics and Logging** category provides monitoring information about resources and their instances, as well as logging information about resource management.

The HARNESS API sits slightly above the pure infrastructure layer (IaaS). It does not attempt to describe application orchestration like TOSCA or deployment like CAMP (leaving these problems to a PaaS layer), but rather, like OCCI and CIMI is more concerned with allocating and linking infrastructure resources. However, unlike these standards (see Section 16.2 for more details), which come with built-in models of different resource "types", such as machines, VMs, networks, and storage devices, the HARNESS API considers all abstract resources to be of the same type. As can be seen in Fig. 16.3, while resources such as XtreemFS cloud storage and Maxeler MPC-X cluster are inherently different from each other, they are handled transparently as a single type resource, having each a corresponding resource manager that provides an implementation of the HARNESS API.

This allows for a more flexible model that can accommodate a wider array of cloud-enabled devices, as well as supporting cross-cutting services such as pricing and monitoring, which do not have to support multiple resource models. Once resources have been allocated in HARNESS, the deployment phase allows each provisioned resource to be handled using resource-specific mechanisms, including APIs, tools and services, to make full use of their capabilities.

16.5 PROTOTYPE DESCRIPTION

In this section, we describe the HARNESS prototype, presented in Fig. 16.8, which has been implemented to validate our approach. The HARNESS prototype has been deployed in two testbeds: Grid'5000, which allows us to explore a large-scale research infrastructure to support parallel and distributed computing experiments [16]; and the Imperial Cluster testbed, which supports clusters of hardware accelerators and heterogeneous storage.

The HARNESS architecture is split into three layers: (1) a *platform* layer that manages applications, (2) an *infrastructure* layer that is responsible for managing cloud resources, and (3) a *virtual execution* layer where applications are deployed and executed. Next, we explain each HARNESS layer in more detail.

16.5.1 THE PLATFORM LAYER

The platform layer automates the choice of resources to run an application: cloud tenants indicate requirements for job completion time and/or monetary cost. The platform then decides which set of resources best satisfy these objectives and constraints. To execute an application in HARNESS, a cloud

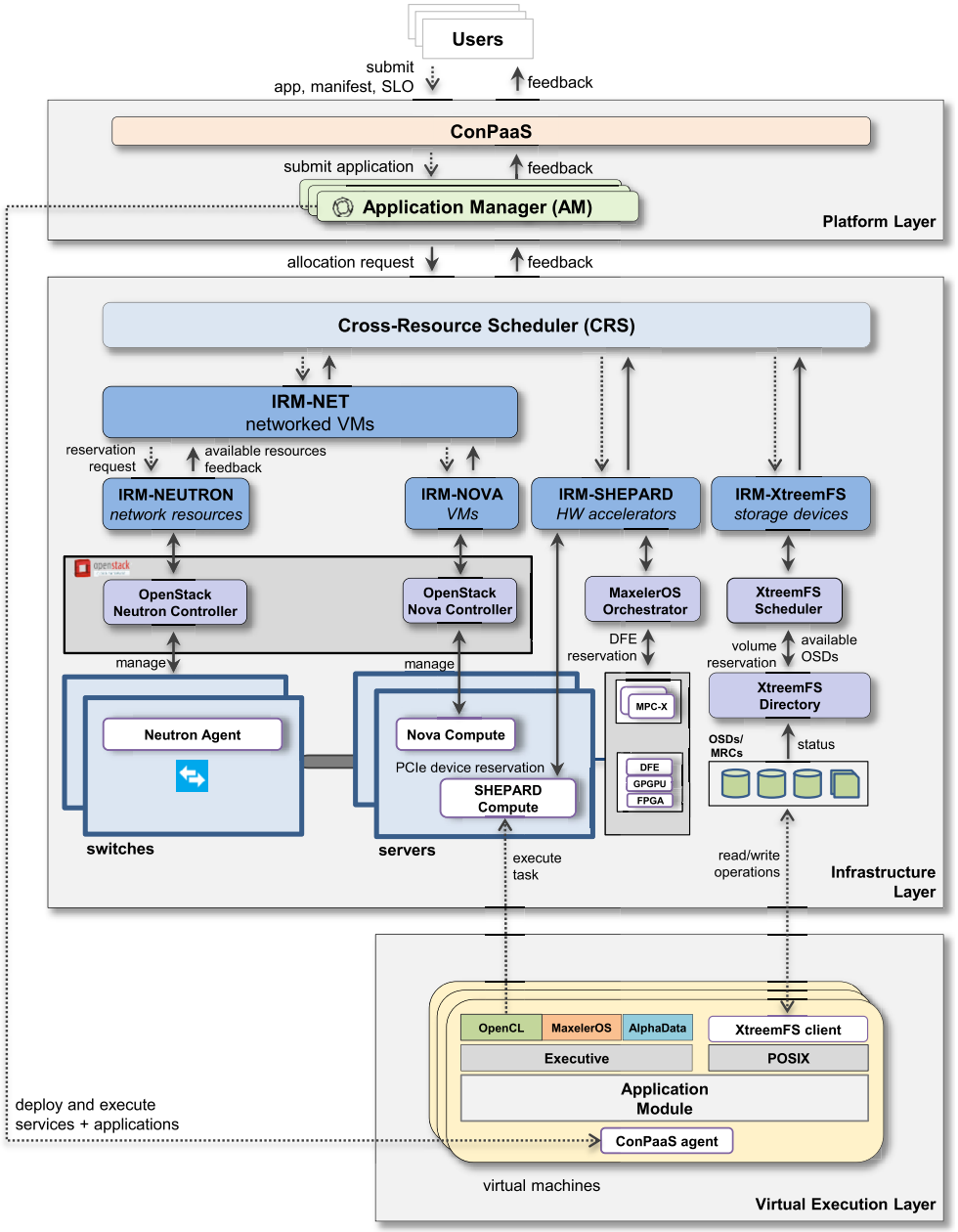


FIGURE 16.8

The HARNESS architecture is composed of three layers: a platform layer that is responsible for managing applications, an infrastructure layer where cloud resources are managed, and a virtual execution layer where applications are deployed and executed

user must supply: (i) an application manifest describing the structure of the application and the types of resources it can potentially execute on; (ii) a service-level objective (SLO) defining nonfunctional requirements for this execution, such as the maximum execution latency or the maximum monetary cost; and (iii) the application binaries and scripts. Once this information is supplied, the HARNESS platform deploys the application over a well-chosen sets of resources such that the manifest and the SLO are respected. As mentioned in Section 16.3, an important part of this process is developing performance models that guide this selection. In HARNESS, we developed several techniques to reduce the profiling effort of arbitrary applications, including taking into account monitoring information to generate high-quality performance models at a fraction of time, as well as extrapolating production-size inputs using reduced-size datasets [25].

