Electronic Preprint for Journal of Information Processing Vol.24 No.2

Invited Paper

Cloud Computing - Current Status and Future Directions

Yukio Tsuruoka[1,a)]

Received: September 2, 2015, Accepted: November 6, 2015

Abstract: Cloud computing is a modern form of advanced information system that has developed with the proliferation of the Internet, broadband access networks and high-speed processors, and it is continuing to spread. Cloud computing enables users to use IT resources, such as processors and storage, through the network, by simply paying a fee and without needing to own servers. Running costs are reduced because resources can be used on-demand only as needed. Since hardware provisioning is not necessary, software development and launching new services can be done quickly. Cloud computing also drives innovation in information systems. For example, cloud computing has made it easy to build cluster systems using virtual machines, which has led to the development of scalable data stores such as object storage and key-value stores. Cloud computing also led to the development of software defined networks and software defined storage, to respond rapidly to the requirements of users and applications. In this paper, the benefits of cloud computing are reviewed and technologies supporting it and new technologies arising from it are outlined. Directions for cloud computing in the future are also discussed.

Keywords: cloud computing, virtualization, large-scale data processing

1. Introduction

Cloud computing refers to forms of information system usage in which data and software are placed on servers on a network and are accessed through the network from clients. It is a modern form of information system that has developed in an environment with increasing network and processor speed and proliferation of the Internet and it is continuing to spread. Cloud computing brings a variety of benefits for enterprises and other organizations. For example, they can use cloud computing to rent virtual machines and storage rather than having their own servers and other equipment. In addition to not incurring equipment costs, running costs can also be reduced because resources are only used when they are needed and payment is by usage. Development and testing are accelerated because new servers can be set up and storage capacity can be allocated rapidly, so time-to-market can be reduced [1].

Cloud computing also drives information system innovation. With the spread of cloud computing, new ways of using information systems and new demand have arisen, and this has resulted in new technologies. For example, it has become easy to build cluster systems using virtual machines, which has resulted in development of scalable data stores such as object storage and key-value stores. Technologies such as software defined networks (SDN) and software defined storage (SDS) have also arisen, enabling hardware resources such as processors, storage and networks to be centrally managed and dynamically allocated and configured through software. This enables rapid response to resource requirements from users and applications.

This paper describes the technologies underlying cloud computing. It is organized as follows. The benefits of cloud computing are summarized in Section 2, and usage models are described in Section 3, virtualization of OS environments is discussed in Section 4, storage in Section 5, networks in Section 6, high-availability technologies in Section 7, security in Section 8, open-source cloud platforms in Section 9, applications in Section 10, and conclusions and future directions are discussed in Section 11.

2. Benefits of Cloud

Virtualization [2] is a central technology of cloud computing. Section 2, discusses the benefits of cloud computing brought through virtualization. Virtualization refers to the realization of virtual machines (VM) on top of a bare-metal (physical) machine (BM). A VM is compatible with a BM, so it can execute almost all machine code that can run on the BM, including the OS. VMs are implemented using system software called a virtual machine monitor or hypervisor. The hypervisor runs on the BM, and the OS runs on a VM administered by the hypervisor. Below, instances of BM are referred to as nodes. Cloud computing has the following benefits due to the properties of the hypervisor and VMs.

Scalability: Cloud computing is a distributed system consisting of resources in units of VMs, so workloads that are too large to be handled by a single node can be handled by simply allocating more nodes to them. This scale-out approach can be implemented at lower cost than scale-up approaches, which would need to replace nodes with higher performance nodes. A single framework can handle tasks of various sizes, provided parallel processing is possible for the tasks. Large-scale data processing using clusters is discussed in Subsection 10.2.

Server consolidation: With conventional BM-based systems, individual servers had to provide resources sufficient to handle

1 Nippon Telegraph and Telephone Corporation, Musashino, Tokyo 180–8585, Japan
a) tsuruoka.yukio@lab.ntt.co.jp

© 2016 Information Processing Society of Japan

the maximum expected workload well. This is larger than the usual workload, so the overall utilization rate is lower. By placing servers on VMs, the number of BMs can be reduced by gathering the VMs on fewer BMs.

Utility computing (*multi-tenancy*): The hypervisor is able to run multiple VMs on a single BM. Individual VMs are isolated from each other, and cannot access the resources of other VMs. This allows individual VMs to be leased to different users. Users do not own the equipment, so the work of purchasing equipment (devices), installing it, and maintaining it is not required, and initial costs are low.

Elasticity: Utility computing enables users to use resources from a resource pool on-demand. Fees are paid according to the quantity used. In this way fluctuating demand can be handled without wasted investment.

Manageability: VM state refers to a set of data including processor registers, memory, storage content, I/O device state and other information. The hypervisor expresses and manages this state information as data. A snapshot of a VM image can be obtained by simply copying this data. Managing VMs as data enables multiple VM images to be managed together, to be managed centrally, and for such management to be automated. This can reduce administration tasks and cost.

Agility: Since VMs are managed as data, provisioning and configuration of new VMs can be done instantly, by simply copying a template (a VM image) and modifying the data. Compared to BM installations, which require purchasing and installing hardware and configuring each device separately, this is remarkably quick. Servers can be set up easily, accelerating software development, testing, and service start up.

High-availability (*HA*), *fault-tolerance* (*FT*): HA systems can be built by having the hypervisor take periodic snapshots of VMs and send them to a backup node, so that recovery can be done quickly when a fault occurs. This is discussed in detail in Section 7. By storing VM snapshots at a remote location, business continuity and disaster recovery can be assured when disasters occur.

Monitoring and control: The hypervisor runs with a higher privilege than the VMs, so it can monitor and control them. For example, it can monitor network I/O of VMs to detect malware infection from the outside or spread of malware from a VM to other machines. It can also isolate infected VMs to prevent damage from spreading. These issues are discussed in detail in Section 8.

3. Usage Models

Clouds can be classified as either public, private, or hybrid, according to how they are used.

With public clouds, cloud service providers (CSPs) provide IT resources as a service to users such as enterprises or developers. Users hire resources whenever and however much they need, paying according to the amount used. Public clouds can be further classified as infrastructure-as-a-service (IaaS), platform-as-a-service (PaaS), or software-as-a-service (SaaS) (**Fig. 1**).

With IaaS, VMs that users use from a remote site are provided. Users do not need to maintain hardware, but they are responsible for administration of code on the VM, including OS and applications. Typical OS environments can also be provided and used as a base.

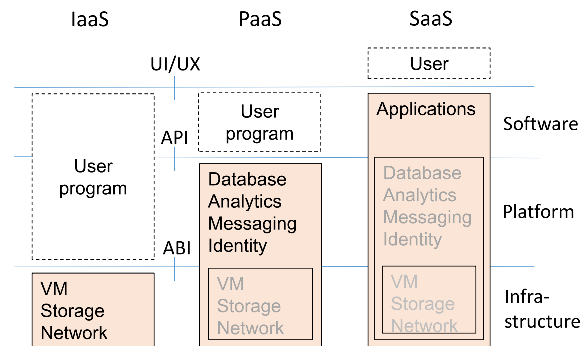

Fig. 1 Usage models of public clouds.

With PaaS, functionality needed for building an application is provided as a service. Examples include databases, identity management, and data analytics platforms. Users use these functions through APIs. Users do not need to maintain hardware or the OS environment, but must build and maintain their application themselves.

With SaaS, a conventional customized system or package software application is provided as a service. The user can outsource management of hardware, OS environment and application to the SaaS provider.

The proportion of administration handled by the provider increases in order from IaaS to PaaS and SaaS, so the added value of the services increases accordingly, but conversely, the degree of freedom for the user decreases. Some SaaS and PaaS provide auto-scale functions that automatically allocate and release resources according to processing requirements.

By using public clouds, enterprises no longer need to own servers. The cost of servers and costs associated with installing them are not incurred, reducing CAPEX. Running costs (OPEX), including power consumed by equipment and air conditioning as well as personnel for operation and maintenance of equipment, are also not incurred. Through rapid server provisioning, service providers can reduce the time from service development to its release. Since fees are charged according to amount used, projects can start small and be expanded proactively, in accordance with risks. Public clouds can be particularly helpful for startups and small and medium-sized businesses (SMBs) that are launching new businesses.

With a private cloud, an enterprise or organization has its own equipment. The role taken by the provider with a public cloud is handled by an internal organization such as the information systems department. Enterprises that already have many server assets can reduce the number of servers through server consolidation, thereby reducing OPEX. Some of the other benefits of public clouds also apply, but not those due to not owning the equipment. Reasons that enterprises may choose to use a private cloud include handling of information that cannot be managed externally, or that network access latency cannot be tolerated. In some cases, private clouds are used temporarily during the transition from using on-premises systems to a public cloud. By first moving servers to VMs, they can later be moved to the public

© 2016 Information Processing Society of Japan

cloud easily due to the portability of VMs.

