The HARNESS Platform: A Hardware- and Network-Enhanced Software System for Cloud Computing

(Preliminary White Paper, 10 November 2015)

FP7 HARNESS consortium[†]

HARNESS advances the state-of-the-art in cloud computing management software by enabling commodity and specialised resources to be deployed side-by-side in data-centre infrastructures.

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-centre infrastructures are built primarily to service distributed N-tier web-based applications that run on commodity hardware, leaving out many scientific and engineering applications which 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, which includes identifying not only the number and type of resources required to accomplish a job, but also their attributes (such as number of CPU cores or data-flow engine topology) at job dispatch time.

To better service workloads stemming from application domains that are currently not supported by modern day data-centres, we have designed and implemented an *enhanced cloud platform stack*, HARNESS, that fully embraces heterogeneity. In particular, the HARNESS cloud platform is designed to be resilient to different types of heterogeneous resources with its modular architecture and a novel API that provides a common interface to resource managers. In particular, new types of 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; as well as more abstract resources, such as QoS-guaranteed network links. Our approach provides full control over what resource features are exposed to cloud tenants, allowing fine-grained allocation requests that can be adjusted to service specific jobs. A key feature of our approach is the support of *resource-agnostic* management, where global managers can schedule resources without having specific knowledge of their inner workings by coordinating with local schedulers that operate more directly on specific classes of resources.

This paper is structured as follows. In Section 2, we explain how the HARNESS cloud computing architecture integrates and manages different types of cloud resources. In Section 3, we describe the implementation of a HARNESS software cloud platform based on components developed by the HARNESS project partners using the aforementioned design architecture. The evaluation of this platform is reported in Section 4. Section 5 concludes this paper.

2 Managing Heterogeneity

One of the 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 (Figure 1), but also generalising this approach beyond the HARNESS resource set. This means designing a cloud computing platform that is resilient to heterogeneity, such that supporting new types of resources does not require a complete redesign of the system.

There are a number of challenges in handling heterogeneity in the context of a cloud computing platform. First, in contrast with commodity CPU-based servers, specialised resources are largely invisible to operating systems, while cloud infrastructure management software such as OpenStack provide very limited support. Heterogeneous compute resources such as GPGPUs and FPGAs are often accessed by applications as I/O devices via library-call and/or runtime interfaces. These devices 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 virtualisation. The second problem relates to creating a system that doesn't require redesigning the architecture or rewriting the allocation algorithms when introducing new types of resources, considering that each type of resource exposes different control mechanisms, as well as interfaces for acquiring feedback about availability and monitoring. More importantly, each type of resource exhibits different semantics for expressing capacity and allocation requests.

[†]*Research funded by the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement number 318521.*

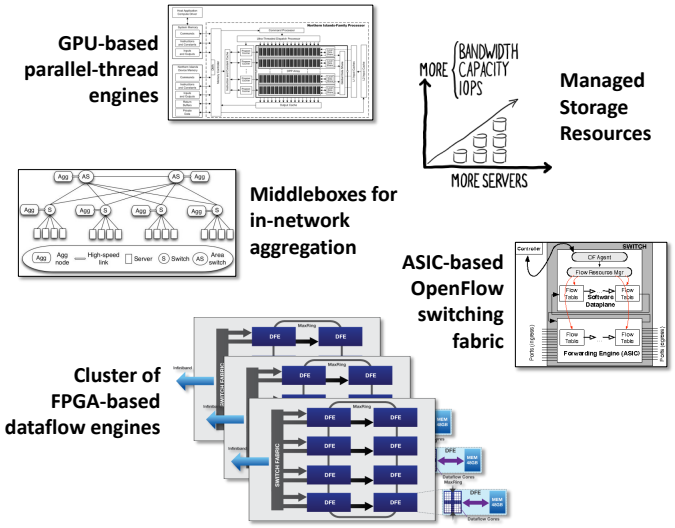


Figure 1: The HARNESS resource set consists of a group of compute, communication and storage resources. Some of these resources include (clockwise from top): (a) GPU accelerators, (b) managed storage volumes with performance guarantees, (c) hybrid switches that can handle millions of access control rules, (c) a cluster of dataflow engines (FPGA) that can be accessed via network, and (d) general-purpose middleboxes that allow application-specific network functions to be performed along network paths.

2.1 Hierarchical Resource Management

To mitigate these problems, we have designed a cloud computing architecture where runtime resource managers can be combined hierarchically, as illustrated in Figure 2. In this organisation, the top levels of the hierarchy are the HARNESS resource managers which service requests using the HARNESS API. At the lower levels of the hierarchy we find vendor resource managers that handle specific devices. 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. In particular, a typical HARNESS platform deployment has a top-level manager that has 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 delegated 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. So, what we see is that at the top level of the hierarchy we have a more agnostic management that can handle a wide range of resource types, and thus have a globalised view of resources available in the system. As we go lower in the hierarchy, we have more localised and resource-specific 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.

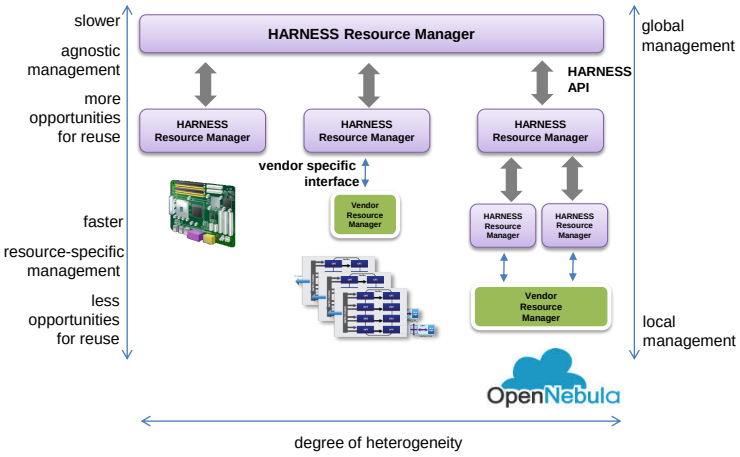


Figure 2: Heterogeneous cloud computing platforms, supporting various types of resources, can be built by composing resource managers that implement the HARNESS API in an hierarchical system.