The platform layer includes two main components: ConPaaS and the Application Manager:

- **ConPaaS** [36] is an integrated runtime environment for elastic cloud platforms. It consists of two key components: (i) a Web server providing a graphical interface where users can submit and manage their applications; and (ii) the *Director* which is in charge of authenticating users and instantiating one Application Manager for each application instance submitted by a user. Once it has created an Application Manager instance, it forwards all subsequent management requests about this application to the Application Manager in charge of the application.
- **The Application Manager (AM)** is in charge of controlling the execution of one particular application. It is a generic and application-agnostic component, and thus a new Application Manager does not need to be developed for every new application. The Application Manager operates in a virtual machine provisioned using the HARNESS cloud resources. This virtual machine contains a specific program in charge of interpreting application manifests and SLOs, building performance models for arbitrary applications, choosing the type and number of resources that an application needs to execute within its SLO, provisioning these resources, deploying the application's code and data in the provisioned resources, and finally collecting application-level feedback during and after execution.

Whenever the *Director* and the Application Manager needs to provision resources in the HARNESS platform (either to create a new Application Manager instance or to run an actual application), it sends a resource provisioning request to the infrastructure layer, which we explain next.

16.5.2 THE INFRASTRUCTURE LAYER

The infrastructure layer is in charge of managing all cloud resources and making them available on demand. Its key components are the CRS (Cross-Resource Scheduler) and the IRMs (Infrastructure Resource Managers). These correspond respectively to the top-level HARNESS resource manager, and the child HARNESS resource managers described in Section 16.4.1. In particular, the CRS handles high-level requests involving multiple resources, while the IRMs are responsible for translating agnostic management requests into resource-specific requests. The currently implemented IRMs are: *IRM-NET* (network resources), *IRM-NOVA* (OpenStack compute resources), *IRM-NEUTRON* (OpenStack network resources), *IRM-SHEPARD* (hardware accelerator resources), and *IRM-XtreemFS* (XtreemFS storage resources). Beneath the IRMs, we have components that manage specific resources, which include OpenStack [10], SHEPARD [30], MaxelerOS Orchestrator [8], and XtreemFS [38].

Below we provide details about some of these components:

- **The Cross-Resource Scheduler (CRS)** is in charge of handling resource provisioning requests [22]. It processes single resource requests, and requests for a *group* of heterogeneous resources with optional placement constraints between resources. For example, an application may request one virtual machine and one FPGA such that the two devices are located close to each other. It uses the network proximity maps provided by *IRM-NET*, and decides which set of physical resources should be chosen to accommodate each request. Once this selection has been made, it delegates the actual provisioning of the resources to corresponding IRMs. Each IRM is in charge of managing some specific type of heterogeneous resources, including VMs, GPGPUs, FPGAs, storage, and network devices.
- **The Network Resource Manager (IRM-NET)** provides the CRS with up-to-date maps of the physical resources which are part of the cloud. These maps contain network proximity measurements realized pairwise between the physical resources, such as latency, and available bandwidth. This information allows the CRS to service allocation requests with placement constraints, such as allocating two VMs with a specific latency requirement between them. This component also handles bandwidth reservations, allowing virtual links to be allocated. Finally, IRM-NET supports subnet and public IP allocations by delegating these requests through IRM-NOVA and IRM-NEUTRON. In particular, users can request one or more subnets and assign VMs to them, and also assign public IPs to individual VMs.
- **The MaxelerOS Orchestrator** supports the allocation of networked DFEs located in MPC-X devices. The MaxelerOS Orchestrator provides a way to reserve DFEs for IRM-SHEPARD. These accelerators are then available to applications over the local network.
- **XtreemFS** is a fault-tolerant distributed file system that provides three kinds of services: (1) the directory service (DIR), (2) the metadata and replica catalog (MRC) server, and (3) the object storage device (OSD) [38]. The DIR tracks status information of the OSDs, MRCs, and volumes. The volume metadata is managed by one MRC. File contents are spread over an arbitrary subset of OSDs. In addition, the XtreemFS Scheduler handles the reservation and release of data volumes to be used by the HARNESS application. Data volumes are characterized by their size, the type of accesses it is optimized for (random vs. sequential), and the number of provisioned IOPS.

16.5.3 THE VIRTUAL EXECUTION LAYER

The virtual execution layer is composed of reserved VMs where the application is deployed and executed (Fig. 16.8). In addition to the application itself, the VMs contain components (APIs and services) that support the deployment and execution processes, including allowing the application to interact with (reserved) resource instances. These components include:

- **The ConPaaS agent**, which performs management actions on behalf of the Application Manager: it configures the VM where the Application Manager resides, installs code/data resources such as GPGPUs, FPGAs and XtreemFS volumes, configures access to heterogeneous resources, starts the application, and finally collects application-level feedback during execution [36];
- **The Executive** is a scheduling process that given a fixed set of provisioned heterogeneous compute resources, selects the most appropriate hardware accelerator for a given application task [20,29];

- **The XtreemFS client** is in charge of mounting XtreemFS volumes in the VMs and making them available as regular local directories [38].