Public and private clouds each have their areas of application and are used accordingly. When public and private clouds are linked and used together, it is called a hybrid cloud. Interclouds, when multiple clouds are interconnected, have also been proposed [3].

Public clouds first appeared in 2006 when Amazon began providing its Amazon Web Services (AWS) [4]. Initially, the SQS message queue service and S3 object storage service were offered, and the EC2 IaaS was added later. As of 2015, AWS has increased to include 46 *1 services.

AWS provides services in 11 regions around the world as of 2015, and each region is independent. Users select regions to receive services and each region has multiple availability zones. An availability zone is a physically separate data center. When there is a fault such as a power outage at the data-center level, resources from a different availability zone can be available.

4. Realizing Virtual OS Environment

To realize a cloud, multiple OSs must be able to run independently in parallel on a single BM. Such virtual OS environments can be implemented using VMs, as discussed earlier, or using containers (**Fig. 2**). Both VMs and containers provide independent OS environments to each user. However, while a VM can run any OS, containers must all run the same OS environment. Conversely, containers are more lightweight than VMs. Overviews of VMs and containers are given below, assuming an x86 architecture.

4.1 Virtual Machines

Virtualization technology was implemented in 1966 in the IBM

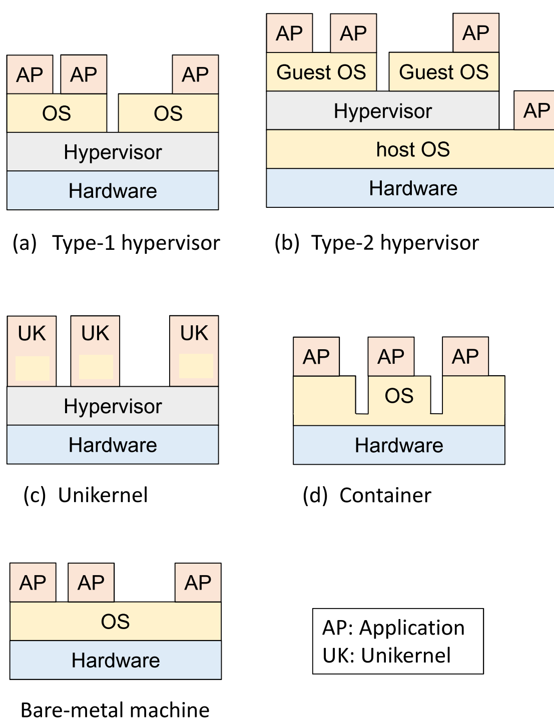

Fig. 2 Types of virtual OS environment.

*1 The number of services in the Tokyo region.

360 mainframe. At the time, it was used for sharing high cost mainframes. The hypervisor for the commodity x86 architecture [5] of today came about as follows. In 1999, VMware Inc. commercialized their VMware Workstation [6]. In 2003, Xen [7], developed at Cambridge University, was released as open-source software. In 2006, KVM [8] developed as an extension of Linux, became a standard feature of Linux. In 2008, Microsoft's Hyper-V hypervisor was incorporated into the Windows Server OS.

4.1.1 Realizing Virtual Machines

A BM has resources including processors, memory, and I/O devices. Typically, processors have control registers and privilege levels for managing these resources. Control registers regulate access to resources. Privilege levels are states of the processor, and the instructions that can be executed are restricted in states with lower privilege. For example, the x86 architecture supports four levels. Level 0 is the highest, and level 3 is the lowest. Control registers are accessed by executing instructions, but those instructions can only be executed at a high privilege level. In a non-virtualized environment, the OS kernel normally runs at level 0, and user programs run at level 3. This enables the OS to manage system resources while preventing user programs from accessing system resources directly.

In virtualized environments, the hypervisor must guarantee that the OS runs on the VM as though on a BM, while limiting the OS privileges to within the VM. In other words, it must prevent a VM from accessing resources of another VM.

To do this, when an OS executes instructions to perform device I/O or change system state, the hypervisor must reinterpret them as relating to the virtual machine in order to execute them. The hypervisor must run with higher privileges than the OS in order to capture instructions executed by the OS, but the OS must run at a high privilege level, or in an equivalent manner.

In 1974, Popek and Goldberg clarified sufficient conditions that an instruction set architecture (ISA) must satisfy to implement virtual machines [9]. This is that sensitive instructions must be included among the privileged instructions. Here, sensitive instructions are the set of instructions that change system state, and privileged instructions are those that generate an exception when executed from an unprivileged level. For example, an instruction that writes to a control register is sensitive, and a privileged instruction can only be executed at the privileged level (Level 0).

Robin et al. studied the x86 architecture at the time, and as a result showed that there were several sensitive non-privileged instructions. In other words, that the sufficient conditions from Popek et al. were not satisfied [10]. VMware Workstation fills in that gap using binary translation [11], [12], which dynamically changes code in memory. Sensitive non-privileged instructions in the kernel code are substituted with privileged instructions and emulated safely by the hypervisor.

4.1.2 Types of Virtualization

Hypervisors are classified into two types. Type-1 hypervisors run directly on the BM. As such, they require I/O drivers, as do OSs. VMware ESXi and Xen are examples of type-1 hypervisors. Instead of having its own I/O drivers, Xen has a special VM called Domain 0 and it uses Linux drivers that run on this VM.

Type-2 hypervisors run on a host OS that runs on the BM.

© 2016 Information Processing Society of Japan

Guest OSs are then run under management by the hypervisor. A benefit for users is that it is easy to install over an existing OS environment. VMware Workstation and Parallels Desktop are examples of type-2 hypervisors.

The code that guarantees the secure isolation of VM environments is called the trusted computing base (TCB) [13]. For type-1, the TCB is only the hypervisor, while for type-2, the host OS is included in the TCB in addition to the hypervisor. Type-1 hypervisors have a smaller TCB and are considered to have less possibility for introduction of vulnerabilities.

VM implementations can be classified as either full virtualization (FV) or paravirtualization (PV).

FV provides an application binary interface (ABI) that is essentially equivalent to a BM, so that a guest OS can be run as is, without modifying the code. The behavior of the BM is virtualized strictly, which requires much more code.

With PV, the guest OS calls the hypervisor code instead of executing sensitive instructions when performing I/O. Such calls are called hypercalls. PV has the benefits of a simple interface between guest OS and hypervisor and low virtualization overhead. On the other hand, the OS must be modified to using hypercalls. This prevents use of an OS that does not have source code available.

The initial release of Xen only supported PV. Xen 3.0 in 2005 added support for FV using hardware support for virtualization, which is described below.

4.1.3 Reducing the Overhead of Virtualization

To implement FV, sensitive instructions, memory, mother board, interrupts, timers, and I/O processing must be virtualized.

Earlier, virtualization of sensitive instructions had to be done by emulation using binary translation, but in 2005 Intel implemented the VT-x [14] processor with hardware support for virtualization and able to execute some of them in hardware. VT-x adds operating modes to the processor and extends the ISA to satisfy the sufficiency conditions from Popek et al. A context switch occurs when switching modes, but the associated overhead has been reduced with improvements in successive processor generations. In 2006, AMD implemented similar functionality with AMD-V. VT-x and AMD-V are not compatible, so the hypervisor must absorb any differences using an abstraction layer in software.

For memory virtualization, normally the OS manages memory by mapping logical addresses to physical addresses through an MMU and page table. The hypervisor must convert accesses to the MMU or page table by the guest OS on the VM to the corresponding VM process. An MMU is emulated. The hypervisor also has a copy of the guest OS page table (shadow page table) and performs conversions between it and the system page table. Changes to the guest OS page table are reflected in the system page table, and page access information is written back to the guest OS. In the second generation of VT-x implemented in 2008, Extended Page Tables (EPT) [5] was introduced, extending address transformation methods to make memory virtualization easier and faster. AMD has also implemented Nested Page Tables (NPT), which are similar to EPT. In virtualization environments, an OS is run for each VM on the BM, so memory usage is tight. By allowing pages to be shared among VMs, the total amount of memory used can be reduced [15].

Virtualization of mother board functions, interrupts, timers, and I/O processing have all been implemented using software emulation, by QEMU [16] for instance. Hardware support has been added for virtualization of I/O processing, making implementation easier and faster. Both AMD and Intel have implemented their own functionality extending their hardware platforms (processors, chipsets, BIOS, etc.) so that I/O devices can be accessed directly from a guest OS. These are called AMD-Vi [17] and Intel VT-d [18] respectively. Implementation of I/O virtualization has also become easier as I/O devices have added virtualization functions that provide separate functions for multiple guest OSs. The PIC-SIG industry organization has created the SR-IOV specification [19] for I/O virtualization of PCI devices for this purpose.