2.2 Agnostic Resource Scheduling

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 recognises 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 some uniformity in semantics presented by the interfaces used to allocate these resources. For example, the authors of [19] assume that a multi-resource allocation can be mapped to a normalised 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 expose far more complicated semantics that cannot be captured by a simple single or multi-dimensional availability metric, and cannot be normalised such that they can be directly understood by a central scheduling algorithm. For some devices, it may be appropriate to allow resources to present different levels of abstraction, or even to represent complex aggregations. Examples include: a (not necessarily homogeneous) collection of machines capable of running some mix of programs or virtual machines (e.g., a cluster), a distributed file service capable of guaranteeing some level of performance or reliability, a virtual network with embedded processing elements, a cluster of FPGA-based dataflow engines with a specific interconnect topology, or a set of network links worth enforced bandwidth and latency QoS requirements.

In HARNESS we develop an approach for describing the problem of scheduling moldable jobs on heterogeneous virtualised resources with non-uniform semantics, under the assumption that the central scheduler (employed by the top-level HARNESS manager) has no knowledge or understanding of how underlying resources function internally, thus allowing its scheduling algorithms to be independent from the type of resources they manage.

To support a central scheduling process that has no specific understanding about the type of resources it manages, we use child

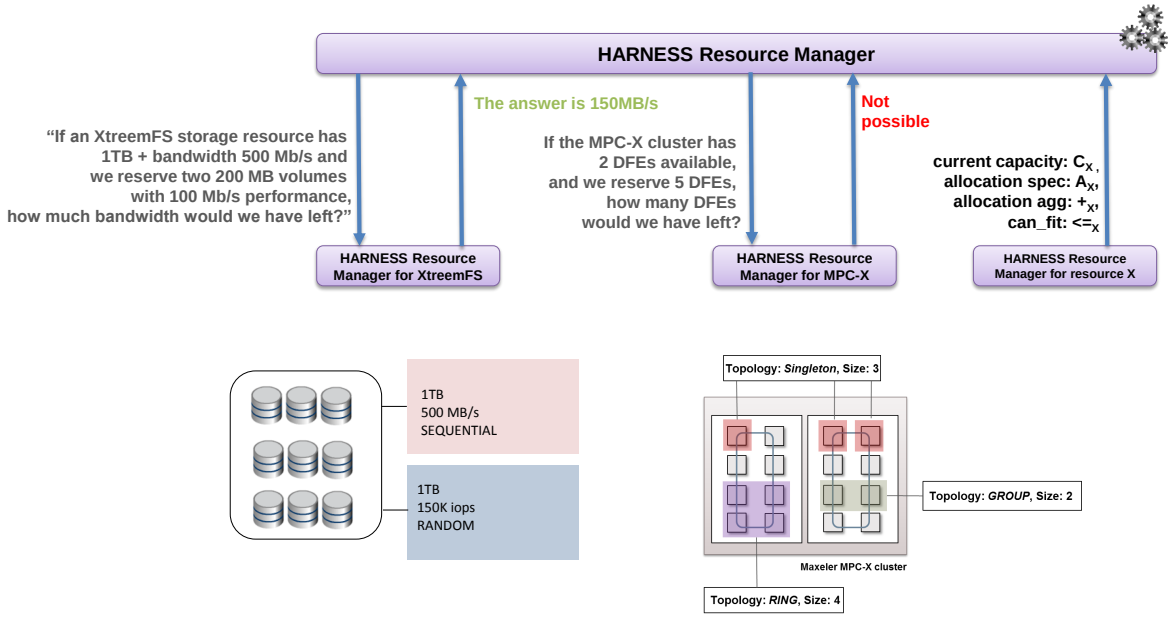


Figure 3: The top-level manager can schedule abstract resources with the support of child resource managers. To support higher-level management, each child resource manager handling a resource x exposes abstract operators ($+_x$, $\leq_x$), the allocation specification (A_x) which defines the space of all valid allocation requests, and the capacity of the resource (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.

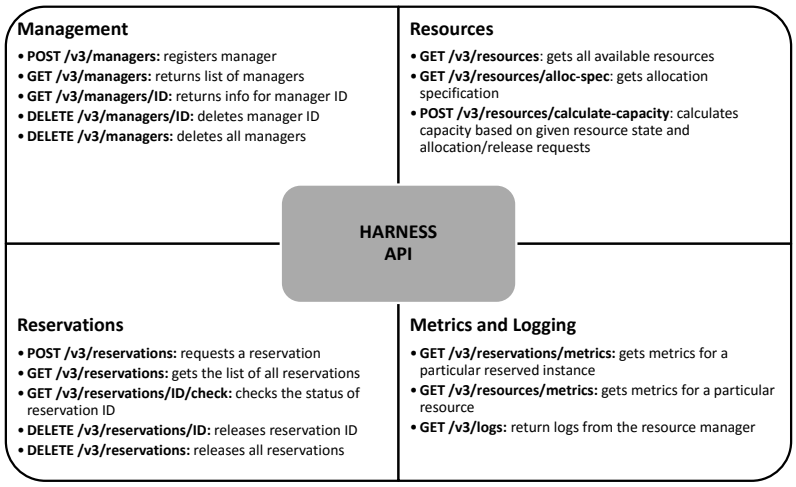


Figure 4: The HARNESS API provides a REST-based interface for HARNESS resource managers to handle heterogeneous resources.

resource managers, as shown in Figure 3, which are in charge of translating an agnostic management request from the top-level manager to a specific request for a particular type of resource. Each child resource manager is responsible for reporting available resources and their respective capacities, performing allocation and deallocation requests, as well as reporting monitoring information (e.g. resource utilisation). One challenge to supporting resource-agnostic scheduling, is that the 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" a resource. For instance, consider the MPC-X cluster shown in Figure 3, where an allocation request can specify not only the number of Dataflow Engines (DFEs) but also their topology (for instance, whether DFEs must be interconnected via a RING topology, or be independent in a GROUP topology). Such requests require knowledge about the internal state of a resource in order to interpret its capacity.

