Heterogeneous Resource Management and Orchestration in Cloud Environments

Dapeng Dong(✉), Huanhuan Xiong, Gabriel G. Castañé, Paul Stack, and John P. Morrison

Department of Computer Science, University College Cork, Cork T12 YN60, Ireland
{d.dong,h.xiong,g.castane,p.stack,j.morrison}@cs.ucc.ie

Abstract. The addition of heterogeneous resources to conventional homogeneous cloud environments has enabled clouds to embrace a wide variety of new applications that heretofore were traditionally confined to specialized computing environments. The enhanced and extended features offered by heterogeneous resources enable service offerings that pose challenges to traditional cloud management throughout the entire service delivery stack. The accelerated uptake of heterogeneous resources is exacerbating these challenges, which no longer can be efficiently addressed in an ad-hoc manner. Therefore, an integrated approach to heterogeneous resource management that is cognizant of the unique advantages of different hardware types is needed. In this paper, two candidate approaches, a platform-integration scheme and a server-integration scheme, are introduced to address this management challenge. The platform-integration scheme integrates and coordinates the management of various coexisting resource managers and associated environments each of which may be managing resources of different types using the most appropriate resource abstraction method. In contrast, the server-integration scheme provides a single, lower level, fine-grained management mechanism across all hardware resource types. Ultimately, the goal of each schemes is to provide a unified view of resources from a capability perspective to consumers.

Keywords: Architecture · Heterogeneous resource · Platform integration Cloud · HPC

1 Introduction

The employment of various advanced technologies, such as virtualization and more recently, containerization, for managing and organizing resources in cloud environments has yielded several distinct system features, such as resource elasticity, system scalability, application load-balancing, configuration flexibility, cost-effective usage models and rapid deployment. Moreover, recent evidence shows an increased demand for support for High Performance Computing (HPC) applications in the cloud. For example, weather forecasting, medical imaging and computational fluid dynamics, that have traditionally been confined to cluster environments are now being migrated to the cloud. To effectively support applications of this type and to demonstrate that comparable performance can be achieved in the cloud, specialized hardware, such as, Graphical Processing Units (GPUs), Many-Integrated-Core processors (MICs) and Data-Flow

Engines (DFEs), and dedicated networking configurations, including 40 Gb/s Ethernet and InfiniBand, are being incorporated into the cloud infrastructure. Consequently, cloud service providers have begun to offer specialized services, for example, Amazon EC2 Cluster Compute, EC2 F1 Instances [1] and Microsoft FPGA-based cloud [2] are all designed to support these high-end applications. The introduction of a wide range of hardware and associated configurations to conventional homogeneous cloud environments is introducing heterogeneity and associated challenges for effectively integrating and efficiently managing heterogeneous resources and the heterogeneity arising from new hardware architectures, diverse computational abilities, diverse power usage patterns, mixed operating system architectures and specialized software libraries. This evolution is having a significant impact on the transitional cloud architectures, and a re-consideration of the organization of the physical hardware resources in the cloud infrastructure layer, the resource management and scheduling approaches in the cloud management layer, and the service orchestration and resource representation in service delivery layer is becoming necessary.

Several cloud management platforms exist for managing virtualized environments (e.g., OpenStack Nova [3]), container environments (e.g., Kubernetes [4], Mesos [5] and Docker Swarm [6,7]), containers in virtualized environments (e.g., Magnum [8]), bare metal servers (e.g., Ironic [9]). These platforms have sufficiently matured and have begun to find practical applications in many public and private clouds. Traditional clouds typically support only one of these platforms and this limits the structure of the cloud environment in terms of hardware diversity. For instance, in a virtualized environment, only certain models and types of computation accelerators (e.g., GPUs) can be accessed by virtual machines with additional configurations on both the underlying hardware (e.g., CPU and motherboard) and software (e.g., Hypervisor and host operating system). In contrast, containers can directly use many of the existing computation accelerators, but have limited features, especially for networking where advanced firewall and load-balancing are noticeably absent. Thus, having multiple abstraction methods simultaneously available in a single cloud deployment [10] is desirable. If a cloud provider supports more than one of these platforms simultaneously, each is provided in isolation from the rest in a manner that effectively partitions the cloud resources among them, thus, creating a situation where those resources can not be shared across platforms.

Without doubt, heterogeneity complicates resource management and resource allocation. In current homogeneous environments, resource allocation is typically formulated using multi-objective optimization equations involving resource availability (e.g., CPU cores, system memory and storage space) and system requirements (e.g., host-affinity and load-balancing). To make decisions in a timely fashion, relaxed algorithms (meta-heuristics or greedy algorithms) are often used. Since heterogeneity offers considerably more features, these calculations become consequently more complex. Improved organization at the system level offers a potential pathway for efficient resource allocation. However, this approach assumes the existence of a unified platform for managing heterogeneous resources. In this paper, two implementation schemes for such a unified platform are introduced. They are referred to as a platform-integration scheme and a server-integration scheme.

This paper is organized as follows. A brief introduction to the background of this work and a consideration of related work are presented in Sect. 2. The proposed unified platform schemes are outlined in Sect. 3, and a use case application demonstrating the platform-integration scheme is given in Sect. 4. Section 5 concludes the paper by highlighting the main ideas of this research and by indicating some potentially fruitful future directions.

2 Background and Related Work

The cloud computing paradigm has shifted the focus of data center management from providing bare-metal resources to providing virtual resources to the end user. These advantages have long been demonstrated in production cloud environments, such as the Amazon EC2, the Microsoft Azure, and the Google Cloud Platforms. Virtualization enhances cloud management by enabling flexible resource configuration and deployment, efficient use of resources, and by offering opportunities for reducing power consumption.

Virtualization and containerization are the two dominant technologies used for managing and abstracting computational resources. Virtualization is generally achieved through abstracting hardware components into logical objects. This abstraction can be realized by hardware emulation and/or by time sharing of a hardware component between multiple processes. A software component that provides virtualization functions is commonly referred to as a *hypervisor*, historically named as a Virtual Machine Manager (VMM). A hypervisor is responsible for providing instances of virtual environments identical to the underlying physical server with minimum performance cost, while retaining full control of the physical resources [11]. It allows for architecturally diverse operating systems to coexist and to simultaneously run on the same physical server. A complete operating system can be installed and run in a virtual environment exactly the same way that it runs on a physical server. This instance of the operating system is often known as a *guest* Virtual Machine (VM) [12]. All hardware resource related operations, which are initiated from guest VMs, are under the control of the hypervisor, and the hypervisor executes these operations on the actual hardware on behalf of each guest VM.