CPU time is allocated to each VM by the hypervisor scheduler, and then further allocated to processes by the guest OSs on the VMs.

Generally, PV has less overhead and is faster than FV that is based on emulation. However, for sensitive instructions and memory virtualization, FV is faster than PV because it utilizes hardware support. PVH, which combines hardware support with PV was introduced in Xen 4.5.

Normally, I/O processing incurs overhead due to interrupt virtualization. This is because a context switch occurs when the hypervisor temporarily receives interrupts intended for the guest OS. Gordon and Amit have proposed Exit-Less Interrupts (ELI) [20], which enable a guest OS to process interrupts directly. ELI increases throughput and latency by a factor of 1.3 to 1.6, and performance on I/O intensive workloads has been shown to achieve 97% to 100% of BM performance.

Unikernel [21] is another technology for reducing overhead in virtualization environments. Unikernel is an approach which generates a small OS environment for a specific application, including only functionality specifically required by that application. Examples include MirageOS [22], ClickOS [23], [24], HaLVM, Clive, LING, Rump kernels, and OSv.

4.2 Containers

Containers provide OS-level virtualization, which uses OS features to create multiple isolated OS environments [25]. In contrast with VMs, there is only one OS, so each OS environment is the same version of the OS. Early implementations include jail [26], which was introduced into FreeBSD 4.0 in 2000, Virtuozzo [27], which was commercialized for Linux by Parallels Inc. *2 in 2002, and Solaris Containers [28] in 2004. Initially, containers were not a Linux kernel feature, so they had to be merged into the kernel when a new version of the kernel was released. To avoid such merge tasks, Parallels released the core of Virtuozzo as the open-source OpenVZ [29] in 2005, contributing to development of the Linux kernel. Functionalities needed to implement containers were incorporated into kernel versions 2.6.15 to 2.6.26, making implementation easy. Linux Containers (LXC) [30] is a container implementation using this standard Linux functionality.

*2 The company name was SWsoft at the time.

© 2016 Information Processing Society of Japan

Containers are used from a higher-level software platform. For example, with the Mesos [31] platform for sharing multiple workloads on a cluster, LXC is used as a mechanism for separating multiple workloads over nodes. Docker [32], which was released in 2013 as a platform for application deployment, initially used LXC for container functionality. Also, CoreOS [33], which was released in 2013 as a platform for deployment on clusters, initially used Docker as its container engine (runtime library).

Note that as of version 0.9 in 2014, Docker no longer uses LXC and the implementation has migrated to libcontainer. CoreOS is also developing the new appcontainer (appc) container specification and is releasing its own engine called Rocket (Rkt). In June 2015, Docker, CoreOS and others initiated the Open Container Initiative (OCI) [*3] [34] industry association to create standard container specifications and the Open Container Specifications [35] were released. Docker provides the runC [36] container runtime implementation based on this specification. runC is able to run Docker images. On the other hand, Google announced the Kubemetes [37] container management platform in 2014 and contributed to the new Cloud Native Computing Foundation (CNCF) in 2015. Both OCI and CNCF organizations are under the Linux Foundation.

Linux functions for implementing containers include kernel namespace [38], Control groups (cgroups) [39], and Linux kernel capabilities [40].

Kernel namespace decides how system resources appear to processes. For example, user ID 0 (root user) in container A and user ID 0 in container B can each be mapped to different non-root users in the host OS.

Cgroups is functionality for allocating hardware resources such as CPU time, memory, and network bandwidth to processes.

Capabilities sets access permissions for individual resources. It enables access control to be configured in smaller units than the conventional distinction between privileged and unprivileged. Isolation of container environments is realized by combining these functions.

Container environments are compact because each environment does not need to have a separate OS. The benefits of containers as a virtual OS environment are that virtualization overhead is low compared to VMs, they can be launched quickly, and they can operate with high density on a single node. On the other hand, they also have the following limitations. All container environments must use the same version of the OS kernel. Performance isolation is also weak. All containers share kernel resources, so a particular container could use up all resources. Limitations can be put on resource use to prevent this, but application behavior must be understood in order to do so.

The TCB for container environments is the OS, which contains much more code than VM environments, for which the TCB is the Type-1 hypervisor only, and the possibility of vulnerabilities being introduced is considered to be higher.

In some container implementations isolation of the environments is not yet complete because the kernel keyrings namespace is not separate, among other reasons.

Xavier et al. [41] compared the performance of three types of container: LXC, OpenVZ, and VServer; to a Xen VM using an HPC workload. The results showed that all containers had less overhead than Xen in memory throughput, disk I/O, network throughput, and network latency. On the other hand, regarding performance isolation, all containers were more affected by other environments on the same node in memory, disk and network performance compared to Xen. All containers and Xen have little processor overhead so processor performance isolation is good.

5. Storage

Processor virtualization has made use of computation resources elastic, so demand has increased to handle storage this way as well. In cloud environments, storage requires the following two points. First, it must be possible to manage resources with agility and flexibility to the same extent as is possible with processors, and second, it must be possible to manage through software it in a unified manner, coordinated with other resources such as processors and networks.

There are two main approaches to storage configurations in cloud infrastructures. One is to use a general, advanced and self-contained storage solution (a product), and the other is to configure a storage subsystem that combines a commodity cluster, having storage devices attached directly to each node, with software to manage them. The latter means that the cluster itself is also a suitable base for a scalable storage subsystem, and this has been used successfully as the data store for large-scale data processing systems such as Hadoop [42]. With cluster-based systems, low-cost commodity drives are used. Faults are expected to occur with these drives, so fault tolerance is implemented in an upper-layer of software. Each of these approaches is described below.

5.1 Storage Solutions

Even before cloud technology became common, various types of resource abstraction in storage had been implemented. For example, a Logical Volume Manager (LVM) would integrate or partition physical drives in order to define logical drives (i.e., logical volumes). Logical Volumes made it easy to change capacity. This eliminated the need to allocate excess capacity before hand and increased utilization of drives. As a result, the number of drives, and consequently CAPEX, were reduced. By managing multiple drives centrally in a resource pool, OPEX could also be reduced. Most storage products provide various additional functions by mapping from logical drives to physical drives. Examples include, RAID to increase throughput or fault tolerance, backups, version control, compression, deduplication, encryption, automated capacity allocation, capacity leveling, load balancing, and hierarchical storage management (HSM). HSM attempts to achieve both high performance and low cost by combining high-speed, high-cost storage (e.g., SSD) with low-speed, low-cost storage (e.g., HDD, tape). Data is automatically moved between different types of drive according to access frequency and QoS policies.

Even before cloud technology became common, storage products had become advanced independent subsystems not reliant on a host system. For example, dedicated Storage Area Net-

*3 Initially called the Open Container Project (OCP).

© 2016 Information Processing Society of Japan

works (SAN) enabled storage to be shared among multiple hosts. New categories of storage have also appeared, incorporating features of cluster-based systems. Examples include scale-out NAS, which integrates drives with controllers so that throughput can be increased by adding equipment, hyper-converged infrastructure, which integrates processors, storage and networking making it easy to build cluster systems, and software defined storage (SDS), which can be controlled through software.

5.2 Cluster-based Storage Systems

Cluster-based data stores can be classified based on the structure of data they handle, including block storage, object storage, file systems, key-value stores, column stores and document stores.

Block storage is accessed in block units, in the same way as physical drives. Examples include Amazon EBS (hereinafter, EBS) [43] and OpenStack Cinder (hereinafter, Cinder) [44].

EBS is a block storage service for the Amazon EC2 IaaS, started in 2008. EC2 does not have persistent storage, so separate storage was needed to save data beyond the life of a VM instance. EBS is attached to an instance for use, but it cannot be attached to multiple instances at the same time. Storage volumes are automatically replicated within their availability zone. Optionally, data and communication can also be encrypted. Throughput and latency can be monitored from a Web administration screen. As of 2015, the maximum capacity for SSD-backed EBS is 16 TB, and the maximum throughput is 4 Gbps when using an EBS-optimized instance.

Cinder is block storage for the OpenStack [45] open-source cloud infrastructure. Similar to EBS, it is used by attaching a volume to a VM. Cinder supports multiple types of back-end storage through drivers. Currently, supported solutions include Ceph [46], GlusterFS [47], Sheepdog [48], [49], LVM, and NFS [50], and they can be used in combination.