16.6 EVALUATION

In this section, we report two cloud deployment scenarios using HARNESS, which are currently not supported by traditional cloud computing platforms:

- **Executing HPC Applications on the Cloud:** This case study (Section 16.6.1) demonstrates how HPC applications such as RTM (Reverse Time Migration) can exploit hardware accelerators and managed storage volumes as cloud resources using HARNESS. These experiments were performed in the Imperial Cluster testbed;
- **Resource Scheduling with Network Constraints:** This case study (Section 16.6.2) demonstrates the benefits of managed networking when deploying distributed applications such as Hadoop MapReduce in a cloud platform. In this scenario, cloud tenants can reserve bandwidth, which directly affects where jobs are deployed. This work was conducted in Grid'5000 with physical nodes located in Rennes and Nantes.

16.6.1 EXECUTING HPC APPLICATIONS ON THE CLOUD

Reverse Time Migration (RTM) represents a class of computationally intensive applications used to process large amounts of data, thus a subclass of HPC applications. Some of the most computationally intensive geoscience algorithms involve simulating wave propagation through the earth. The objective is typically to create an image of the subsurface from acoustic measurements performed at the surface. To create this image, a low-frequency acoustic source is activated and the reflected sound waves are recorded, typically by tens of thousands of receivers. We term this process a *shot*, and it is repeated many thousands of times while the source and/or receivers are moved to illuminate different areas of the subsurface. The resulting dataset is dozens or hundreds of terabytes in size.

Our experiments were conducted on the Imperial Cluster testbed, which includes 3 physical compute machines, a DFE cluster harboring 24 DFEs, and standard HDD and SSD storage drives. The dataset used in our experiments are based on the Sandia/SEG Salt Model 45 shot subset.[1]

Deploying RTM on a heterogeneous cloud platform. In this experiment, we demonstrate how an HPC application, such as RTM, is deployed and executed in the HARNESS cloud platform. The RTM binary, along with its deployment and initialization scripts, are compressed into a tarball. This tarball is submitted to the HARNESS cloud platform along with the *application manifest*. The application manifest describes the resource configuration space, which allows the Application Manager to derive a valid resource configuration to run the submitted version of RTM.

When the application is submitted, the HARNESS platform creates an instance of the Application Manager which oversees the life-cycle of the application. The Application Manager operates in two modes. In the *profiling* mode, the Application Manager creates a performance model by running the

[1] The Sandia/SEG Salt Model 45 shot dataset can be downloaded here: http://wiki.seg.org/wiki/SEG_C3_45_shot.

application on multiple resource configurations and capturing the execution time of each configuration. Associated with the performance model, we have a pricing model which indicates the monetary cost of using a specific resource configuration. With the performance and pricing models, the application manager can translate cost and performance objectives specified in the SLO (e.g., to execute the fastest configuration) into a resource configuration that can best achieve these objectives.

For this experiment, the Application Manager deployed RTM on different resource configurations, varying the number of CPU cores (from 1 to 8), RAM sizes (1024 and 2048 MB), number of dataflow engines (from 1 to 4) and storage performances (10 and 250 MB/s). The pricing model used is as follows:

$$cost(c) = c.num\_dfes \times 9 \times 10^{-1} + c.cpu\_cores \times 5 \times 10^{-4} + c.mem\_size \times 3 \times 10^{-5} + c.storage\_perf \times 10^{-5}$$

where for a given configuration c, $c.num\_dfes$ represents the number of DFEs, $c.cpu\_cores$ corresponds to the number of CPU cores, $c.mem\_size$ corresponds to the size of RAM (MB), and $c.storage\_perf$ the storage performance (MB/s). The resulting cost is in €/s. The pricing models presented in this section are loosely based on the current offerings from Amazon EC2 [15]; however, they can be arbitrary and can be updated dynamically to reflect various factors, such as resource availability. The subject of cloud resource pricing is complex, specially when considering heterogeneous resources, and is outside the scope of this chapter.

Fig. 16.9 presents the performance model generated by the Application Manager using the pricing model specified above. The Application Manager automatically selected and profiled 28 configurations, with 5 of these configurations identified as part of the Pareto frontier. The number of profiled configurations is dependent on the profiling algorithm used by the Application Manager [22]. For each configuration, the Application Manager reserves the corresponding resources, and deploys the application. The initialization script supplied with the application automatically detects the configuration attributes, and configures the application to use these resources. For instance, if the configuration specifies 8 CPU cores, then the initialization script configures the `OMP_NUM_THREADS` environment variable to that number, and allow the application to fully utilize all provisioned CPU resources.

Fig. 16.9 highlights four configurations in the top-right quadrant, which correspond to the slowest and most expensive configurations, and thus the least desirable of all the configurations identified. This is due to the use of slow storage (9 MB/s) which dominates the performance of the job despite the use of DFEs. At the bottom-right quadrant, there are five configurations highlighted that are relatively inexpensive, however they run relatively slow since they do not employ DFEs. Finally, in the center of the figure, we find three highlighted configurations which use a limited number of CPU cores and DFEs, but they do not provide the best trade-off between price and execution time. Instead, the five configurations that provide the best trade-offs are those in the Pareto frontier (see Table 16.2).

With the above configurations and the corresponding pricing, the Application Manager can service SLO-based requests. For instance, if the user requests the fastest configuration under €0.25, the Application Manager would select configuration *C*, while *A* would be identified as the cheapest configuration.

Exploiting different DFE topology reservations. The RTM job deployed in HARNESS is both *moldable* and *malleable* [21]. A *moldable* job can adapt to different resource configurations at the start of the program. A *malleable* job, on the other hand, can be reconfigured at run-time during program