Therefore while a given resource x has some finite capability or *capacity* (as resources with infinite capacities can be safely ignored by definition), it may not be possible to fully represent this capacity as a singleton or vector of numerical values. Rather than modelling finite resource capacities as integers or vectors, we assume a more abstract representation, wherein the top-level manager has access, through the child manager managing resource x, to an

operation of composition ($+_x$) that groups allocation requests, a partial-ordering relation ($\leq_x$) to reason about resource capacity, the allocation specification (A_x) which defines the space for all valid allocation requests, and the capacity of x, denoted C_x. Then for any ordered sequence of requests to acquire or release allocations on a resource, x, it is possible to compute deterministically whether or not that set of requests could be serviced by x.

In practice, this means that the top-level resource manager can use these abstract operators during the scheduling phase to derive a set of resources that can best service an allocation request without having to directly interpret the capacity of each type of resource. In Figure 3 we illustrate the interactions between the top-level and child resource managers during the scheduling process.

2.3 HARNESS API

The HARNESS API is an interface implemented by all HARNESS resource managers to handle a specific type of resource. It mostly follows the RESTful style, where interactions between components are handled through HTTP requests. Resource managers supporting this API can be combined in an hierarchical system to support a heterogeneous cloud computing platform that can handle various types of resources, from physical clusters of FPGAs to conceptual resources such as virtual links.

While the implementation of the API is specific based on the nature of the resource, the API makes a few assumptions about its nature. First, a resource has capacity which can be finite or infinite; Second, a resource operates on a specific allocation space; Third, there is a function that can compute (or estimate) changes in capacity based on an allocation request; Fourth, we can create and destroy instances of a resource (e.g. virtual machine); Fifth, resources can be monitored with feedback provided on specific metrics.

We have grouped the HARNESS API functions in four categories (Figure 4):

- **Management.** Allow resource managers to connect to other HARNESS resource managers in a hierarchical organisation.
- **Resources.** Provide information about the state of available resources, and meta-information about allocation requests and resource capacity.
- **Reservations.** Allow resource instances to be created and destroyed.
- **Metrics and Logging.** Provide monitoring information about resources and their instances, and also logging information.

When designing the HARNESS API, we have considered existing cloud standards. One key difference between the cloud and its predecessor, the grid, is that cloud standards are often de-facto and based on simple interfaces and running code, rather than being meticulously specified in advance by committees representing the interests of various stakeholders (as was the case for the grid). In the early days of the cloud, open source projects would either attempt to mimic the not-fully-documented interfaces of commercial implementations [6] or simply create their own APIs (e.g., OpenStack Nova). While this approach has allowed for the rapid evolution of cloud service offerings, it has also resulted in instability and the proliferation of a widespread set of similarly functional, but incompatible interfaces [2].

Attempts to standardise on cloud interfaces have been spear-headed by a number of organisations, most notably the OASIS, the DMTF, the OGF and the SNIA. While cloud vendors have been slow to fully embrace these projects, third party projects such as Apache Libcloud [4] and Deltacloud [3], which provide thin translation layers between incompatible APIs have, with some caveats, made it possible for end-users to escape vendor lock-in. The current state of the major available open cloud interface standards is summarised in Table 1, and described below:

- **The TOSCA [17]** 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 not only describe how their distributed application should be deployed (in terms of which services need to communicate with each other), but also how management actions should be implemented (e.g., start the database before the web service, but after the file server).
- **The CAMP [16]** 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, individual virtual machines, etc.
- **The CIMI [5]** standard was developed by the CMWG of DMTF. As with most cloud standards it specifies 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 pre-defined 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 CDMI [20]** international standard specifies "the interface to access cloud storage and the data stored therein". CDMI was developed by SNIA, an industry group with an obvious interest in promoting standard interfaces for cloud storage. It defines *containers*, which represent abstract storage space in which files can be stored, and *data objects*, which roughly correspond to files. The standard implementation uses REST over HTTP. It provides similar functionality to Amazon's S3 interface [21].
- **The OCCI** standard comes from OGF, an organisation that was previously very active in defining grid computing

Specification	Standards Org.	Version	Focus
TOSCA	OASIS	1.0	orchestration
CAMP	OASIS	1.1	deployment
CIMI	DMTF	2.0	infrastructure
OCCI	OGF	1.1	infrastructure
CDMI	SNIA	1.1.1	storage infrastructure

Table 1: Current Open Cloud Interface Standards

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, OpenStack Nova, or CIMI. However, OCCI is designed to be flexible, and the core specification can be used with extension standards to define new resource types.

The HARNESS API sits slightly above the pure IaaS layer. It does not attempt to describe application orchestration like TOSCA or deployment like CAMP (leaving these problems to the AM), but rather, like OCCI and CIMI is more concerned with allocating and linking infrastructure resources. However, unlike these standards, 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.

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 conform to multiple resource models. Once resources have been allocated in HARNESS, the deployment phase allows each provisioned resource to be handled using specific APIs, tools and services to make full use of their capabilities.

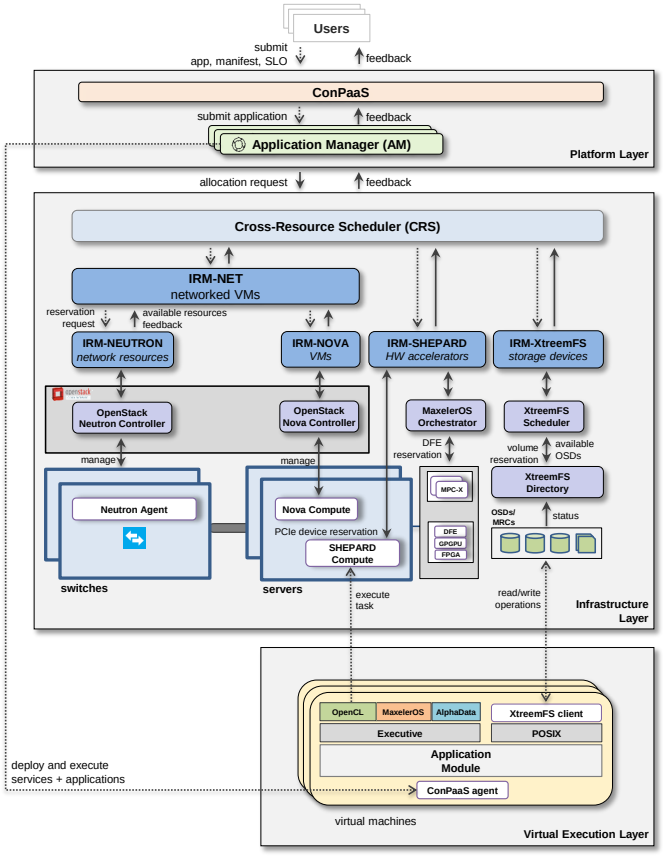


Figure 5: The HARNESS architecture is composed by three layers: a Platform layer that is responsible for managing applications, an Infrastructure layer where cloud resources are managed, and a Virtual Execution layer where applications are deployed and executed.