Containerization is another type of virtualization. Technically, containerization technology provides isolated application execution environments at the operating system level. It uses Linux native functions, mainly the control groups (*cgroups*) and the *namespace*, to isolate applications/processes. Because containers do not use hypervisor-like middleware, multiple container applications can share common libraries and hardware drivers installed on the host operating system. This also makes a container application lightweight and easier for it to access specialized hardware, such as GPUs and MICs. On the other hand, a container application is less secure because of its shared environment and offers less functionality; it is essentially an application wrapping mechanism. In practical deployments, selecting either virtualization or containerization technologies for managing a cloud environment depends on the business goals set out by Cloud Service Provider (CSP). From an architecture design perspective, containers are suitable for running applications, whereas VMs are suitable for building virtual Information Technology (IT) infrastructures.

In a cloud environment, each virtualization or containerization technology provides desired features such as elasticity, scalability and high-availability. This requires a hyper-level management framework that can coordinate virtual resources across physical servers at large scale. The following sections review the architectural design of several widely adopted Infrastructure-as-a-Service (IaaS) management frameworks focusing on computational resource management.

2.1 Virtualization Management Frameworks

OpenStack [3] is an open-source cloud platform focusing on the management of virtualized environments. In particular, for managing computational resources, OpenStack uses a front-end API server for receiving and responding to requests for resources. Allocating a computational resource will require various other components be associated with it, such as, networking, storage and security groups. This can be a very complex process when multiple simultaneous requests, with different configurations trying to acquire globally available resources are made. In order to reduce this complexity of this process, requests are forwarded to mediator service, known as the *nova-conductor*. The *nova-conductor* coordinates various components (e.g., networking, image, storage, and compute) for each request, and multiple instances of the *nova-conductor* can be created to deal with a high-volume of requests. The *nova-conductor* first uses a scheduler service (the *nova-scheduler*) to locate a group of potential physical server(s) that meet specified requirements, specified requirements, such as, the number of CPU cores, the size of memory, and the required storage space. Subsequently, those candidate physical servers are further filtered, in a iterative manner, based on the preferences and criteria (also know as *weights*) specified by the user and/or the CSP. The requested resources will subsequently be deployed by the *nova-compute* service (by calling Hypervisor specific APIs) on the most appropriate physical servers [13].

OpenNebula [14] provisions VMs in a similar manner to OpenStack. It uses a *front-end* service to deal with requests and resource management. A request for VMs is first formulated into a VM template, this template is forwarded to a *scheduler* service [15], which selects available resources based on the system requirements and/or user preferences, such as, Packing, Striping and Load-aware polices [16]. The *front-end* service coordinates VM deployment on the selected server(s) by calling Hypervisor specific APIs.

Nimbus [17] is another IaaS management framework for scientific users. It takes a simpler approach for managing resources than those mentioned above. Architecturally, Nimbus consists of three main core components including a *Nimbus IaaS central service*, a storage service (*Cumulus*) and *VMM control services*, running on each server basis. The *Nimbus IaaS central service* acts as a middleware between cloud end-users and cloud resources. From an end-user perspective, the central service is the server that deals with requests for resources; from a CSP point of view, the central service is a client that initiates requests for resource deployment to the *VMM control service(s)* on the selected server(s). Thus, Nimbus implements a client-server model. The client-server model greatly simplifies the Nimbus architecture and provides a robust platform. On the other hand, this model may limit scalability. Nevertheless, Nimbus supports cloud federations that can be formed by different cloud platforms and this is achieved through a centralized account management service [18].

2.2 Containerization Management Frameworks

Borg [19] is a former proprietary Google platform used for managing large-scale container environments. Borg manages tens of thousands of servers simultaneously. The Borg architecture consists of three main components including *Borg masters*, *job schedulers*, and *Borglet* agents. A typical Borg instance consists of a single *Borg master*, a single *job scheduler* and multiple *Borglet* agents. The *Borg master* is the central point for managing and scheduling jobs and requests. A *Borg master* and *job scheduler* are replicated in several copies for purposes of high-availability, however, only a single *Borg master* and a single *job scheduler* are active in the system at any one time. The *Borg master* is responsible for dealing with requests for deploying jobs and the *job scheduler* searches for suitable servers to host tasks. The actual deployment of the job is carried out by a *Borglet* agent on the selected server (multiple tasks on the same server are separated and distinguished by the Linux kernel functions *cgroups* and *namespace*).

Borg presents a centralized management approach. This also requires *Borg masters* and *job schedulers* (the original and all replicas used for high-availability) to be large enough to scale out as required. The Borg *job scheduler* may potentially manage a very high volume of jobs simultaneously, this has made Borg more suitable for long-running services and batch jobs, since those job profiles reduce the load on the job scheduler.

Omega [20] is an enhancement of Borg system's scheduler architecture. It employs multiple schedulers working in parallel to speed up resource allocation and job scheduling. Each scheduler maintains the complete state of all available resources and decisions are made by each scheduler, independently. Conflicting resource allocations will be determined in resource deployment phase, and one or both of the conflicting requests will be returned to their originating scheduler for rescheduling. Kubernetes [21,22] is the most recent evolution of Google's data center management technology. Architecturally, Kubernetes implements a master-worker model. The master runs an *API service* for dealing with requests, a cluster state maintenance service (*Etcd*) for tracking data center resource information, and a scheduler for locating resources. On each computational node, a local container management service (*kubelet*) is used for managing container life-cycles and a network proxy service (Proxy) is used for establishing inter- and intra-communications between containers and the Internet. Notably, containers are not managed individually. A collection of containers is organized together and managed as a single entity. This is commonly referred to as a *Pod*. The concepts of the *Pod* reflects the service management philosophy that a large cloud application should be decomposed into a set of self-contained services; each service carries a single function per container basis; and all services belonging to an application should be managed together. Additionally, the use of *Pods* also makes Kubernetes more scalable.

Docker Swarm [6,7] mimics *Pod* concept. It provides a flexible and easy way for building virtual clusters on demand in which cluster members can be distributed across physical servers. Members of a swarm cluster are connected through a designated overlay network. A *swarm* is conceptually a virtual cluster. Common services or services belong to the same application, can be managed and grouped into the same *swarm*.