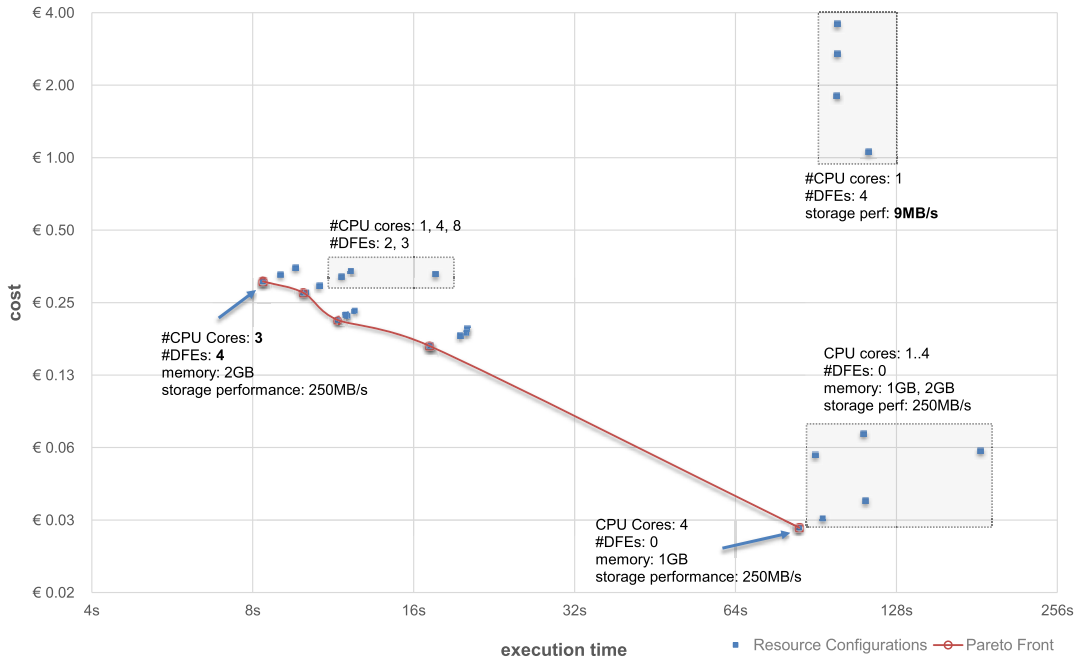


FIGURE 16.9

Performance model for the RTM application automatically generated by the Application Manager

Table 16.2 Five pareto configurations for the RTM application (see Fig. 16.9)

#ID	#DFEs	#CPU cores	RAM (MB)	Storage speed (MB/s)	Execution time (s)	Price (€)
A	0	4	1024	250	84	0.03
B	1	8	2048	250	17	0.17
C	2	1	1024	250	12	0.21
D	3	3	2048	250	10	0.27
E	4	3	2048	250	8	0.31

execution. Both types of jobs provide more flexibility than a *rigid* job which is designed to run on single resource configuration. In the following experiment, we further explore RTM's moldable and malleable properties.

Our implementation of the HARNESS platform supports the *DFE cluster* resource, as presented in Fig. 16.3, with two types of DFE allocation requests: *GROUP* and *RING*. As previously explained, a request for a *GROUP* of N DFEs would provision N DFEs within the same MPC-X box, while requesting N DFEs of a *RING* topology would provision N interconnected DFEs.

Fig. 16.10 shows the performance of a single RTM shot using different problem dimensions and number of DFEs. Multiple DFEs are connected via RING. The characterization of these jobs is sum-

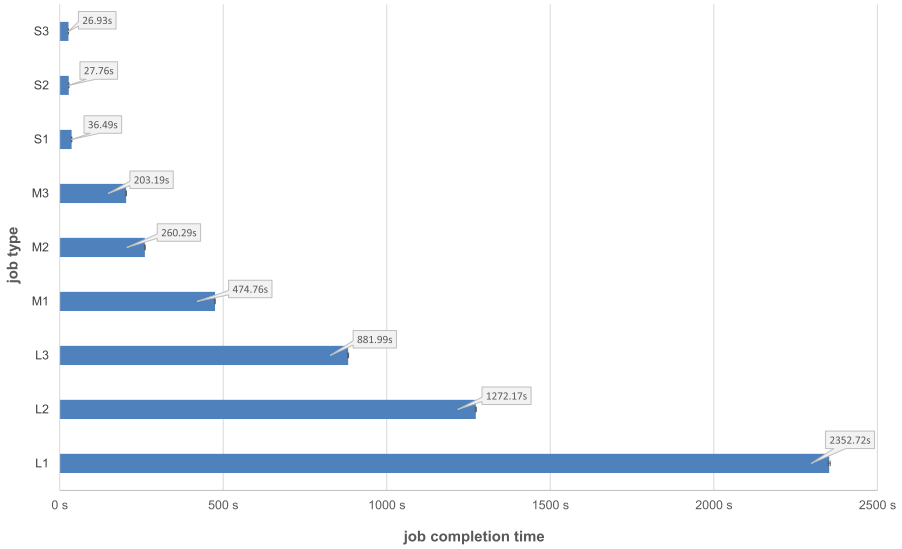


FIGURE 16.10

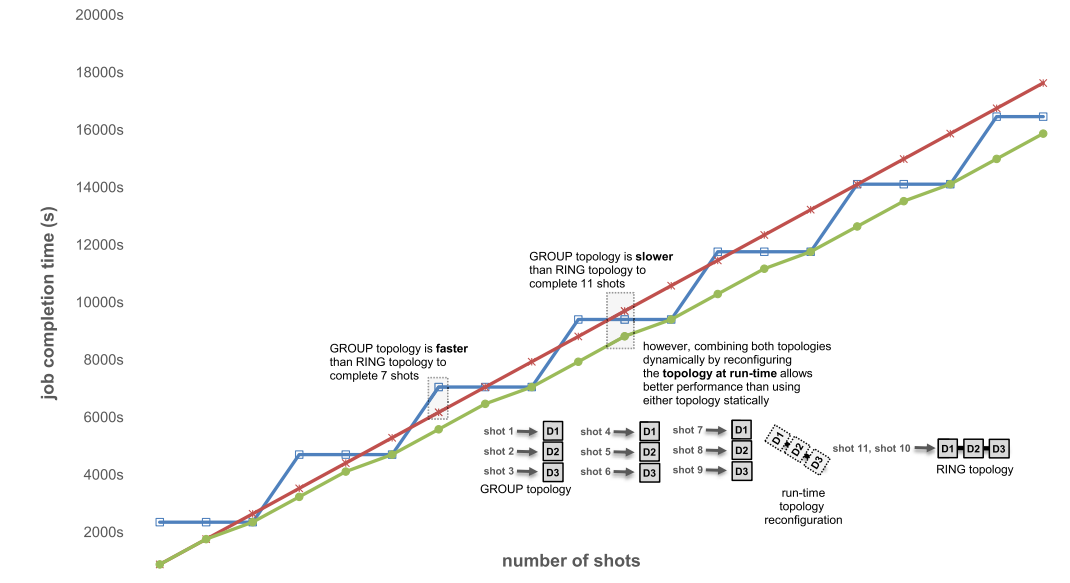
Performance of a single RTM shot using different problem dimensions (S, M, L) and number of DFEs (1, 2 and 3)