3 Prototype Description

In this section we describe the HARNESS prototype, which we present in Figure 5. The components of this architecture were designed and implemented by 4 HARNESS teams, which target a specific technical area: compute, communication, storage and platform. Each technical team developed their software components autonomously, however the integration process required each component to adhere to the HARNESS API (Section 2.3).

The HARNESS prototype described in this section has been implemented to validate our approach, and 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, and the Imperial 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 resources, and (3) a *virtual execution* layer where the applications actually run. Next, we explain each HARNESS layer in more detail.

3.1 The Platform Layer

The Platform layer includes the *Frontend*, the *Director* and the Application Manager components:

- **The Frontend** is a Web server that provides a graphical interface where users can manage their applications. To deploy an application, a user must upload an application manifest and a SLO (Service-Level Objective) document [11]. The application manifest describes the structure of an application and the resources it needs, while the SLO specifies the functional and non-functional parameters of one particular execution of the application. The purpose of the *Frontend* is to provide an intuitive graphical user interface for the end users, and translate all user actions into HTTPS/JSON requests which are sent to the *Director*.
- **The Director** 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 [12]. 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 SLO, 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* or an Application Manager needs to provision resources in the HARNESS platform (either to create a new application manager or to run an actual application), it sends a resource provisioning request to the Infrastructure layer.

3.2 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 2.2. 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*, *IRM-NOVA*, *IRM-NEUTRON*, *IRM-SHEPARD* and *IRM-XtreemFS*. Beneath the IRMs, we have components that manage specific resources, which include OpenStack, SHEPARD, MaxelerOS orchestrator and the XtreemFS scheduler.

- **The Cross-Resource Scheduler (CRS)** is in charge of handling resource provisioning requests [12]. In particular, 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 Nova Resource Manager (IRM-NOVA)** is in charge of managing virtual machines running on traditional server machines. It is a thin layer that converts resource-agnostic provisioning requests issued by the CRS into OpenStack Nova requests. Managing the creation of virtual machines over a pool of physical machines is a well-studied problem, so we rely on the popular and well-supported OpenStack Nova[1] to supply this functionality.
- **The SHEPARD Resource Manager (IRM-SHEPARD)** is in charge of managing hardware accelerator resources: GPGPUs, FPGAs and DFEs. It therefore translates resource-agnostic provisioning requests issued by the CRS into requests for locally installed accelerators, and networked DFEs via the MaxelerOS Orchestrator.
- **The Networked Resource Manager (IRM-NET)** provides the CRS with up-to-date maps of the physical resources which are part of the cloud. In particular, this map contains network proximity measurements realised 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 particular latency requirement. 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 Neutron Resource Manager (IRM-NEUTRON)** is in charge of managing subnet and public IP resources. It is a thin layer that converts resource-agnostic provisioning requests issued by the CRS into OpenStack Neutron requests.
- **The XtreemFS Resource Manager (IRM-XtreemFS)** is in charge of managing data storage device reservations. It therefore translates resource-agnostic provisioning requests issued by the CRS into concrete requests that can be processed by the XtreemFS Scheduler.

1. https://www.openstack.org/

The following components are part of the infrastructure and handle specific types of resources:

- **OpenStack Nova Controller** is the interface by which IRM-NOVA can request the creation/deletion of virtual machines on specific physical machines. It is a standard OpenStack component that we adapted to accept the placement decisions made by the CRS. The OpenStack Nova controller operates by issuing requests to the OpenStack Nova Compute components installed on each physical host.
- **OpenStack Nova Compute** is a daemon running on each physical host of the system, and which controls the management of virtual machines within the local physical machine. It is a standard OpenStack component.
- **The MaxelerOS Orchestrator** supports the allocation of networked DFEs located in MPC-X chassis. The MaxelerOS Orchestrator provides a way to reserve DFEs for IRM-SHEPARD. These accelerators will then be available to applications over the local network.
- **SHEPARD Compute** is a daemon running in each physical host, which provides information to *IRM-SHEPARD* about available hardware accelerators installed in the physical host, and performs reservation of those resources.
- **OpenStack Neutron Controller** is the interface by which IRM-NEUTRON can request the creation/deletion of subnet and public IP resources. The OpenStack Neutron Controller operates by issuing requests to the OpenStack Neutron Agent components installed on each physical host.
- **OpenStack Neutron Agent** is a daemon running on each physical host of the system, which controls the management of several services including DHCP and L3 networking. It is a standard OpenStack component.
- **XtreemFS** is fault-tolerant distributed file system that provides three kinds of services: (1) a DIR, (2) MRCs, and (3) OSDs [14], [18]. 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.
- **The XtreemFS Scheduler** handles the reservation/release of data volumes to be used by the HARNESS application [10]. Data volumes are characterised by their size, the type of accesses it is optimised for (random vs. sequential), and the number of provisioned IOPS. The XtreemFS Reservation Scheduler is in charge of ensuring data volume creation such that these requested properties are respected. The actual creation/deletion of data volumes is handled using the XtreemFS DIR, while the actual data and meta-data are respectively stored in the OSDs and MRC.

3.3 The Virtual Execution Layer

The Virtual Execution layer is composed of reserved VMs where the application is deployed and executed (Figure 5). In addition to the application itself, the VMs contain components (APIs and services) that support the deployment process, and also allow 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 its VM, 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.
- **The Executive** is a process that given a set of provisioned heterogeneous compute resources, selects the most appropriate resource for a given application task [9].
- **The XtreemFS client** is in charge of mounting XtreemFS volumes in the VMs and making them available as regular local directories [10].

4 Evaluation

In this section we report two case studies:

- **Executing HPC Applications on the Cloud:** This case study (Section 4.1) 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 4.2) demonstrates the benefits of managed networking when deploying distributed applications such as Hadoop MapReduce in a cloud platform. In this scenario, cloud tenants are able to reserve bandwidth which affect where jobs are deployed. This work was conducted in Grid'5000, with physical nodes located in Rennes and Nantes.