Architecturally, a *swarm* consists of a *swarm master* and *swarm workers*. Any available computational nodes in a data center can freely join and leave a swarm, and provides the basis for application/service elasticity.

Mesos [5] is another management platform that is based on a master/worker architecture. Mesos enables multiple different scheduling frameworks to manage the same environment. This is achieved by employing a coordinator service that assigns controls on resources to a single scheduler during its decision making processes. This can potentially lead to an inefficient use of resources when the request is lightweight and available resources are significantly large.

Alibaba Inc. has created the Fuxi [23] platform for supporting its large scale worldwide *e*-commerce business. The Fuxi architecture design focuses on scalability and fault tolerance. It consists of three main components including a *FuxiMaster*, a *FuxiAgent* and an *ApplicationMaster* on each physical server. The *FuxiMaster* is responsible for receiving and responding to job requests and for locating a *FuxiAgent* suitable for each individual job. The designated *FuxiAgent* spawns an *ApplicationMaster* to handle the job and potentially split the job into smaller tasks depending on the job type. The *ApplicationMaster* then initiates requests to the *FuxiMaster* for resources. When the *FuxiMaster* returns a list of *FuxiAgents* that contain sufficient resources for the job, the *ApplicationMaster* starts issuing commands directly to the selected *FuxiAgents* to start the job. In comparison with common container technologies that use *cgroups* to control resource assignment and kernel *namespace* for isolating task execution environments, Fuxi continuous to use *cgroups* to control resource assignment, but makes use of independent *sandboxes* for isolating task execution environments.

2.3 Bare-Metal Management Frameworks

Although virtualization and containerization are the main technologies used for managing resources, in many situations, managing bare-metal resources are still important. For example, provisioning data center infrastructure and providing high-performance servers for heavily loaded database systems and specialized computation accelerators remain an important activity.

Bare-metal server management is technically different from managing containerized and virtualized environments. Since bare-metal servers do not have pre-installed operating systems, vendor-specific chip-level management modules such as, Intelligent Platform Management Interface (IPMI) and Preboot Execution Environment (PXE) must be used. In general, provisioning a new server requires the sending of a request to an API server, a controller service is then invoked to identify a target physical server by matching user specified criteria, such as, CPU architecture and system memory size. The controller service then prepares for the operating system images or *ramdisk* to be installed on the selected server and issues IPMI commands to the server for network booting and operating system image or *ramdisk* installation via PXE. After the image loading and installation processes, the status and access methods are handed to the end-user. Ironic [9], Razor [24], and Foreman [25] are several typical implementations of such a scheme.

In summary, modern data center management platforms still operate a client-server model and this model continuous to scales. Architecturally, those management platforms consist of three common components including a front-end facing API server, a coordinator service (e.g., schedulers and resource management), and back-end agents (e.g., Hypervisors, *Borglet* and *Kubelet*). To further improve on scalability, cloud federation can be used and is commonly implemented through centralized account management mechanisms incorporated with networking inter-routing schemes.

3 Heterogeneous Resource Integration

Existing data center management platforms typically employ a single resource abstraction method (such as, vitalization or containerization). These are efficient and effective for managing homogeneous resources. The increasing demand for supporting versatile high-performance accelerators and high-throughput network connections are changing the nature of a data center from a homogeneous environment to one that is more heterogeneous. This poses challenges to existing data center management platforms and how they accommodate various types of computational hardware resources. Orchestrating services and resources with complex configurations to meet user- and/or system-specific requirements is thus becoming increasingly more difficult. As system functions become more versatile, the complexity of the system is also increased. However, this complexity must be made transparent to end-users. Consequently this requires an adjustment the paradigms used for delivering services. In the following sections, a blueprint-oriented service delivery model and two integration schemes (a platform-integration scheme and a server-integration scheme) for managing and unifying heterogeneous resources in cloud environments are introduced.

3.1 Service Delivery Model

To fully exploit versatile service and resource options offered by heterogeneous resource, a careful and a considered approach to manage these resources is necessary. This can be challenging for both the service provider and for the service consumer, especially, when the components of a cloud application may require the deployment on different types of resources. Moreover, leaving aside the difficulties of working with heterogeneous hardware environment, expert knowledge related to the deployment of cloud application components is usually required to fully exploit these hardware resources and accelerators. Configuration complexity and deep domain-specific knowledge should be made transparent to end-users. Thus, an approach is taken that allows end-users to compose their tasks into a workflow of constituent service(s). Workflows of this kind are often referred to as *blueprints*.

An application blueprint can be visualized as a graph that expresses the business logic and intra-relationships of application components. Deploying an application blueprint effectively deploys the set of services that comprise the application. In an homogeneous cloud, deploying a set of services results in each being hosted by resources of a single type. This represented the state of the industry. However, in making the transition to an heterogeneous cloud a blueprint involves assigning services to the most appropriate heterogeneous resources. Depending on the nature of the application and/or user preferences, a service or a group of services can thus be assigned to the same resource or indeed to different resources of different types, when necessary. This advancement requires adjusting the resource provisioning models appropriately and is addressed in Sects. 3.3 and 3.4.

3.2 System Workflow

Given the blueprint-oriented service delivery model, the system workflow is shown in Fig. 1. An application blueprint is composed by end-users and first submitted to an API Server. The API Server is responsible for receiving and responding to end-user requests and forward the a blueprint to an instance of a Resource Coordinator. Many Resource Coordinators may potentially work in parallel to load-balance large volume of requests. Each application blueprint is processed by a single Resource Coordinator. The Resource Coordinator decomposes the blueprint into sub-groups of resource requests according to the resource abstraction types. For example, a complex blueprint may consists of many services and each of the service may require to be deployed on different types of hardware resources. For instance, a blueprint may be described as an application that requires front-end web servers to collect data which is subsequently processed using accelerators, thus, the resources required for this blueprint deployment may be a set of VMs running on CPUs, and a set of containers running on servers having Xeon Phi co-processors. After the blueprint decomposition process, the Resource Coordinator analyses the relationships between sub-groups of the resource requests and makes further amendments to the blueprint. The amendments are mainly made for realizing communications between sub-groups. The Resource Coordinator then forwards each sub-group of resource requests to designated Virtual Resource Partitions (VRPs) that are managed by corresponding management platforms. Details about VRPs and management platforms are given in Sects. 3.3, and 3.4.