Object storage is a storage architecture that processes data in units of objects composed of data and metadata. Data consists of variable-length byte arrays, while meta data consists of keys, values and attributes. By separating data and metadata, data can be accessed directly, and metadata is used for extensibility and optimization. Objects are identified using globally unique identifiers. The object storage concept was described by Gibson et al. [51] in 1997 and has been implemented in many systems since then. Amazon S3 (hereinafter, S3) [52] is one example of a commercial service. OSS examples include Lustre [53] and OpenStack Swift (hereinafter, Swift) [54]. Ceph, GlusterFS and Sheepdog, as mentioned above, also have object storage interfaces. There are also storage products that have an object storage architecture. Most object storage provides an S3-compatible API.

GlusterFS provides a file system interface. It can also be used as object storage or block storage.

A key-value store (KVS) stores data in key-value pairs. One example is Dynamo [55].

A column store stores data in column units representing tables of rows and columns. In contrast with RDBMSs, which store data in row units, they facilitate rapid addition of new columns and analysis that accesses data from particular columns only. They are also able to store sparse tables compactly. On the other hand, they are not suited to transaction processing. One example is Cassandra [56]. Bigtable and Hbase, which are used as large-scale data processing platforms, can also be classified as column stores. These are discussed in Section 10.2.

A document store stores data in units of documents. One example is MongoDB [57].

The data structures for KVS, column store and document store are similar, so they can be considered as types of KVS. They also have common characteristics, including scalability due to relaxed consistency requirements, flexible data structures, and ability to build low-cost systems using inexpensive drives due to fault tolerance.

6. Networking

Virtualization of networking has also become necessary in cloud infrastructures as processing has been virtualized. For example, provisioning of VMs requires network configuration and dynamic allocation of bandwidth according to workload (bandwidth on demand), and in multi-tenant environments, networks for each tenant must be kept separate. Software defined Networks (SDN) [58] is a key idea for managing networks for cloud infrastructures. SDN is an architecture that enables networks to be created, deleted and configured through software control. SDNs separate network functions into the *forwarding function* and the *controller*. The forwarding function transports packets, and the controller controls the forwarding function.

The forwarding function is implemented in switches and routers. The controller is implemented as a software module. The behavior of the forwarding function is programmably controlled by the controller, allowing networks to be controlled rapidly and dynamically according to the requirements of higher layers. It also enables automated control and centralized device management within a data center.

OpenFlow [59] is a protocol regulating the interface between forwarding function and controller. OpenFlow was developed to facilitate network research and began from research by Casado, who was at Stanford University in 2006. After OpenFlow was proposed, the concept of SDNs grew and became widely known. The initial specification was set in 2011 and specifications [60] are currently set by the Open Network Foundation, an industry organization.

One way to configure a network between two VMs in a data center using OpenFlow is to configure all switches on the path between the VMs. In this case, all switches would have to support OpenFlow.

Another way is to use overlay networks. Note that hypervisors have a virtual switch, which is a software implementation of a switch. An overlay network is created by tunneling between the virtual switches of the hypervisors hosting the two VMs. Then, network configuration is done on that overlay network, which is to say, on the two virtual switches. As an example, if VM 1a and VM 1b are running on node 1 and VM 2a and VM 2b are on node 2, a virtual network for VM 1a and VM 2a and another virtual network for VM 1b and VM 2b could be created. This method has the benefit that there are no changes to the underlying phys-

© 2016 Information Processing Society of Japan

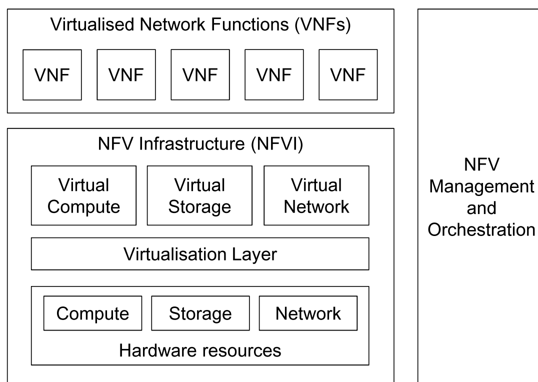

Fig. 3 NFV framework.

ical network. Tunneling protocols used include VXLAN [61], STT [62], GRE [63], and DOVE.

In addition to vendor products, there are also OSS implementations of OpenFlow controllers. Examples include Ryu [64], Trema [65], Open-Daylight controller [66], OpenContrail controller [67], and Floodlight [68].

In addition to vendor products, there are also OSS implementations of OpenFlow switches. Examples include Open-vSwitch (OVS) [69], [70] and Lagopus [71]. OVS was merged with Linux kernel 3.3 in 2012. Both OVS and Lagopus increase processing speed by using the Intel Data Plane Development Kit (DPDK) [72], which is a packet processing library. DPDK is run in the user space. By using polling for reception, overhead caused by interrupts can be avoided. It also uses DMA to interact directly with the NIC, not using a kernel buffer. This reduces overhead and increases throughput for small packets.

NFV [73] is a specification for implementing, dynamically organizing, and managing the *Network Function* (NF) using commodity hardware and virtualization, and is standardized by ETSI [74]. The NFV framework is shown in the **Fig. 3**. In the figure, *Virtualised Network Function* (VNF) is a software implementation of the NF. The *NFV Infrastructure* (NFVI) runs the VNF, so it includes hardware resources and software that virtualizes it. *NFV Management and Orchestration* manages lifecycles of VNFs as well as hardware resources and software orchestration in the NFVI. By separating software and hardware, improvements to both are made separately. The goals of NFV include implementing NF at low cost using commodity hardware and virtualization, connecting VNF and hardware with flexibility, developing services rapidly through software, automating operations, reducing power consumption by consolidating workloads, and multi-vendor solutions through open standards.

OPNFV [75] is an example of an OSS implementation of NFV. It integrates Ceph, KVM, OpenDaylight [76], OpenStack and Open vSwitch to form a carrier-grade NFV platform.

Conventional network switches provide both hardware and the software to control it. In contrast, white-box switches provide only the hardware, and the software can be selected by the user. This provides the benefits of freedom and the possibility of centralized management. Device cost should also decrease. On the other hand, the possibility of attack by malware increases with more freedom [77], and it is more difficult to detect such attacks. For these reasons, they must be managed carefully.

In large-scale data-center networks, it is important to prepare enough bandwidth. Network topologies such as Fat-trees [78] and Clos networks [79], [80] are used as well as tree structures. Al-Fares et al. have proposed an architecture based on fat-trees that achieves high aggregate bandwidth using commodity switches [81]. Mysore et al. have proposed the Portland [82] Layer 2 forwarding and routing protocol, which is scalable and fault tolerant. Google data centers achieve bisection bandwidths over 1 Pbps using multi-stage Clos networks and a centralized control protocol [83].

7. High-availability and Fault-tolerance

Availability is a basic requirement of a service. For operation of large-scale systems on a cloud, hardware failure rates increase, and spread of the effects of a fault is an issue. For these reasons, high availability (HA), which minimizes downtime due to faults, and fault tolerance (FT), which suppresses faults and maintains processing, are important. Systems that are fault tolerant at the host level have redundant hosts, so that if the primary host stops, processing switches to a backup host. To achieve this, the states of primary and backup hosts must be synchronized.

Bressoud et al. [84] proposed an FT system using VMs in 1995. This was a lock-step system controlled such that primary and backup VMs executed the same instructions in parallel. Both VMs were given the same input and maintained deterministic execution in parallel. Non-deterministic processing such as reading timers was emulated by the hypervisor so that both VMs produced the same result. Interrupts were also generated at the same time. This was implemented in the hypervisor layer, so no special hardware was required. No changes to OS or applications were required either. However, maintaining determinacy was dependent on the instruction set architecture (ISA). The authors implemented a PA-RISC ISA prototype system. The hypervisor code was on the order of 24 thousand lines of code (24 KLOC). Note that processing speed reduced by approximately half due to overhead.

In 2002, Sapuntzakis et al. [85] proposed a method that encapsulates VM state and rapidly migrates it to a different host. They discussed methods for reducing the amount of transferred data including copy-on-write to disk, ballooning to handle unused memory efficiently, paging methods for getting only the blocks needed, and using a hash to detect matches in the destination block. The main objectives of the proposed method were to move computing environments as users move and improve manageability of environments by encapsulating them, but they also suggested applications for failover using portability of VMs.

In 2005, Clark et al. [86] proposed a method for live migration of running VMs to different BMs, which they implemented on Xen. During the proposed pre-copy, memory pages are first partitioned and repeatedly sent to the destination BM without stopping VM execution. During this time, any changed pages are resent. In the final phase, the VM is stopped, the remaining changed pages are sent, and VM execution is resumed. To users, the time required by the final phase is down time, and is

© 2016 Information Processing Society of Japan

less than the total migration time from start to end of the migration. There is a tradeoff between reducing down time and reducing total migration time, which was considered in the design. This paper proposed the concept of a writable working set. By estimating the writable working set for the workload, the number of pre-copy rounds is decided. The migration traffic affects VM processing, so a rate-adaptive algorithm is used to adjust the transmission rate. The implementation achieved migration of an interactive workload on a commodity cluster with down time of 60 ms. Live migration of VMs has much in common with VM replication for HA and FT, and can be applied to them as well.