4.1 Executing HPC Applications on the Cloud

In this section, we evaluate the ability to deploy an HPC application on the HARNESS cloud platform. For this purpose we have selected Reverse Time Migration (RTM), one of the HARNESS demonstrators. 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. The problem of transforming this dataset into an image is computationally intensive and can be approached using a variety of techniques.

The experiments were conducted in the Imperial Cluster testbed, which includes three CPU machines, a DFE cluster harbouring 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, which can be downloaded here: http://wiki.seg.org/wiki/SEG_C3_45_shot.

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 initialisation scripts, are compressed into a tarball. This tarball is submitted to the HARNESS cloud platform frontend, 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 application on multiple resource configurations and capturing the execution time of each configuration. Associated with the performance model, we use a cost model which given a resource configuration, provides the cost to provision that configuration per unit of time. With the performance and cost models, the application manager can translate cost and performance objectives specified in the SLO, e.g. 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 (1024MB and 2048MB), number of dataflow engines (from 1 to 4) and storage performances (10MB/s and 250MB/s). The cost model is as follows:

$$cost(c) = c.num\_dfes \times 9e-1 + c.cpu\_cores \times 5e-4 + c.mem\_size \times 3e-5 + c.storage\_perf \times 1e-05$$

where for a given configuration c, $c.num\_dfes$ represents the number of DFEs, 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 euros/second.

Figure 6 shows the performance model generated by the Application Manager using the cost model described above. The application manager automatically selected and profiled 28 configurations, with 5 of these configurations part of the Pareto front. The number of profiled configurations is dependent on the profiling algorithm used by the Application Manager [13]. For each configuration, the application managers reserves the corresponding resources, and deploys the application. The initialisation script supplied with the application automatically detects the configuration attributes, and adapts the application to it. For instance, if the configuration specifies 8 CPU cores, then the initialisation script configures the `OMP_NUM_THREADS` environment variable to that number, to allow the application to fully utilise provisioned CPU resources.

As shown in Figure 6, there are four configurations highlighted at the top-right quadrant, which are the most expensive and slowest configurations, and thus the least desirable configurations. This is due to the use of slow storage (9MB/s) which dominates the performance of the job despite the use of dataflow engines. At the bottom-right quadrant, we highlight five configurations that are relatively inexpensive, however they perform slowly since they do not use dataflow engines. Finally, in the centre of the figure, we highlight three configurations, which use a limited number of CPU cores and DFEs, but they do not provide the best trade-off

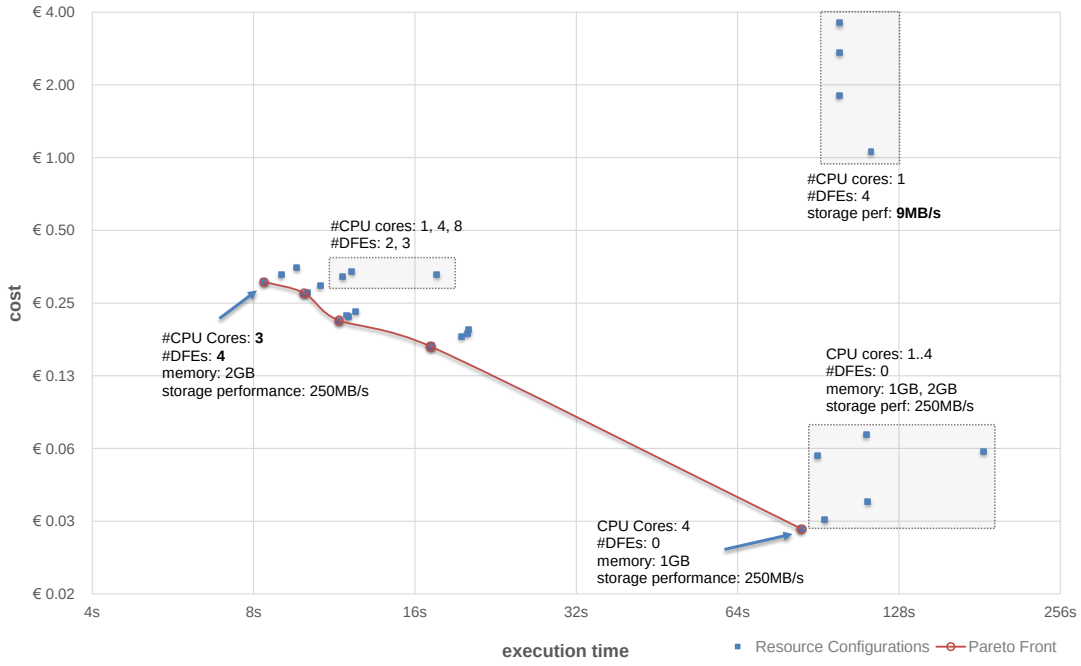


Figure 6: Performance model for the RTM demonstrator automatically generated by the Application Manager.

between price and execution time. Instead, the five configurations that provide the best trade-offs are those in the Pareto front, as listed below:

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

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 selects configuration C, while the cheapest configuration is A. The performance model in our example works only to a specific problem size and configuration, with other problem sizes requiring additional profiling information.

Exploiting different DFE topology reservations. As we have seen in the previous experiment, the RTM job deployed in HARNESS is *moldable* [7]. A *moldable* job can adapt to different resource configurations, thus being flexible in the number and type of resources provisioned. While a moldable job can only adjust to available resources at the start of the program, a *malleable* job can be reconfigured at run-time during program execution. Both types of jobs provide more flexibility than a *rigid* job which only executes on a single resource configuration. In this experiment, we wish to explore how RTM behaves when we exploit its moldable and malleable properties.

Our current implementation of the HARNESS platform supports the *DFE cluster* resource, as shown in Figure 7. A DFE cluster is composed by an arbitrary number of MPC-X boxes. An MPC-X box harbours a set of DFEs which are connected via a RING topology. This allows each DFE to connect directly to the CPU host, as well as with adjacent DFEs. The CPU host connects to the DFE cluster via Infiniband.