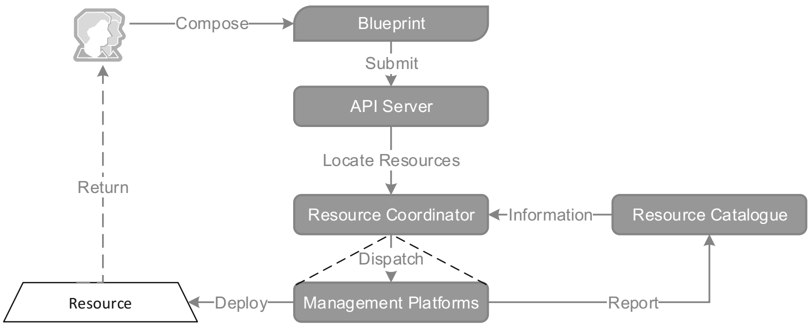


Fig. 1. System workflow.

3.3 The Platform-Integration Scheme

Different types of hardware resources require appropriate resource management techniques. The mechanisms for integration heterogeneous resources and their respective

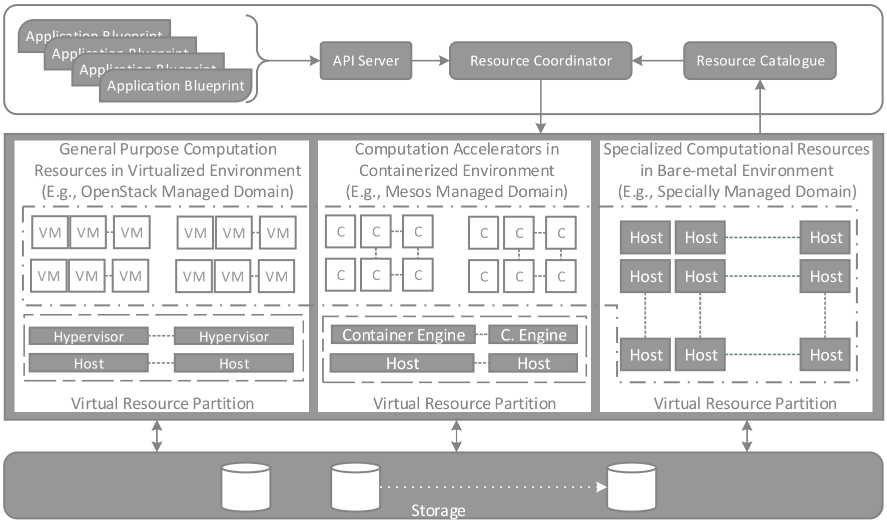

Application Blueprint
API Server
Resource Coordinator
Resource Catalogue
General Purpose Computation Resources in Virtualized Environment (E.g., OpenStack Managed Domain)
Computation Accelerators in Containerized Environment (E.g., Mesos Managed Domain)
Specialized Computational Resources in Bare-metal Environment (E.g., Specially Managed Domain)
VM
C
Host
Hypervisor
Container Engine
C. Engine
Virtual Resource Partition
Storage

Fig. 2. Managing and accommodating heterogeneous hardware resources through multiple integrated platforms.

management techniques in a single unified scheme. An overview of the proposed platform-integration scheme is shown in Fig. 2. In this scheme, hardware resources are virtually partitioned based on the resource abstraction/access methods (virtualization, containerization and bare metal) most appropriate for the respective hardware type. A corresponding management framework is then adopted to manage groups of hardware of the same type. A central Resource Coordinator component is provided as an interface to be used by end-users to deploy applications on the underlining resources. More importantly, the Resource Coordinator component coordinates the deployment for the application blueprint components on, potentially, various types of resources across those virtual partitions.

Heterogeneous hardware resources are managed through the designated platforms. This may raise interoperability issues, however, as each platform manages a virtual resource partition, in the same management domain, the resulting interoperability issues reduced to a technical integration action and are not exacerbated by having to consider the interests of multiple entities. Figure 2 shows how the integration scheme may use OpenStack to manage virtual environment, Mesos to manage container environment, and Ironic [9] to manage bare-metal servers. Each platform offers a different set of Application Programming Interfaces (APIs) and utilities for similar resource management operations, such as, creating virtual machines and/or containers. The Resource Coordinator uses a Plug & Play Interface that defines a set of common operations for managing underlying resources, and these operations are then translated to platform-specific API calls or commands using the Plug & Play implementation modules to carry out service deployment processes. Additionally, storage systems are organized and managed independently. Processing units can be easily configured to use volume-based and/or network attached storage systems.

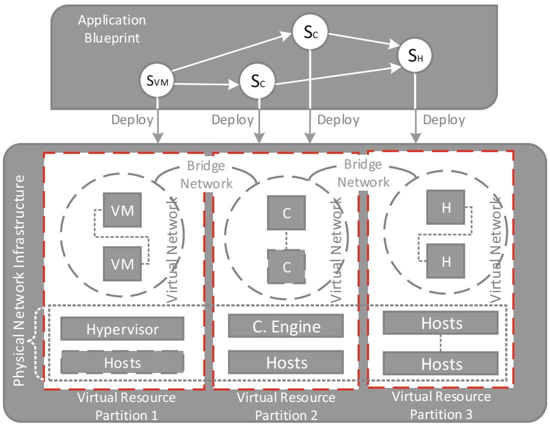


Fig. 3. Networking integration scheme using bringing network.

Networking Integration Strategy. Two schemes are available for networking integration. The first scheme is to treat networking in each VRP, independently, as shown in Fig. 3. Application components are deployed independently in their corresponding VRP and virtual networks are created accordingly within each VRP. At the same time, network bridges are created in order to establish communication channels across VRPs. The scheme does not require any modification to the respective resource management platforms. This gives the flexibility for integrating other resource management platforms, for example, Kubernetes and Docker Swarms, with existing environment. The concerns about this scheme arise from the differences associated with each of the networking approaches taken by each of the respective resource management platforms. Considering that different platforms offer different types of networking services at various level, for example, an OpenStack managed network uses the Neutron framework, which offers rich functionalities including firewalls, load-balancers, and security groups, etc., these may not be available in the container environment if it is managed by Mesos. As the available functional components are different from platform to platform, this will affect how an application blueprint can be created.