In 2008, Cully et al. [87] proposed the Remus FT system based on VM replication. Remus requires several seconds of down time during failover, but preserves state including network connections. Protected software is encapsulated in a virtual machine and a VM snapshot is replicated to the backup host in 25 ms. Note that even though the system replays on the backup host from a checkpoint when a fault occurs, the same output is not guaranteed due to non-determinism. For this reason, external output cannot be released until the state that produced the output has been replicated on the backup host. One way to achieve this is to synchronize when there is external output. Instead of this, Remus sets synchronization points every 25 ms and buffers output till the next point. This reduces output latency and run-time overhead. Note that writing to disk is also duplicated on the backup host, and writing is done upon synchronization. Remus is implemented on Xen using live migration functions and is a standard feature as of Xen 4.0 [88].

In 2008, Tamura et al. proposed the Kemari [89] FT system, which replicates VM snapshots on a backup host, similar to Remus. In contrast with Remus, it performs synchronization when there is external output. This eliminates the buffering latency required by Remus. On the other hand, when output frequency is low, snapshots that must be replicated become larger and the latency for one synchronization increases. Conversely, when output frequency exceeds the Remus synchronization frequency, synchronization overhead becomes higher than that of Remus.

In 2013, Dong et al. proposed the COLO [90] FT system, which performs lock-step execution with coarse granularity. With COLO, the VM on the primary host (called P) executes in parallel with the VM on the secondary host (S). Both receive the same input, and external outputs are compared before being released. When P and S output the same data, it is released externally and execution continues. When outputs differ, the output from P is released after stopping P, taking a snapshot, and using it to synchronize S. Then execution of P and S is re-started. Note that even if the outputs match, the states of P and S are not necessarily the same. In fact, any differences in state cannot be observed externally, and no inconsistencies arise, even after failover from P to S. COLO performs lock-step operation when there is external output. There is overhead from distributing input and comparing output. This is less overhead than performing lock-step operation at the instruction level, as with the method from Bressoud. It is also less overhead than Remus, which performs periodic synchronization, as long output from P and S match. However, when outputs do not match, the full state of P must be synchronized, resulting in delay. In Ref. [90], optimizations to increase the rate at which outputs match are proposed, such as ignoring packet timestamps and comparing outputs by client to absorb any differences in the order of output caused by the concurrent programs. COLO is a standard feature of Xen as of Xen 4.5.

8. Security

There are three perspectives on security regarding cloud computing.

The first is from the perspective of system operation. To maintain systems in a secure state, activity such as network input and output must be monitored and security patches applied, and these can be complex tasks. In fact, many incidents result from incompetent operation. Users can outsource OS-level security administration to a provider by using PaaS or SaaS. This can maintain the level of operational security. On a private cloud, VM images can be centrally managed, unifying security management and making policy enforcement easier.

The second perspective is security of virtualized systems. Security of virtualized systems depends on the hypervisor, which is the TCB. Generally, the hypervisor has a smaller code base than the OS. For example, the source code for Xen is approximately 100 KLOC. This means there is less likelihood that vulnerabilities will be introduced in a hypervisor than an OS, and isolation of environments should be more robust. On the other hand, if a vulnerability is found in the hypervisor, there is a risk that security for all VMs it manages will be compromised. It is also difficult to detect hypervisor malware infections from a guest OS environment [91], [92], [93], [94]. Various methods for increasing the security of virtualized environments have been proposed.

Garfinkel et al. proposed an integrity verification method to check that system software comprising the TCB has not been tampered with Refs. [95], [96]. The integrity verification computes a hash value of the code when system software such as the boot loader or hypervisor is launched and during execution [97] and compares it with a stored value computed beforehand. The hash value is digitally signed to guarantee its validity. This feature is used in the Windows 8 secure boot feature.

Steinberg et al. have proposed the NOVA micro hypervisor based virtualization architecture [98]. The kernel mode micro hypervisor is 9 KLOC. This is smaller than both Xen and Hyper-V, which are approximately 100 KLOC. Performance drops slightly because virtualization is implemented in the user space, but it simplifies the interface and the TCB is smaller. This reduces the attack surface, increasing security.

Szefer at al. have proposed the noHype virtualization system that runs multiple VMs in parallel and runs guest OSs directly on the hardware [99]. With noHype, processor and memory are allocated to a VM before it is executed and the OS is modified so that all resources are determined before the OS is launched. Guest OSs access virtualized I/O devices directly. This reduces the cases when control passes from VM to hypervisor, thereby reducing opportunities for malware infection from guest OS to the hypervisor.

In 2013, Yarom et al. identified a side-channel attack called FLUSH+RELOAD [100], in which the internal state of VM run-

© 2016 Information Processing Society of Japan

ning on the same processor could be guessed by observing processor L3 cache access patterns. In combination with a Web server attack, it enables encryption keys in a target VM to be guessed. There are various conditions for a successful attack, including the need for the target source code, and that attack code must be running on the same processor as the target VM, but the attach shows one way that information can leak through the processor. Special consideration is needed when implementing highly confidential processing such as encryption.

The third perspective is monitoring and control of VMs when using virtualization. The hypervisor runs with higher privilege than the VMs and controls them. Hypervisors not only realize VM isolation, they can also inspect and interpose on VM activities. This is called virtual machine introspection (VMI) [101]. VMI is able to detect attacks from within the LAN that cannot be detected by a firewall located at the border of intranet, and individual VMs can be isolated individually from the LAN. A VMI library called libVMI [102] has also been developed, supporting the hypervisors in Xen, KVM and others.

9. Open Source Cloud Platforms

Most components comprising cloud infrastructures have been developed as OSS. For example, there is infrastructure, including hypervisors, OSs, and data stores, as well as tools for resource management, configuration management, deployment, and monitoring. There are also cloud platforms providing the software stack needed to build private or public clouds, including Eucalyptus [103], Open Nebura [104], CloudStack [105], and OpenStack [45].

The OpenStack project was started by Rackspace Hosting and NASA in July, 2010. As of 2015, it is being developed with a six-month release cycle under management of the OpenStack Foundation, which has over 500 member enterprises. There are projects to develop various components, including Nova (compute), Swift (object storage), Glance (imaging service), Keystone (identity service), Horizon (dashboard), Newtron (networking), and Cinder (block storage). OpenStack APIs are compatible with Amazon EC2 and S3, and can be used with major Linux distributions. It is being used with large scale environments on the order of 200,000 cores.

Cloud Foundry [106] is an OSS PaaS platform. The Cloud Foundry Foundation was established in 2014. Projects include DIEGO (execution environment), LATTICE (container management), and BOSH (lifecycle management tool).

10. Applications

10.1 High Performance Computing

Initially, cloud computing was applied in fields such as Web servers and enterprise systems with small to medium workloads, and resources provided by cloud systems were suitable for that scale of workload. However, the reliability, scalability, and cost of cloud systems were also useful for other fields such as high-performance computing (HPC).

Around 2010, Ostermann [107], Iosup [108], and Jackson [109] each evaluated the performance of cloud systems when applied to scientific computing. The results showed that network latency can reduce performance, and in particular, performance drops as VM instances are added due to reduced network performance and this can lead to reduced scalability. EC2 is often suitable for scientific computing, but for large problems using 1,024 or more cores, performance drops by an order of magnitude. Performance can be improved by tuning programs to consider cache size and using high-speed interconnects. The cloud model can also be extended to HPC by utilizing BM resources accessible in a cloud. Note that in EC2, high performance instances, 10 Gbps interconnects, and high-speed I/O through SR-IOV are currently available for HPC. In 2011, the EC2 cluster ranked 42nd in the top 500 supercomputing sites [110]. The instance was cc2.8xlarge (Xeon 8C 2.6 GHz, 10G Ethernet), with 17,024 cores.

10.2 Large Scale Data Processing

Large-scale data processing infrastructures have some points in common with clouds in that they are scalable cluster-based systems, and are one application of cloud systems.

Google developed its Google FS (GFS) scalable distributed file system to perform the high-speed crawling and indexing required for its search services, and published it in 2003 [111], [112]. The system uses commodity hardware, so faults are assumed to occur under normal operation. For the assumed workload, many clients will write to the same file in parallel, and multiple clients will also read from the file in parallel. Most writes will be appends, overwrites will be rare, and most reads will be sequential. Beyond these, the file system was designed to have fault tolerance and to prioritize high aggregated throughput rather than low latency.