With the HARNESS platform, cloud tenants can reserve an

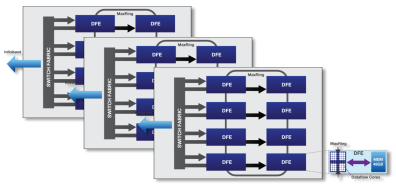
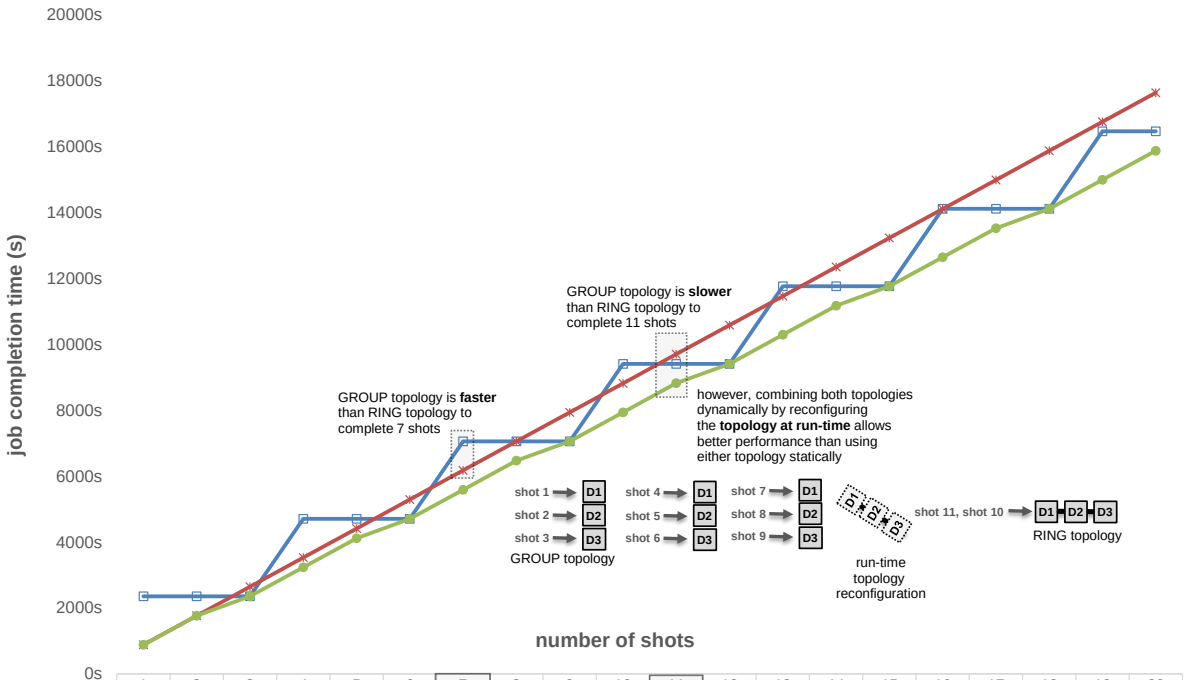


Figure 7: A DFE cluster is composed by an arbitrary number of MPC-X boxes. In this figure, a DFECluster manages 3 MPC-X boxes with 8 DFEs each. The DFEs in each MPC-X are connected physically via a RING interconnect.

arbitrary number of DFEs (subject to availability) and its topology using the application manifest. In our current implementation, we support three types of topologies: *SINGLETON*, *GROUP* and *RING*. More specifically, when reserving a *SINGLETON* with N DFEs, the platform (through MaxelerOS Orchestrator) returns DFEs from any MPC-X box. Reserving a *GROUP* with N DFEs returns resources within the same MPC-X box. Finally, reserving N DFEs with a *RING* topology results in N DFEs that are interconnected via ring interconnect. The difference between a SINGLETON and a GROUP reservation is that the latter allows a group of physical DFEs to be operated as a single virtual DFE [8].

Figure 8 shows the performance of a single RTM shot using different problem dimensions and number of DFEs. Multiple DFEs are connected via RING. The characterisation of these jobs is summarised in Table 2. 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 utilise 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 multi-shot RTM job. Figure 9 shows the results



	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 9: The effect of different topologies on RTM performance.

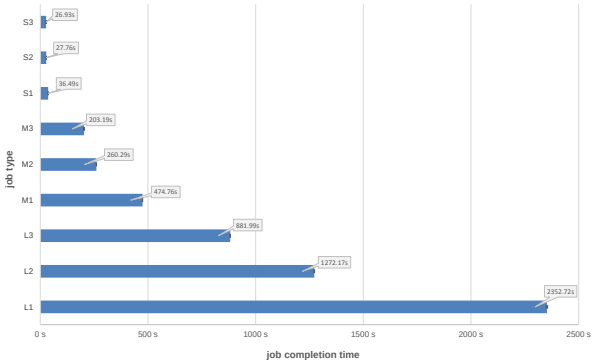


Figure 8: Performance of a single RTM shot using different problem dimensions (S,M,L) and number of DFEs (1, 2 and 3).

Table 2: 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

of completing an RTM job with varying number of shots. Each shot has a dimension of 600 × 600 × 600, running in 6000 iterations. For each number of shots, we compare the performance of running 3 DFEs independently (GROUP) against 3 DFEs interconnected (RING). The 3 independent DFEs can process three shots in parallel, while the 3 interconnected DFEs, working as a single compute resource, can only process shots sequentially. Each independent DFE is capable of computing a shot in 2352.7s, while the interconnected DFEs process a shot in 881.99s. 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 job is moldable, it can be adapted to run with one topology or another depending on the required number of shots.

We can further speed-up the computation by making the RTM job *malleable*, such that it adapts at runtime. As can be seen in Figure 9, depending on the number of shots, we can combine both types of topologies to reduce the execution. For instance, for 11 shots, we can execute 9 shots in three sequences in parallel followed by 2 shots done with the three DFEs interconnected. This combination yields the best performance (8821.1s) than if we had selected a static configuration (9410.9s for the parallel configuration and 9701.9s for the interconnected configuration). We expect larger problems to greatly benefit from this flexibility.