The second scheme employs the Neutron framework [26] for building and managing virtual network infrastructure. Figure 4 shows the simplified networking plan. All hardware resources are connected to the same physical networking infrastructure, but logically, they are managed by corresponding platforms, independently. From an end-user point of view, all resources are in a single resource pool. In the case that multiple components of a single application-blueprint need to be deployed on both VMs and containers, which are managed by different platforms, this requires a dedicated virtual network for the entire application-blueprint over the end-user (*tenant*) network. Thus, there is a need for a unified virtual network infrastructure management framework to be installed across all platforms, horizontally. In addition, the *tenant* networks must be managed in a seamless fashion. The second networking planning scheme adopts OpenStack Neutron for this purpose. In general, frameworks and services developed under the OpenStack *Big Tent Governance* natively support Neutron services. In contrast,

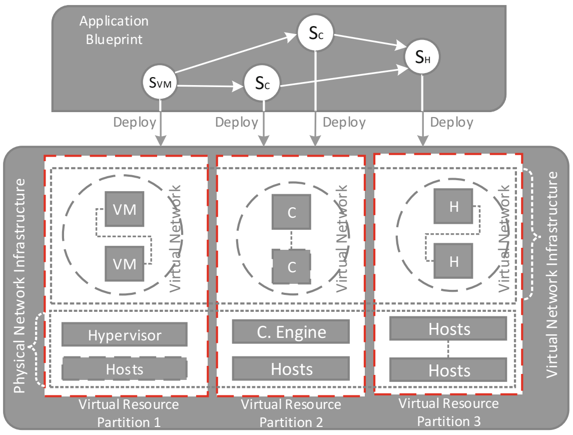


Fig. 4. Networking integration scheme using unified network framework across multiple platforms.

container technologies such as Kubernetes, Mesos, and Docker Swarm employ different networking models. For example, Kubernetes can use Flannel [27], Weave Net [28] frameworks operating in various modes; Docker uses libnetwork [29] by default. In the context of this work, the Kuryr network driver [30] is employed to link Neutron and container networks. Thus, end-users will experience seamless connections between all types of heterogeneous hardware resources.

3.4 The Server-Integration Scheme

In the second integration scheme, heterogeneous hardware is organized on a per server basis, as shown in Fig. 5. Each physical server is equipped with both general purpose processors and computation accelerators where they are applicable. Two types of networking interfaces, including high-speed interface(s) (e.g., InfiniBand or 40 Gb/s Ethernet) and standard-speed interface(s) (e.g., 1/10 Gb/s Ethernet) are also installed on a per server basis. Different types of networks are connected to their corresponding dedicated networking switches. In this configuration, a physical server is capable of offering high-performance computational resources for HPC applications as well as economical computational resources for general applications such as Web services.

The mixed hardware configuration also requires both container engine and hypervisor to coexist on the same physical server. This is because, in a virtualized environment, access to specialized computational accelerators (e.g., MICs, GPUs and DFEs), especially when dealing with various types and models of those accelerators, from a VM can be very problematic. It is generally requires both software (including operating system and hypervisor) and hardware (including CPU and motherboard) to support for passing through specialized accelerators to VMs. In contrast, a container application can directly use accelerators that have been already recognized by the underly host operating systems. Nevertheless, trade offs need to be taken into account when using different resource abstraction/access methods. For examples, VMs, as a complete operating

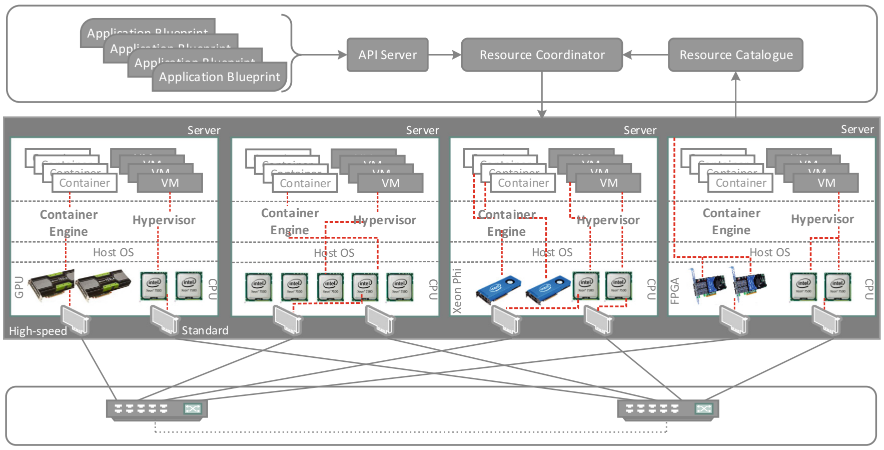


Fig. 5. Managing and accommodating heterogeneous hardware resources through hardware integration on a per server basis.

system, can provide all features that a standard operating system offers; a container provides a light-weight application execution environment which may result in better performance, but may be less flexible and secure. For applications to experience native performances or strict secure environment, for example heavily loaded database systems and banking transaction processing systems, the option for access to bare-metal servers is often needed. Thus, the coexistence of various resource abstraction methods on each individual server is desirable. Note that the coexistence of both container engine and hypervisor may affect the choices for selecting the types of hypervisor (Classic System VMs or Hosted VMs) [12].

At the management layer, all resources are registered with a Resource Catalogue and compatible computational resources are logically grouped together from an application perspective. Depending on the characteristics of an application blueprint to be deployed, the Resource Coordinator uses the information from the Resource Catalogue to make decisions on how and where to provision resources. One of the key features of the Server-Integration scheme is that it allows for various software and hardware components, including types of operating system, hypervisors, container engines, general purpose processors, computational accelerators, and different types of networking connections to be dynamically and flexibly combined together to meet application and system requirements.

3.5 Summary

The platform-integration scheme can provision heterogeneous resources through the integration of various existing platforms in which each platform manages a set of homogeneous hardware resources, independently. Globally, all types of resources are virtually presented in a unified resource pool to end-users. The use of existing management platforms provides a solution for rapid construction of a heterogeneous cloud. Most of the architectural components such as telemetry, fine-grained resource scheduling, and resource management, are reused. It is must be noted that, in some circumstances, for example, an orchestrated service that are deployed on various types of resources across different platforms, may encounter network congestion issues, especially for those high-throughput HPC alike applications. Additionally, as each management platforms (e.g., OpenStack and Mesos) have their built-in resource schedulers, the platform-integration scheme is limited on how they control and optimize resources at a coarse-grained level. In contrast, the server-integration scheme is more flexible. Multiple management platforms can be configured to simultaneously manage their corresponding resource types on the same server across the data center, provided no conflicts between them; or a customized platform to manage the system in a more dedicated manner, providing all necessary auxiliary services such as telemetry and resource manager. Additionally, as the coexistence of various types of resources on each server, more diverse applications and system requirements can be more easily met by *wiring* appropriate components.