Table 16.3 Three classes of RTM jobs using different number of DFEs

Design	Configuration	Dimension	#iterations
S1	1×DFE	200 × 200 × 200	2000
S2	2×DFEs (ring)	200 × 200 × 200	2000
S3	3×DFEs (ring)	200 × 200 × 200	2000
M1	1×DFEs	400 × 400 × 400	4000
M2	2×DFEs (ring)	400 × 400 × 400	4000
M3	3×DFEs (ring)	400 × 400 × 400	4000
L1	1×DFEs	600 × 600 × 600	6000
L2	2×DFEs (ring)	600 × 600 × 600	6000
L3	3×DFEs (ring)	600 × 600 × 600	6000

marized in Table 16.3. We can see that the number of DFEs makes little impact on smaller jobs, such as $S1$, $S2$, and $S3$. This is due to the fact that smaller workloads will not be able to fully utilize multiple DFEs. Larger jobs, on the other hand, scale better and are able to exploit larger number of DFEs. For instance, $S3$ is only 0.7 times faster than $S1$, while $L3$ is 2.6× faster than $L1$.

Let us now focus on the impact of the DFE topology on completing a multishot RTM job. Fig. 16.11 shows the results of completing an RTM job with varying number of shots. Each shot has a dimension of $600 \times 600 \times 600$, running in 6000 iterations. For each number of shots, we compare the performance of using 3 independent DFEs (GROUP) against 3 interconnected DFEs (RING). The former can pro-



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
GROUP	2352.7	2352.7	2352.7	4705.4	4705.4	4705.4	7058.2	7058.2	7058.2	9410.9	9410.9	9410.9	11764	11764	11764	14116	14116	14116	16469	16469
RING	881.99	1764	2646	3528	4410	5291.9	6173.9	7055.9	7937.9	8819.9	9701.9	10584	11466	12348	13230	14112	14994	15876	16758	17640
Reconfiguration	881.99	1764	2352.7	3234.7	4116.7	4705.4	5587.4	6469.4	7058.2	7940.2	8822.1	9410.9	10293	11175	11764	12646	13528	14116	14998	15880

FIGURE 16.11

The effect of different DFE topologies on RTM performance

cess three shots in parallel, while the latter, working as a single compute resource, can only process each shot sequentially. Each independent DFE computes a shot in 2353 s, while the interconnected version computes a shot in 882 s. It can be seen from the figure that depending on the total number of shots, one of the topologies is more efficient than the other. For 7 shots, the independent DFEs run faster, while for 11 shots the interconnected DFEs run faster. Since RTM jobs are moldable, we can optimize their performance by selecting the topology that can provide the best performance according to the number of shots.

We can further speed-up the computation by configuring the RTM job to be *malleable*, so that it adapts during runtime. As can be seen in Fig. 16.11, depending on the number of shots, we can combine both types of topologies to reduce the completion time. For instance, for 11 shots, we can execute 9 shots in three sequences in parallel followed by 2 shots computed with the three DFEs interconnected. This combination yields the best performance (8821 s) when compared to a static configuration using the parallel configuration (9411 s) or the interconnected configuration (9702 s). Malleable jobs can be automatically managed by a runtime scheduler, which decides on the most optimal topology given a set of allocated resources. In our HARNESS prototype, the *Executive* component is responsible for this decision process (see Section 16.5.3).

16.6.2 RESOURCE SCHEDULING WITH NETWORK CONSTRAINTS

The purpose of this case study is to investigate the benefits of managed networking when deploying a distributed application such as the Hadoop-based AdPredictor [23] on a large-scale testbed such as Grid'5000. AdPredictor represents a class of modern industrial-scale applications, commonly known as *recommender systems*, that target either open-source or proprietary on-line services. For example, one such service is Mendeley [26], a free reference organizer and academic social network that recommends related research articles based on user interests. Another such service is Bing [27], a commercial online search engine that recommends commercial products based on user queries. In general, items are matched with users and, due to the modern "data deluge", these computations are usually run in large-scale data centers. The ACM 2012 KDD Cup *track2* dataset [14] has been used to evaluate AdPredictor.

Grid'5000 is a large-scale multi-site French public research testbed designed to support parallel and distributed computing experiments. The backbone network infrastructure is provided by the French National Telecommunication Network for Technology, Education and Research RENATER, which offers 11,900 km of optic fiber links and 72 points of presence.

Fig. 16.12 reports the throughput of one MapReduce worker over time where the steady throughput consumption peaks at approximately 200 Mbit/s, excluding any bursts during shuffling. Fig. 16.13 presents the results of an AdPredictor job using the same configuration by varying the available bandwidth between the worker compute hosts. The application exhibits degradation below 200 Mbit/s, which is consistent with our measurements in Fig. 16.12.

Consequently, when a network-bound application, such as AdPredictor, is deployed on a cloud platform that provides *no network performance guarantees*, it will have its performance severely affected if resources have been reserved in a low-bandwidth environment. Note that services like Microsoft Azure [9] enforce a maximum bound on outgoing traffic bandwidth depending on the VM type chosen, not allowing compute and network resources to be tailored independently according to the application needs. HARNESS addresses these issues by exposing network bandwidth and latency as independent resources and constraints, respectively. In this context, applications may define and submit their desired network performance guarantees and the underlying infrastructure will provision them, if available. This highlights a key principle of HARNESS, which allows specialized and commodity resources to be treated as first-class cloud entities.

