Liquid: A Scalable Deduplication File System for Virtual Machine Images

Xun Zhao, Yang Zhang, Yongwei Wu, *Member, IEEE*, Kang Chen, Jinlei Jiang, *Member, IEEE*, and Keqin Li, *Senior Member, IEEE*

Abstract—A virtual machine (VM) has been serving as a crucial component in cloud computing with its rich set of convenient features. The high overhead of a VM has been well addressed by hardware support such as Intel virtualization technology (VT), and by improvement in recent hypervisor implementation such as Xen, KVM, etc. However, the high demand on VM image storage remains a challenging problem. Existing systems have made efforts to reduce VM image storage consumption by means of deduplication within a storage area network (SAN) cluster. Nevertheless, an SAN cannot satisfy the increasing demand of large-scale VM hosting for cloud computing because of its cost limitation. In this paper, we propose Liquid, a scalable deduplication file system that has been particularly designed for large-scale VM deployment. Its design provides fast VM deployment with peer-to-peer (P2P) data transfer and low storage consumption by means of deduplication on VM images. It also provides a comprehensive set of storage features including instant cloning for VM images, on-demand fetching through a network, and caching with local disks by copy-on-read techniques. Experiments show that Liquid's features perform well and introduce minor performance overhead.

Index Terms—Cloud computing, deduplication, file system, liquid, peer to peer, storage, virtual machine

1 Introduction

Cloud computing is widely considered as potentially the next dominant technology in IT industry. It offers simplified system maintenance and scalable resource management with *Virtual Machines* (VMs) [15]. As a fundamental technology of cloud computing, VM has been a hot research topic in recent years. The high overhead of virtualization has been well addressed by hardware advancement in CPU industry, and by software implementation improvement in hypervisors themselves.

Typically in cloud computing platforms, new VMs are created based on template images. In most cases, only a limited number of combinations of OS and software applications will be used. Preparing template images for such combinations would be enough for most users' needs, which is a common practice adopted by cloud computing pioneers like Amazon EC2 [1].

The growing number of VMs being deployed leads to increased burden on the underlying storage systems. To ensure that advanced VM features like migration and high availability could work fluently, VM images need to be accessible from more than one host machine. This leads to the common practice of storing VM images on shared network storage such as network-attached storage (NAS) and SAN, while the host machine's direct-attached storage (DAS) is only used for ephemeral storage. The problem of such an approach is that network storage systems usually cost several times more than DAS, and they have high demand on network IO performance. Moreover, the critical need to store thousands of VM images would be an extremely challenging problem for network storage systems because of the significant scale of storage consumption.

Studies have shown that the storage consumption issue brought by a large number of VM images could be addressed by deduplication techniques [17], [20], which have been extensively used in archival systems [32]. Existing systems have made efforts to address this issue on a SAN cluster by deduplication [10]. It is operated in a decentralized fashion, such that deduplication is done at the host machines running VMs, and unique data blocks are then stored on the SAN cluster. However, SANs are very expensive, and thus difficult to satisfy the ever-growing need of VM image storage in the future.

In this paper, we have proposed Liquid, which is a distributed file system particularly designed to simultaneously address the above problems faced in large-scale VM deployment. Its client side breaks VM images into small data blocks, references them by their fingerprints (calculated during a deduplication process), and uses deduplication techniques to avoid storing redundant data blocks. The deduplicated data blocks are then saved to a group of data servers, and the set of fingerprints is saved to a meta server. When a VM image is to be accessed, a Liquid client downloads its set of fingerprints from the meta server, fetches data blocks from data servers and peer clients in a P2P fashion, and exports an integrated VM image layout to hypervisors. Liquid's P2P data block transfer protocol reduces requests directly issued to data servers, uses DAS on each host machine more effectively, and guarantees high scalability of the whole system.

• *X. Zhao, Y. Zhang, Y. Wu, K. Chen, and J. Jiang are with the Department of Computer Science and Technology, Tsinghua National Laboratory for Information Science and Technology (TNLIST) Tsinghua University, Beijing 100084, China, and also with the Research Institute of Tsinghua University, Shenzhen 518057, China. E-mail: {zhaoxun09@mails., wuyw@, chenkang@, jjlei@}tsinghua.edu.cn.*
• *K. Li is with the Department of Computer Science, State University of New York, New Paltz, NY 12561 USA. E-mail: lik@newpaltz.edu.*

Manuscript received 9 Mar. 2013; revised 28 May 2013; accepted 26 June 2013. Date of publication 2 July 2013; date of current version 21 Mar. 2014. Recommended for acceptance by W. Jia.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2013.173

In order to support fast VM deployment, Liquid provides features such as fast VM image cloning and on-demand data block fetching. This allows creating a new VM from template image in a few milliseconds, and reduce network IO by only fetching data blocks when necessary. The copy-on-read technique is used to reduce network IO by caching previously accessed data blocks on DAS. This also results in the benefit of high availability of data blocks for P2P transfer.

The main contributions of this paper are summarized as follows.

1. We propose a deduplication file system with low storage consumption and high-performance IO, which satisfies the requirements of VM hosting.
2. We provide a P2P data block sharing scheme, which is highly scalable for large-scale deployment of VM images.
3. We develop additional techniques for reducing network IO and expediting VM creation, including fast VM image cloning, on-demand data block fetching, and copy-on-read, together with fault tolerance techniques for high availability.