4 Experiment

The initial implementation and the deployment of the proposed schemes have been realized in the context of CloudLightning project [31]. In this paper, a use case based on the Intel's Ray-Tracing application [32] is used to demonstrate the need for a unified platform to manage a cloud environment composed of heterogeneous resources.

4.1 Testbed Configuration

The experimental environment consists of an OpenStack managed virtualization environment (Newton release) which consists of eight Dell C6145 compute servers in total having 384 cores, 1.4 TB RAM, 12 TB storage and a Mesos managed Docker container environment (v17.04.0-ce) which consists of five IBM 326e servers in total having 10 cores, 40 GB RAM, 200 GB storage. In this deployment configuration, all physical servers have multiple dedicated network connections to three different networks including a *public*, a *private* and a *bridge* network. The *public* network connects to the Internet, the *private* networks are private to OpenStack or Mesos, respectively, the *bridge* network provides interconnections between virtual machines (managed by OpenStack) and containers (managed by Mesos). In the context of OpenStack, the *private* network is equivalent to the Neutron *Tenant* network, the *public* and *bridge* networks are the Neutron *Provider* networks. In the Mesos managed Docker environment, three Docker Bridge networks are created with each connecting to the *public*, *private* and *bridge* network, respectively. This deployment configuration is flexible to allow for future platforms, if needed, to be integrated with the existing environments. The detailed testbed layout and network configuration are shown in Fig. 6.

4.2 Use Case Blueprint

The Intel's Ray-Tracing application use case is composed of two parts, the first part is the Ray-Tracing engine and the second part is a Web interface. Both the engine and the Web interfaces should be respectively deployed on the most appropriate back-end

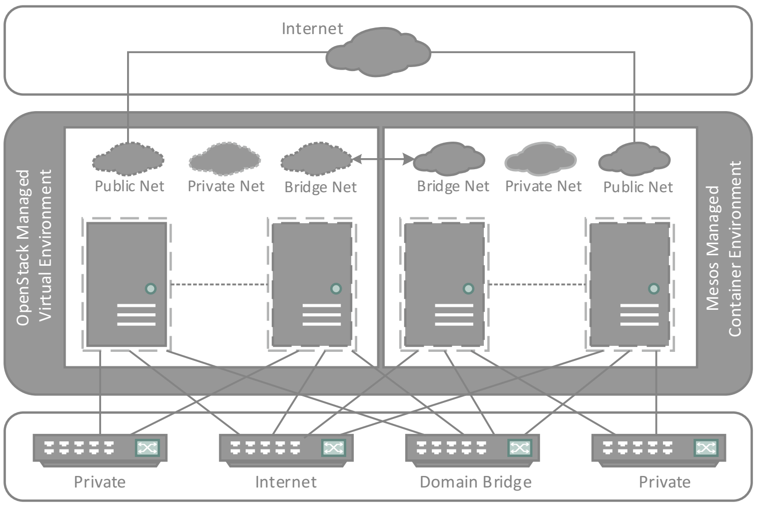


Fig. 6. Testbed layout and network configuration.

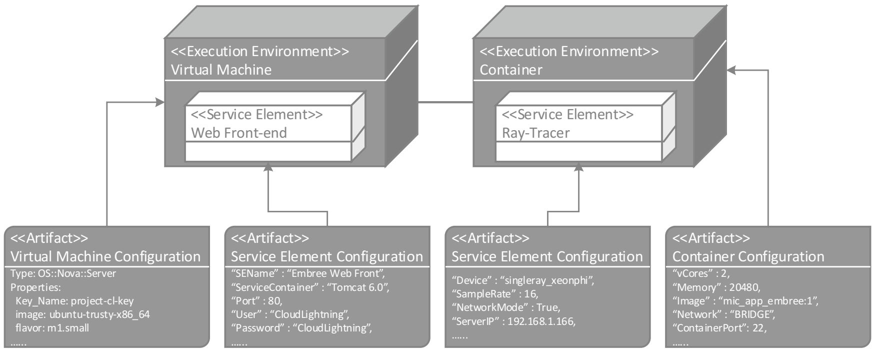
resources. For example, it has been demonstrated that the Ray-Tracing application can gain much better performance with MICs [33,34] comparing to general purpose CPUs. In order to use MICs, applications are generally required to be deployed in containers or directly on bare metal servers. And it is economically reasonable for deploying a Web server on a VM (providing more secured environment) that is configured with general purpose CPU processors. Thus, in the experiment, a blueprint is constructed which specifies that the Web interface should be deployed on VMs and the Ray-Tracing engine should be deployed in a container.

The graphical representation of the use case blueprint is shown in Fig. 7. The blueprint is also expressed in EXtensible Markup Language (XML) for machine interpretation. A blueprint consists of four main components:

1. *Execution Environments*, specifying the resource types such as virtual machines, containers, bare metal and so on,
2. *Service Element*, detailing the software component(s) to be deployed in an Execution Environment,
3. *Artifacts*, containing configurations for each Execution Environment or Service Element,
4. Connections, specifying the connectivity between Execution Environments and Service Elements.

The Resource Coordinator is responsible for parsing, decomposing and transforming blueprint components to a format, that can be understood by the underlying cloud management platforms, to facilitate application deployment.