In GFS, files are partitioned into and managed in 64 MB chunks. Chunks are identified with a globally unique 64 bit ID. Chunks also have version numbers and checksums attached to them.

The system is composed of one master, multiple chunk servers, and multiple clients. The master has metadata, including the directory hierarchy and chunk locations. The chunk servers hold the chunks. For fault tolerance, chunks are replicated on three chunk servers. The metadata on the master is held in memory. In preparation for a master fault, the metadata change history is saved. The master also performs chunk repositioning and garbage collection, among other tasks.

GFS separates data and control in order to simplify data flow. Data does not go through the master and is exchanged directly between clients and chunk servers. Data replication is performed by the chunk servers.

In addition to the usual read and write file operations, a record append operation is introduced. Record append adopts a week consistency model in order to gain throughput.

Google also developed Colossus (GFS2) [113] as a successor to GFS. Colossus was designed with low latency for real-time processing, a 1 MB chunk size, and eliminating any single point of failure.

In 2004, Google announced MapReduce [114] as a programming model for large-scale data processing. With MapReduce, the user describes the process using a map function and a reduce function. The map function generates a set of intermediate data

© 2016 Information Processing Society of Japan

(key-value pairs) from the input, and the reduce function takes that set of key-value pairs as input and computes outputs that summarize it. For example, to count the frequency of words appearing in a text, the map function would divide the input text into words and create a key value pair for each word as output. Here, the key would be the word and the value would be a frequency of 1. The key-value pairs are classified by key (by word), and input to the reduce function. The reduce function computes totals from the input values and outputs the frequency of each word. Multiple workers process both the map function and the reduce function. Parallelization and load balancing is done automatically by the MapReduce library. A locality optimization, which co-locates data and compute nodes, is also performed. MapReduce was implemented on a cluster of thousands commodity PCs. Directly connected IDE drives were used for storage and GFS was used for reliability. MapReduce is used for tasks including Web page analysis, large-scale machine learning, satellite image processing, and machine translation. As of 2008, over 10,000 MapReduce programs had been written, and 20 PB of data per day had been processed on the cluster in 100,000 jobs every day.

Google also developed a scalable distributed data store called Bigtable [115] for use by its own services, which was published in 2006. By designing their own data store, they were able to support scalability and they gained design freedom to eliminate bottlenecks. Bigtable features scalability to support thousands of commodity servers and handle several petabytes of data, it supports various data-size and latency requirements so it can be used with many applications, and it uses a simplified data model rather than a relational data model. Data is stored as multi-dimensional tables, and is identified by row, column and timestamp. Data consists of byte arrays (character strings), and keys are character strings specifying row and column. GFS is used to store data and logs and Chubby is used as a distributed lock service.

Bigtable is used for Web indexing, and services such as Gmail, YouTube, Google Maps, and Google Earth, and as a platform for MapReduce. It has been in use since April, 2005, and was being used from 60 projects as of August, 2006. It required seven person-years for design and implementation.

In 2005, inspired by the Google GFS and MapReduce papers, Cutting et al. implemented them as OSS. Currently, this set of modules is being developed by the OSS community as Apache Hadoop [42] (hereinafter, “Hadoop”).

Several projects are included within the Apache Hadoop project.

The HDFS project is developing the Hadoop distributed file system (HDFS) [116]. HDFS was designed under almost the same assumptions as GFS, but the implementation is slightly different; some tasks are handled by the clients rather than by the servers.

The MapReduce project is developing the MapReduce large-scale data processing platform. MapReduce can be used with file systems other than HDFS and on public clouds. However, it does not perform well unless locality optimization is done.

In 2008, Hadoop broke the world record for sorting. It performed a 1 TB sort in 209 seconds using a 910-node cluster. In 2008, Yahoo disclosed that it was generating its Web search data using a Hadoop application on a 10,000-core cluster. In 2012, Facebook was maintaining 100 PB of data on a Hadoop cluster and adding 0.5 PB every day.

11. Summary and Future Directions

This paper has described technologies supporting cloud computing. After summarizing cloud benefits and categories of use, it outlines technologies for virtualization, storage, networking, high-availability, security, an open-source software stack, and applications for HPC and large-scale data processing. Four considerations for future directions are discussed below.

The first is expansion of platform functionality. Using cloud computing to lease IT resources, anyone with an idea can create a service from it. In the future, various platform functions are expected to appear that will make this even easier. For example, functions that will make it simple for people without system expertise to use them. Currently, tools for software development and collaboration, analytics platforms, machine learning, stream processing platforms based on lambda architecture, and marketplaces are being provided as cloud services. These combinations will be organized as design patterns and become established as architectures covering layers from business workflow to software organization.

The second is development of open source and Software Defined Anything (SDx). In addition to OSS, open innovation will accelerate through standardization of hardware platforms, with OCP [117] for instance. Administration will continue to get easier through SDN, SDS, and Software Designed Data Centers (SDDC), and it will become possible to manage a wider range of resources centrally.

The third is that versatility will increase. In addition to general server resources, resources specialized for particular applications will be usable on cloud systems. GPU virtualization and heterogeneous computing are possibilities.

Finally, new systems will be redefined and globally optimized, incorporating new elemental technologies and new requirements. SSD prices are dropping and they continue to replace disk storage [118], and high-speed persistent memory is expected to become practical soon. Interconnects between processors are also expected to increase in speed. Optimized systems in these environments also need to be designed. Resource management functions are duplicated in hypervisors and OSs. There are two technologies, unikernel and containers, that remove one or the other. There is still room for optimization of technology in this area. Regarding storage and networking, there will be further work on whether to ensure fault tolerance and other added values at the equipment level, or to implement them with commodity hardware and software, comparing them on operations costs and other issues.

Global data and traffic is concentrated at data centers and the volume is increasing exponentially, approximately doubling every two years. Networks and protocols suited to those sorts of data and traffic distributions are needed.

Acknowledgments The authors would like to thank Mr.Seiji Kihara, Mr.Toshio Hitaka and anonymous referees for their helpful comments.

© 2016 Information Processing Society of Japan

References

[1] IFS, WHY BUSINESS AGILITY MATTERS, RIGHT NOW (online), available from ⟨http://www4.ifsworld.com/business-agility-now⟩ (accessed 2015-07-29).

[2] Smith, J. and Nair, R.: *Virtual Machines: Versatile Platforms for Systems and Processes*, Elsevier (2005).

[3] Bernstein, D., Ludvigson, E., Sankar, K., et al.: Blueprint for the Intercloud - Protocols and Formats for Cloud Computing Interoperability, *Proc. ICIW'09*, pp.328–336, IEEE (2009).

[4] Amazon, AWS Documentation (online), available from ⟨https://aws.amazon.com/documentation/⟩ (accessed 2015-07-29).

[5] Intel: Intel 64 and IA-32 Architectures Software Developer's Manual.

[6] Sugerman, J., Venkitachalam, G. and Lim, Beng-Hong: Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor, *Proc. USENIX ATC'01, General Track*, pp.1–14 (2001).

[7] Barham, P., Dragovic, B., Fraser, K., et al.: Xen and the Art of Virtualization, *Proc. SOSP'03*, pp.164–177, ACM (2003).

[8] Kivity, A., Kamay, Y., Laor, D., et al.: kvm: the Linux Virtual Machine Monitor, *Proc. Linux Symposium*, Vol.1, pp.225–230 (2007).

[9] Popek, G.J. and Goldberg, R.P.; Formal Requirements for Virtualizable Third Generation Architectures, *Comm. ACM*, Vol.17, No.7, pp.412–421, ACM (1974).

[10] Robin, J.S. and Cynthia E.I.: Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor, *Proc. USENIX Security'00*, pp.129–144 (2000).

[11] Sites, R.L., Shernoff, A., Kirk, M.B., et al.: Binary Translation, *Comm. ACM*, Vol.36, No.2, pp.69–81, ACM (1993).

[12] Adams, K. and Agesen, O.: A Comparison of Software and Hardware Techniques for x86 Virtualization, *Proc. ASPLOS'06*, pp.2–13, ACM (2006).

[13] DEPARTMENT OF DEFENSE TRUSTED COMPUTER SYSTEM EVALUATION CRITERIA (TCSEC), DoD 5200.28-STD (1985).

[14] Uhlig, R., Neiger, G., Rodgers, D., et al.: Intel Virtualization Technology, *Computer*, Vol.38, No.5, pp.48–56, IEEE (2005).

[15] Miloś, G., Murray, D.G., Hand, S., et al.: Satori: Enlightened page sharing, *Proc. USENIX ATC'09*, pp.v1–14 (2009).