4.2 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 on a large-scale testbed such as Grid'5000. AdPredictor represents a class of modern industrial-scale applications, commonly known as *recommender systems*,

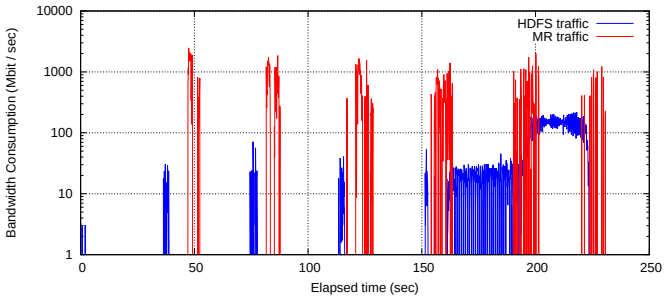


Figure 10: AdPredictor throughput over time

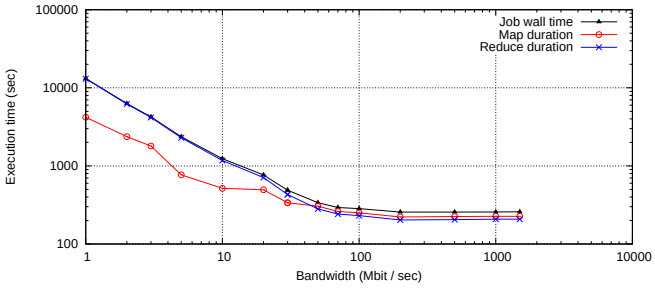


Figure 11: AdPredictor performance vs bandwidth

that target either open-source or proprietary on-line services. For example, one such service is Mendeley [15], a free reference organiser and academic social network that recommends related research articles based on user interests. Another such service is Bing, a commercial on-line 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 centres. The ACM 2012 KDD Cup *track2* dataset [1] has been used to evaluate AdPredictor.

Grid5000 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 120 optic fiber links and 72 points of presence.

Figure 10 reports the throughput of one MapReduce worker over time, where the steady throughput consumption peaks at approximately 200 Mbit/sec, excluding any bursts during shuffling. On the other hand, Figure 11 presents the results of an AdPredictor job using the same configuration by varying the available bandwidth between the worker compute hosts. It can be seen that the application exhibits degradation below 200 Mbit/sec, which is consistent with our measurements in Figure 10.

Consequently, a tenant deploying an AdPredictor application on a cloud platform that does not provide any guarantees could have his resources reserved in a low-bandwidth environment, hindering the performance of her application. HARNESS addresses these issues by exposing network bandwidth and latency as resources and constraints, respectively. Applications may define and submit their desired network performance guarantees and the underlying infrastructure will provision them, if possible.

Next, we report three scenarios conducted in Grid'5000: (a) scheduling without network constraints; (b) scheduling using bandwidth reservation requests; (c) scheduling with service-level

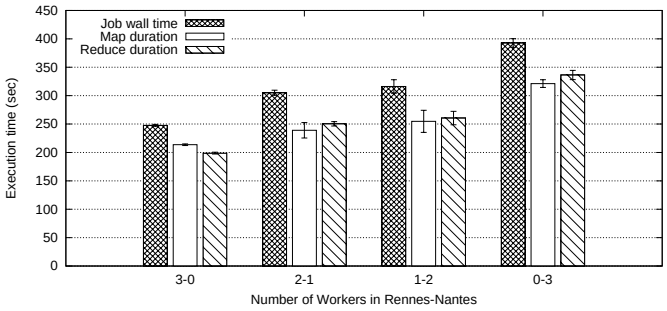


Figure 12: AdPredictor performance vs resource placement

objectives. The testbed used consists of 8 physical nodes, across two different sites: *Rennes* (4 nodes) and *Nantes* (4 nodes). While both sites offer high-speed gigabit connectivity of 1500 Mbit/sec, we have emulated heavy network congestion in Nantes, so that the throughput of any network flow in Nantes is limited to 500 Mbit/sec.

Scheduling without network constraints. In the first experiment we request 1 *master* and three *worker* instances without specifying network constraints. Figure 12 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 congested Nantes cluster, or across both sites. Nevertheless, in this scenario the tenant has no control over the final placement.

Scheduling using bandwidth reservation requests. In order to eliminate the suboptimal placements presented in Figure 12, labelled 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 the following allocation request in the application manifest:

```
{
    "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
    },
```

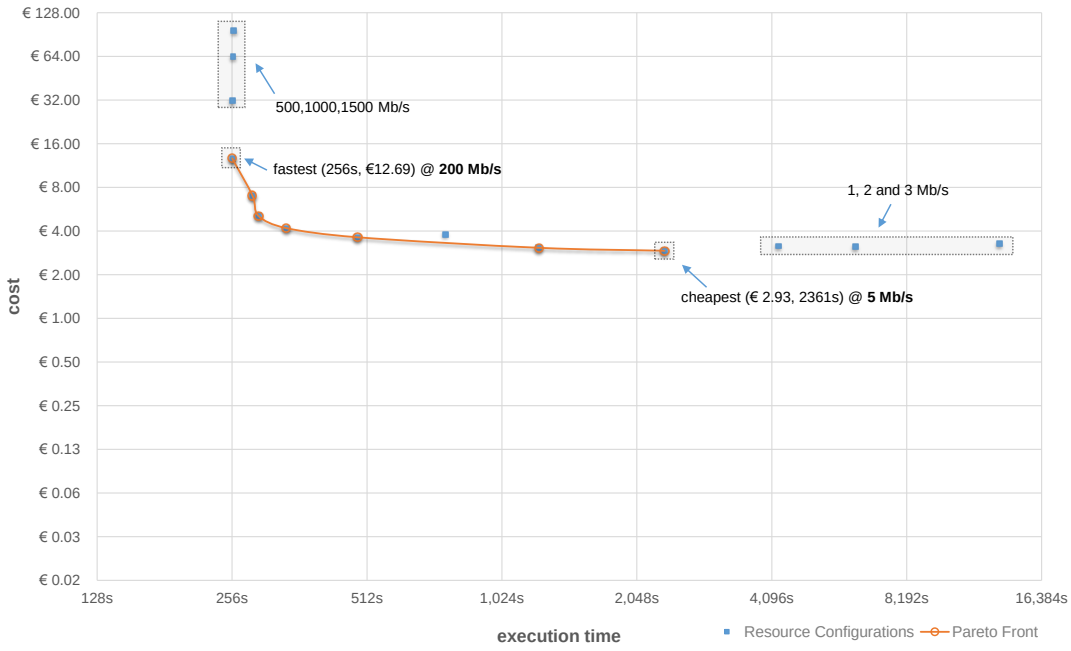


Figure 13: Performance/cost model generated for AdPredictor running on Grid'5000

```
    "Link2": {
        "Type": "Link",
        "Source": "Worker2",
        "Target": "Worker3",
        "Bandwidth": 300
    },
    "Link3": {
        "Type": "Link",
        "Source": "Worker1",
        "Target": "Worker3",
        "Bandwidth": 300
    }
}
```

This request demands a reservation of 300 Mbit/sec 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.