The Resource Coordinator categorizes Execution Environments of the blueprint in to groups based on resource types (EE-Group), such as virtual machines, containers, or bare-metal. Within each EE-Group, Execution Environments are further partitioned



```
Type: OS::Nova::Server
Properties:
  Key_Name: project-cl-key
  image: ubuntu-trusty-x86_64
  flavor: m1.small
......
```

```
"SEName" : "Embree Web Front",
"ServiceContainer" : "Tomcat 6.0",
"Port" : 80,
"User" : "CloudLightning",
"Password" : "CloudLightning",
......
```

```
"Device" : "singleray_xeonphi",
"SampleRate" : 16,
"NetworkMode" : True,
"ServerIP" : 192.168.1.166,
......
```

```
"vCores" : 2,
"Memory" : 20480,
"Image" : "mic_app_embree:1",
"Network" : "BRIDGE",
"ContainerPort": 22,
......
```

Fig. 7. Ray-Tracing application blueprint.

into sub-groups based on the connectivities (C-Group), for example, if another given blueprint consists of three virtual machines without specifying connections between them, then this blueprint will be partitioned into one EE-Group and three C-Group within that EE-Group. In the use case scenario described there, there are two EE-groups, and one C-Group within each EE-Group. This grouping can be determined by formulating the blueprint topology into a graph $G(V, E)$, then connectivities between Execution Environments can be identified using the Union-Find algorithm, as illustrated in Algorithm 1. Where V indicates the vertices in the graph corresponding to the Execution Environments in the blueprint, and E denotes the edges in the graph corresponding to the connections between Execution Environments.

The algorithm assumes the connections are symmetric (if Execution Environment A is connected to Execution Environment B, then B is connected to A) and transitive (if Execution Environment A is connected to B, B is connected to C, then A is connected to C). Additional constraints can be added to make blueprint connections asymmetric and/or non-transitive.

When the grouping process is completed, the Resource Coordinator seeks for connections between C-Groups within each EE-Group. A connection between a pair of Execution Environments in different C-Groups indicates that the both Execution Environments should be placed in a *bridge* network or need to be attached to a *bridge* network, to establish cross platform communications. Execution Environments in a completely isolated C-Group should be placed in a *private* network, if Internet access is desired, then each Execution Environment must be attached to the *public* network, independently. Once the networks are identified, Execution Environments with their corresponding configurations in each EE-Group are transformed into deployment templates that are compatible with the corresponding management platforms. The Resource Coordinator initiates the deployment process and subsequently manages the life-cycle of the application blueprint. Listing 1.1 shows the deployment template for the Intel Ray-tracing Web front-end, in YAML format. This template is converted from the use case blueprint (as shown in Fig. 7) to make it compatible with corresponding platforms. In this use case, the Ray-tracing Web front-end is configured to be deployed on a VM, and VMs are managed by OpenStack. Listing 1.2 shows the deployment template for

Algorithm 1. Identifying Connectivity Groups (C-Groups) in each Execution Environment (EE-Group) using Union-Find Algorithm.

```
/* Identify C-Groups in a EE-Group                                    */
Data: EE-Group
Result: List{C-Group{v.idx}}
$G_{EE-Group}$(V, E); size ← V.size(); topology = new int[size];
for i ← 0 to size-1 do
    topology[i] = i;
end
foreach Edge e($v_i$, $v_j$) : E do
    if connected($v_i$, $v_j$) then
        continue;
    end
    else
        $idx_i$ = find($v_i$); $idx_j$ = find($v_j$);
        if $idx_i$ == $idx_j$ then
            return
        end
        else
            for q ← 0 to size-1 do
                if topology[i] == $idx_i$ then
                    topology[i] = $idx_j$
                end
            end
        end
    end
end
/* Store topology to C-Groups                                         */
cgroupSize ← number of unique values in topology;
for i ← 0 to cgroupSize-1 do
    new $cGroup_i$()
end
for i ← 0 to topology.lenght-2; i++ do
    if topology[i] == -1 then
        continue
    end
    $cGroup_k$.add(i);
    for j ← i+1 to topology.length-1; j++ do
        if topology[j] == topology[i] then
            $cGroup_k$.add(j); topology[j] == -1
        end
    end
    topology[i] == -1; k++;
end
/* Determine which connectino group an Execution
   Environment belongs to                                             */
Function find(v)
    return topology[v.index]
/* Detect whether two Execution Environments are connected
   */
Function connected(v1, v2)
    return find(v1) == find(v2)
```

the Intel Ray-tracing back-end application, in JSON format. This templates is also converted from the use case blueprint, but the template is made compatible with Mesos/-Marathon for the application deployment in containers.

Listing 1.1. Intel Ray-tracing Web front-end deployment template in YALM format.

```
blueprint-id: c97e718674c34adf815316ad4cec93cf
heat_template_version: 2016-10-14
resources:
   embree_web_frontend:
      type: OS::Nova::Server
      properties:
         image: Ubuntu14.04_LTS_svr_x86_64
         flavor: m1.small
         key_name: cl-project
         networks:
            - network: bridge-provider
user_data:
   template: |
      #!/bin/bash -v
      apt -y install httpd
......
```

Listing 1.2. Ray-Tracer in Mesos managed Docker containers using Marathon.

```
{
  "blueprint-id" : "c97e718674c34adf815316ad4cec93cf",
  curl -X POST -H "Content-type:␣application/json"
  marathon:8080/v2/apps -d
  {
    "id" : "embree",
    "cpus" : 2,
    "mem" : 10240.0,
    "container" :
    {
      "type" : "DOCKER",
      "docker":
      {
        "image" : "mic-app-embree:1",
        "network": "BRIDGE",
        "portMappings":
        [
          {
            "containerPort":22,
            "hostPort":0
          }
        ]
      }
    }
  }
}
```

5 Conclusions

Conventional cloud environments typically consist of homogeneous resources. Driven by consumer needs and technological advances, this situation is gradually changing. Heterogeneity in resource types is being introduced, and this poses challenges to traditional resource management mechanisms which aspire to seamlessly deliver the advantages associated with novel heterogeneous architectures to the end user. In response to this transition, a platform-integration scheme and a server-integration scheme have been presented. The platform-integration scheme describes a hyper-level management approach by integrating, and coordinating various coexisting cloud management platforms. Each of these platforms manages hardware resources of a particular type, characteristics, and an abstraction method most appropriate for its management. A use case experiment was presented to demonstrate how an application blueprint can be deployed and managed in an heterogeneous environment implementing the platform-integration scheme. The experiment also illustrates the benefit of having a management framework providing a unified view of heterogeneous resources. In contrast, the server-integration scheme is employed at the lowest level in the service delivery stack and performs fine-grained resource optimization and flexible service orchestration. This is achieved by reorganizing the hardware components on each physical server and by resource grouping at data center infrastructure level. The development of a use case to illustrate the server-integration scheme requires specialized hardware capabilities including I/O virtualization. Moreover, specialized configurations and software libraries are required to support hardware accelerator pass-through technologies. This use case will be developed in future work.