[16] Bellard, F.: QEMU, a Fast and Portable Dynamic Translator, *Proc. USENIX ATC'05, FREENIX Track*, pp.41–46 (2005).

[17] AMD: AMD I/O Virtualization Technology (IOMMU) Specification, 48882-Rev 2.62-February 2015 (2015).

[18] Intel: Intel Virtualization Technology for Directed I/O - Architecture Specification, rev2.3 (Oct. 2014).

[19] PCI-SIG: Single Root I/O Virtualization and Sharing Specification Revision 1.0 (2007).

[20] Gordon, A., Amit, N., Har'El, N., et al.: ELI: Bare-Metal Performance for I/O Virtualization, *Proc. ASPLOS'12*, pp.411–422, ACM (2012).

[21] Madhavapeddy, A. and Scott, D.J.: Unikernels: Rise of the Virtual Library Operating System, *ACM Queue*, Vol.11, No.11, pp.30–44, ACM (2013).

[22] Madhavapeddy, A., Mortier, R., Rotsos, C., et al.: Unikernels: Library Operating Systems for the Cloud, *Proc. ASPLOS'13*, pp.461–472, ACM (2013).

[23] NEC: Tiny, Agile Virtual Machines for Network Processing (online), available from ⟨http://cnp.neclab.eu/clickos/⟩ (accessed 2015-07-29).

[24] Martins, J., Ahmed, M., Raiciu, C., et al.: ClickOS and the Art of Network Function Virtualization, *Proc. NSDI'14*, pp.459–473, USENIX Association (2014).

[25] Soltesz, S., Potzl, H., Fiuczynski, M.E., et al.: Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors, *Proc. EuroSys'07*, pp.275–287, ACM (2007).

[26] Kamp, P.H. and Watson, R.N.M.: Jails: Confining the omnipotent root., *Proc. SANE'00*, Vol.43, pp.116–126 (2000).

[27] Odin: Virtuozzo (online), available from ⟨http://www.odin.com/products/virtuozzo/⟩ (accessed 2015-07-29).

[28] Price, D. and Tucker, A.: Solaris Zones: Operating System Support for Consolidating Commercial Workloads, *Proc. LISA'04*, pp.241–254, USENIX Association (2004).

[29] OpenVZ (online), available from ⟨https://openvz.org/⟩ (accessed 2015-07-29).

[30] Linux Containers (online), available from ⟨https://linuxcontainers.org/⟩ (accessed 2015-07-29).

[31] Hindman, B., Konwinski, A., Zaharia, M., et al.: Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center, *Proc. NSDI'11*, pp.22–35 (2011).

[32] Docker (online), available from ⟨https://www.docker.com/⟩ (accessed 2015-07-29).

[33] Core OS (online), available from ⟨https://coreos.com/⟩ (accessed 2015-07-29).

[34] Open Container Initiative (online), available from ⟨https://www.opencontainers.org/⟩ (accessed 2015-07-29).

[35] Open Container Specifications (online), available from ⟨https://github.com/opencontainers/specs⟩ (accessed 2015-07-29).

[36] runc (online), available from ⟨https://github.com/opencontainers/runc⟩ (accessed 2015-07-29).

[37] Kubernetes (online), available from ⟨http://kubernetes.io/⟩ (accessed 2015-07-29).

[38] NAMESPACES(7), Linux Programmer's Manual (online), available from ⟨http://man7.org/linux/man-pages/man7/namespaces.7.html⟩ (accessed 2015-07-29).

[39] CGROUPS (online), available from ⟨https://www.kernel.org/doc/Documentation/cgroups/⟩ (accessed 2015-07-29).

[40] CAPABILITIES(7), Linux Programmer's Manual (online), available from ⟨http://man7.org/linux/man-pages/man7/capabilities.7.html⟩ (accessed 2015-07-29).

[41] Xavier, M.G., Neves, M.V., Rossi, F.D., el al.: Performance Evaluation of Container-based Virtualization for High Performance Computing Environments, *Proc. PDP'13*, pp.233–240, IEEE (2013).

[42] White, T.: Hadoop: The definitive guide, O'Reilly Media, Inc. (2012).

[43] Amazon EBS (online), available from ⟨https://aws.amazon.com/ebs/details/⟩ (accessed 2015-07-29).

[44] OpenStack Operations Guide, 6. Storage Decisions (online), available from ⟨http://docs.openstack.org/openstack-ops/openstack-ops-manual.pdf⟩ (accessed 2015-07-29).

[45] Open source software for creating private and public clouds (online), available from ⟨https://www.openstack.org/⟩ (accessed 2015-07-29).

[46] Weil, S.A., Brandt, S.A., Miller, E.L., et al.: Ceph: A Scalable, High-Performance Distributed File System, *Proc. OSDI'06*, pp.307–320, USENIX Association (2006).

[47] GLUSTER (online), available from ⟨http://www.gluster.org/⟩ (accessed 2015-07-29).

[48] Morita, K.: Sheepdog: Distributed storage system for qemu/kvm, LCA 2010 DS&R miniconf (2010).

[49] Sheepdog (online), available from ⟨https://sheepdog.github.io/sheepdog/⟩ (accessed 2015-07-29).

[50] RFC 7530, Network File System (NFS) Version 4 Protocol, Mar. 2015.

[51] Gibson, G.A., Nagle, D.F., Amiri, K., et al.: File Server Scaling with Network-Attached Secure Disks, *Proc. SIGMETRICS'97*, pp.272–284, ACM (1997).

[52] Amazon S3 (online), available from ⟨https://aws.amazon.com/s3/details/⟩ (accessed 2015-07-29).

[53] Braam, P.J.: The Lustre Storage Architecture, Cluster File Systems, Inc. (2004).

[54] Welcome to Swift's documentation! (online), available from ⟨http://docs.openstack.org/developer/swift/⟩ (accessed 2015-07-29).

[55] DeCandia, G., Hastorun, D., Jampani, M., el al.: Dynamo: Amazon's Highly Available Key-value Store, *Proc. SOSP'07*, pp.205–220, ACM (2007).

[56] Lakshman, A. and Malik, P.: Cassandra - A Decentralized Structured Storage System, *ACM SIGOPS Operating Systems Review*, Vol.44, No.2, pp.35–40, ACM (2010).

[57] monogoDB (online), available from ⟨https://www.mongodb.org/⟩ (accessed 2015-07-29).

[58] ONF: Software-Defined Networking: The New Norm for Networks (online), available from ⟨https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf⟩ (accessed 2015-07-29).

[59] McKeown, N., Anderson, T., Balakrishnan, H., et al.: OpenFlow: Enabling Innovation in Campus Networks, *Proc. SIGCOMM'08*, pp.69–74, ACM (2008).

[60] ONF: OpenFlow Switch Specification Version 1.3.4 (online), available from ⟨https://www.opennetworking.org/sdn-resources/openflow/57-sdn-resources/onf-specifications/openflow/⟩ (accessed 2015-07-29).

[61] RFC7348 Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks.

[62] Internet-Draft: A Stateless Transport Tunneling Protocol for Network Virtualization (STT) (online), available from ⟨https://www.ietf.org/archive/id/draft-davie-stt-06.txt⟩ (accessed 2015-07-29).

[63] RFC2784 Generic Routing Encapsulation (GRE)

[64] Build SDN Agilely (online), available from ⟨http://osrg.github.io/ryu/index.html⟩ (accessed 2015-07-29).

[65] Trema (online), available from ⟨http://trema.github.io/trema/⟩ (accessed 2015-07-29).