We report three scenarios conducted in Grid'5000: (a) allocating resources without network constraints; (b) allocating resources using bandwidth reservation requests; and (c) allocating resources with service-level objectives. The testbed used for these experiments consists of 8 physical compute nodes across two different sites: *Rennes* (4 nodes) and *Nantes* (4 nodes). While both sites offer high-speed gigabit connectivity of 1500 Mbit/s, we have emulated heavy network congestion in Nantes, so that the throughput of any network flow in Nantes is limited to 500 Mbit/s.

Allocating without network constraints. In the first experiment we request 1 *master* and three *worker* instances without specifying network constraints. Fig. 16.14 presents all the possible configurations that may result from this allocation request, as each worker may be placed either in Rennes or Nantes. If the tenant specifies no network constraints, any one of these placements may be possible, therefore the end-user will experience a considerable variability in her application's performance over multiple deployments on the same cloud platform. It is evident that the application's execution time is dependent on whether the workers are deployed in the high-bandwidth Rennes cluster, in the con-

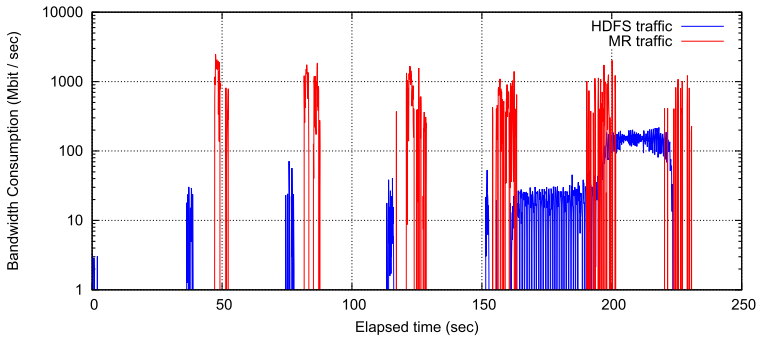


FIGURE 16.12

AdPredictor throughput over time

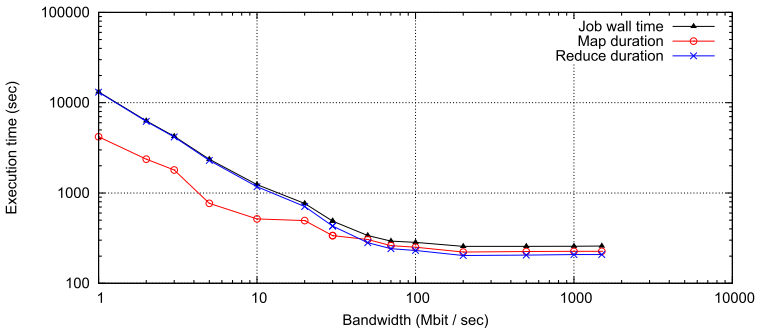


FIGURE 16.13

AdPredictor performance vs bandwidth

gested Nantes cluster, or across both sites. Nevertheless, in this scenario the tenant has no control over the final placement.

Allocating using bandwidth reservation requests. In order to eliminate the suboptimal placements presented in Fig. 16.14, labeled as "2-1", "1-2" and "0-3", which involve at least one of the workers being placed in the congested cluster of Nantes, the tenant can specify a request demanding a reservation of 300 Mbit/s between workers `Worker1` and `Worker2`, `Worker1` and `Worker3`, and `Worker2` and `Worker3`. Consequently, the CRS takes into account this bandwidth reservation request when allocating VMs (containers) for the workers, therefore eliminating the suboptimal placements and deploying all workers in Rennes under the conditions presented in this experiment. Listing 1 presents the reservation request for this scenario, in which four VMs (labeled Master and Worker1–4, respectively) are requested with specific computation requirements, and three resources of type "Link" specify minimum bandwidth requirements between Worker-labeled VMs. Recall that resources of different types can be requested independently from each other and tailored to the specific requirements of an application.