Scheduling with service-level objectives. In our 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 cost of reserving resources. In HARNESS, such objectives are specified as a service-level objective (SLO). In order for the HARNESS platform to derive a resource configuration that can meet performance or cost objectives, it needs to have a performance/cost model. This can be automatically generated by the Application Manager, or provided by the user. For this experiment, we have generated a performance model based on profiling AdPredictor with different bandwidth reservation requests.

The cost model is as follows:

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

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

Figure 13 presents the performance/cost model of AdPredictor running on Grid'5000. It contains 14 points where we vary the bandwidth requirements from 1Mb/s to 1.5Gb/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 front. These configurations are then selected to satisfy objectives in terms of pricing (the cheapest configuration costs €2.93 but requires 39 minutes to complete) or in terms of completion time (the fastest configuration completes the job in 256s but costs €12.69).

5 Conclusion

In this paper we presented the HARNESS integrated platform architecture, including the final implementation of the HARNESS cloud platform, which is capable of managing resources such as CPUs, DFEs, network middleboxes, and SSDs. The HARNESS architecture is based on a novel hierarchical management approach, designed to make cloud platform systems resilient to new forms of heterogeneity—introducing new types of resources does not result in having to redesign the entire system.

We developed a fully functional prototype of a HARNESS cloud platform based on our approach, with the following two layers:

- The HARNESS platform layer automates the process of selecting resources to satisfy application-specific goals (e.g., low completion time and/or low monetary cost), as specified by the cloud tenants. To do so, it resorts to automatic application profiling to build performance models. The platform takes advantage of 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 HARNESS infrastructure layer is managed by a collection of resource managers, each designed for a specific type of devices incorporated in the HARNESS cloud. These managers deal with provisioning and making effective use of reserved resources on behalf of a cloud tenant. Management technologies include: (a) MaxelerOS orchestrator for networked DFE reservations; (b) SHEPARD for hardware accelerator reservations; and (c) XtreemFS for heterogeneous storage reservations. A cross-resource scheduler enfolds all these resource-specific managers to optimise multi-tenant reservation requests alongside (optional) network placement constraints.

To maximise resource utilisation, we develop a set of virtualisation technologies that allow physical devices, some of them originally designed to be single-tenant, to be effectively shared in a multi-tenant environment. Our current virtualisation technologies include: (a) a virtualised DFE, backed up by a group of physical DFEs that can grow or shrink depending on the workload; (b) an XtreemFS virtualised storage object that enables multiple reservations with different requirements; and (c) network virtualisation exposing tenants the ability to deploy general network functions (e.g., Layer-2 switching, Layer-3 forwarding) and application-specific network functions (e.g., an HTTP load balancer).

By fully embracing heterogeneity, HARNESS allows a more complex and richer context in which to make price/performance trade offs, bringing wholly new degrees of freedom to the cloud resource allocation and optimisation problem.

ACKNOWLEDGMENT

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).

REFERENCES

[1] ACM SIGKDD. Predict the click-through rate of ads given the query and user information. Available at http://www.kddcup2012.org/c/kddcup2012-track2/.
[2] Apache Software Foundation. Apache Deltacloud drivers. Available at https://deltacloud.apache.org/drivers.html.
[3] Apache Software Foundation. Apache Deltacloud Project. Available at http://deltacloud.apache.org.
[4] Apache Software Foundation. Apache Libcloud Project. Available at http://libcloud.apache.org.
[5] Cloud Management Working Group (CMWG). Cloud Infrastructure Management Interface (CIMI) Specification. Available at http://www.dmtf.org/standards/cmwg.
[6] Eucalyptus Project. Available at http://www.eucalyptus.com/.
[7] D. G. Feitelson and L. Rudolph. Towards convergence in job schedulers for parallel supercomputers. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, IPPS '96, pages 1–26. Springer-Verlag, 1996.
[8] FP7 HARNESS Consortium. Monitoring and virtualisation report. Project Deliverable D3.2, 2014.
[9] FP7 HARNESS Consortium. Experimental Hardware Prototype and Report (initial). Project Deliverable D3.4.1, 2013.
[10] FP7 HARNESS Consortium. Data storage component (initial). Project Deliverable D5.4.1, 2013.
[11] FP7 HARNESS Consortium. Application characterisation report. Project Deliverable D6.1, 2013.
[12] FP7 HARNESS Consortium. Heterogeneous platform implementation (initial). Project Deliverable D6.3.1, 2013.
[13] FP7 HARNESS Consortium. Heterogeneous platform implementation (updated). Project Deliverable D6.3.3, 2015.
[14] F. Hupfeld, T. Cortes, B. Kolbeck, J. Stender, E. Focht, M. Hess, J. Malo, J. Marti, and E. Cesario. The XtreemFS architecture—A case for object-based file systems in Grids. *Concurrency and Computation: Practice and Experience*, 20(17):2049–2060, 2008.
[15] Mendeley. Available at http://www.mendeley.com/.
[16] 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.
[17] 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.
[18] J. Stender, M. Berlin, and A. Reinefeld. XtreemFS – a file system for the cloud. In D. Kyriazis, A. Voulodimos, S. V. Gogouvitis, and T. Varvarigou, editors, *Data Intensive Storage Services for Cloud Environments*. IGI Global, 2013.
[19] M. Stillwell, F. Vivien, and 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.
[20] Storage Networking Industry Association (SNIA). Cloud Data Management Interface (CDMI) v1.1.1. Available at http://www.snia.org/sites/default/files/CDMI_Spec_v1.1.1.pdf.
[21] Storage Networking Industry Association (SNIA). S3 and CDMI. Available at http://www.snia.org/sites/default/files/S3-like_CDMI_v1.0.pdf.