[66] OpenDaylight Controller (online), available from ⟨https://wiki.opendaylight.org/view/OpenDaylight_Controller:Main⟩

© 2016 Information Processing Society of Japan

(accessed 2015-07-29).
[67] OpenContrail (online), available from ⟨http://www.opencontrail.org/⟩ (accessed 2015-07-29).
[68] Project Floodlight (online), available from ⟨http://www.projectfloodlight.org/floodlight/⟩ (accessed 2015-07-29).
[69] Pfaff, B., Pettit, J. and Koponen, T.: Extending Networking into the Virtualization Layer, *Proc. HotNets'09*, ACM (2009).
[70] OvS Open vSwitch (online), available from ⟨http://openvswitch.org/⟩ (accessed 2015-07-29).
[71] Lagopus switch (online), available from ⟨https://lagopus.github.io/⟩ (accessed 2015-07-29).
[72] DPDK (online), available from ⟨http://dpdk.org/⟩ (accessed 2015-07-29).
[73] ETSI GS NFV 002: Network Functions Virtualisation (NFV); Architectural Framework.
[74] ETSI: Network Functions Virtualisation (online), available from ⟨http://www.etsi.org/technologies-clusters/technologies/nfv⟩ (accessed 2015-07-29).
[75] OPNFV (online), available from ⟨https://www.opnfv.org/software⟩ (accessed 2015-07-29).
[76] OPEN DAYLIGHT (online), available from ⟨https://www.opendaylight.org/⟩ (accessed 2015-07-29).
[77] Pickett, G.: Stay Persistent in Software Defined Networks, Blackhat USA (2015).
[78] Leiserson, C.E.: Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing, *IEEE Trans. on Comput.*, Vol.C-34, No.10, pp.892–901, IEEE (1985).
[79] Clos, C.: A Study of Non-Blocking Switching Networks, *Bell Sys. Tech. J.*, Vol.32, No.2, pp.406–424 (1953).
[80] Dally, W.J. and Towles, B.P.: *PRINCIPLES AND PRACTICES OF INTERCONNECTION NETWORKS*, Elsevier (2004).
[81] Al-Fares, M., Loukissas, A. and Vahdat, A.: A Scalable, Commodity Data Center Network Architecture, *Proc. SIGCOMM'08*, pp.63–74, ACM (2008).
[82] Mysore, R.N., Pamboris, A., Farrington, N., et al.: PortLand: A Scalable Fault-Tolerant Layer 2 Data Center Network Fabric, *Proc. SIGCOMM'09*, pp.39–50, ACM (2009).
[83] Singh, A., Ong, J., Agarwal, A., et al.: Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network, *Proc. SIGCOMM'15*, pp.183–197, ACM (2015).
[84] Bressoud, T.C. and Schneider, F.B.: Hypervisor-based Fault-tolerance, *ACM Trans. Comput. Syst.* (*TOCS*), Vol.14, No.1, pp.80–107, ACM (1996).
[85] Sapuntzakis, C.R., Chandra, R., Pfaff, B., et al.: Optimizing the Migration of Virtual Computers, *Proc. OSDI'02*, pp.377–390, USENIX Association (2002).
[86] Clark, C., Fraser, K., Hand, S., et al.: Live Migration of Virtual Machines, *Proc. NSDI'05*, pp.273–286, USENIX Association (2005).
[87] Cully, B., Lefebvre, G., Meyer, D., et al.: Remus: High Availability via Asynchronous Virtual Machine Replication, *Proc. NSDI'08*, pp.161–174, USENIX Association (2008).
[88] Remus (online), available from ⟨http://wiki.xen.org/wiki/Remus⟩ (accessed 2015-07-29).
[89] Tamura, Y., Sato, K., Kihara, S., et al.: Kemari: Virtual Machine Synchronization for Fault Tolerance, *Proc. USENIX ATC'08* (*Poster Session*), USENIX Association (2008).
[90] Dong, Y., Ye, W., Jiang, Y., et al.: COLO: COarse-grained LOck-stepping Virtual Machines for Non-stop Service, *Proc. SoCC'13*, Article No.3, ACM (2013).
[91] King, S.T., Chen, P.M., Wang, Y.-M., et al.: SubVirt: Implementing malware with virtual machines, *Proc. IEEE Symp. Security and Privacy* (*S&P*)*'06*, IEEE (2006).
[92] Zovi, D.A.D.: Hardware Virtualization Rootkits, Black Hat USA (2006).
[93] Rutkowska, J.: Security Challenges in Virtualized Environments, RSA Conference 2008 (online), available from ⟨http://invisiblethingslab.com/itl/Resources.html⟩ (accessed 2015-07-29).
[94] Perez-Botero, D., Szefer, J. and Lee, R.B.: Characterizing Hypervisor Vulnerabilities in Cloud Computing Servers, *Proc. Workshop on Security in Cloud Computing* (*SCC'13*), ACM (2013).
[95] Garfinkel, T., Pfaff, B., Chow, J., et al.: Terra: A Virtual Machine-Based Platform for Trusted Computing, *Proc. SOSP'03*, pp.193–206, ACM (2003).
[96] Sailer, R., Zhang, X., Jaeger, T., et al.: Design and Implementation of a TCG-based Integrity Measurement Architecture, *Proc. USENIX Security Symp. '04*, USENIX Association (2004).
[97] Grawrock, D.: *Dynamics of a Trusted Platform: A building block approach*, Intel Press (2009).
[98] Steinberg, U. and Kauer, B.: NOVA: A Microhypervisor-Based Secure Virtualization Architecture, *Proc. EuroSys'10*, pp.209–222, ACM (2010).
[99] Szefer, J., Keller, E., Lee, R.B., et al.: Eliminating the Hypervisor Attack Surface for a More Secure Cloud, *Proc. CCS'11*, pp.401–412, ACM (2011).
[100] Yarom, Y. and Falkner, K.E.: Flush+Reload: A High Resolution, Low Noise, L3 Cache Side-Channel Attack, *Proc. USENIX Security Symp. '14*, pp.719–732, USENIX Association (2014).
[101] Garfinkel, T. and Rosenblum, M.: A Virtual Machine Introspection Based Architecture for Intrusion Detection, *Proc. NDSS'03*, pp.191–206 (2003).
[102] LibVMI (online), available from ⟨http://libvmi.com/⟩ (accessed 2015-07-29).
[103] Nurmi, D., Wolski, R., Grzegorczyk, C., et al.: The Eucalyptus Open-source Cloud-computing System, *Proc. CCGRID'09*, pp.124–131, IEEE (2009).
[104] Milojičić, D., Llorente, I.M. and Montero, R.S.: OpenNebula: A Cloud Management Tool, *IEEE Internet Computing*, Vol.15, No.2, pp.11–14, IEEE (2011).
[105] Apache CloudStack (online), available from ⟨https://cloudstack.apache.org/⟩ (accessed 2015-07-29).
[106] Cloud Foundry (online), available from ⟨https://www.cloudfoundry.org/⟩ (accessed 2015-07-29).
[107] Ostermann, S., Iosup, A., Yigitbasi, N., et al.: A Performance Analysis of EC2 Cloud Computing Services for Scientific Computing, *Proc. Cloudcomp'09*, LNICST, Vol.34, pp.115–131, Springer Berlin Heidelberg (2010).
[108] Iosup, A., Ostermann, S., Yigitbasi, M.N., el al.: Performance Analysis of Cloud Computing Services for Many-Tasks Scientific Computing, *IEEE Trans. Parallel and Distributed Systems*, Vol.22, No.6, pp. 931–945, IEEE (2011).
[109] Jackson, K.R., Ramakrishnan, L., Muriki, K., et al.: Performance Analysis of High Performance Computing Applications on the Amazon Web Services Cloud, *Proc. CloudCom'10*, pp.159–168, IEEE (2010).
[110] TOP 500 AMAZON EC2 CLUSTER COMPUTE INSTANCES (online), available from ⟨http://www.top500.org/system/177457⟩ (accessed 2015-07-29).
[111] Ghemawat, S., Gobioff, H. and Leung, S.T.: The Google File System, *Proc. SOSP'03*, pp.29–43, ACM (2003).
[112] McKusick, K. and Quinlan, S.: GFS: Evolution on Fast-forward, *ACM Queue*, Vol.7, No.7, ACM (2009).
[113] Fikes, A.: Storage Architecture and Challenges, *Talk at the Google Faculty Summit* (2010).
[114] Dean, J. and Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters, *Comm. ACM*, Vol.51, No.1, pp.107–113, ACM (2008).
[115] Chang, F., Dean, J., Ghemawat, S., et al.: Bigtable: A Distributed Storage System for Structured Data, *ACM Trans. Comput. Syst.* (*TOCS*), Vol.26, No.2, Article 4, ACM (2008).
[116] Shvachko, K., Kuang, H., Radia, S., et al.: The Hadoop Distributed File System, *Proc. Symp. Mass Storage Systems and Technologies* (*MSST*) *2010*, pp.1–10, IEEE (2010).
[117] Open Compute Project (online), available from ⟨http://www.opencompute.org/⟩ (accessed 2015-07-29).
[118] Meza, J., Wu, Q., Kumar, S., et al.: A Large-Scale Study of Flash Memory Failures in the Field, *Proc. SIGMETRICS'15*, ACM (2015).

Yukio Tsuruoka received his B.E. and M.E. degrees from the University of Electro-Communications in 1985 and 1987, respectively. He joined Nippon Telegraph and Telephone Corporation in 1987. He received the Dr. Eng. from the University of Electro-Communications in 2001. He was a visiting professor, the University of Electro-Communications, Graduate School of Information Systems. He is presently a senior research engineer in NTT Software Innovation Center. His research interests include information security, cloud computing and mobile computing. He is a member of the IPSJ and the IEICE.

© 2016 Information Processing Society of Japan