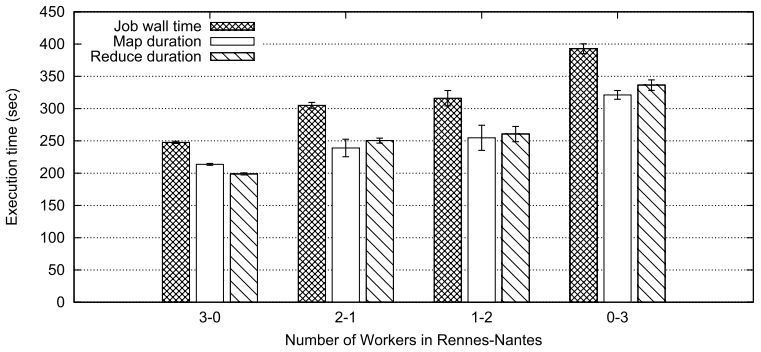


FIGURE 16.14

AdPredictor performance vs resource placement

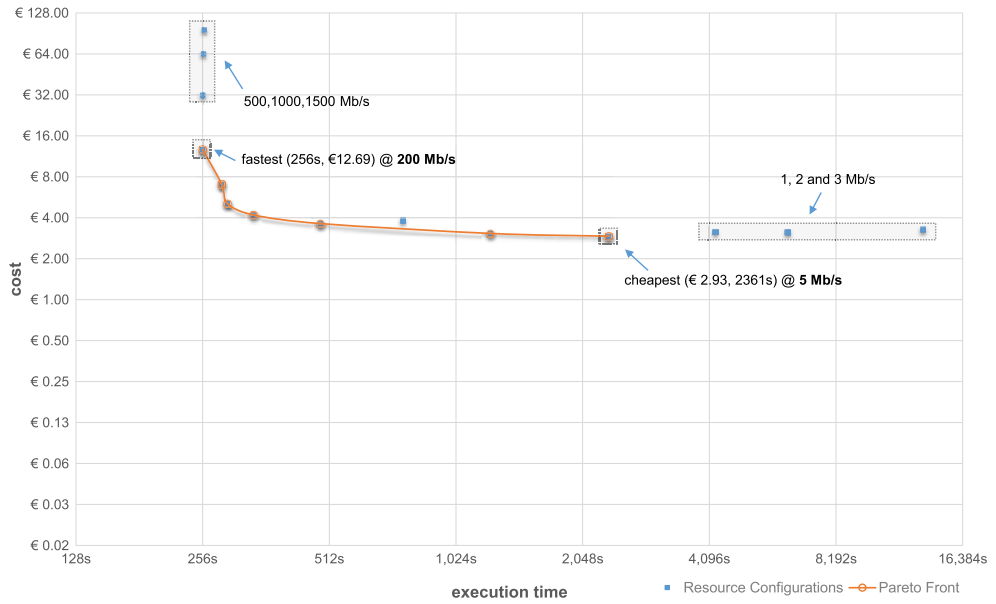


FIGURE 16.15

Performance model generated for AdPredictor running on Grid'5000

Allocating with service-level objectives. In the previous experiment, we requested bandwidth reservation in the application manifest to deploy a Hadoop-based AdPredictor job. In a real scenario, the cloud tenant is more concerned about job completion time and the price of reserving resources. In order for the HARNESS platform to derive a resource configuration that can meet performance

```
{
    "Master": {
        "Type": "Machine",
        "Cores": 4,
        "Memory": 4096
    },
    "Worker1": {
        "Type": "Machine",
        "Cores": 12,
        "Memory": 16384
    },
    "Worker2": {
        "Type": "Machine",
        "Cores": 12,
        "Memory": 16384
    },
    "Worker3": {
        "Type": "Machine",
        "Cores": 12,
        "Memory": 16384
    },
    "Link1": {
        "Type": "Link",
        "Source": "Worker1",
        "Target": "Worker2",
        "Bandwidth": 300
    },
    "Link2": {
        "Type": "Link",
        "Source": "Worker2",
        "Target": "Worker3",
        "Bandwidth": 300
    },
    "Link3": {
        "Type": "Link",
        "Source": "Worker1",
        "Target": "Worker3",
        "Bandwidth": 300
    }
}
```

Listing 1: Reservation request for compute and network resources

or cost objectives, it needs to have a performance model. For this experiment, we have generated a performance model based on profiling AdPredictor with different bandwidth reservation requests.

The pricing model used in our experiments is as follows:

$$cost(c) = c.cpu\_cores \times 5 \times 10^{-4} + c.mem\_size \times 3 \times 10^{-5} + c.bandwidth \times 2 \times 10^{-1}$$

The bandwidth unit is Mb/s. With this pricing model, the price of 1 Gb/s bandwidth is roughly equal to one VM with 4 cores and 8 GB RAM.

Fig. 16.15 presents the performance model of AdPredictor running on Grid'5000. It contains 14 points where we vary the bandwidth requirements from 1 Mb/s to 1.5 Gb/s, while maintaining the same compute and storage configuration. It can be seen that different bandwidth reservations have an impact in both pricing and performance. Not all configurations provide a good trade-off between price

and execution time, and they are discarded. The remaining configurations, 7 in total, are part of the Pareto frontier. These configurations are then selected to satisfy objectives in terms of pricing (the cheapest configuration costs €2.93 but requires 2361 s to complete) or in terms of completion time (the fastest configuration completes the job in 256 s but costs €12.69).

16.7 CONCLUSION

In this chapter, we presented the HARNESS cloud computing architecture, which supports two distinct layers:

- The *platform layer* manages applications on behalf of cloud tenants. More specifically, it automates the process of selecting a resource configuration that can best satisfy application-specific goals (e.g., low completion time), with each configuration having its own cost, performance, and utilization footprint. To achieve this, the platform layer resorts to application profiling to automatically build performance models. The platform also exploits the fact that application performance characteristics may be observed using smaller inputs, so it employs extrapolated application profiling techniques to reduce the time and cost of profiling;
- The *infrastructure layer* manages heterogeneous resources on behalf of cloud providers. This layer uses a resource management model in which all types of resources are handled as first-class entities, as opposed to the VM-centric model employed by current cloud providers. The infrastructure layer is based on a *multitier management* approach, designed to make cloud computing systems open and resilient to new forms of heterogeneity. This way, introducing new types of resources does not result in having to redesign the entire system. The various resource managers that make up the HARNESS infrastructure are governed by a single API specification that handles all types of resources uniformly. Thus, a new type of cloud resource can be incorporated into the HARNESS infrastructure by providing an implementation of the HARNESS API.

We developed a fully working prototype of the HARNESS cloud computing platform. Our infrastructure layer implementation relies on the following technologies: (a) MaxelerOS Orchestrator for networked DFE reservations; (b) SHEPARD for hardware accelerator reservations; (c) XtreemFS for heterogeneous storage reservations; (d) IRM-NET for network link reservations; and (d) a cross-resource scheduler (CRS) which enfolds all these resource-specific managers to optimize multiple reservation requests alongside (optional) network placement constraints.

Our prototype was evaluated using two testbeds: (1) a heterogeneous compute and storage cluster that includes FPGAs and SSDs where we deployed an HPC application (Reverse-Time Migration), and (2) Grid'5000, a large-scale distributed testbed that spans France to which we deployed a machine learning application (AdPredictor). In our evaluation, we demonstrated how HARNESS fully embraces heterogeneity, allowing the side-by-side deployment of commodity and specialized resources. This support increases the number of possible resource configurations in which an application can be deployed to, bringing wholly new degrees of freedom to the cloud resource allocation and optimization problem.

PROJECT RESOURCES

The source-code of most of the HARNESS prototype components and deployment projects has been released to the public. In particular, the software projects that create the unified HARNESS platform can be downloaded from our GitHub page (https://github.com/harnesscloud). Free-standing software projects created or extended by HARNESS, such as ConPaaS (http://www.conpaas.eu) and XtreemFS (http://www.xtreemfs.org), have their own independent software download sites. A more detailed description of each component available for downloading is found in http://www.harness-project.eu/?page_id=721. Video demonstrations of our final prototype can be accessed in http://www.harness-project.eu/?page_id=862. Technical outcomes of the project not covered in this chapter can be found in the HARNESS whitepaper [24]. Finally, a list of all our project publications and technical reports can be found in our public website: http://www.harness-project.eu/.