The candidate heterogeneous cloud management solutions proposed here, while still in the early stage of the development, provide realistic solutions to the complex problem of heterogeneous resource management. The platform-integration scheme can more readily be exploited, since it integrates and manages a multiplicity of extant technologies. The server-integration scheme, being a lower-level solution, is more specialized in its requirements from both the hardware and software environments. Hence, it can be seen as more of a longer-term solution.

To efficiently and effectively manage an heterogeneous cloud as an holistic entity, re-consideration of physical server design (incl. on-board computation accelerator integration, a good balance between computation accelerator and general purpose processing capacity, and a redesign of the cooling system), heterogeneous environment management (incl. neural network based resource management and collective intelligence based autonomic computing), service delivery model and cloud application development methodology (incl. a unified view of heterogeneous resources and a script-less application development) should all be addressed in a coherent and integrated manner. This is the challenge for the designers of the emerging heterogeneous cloud.

Acknowledgment. This work is funded by the European Unions Horizon 2020 Research and Innovation Programme through the CloudLightning project under Grant Agreement Number 643946.

References

1. Barr, J.: Developer Preview-EC2 Instances (F1) with Programmable Hardware. Amazon Web Services (2016)
2. Russinovich, M.: Inside the Microsoft FPGA-based configurable cloud. Microsoft Developer Network (MSDN) (2017)
3. OpenStack, L.: The openstack project (2011)
4. Kubernetes (2017). http://kubernetes.io/
5. Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A.D., Katz, R., Shenker, S., Stoica, I.: Mesos: a platform for fine-grained resource sharing in the data center. In: Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation, NSDI 2011, pp. 295–308. USENIX Association, Berkeley (2011)
6. Merkel, D.: Docker: lightweight Linux containers for consistent development and deployment. Linux J. **2014**, 2 (2014)
7. Turnbull, J.: The Docker Book. Lulu.com, Morrisville (2014)
8. OpenStack Magnum (2017). http://git.openstack.org/cgit/openstack/magnum
9. OpenStack Ironic (2016). https://docs.openstack.org/ironic/latest/
10. Dong, D., Stack, P., Xiong, H., Morrison, J.P.: Managing and unifying heterogeneous resources in cloud environments. In: Proceedings of the 7th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER, pp. 143–150. INSTICC, ScitePress (2017)
11. Popek, G.J., Goldberg, R.P.: Formal requirements for virtualizable third generation architectures. Commun. ACM **17**, 412–421 (1974)
12. Smith, J.E., Nair, R.: The architecture of virtual machines. Computer **38**, 32–38 (2005)
13. OpenStack: Architecture design guide. Technical report 15.0.0 (2017)
14. Fontán, J., Vázquez, T., Gonzalez, L., Montero, R.S., Llorente, I.: OpenNebula: the open source virtual machine manager for cluster computing. In: Open Source Grid and Cluster Software Conference, vol. 86 (2008)
15. OpenNebula: OpenNebula 5.2 deployment guide. Technical report 5.2.1, OpenNebula Systems (2017)
16. OpenNebula: OpenNebula 5.2 operation guide. Technical report 5.2.1, OpenNebula Systems (2017)
17. Nimbus (2017). http://www.nimbusproject.org
18. Keahey, K., Armstrong, P., Bresnahan, J., LaBissoniere, D., Riteau, P.: Infrastructure outsourcing in multi-cloud environment. In: Proceedings of the 2012 Workshop on Cloud Services, Federation, and the 8th Open Cirrus Summit, FederatedClouds 2012, pp. 33–38. ACM, New York (2012)
19. Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., Wilkes, J.: Large-scale cluster management at Google with Borg. In: Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, pp. 18:1–18:17. ACM, New York (2015)
20. Schwarzkopf, M., Konwinski, A., Abd-El-Malek, M., Wilkes, J.: Omega: flexible, scalable schedulers for large compute clusters. In: Proceedings of the 8th ACM European Conference on Computer Systems, EuroSys 2013, pp. 351–364. ACM, New York (2013)
21. Rensin, D.K.: Kubernetes - Scheduling the Future at Cloud Scale, 1005 Gravenstein Highway North Sebastopol, CA 95472 (2015)
22. Burns, B., Grant, B., Oppenheimer, D., Brewer, E., Wilkes, J.: Borg, Omega, and Kubernetes. Commun. ACM **59**, 50–57 (2016)
23. Zhang, Z., Li, C., Tao, Y., Yang, R., Tang, H., Xu, J.: Fuxi: a fault-tolerant resource management and job scheduling system at Internet scale. Proc. VLDB Endow. **7**, 1393–1404 (2014)

24. Razor Server (2017). https://github.com/puppetlabs/razor-server
25. Foreman (2017). https://theforeman.org
26. OpenStack Neutron (2017). https://github.com/openstack/neutron
27. Flannel (2017). https://github.com/coreos/flannel
28. Weaveworks WeaveNet (2017). https://www.weave.works/docs/net/latest/introducing-weave/
29. Libnetwork (2017). https://github.com/docker/libnetwork
30. Kuryr (2017). http://docs.openstack.org/developer/kuryr/
31. Lynn, T., Xiong, H., Dong, D., Momani, B., Gravvanis, G., Filelis-Papadopoulos, C., Elster, A., Khan, M.M.Z.M., Tzovaras, D., Giannoutakis, K., Petcu, D., Neagul, M., Dragon, I., Kuppudayar, P., Natarajan, S., McGrath, M., Gaydadjiev, G., Becker, T., Gourinovitch, A., Kenny, D., Morrison, J.: CloudLightning: a framework for a self-organising and self-managing heterogeneous cloud. In: Proceedings of the 6th International Conference on Cloud Computing and Services Science, pp. 333–338 (2016)
32. Intel Embree (2017). https://embree.github.io
33. Benthin, C., Wald, I., Woop, S., Ernst, M., Mark, W.R.: Combining single and packet-ray tracing for arbitrary ray distributions on the Intel MIC architecture. IEEE Trans. Visual Comput. Graph. **18**, 1438–1448 (2012)
34. Wald, I.: Fast construction of SAH BVHs on the intel many integrated core (MIC) architecture. IEEE Trans. Visual Comput. Graph. **18**, 47–57 (2012)