REFERENCES

[1] Amazon Web Services. Available at http://aws.amazon.com/.
[2] CELAR project: automatic, multi-grained elasticity provisioning for the cloud. Available at http://www.celarcloud.eu/.
[3] CloudLightning project: self-organising, self-managing heterogeneous cloud. Available at http://cloudlightning.eu/.
[4] CloudSpaces project: an open service platform for the next generation of personal clouds. Available at http://cloudspaces.eu/.
[5] Google App Engine. Available at https://developers.google.com/appengine/.
[6] Google Compute Engine. Available at https://cloud.google.com/products/compute-engine/.
[7] LEADS project: large-scale elastic architecture for data as a service. Available at http://www.leads-project.eu/.
[8] Maxeler Technologies: maximum performance computing. Available at http://maxeler.com/.
[9] Microsoft Azure Services Platform. Available at http://www.azure.net/.
[10] OpenStack: open source software for creating private and public clouds, http://openstack.org.
[11] PaaSage project: a model-based cross cloud development and deployment platform. Available at http://www.paasage.eu/.
[12] The BigFoot project: an OpenStack based analytics-as-a-service solution. Available at http://bigfootproject.eu/.
[13] Venus-C project: virtual multi-disciplinary environments using cloud infrastructures. Available at http://www.venus-c.eu/.
[14] ACM SIGKDD, Predict the click-through rate of ads given the query and user information. Available at http://www.kddcup2012.org/c/kddcup2012-track2/.
[15] Amazon EC2 pricing. Available at http://aws.amazon.com/ec2/pricing/.
[16] D. Balouek, et al., Adding virtualization capabilities to the Grid'5000 testbed, in: Cloud Computing and Services Science, vol. 367, 2013, pp. 3–20.
[17] G. Birkhoff, Lattice Theory, vol. 25, American Mathematical Soc., 1940.
[18] Cloud Management Working Group (CMWG), Cloud Infrastructure Management Interface (CIMI) specification. Available at http://www.dmtf.org/standards/cmwg.
[19] CloudFoundry. Available at http://www.cloudfoundry.com/.
[20] J.G.F. Coutinho, O. Pell, E. O'Neill, P. Sanders, J. McGlone, P. Grigoras, W. Luk, C. Ragusa, HARNESS project: managing heterogeneous computing resources for a cloud platform, in: Reconfigurable Computing: Architectures, Tools, and Applications, Springer, 2014, pp. 324–329.
[21] D.G. Feitelson, L. Rudolph, Towards convergence in job schedulers for parallel supercomputers, in: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing, IPPS'96, Springer-Verlag, 1996, pp. 1–26.
[22] FP7 HARNESS Consortium, Heterogeneous Platform Implementation (updated). Technical Report D6.3.3, 2015.
[23] T. Graepel, J.Q. Candela, T. Borchert, R. Herbrich, Web-scale Bayesian click-through rate prediction for sponsored search advertising in Microsoft's Bing search engine, in: Proceedings of the 27th International Conference on Machine Learning (ICML-10), 2010, pp. 13–20.
[24] HARNESS white paper. Available at http://www.harness-project.eu/wp-content/uploads/2015/12/harness-white-paper.pdf.

[25] A. Iordache, E. Buyukkaya, G. Pierre, Heterogeneous resource selection for arbitrary HPC applications in the cloud, in: Proceedings of the 10th International Federated Conference on Distributed Computing Techniques (DAIS 2015), June 2015.
[26] Mendeley. Available at http://www.mendeley.com/.
[27] Microsoft Bing. Available at http://www.bing.com/.
[28] X. Niu, J.G.F. Coutinho, W. Luk, A scalable design approach for stencil computation on reconfigurable clusters, in: Proceedings of the IEEE on Field Programmable Logic and Applications (FPL), 2013.
[29] E. O'Neill, J. McGlone, J.G.F. Coutinho, A. Doole, C. Ragusa, O. Pell, P. Sanders, Cross resource optimisation of database functionality across heterogeneous processors, in: Proc. of the 12th IEEE International Symposium on Parallel and Distributed Processing with Applications, 2014.
[30] E. O'Neill, J. McGlone, P. Milligan, P. Kilpatrick, SHEPARD: scheduling on heterogeneous platforms using application resource demands, in: 2014 22nd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), Feb. 2014, pp. 213–217.
[31] Open Grid Forum (OGF), Open Cloud Computing Interface (OCCI) specification. Available at http://occi-wg.org.
[32] OpenShift. Available at https://www.openshift.com/.
[33] Organization for the Advancement of Structured Information Standards (OASIS), Cloud application management for platforms (CAMP) v1.1. Available at http://docs.oasis-open.org/camp/camp-spec/v1.1/camp-spec-v1.1.html.
[34] Organization for the Advancement of Structured Information Standards (OASIS), Topology and orchestration specification for cloud applications (TOSCA) v1.0. Available at http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.html.
[35] O. Pell, O. Mencer, H.T. Kuen, W. Luk, Maximum performance computing with dataflow engines, in: High-Performance Computing Using FPGAs, 2013, pp. 747–774.
[36] G. Pierre, C. Stratan, ConPaaS: a platform for hosting elastic cloud applications, IEEE Internet Comput. 16 (5) (Sept. 2012) 88–92.
[37] Rackspace open cloud. Available at https://www.rackspace.com/cloud.
[38] J. Stender, M. Berlin, A. Reinefeld, XtreemFS – a file system for the cloud, in: Data Intensive Storage Services for Cloud Environments, IGI Global, 2013.
[39] M. Stillwell, F. Vivien, H. Casanova, Virtual machine resource allocation for service hosting on heterogeneous distributed platforms, in: Proceedings of the 26th International Parallel and Distributed Processing Symposium, May 2012.
[40] T. Yu, B. Feng, M. Stillwell, J.G.F. Coutinho, et al., Relation-oriented resource allocation for multi-accelerator systems, in: International Conference on Application-Specific Systems, Architectures and Processors (ASAP), 2016.

ACKNOWLEDGEMENTS

Simulations presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several universities as well as other funding bodies (see https://www.grid5000.fr). The HARNESS Project was supported by the European Commission Seventh Framework Programme, grant agreement no 318521.