Therefore, Liquid is a file system which achieves good performance in handling multiple challenges of VM creation.

The rest of this paper is organized as follows. Section 2 provides background information. Section 3 presents the design and implementation of Liquid. Section 4 evaluates the performance of Liquid. Section 5 reviews related work. Section 6 concludes the paper.

2 Background

2.1 VM Image Formats

There are two basic formats for VM images. The *raw image format* is simply a byte-by-byte copying of physical disk's content into a regular file (or a set of regular files). The benefit of raw format images is that they have better IO performance because their byte-by-byte mapping is straightforward. However, raw format images are generally very large in size, since they contain all contents in the physical disks, even including those blocks which never really get used.

The other one is the *sparse image format*. Instead of simply doing a byte-by-byte copy, the sparse image format constructs a complex mapping between blocks in physical disks and data blocks in VM images. For special blocks, such as those containing only zero bytes, a special mark is added on the block mapping, so that the blocks do not need to be stored, since their content could be easily regenerated when necessary. This will help reduce the size of newly created VM images, since most blocks inside the images would never be used, which only contain zero bytes, and hence, do not need to be stored. Manipulations on the block mapping and a VM image bring advanced features such as snapshotting, copy-on-write images [22], etc. However, the block mapping in sparse format also results in worse performance of IO compared with raw images. Interpreting the block mapping introduces additional overhead, and it generally breaks sequential IO issued by hypervisors into random IO on VM images, which significantly impairs IO performance.

Both formats are widely used in hypervisors such as Xen, KVM, VirtualBox, etc. All these hypervisors support raw format natively. Xen and KVM support the *qcow2* sparse format, which has two levels of block mapping and a block size of 256 KB. VirtualBox supports the *vdi* sparse format, which has one level of block mapping and a coarser granularity of data block size at 1 MB.

2.2 Deduplication Techniques

Data deduplication is a specialized data compression technique for eliminating duplicate copies of repeating data [8]. It aims at improving storage utilization. In the process of deduplication, unique blocks of data are usually identified by an analyzed fingerprint from their content. Whenever the fingerprint of a data block is calculated, it is compared with a stored fingerprint database to check for a match. This data block will be defined as a redundant data block, if an identical fingerprint is found. A redundant data block is replaced with a reference to the stored data block, instead of storing the same content multiple times. This technique will be highly effective when the original data set is redundant. For archival systems, research has shown that deduplication could be more effective than conventional compression tools [20].

The basic unit for deduplication could be a whole file, or sections inside a file. For the latter case, there are two methods to break a file into sections, namely, fixed size chunking and variable size chunking [32]. The *fixed size chunking* method splits the original file into blocks of the same size (except the last block). The *variable size chunking* method adopts a more complicated scheme, by calculating Rabin fingerprint [5] of a sliding window on file content, and detects more natural boundaries inside the file. Compared with variable size chunking, fixed size chunking will have better read performance, but it cannot handle non-aligned insertion in files effectively. Variable size chunking has been widely applied in archival systems, which deals with data that are rarely accessed [32]. For VM images, research has shown that fixed size chunking is good enough in measure of deduplication ratio [17], [20].

3 Design and Implementation

3.1 Assumptions

Here are a few assumptions for the VM images and their usage when Liquid is designed.

1. A disk image will only be attached to at most one running VM at any given time.
2. The disk images are mainly used to store OS and application files. User generated data could be stored directly into the disk image, but it is suggested that large pieces of user data should be stored into other storage systems, such as SAN. Temporary data could be saved into ephemeral disk images on DAS.

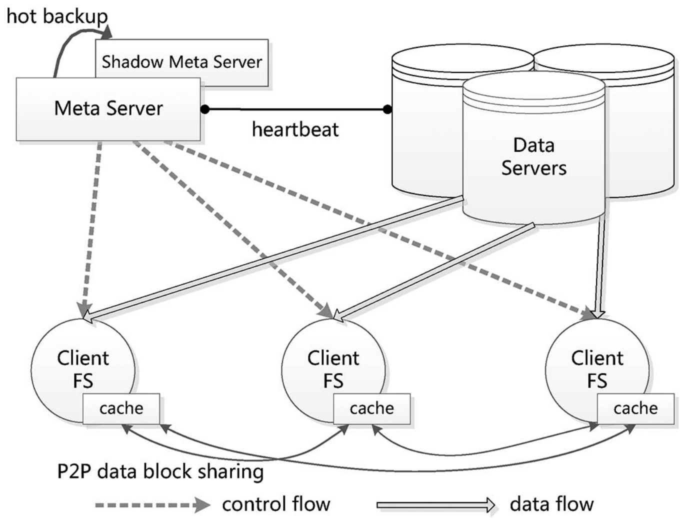


Fig. 1. Liquid architecture.

Assumption 1 is straightforward, as in physical world, one hard drive cannot be attached to multiple machines at the same time. Assumption 2 is aimed at achieving higher deduplication ratio and better IO performance for temporary data [30].

3.2 System Architecture

Liquid consists of three components, i.e., a single meta server with hot backup, multiple data servers, and multiple clients (see Fig. 1). Each of these components is typically a commodity Linux machine running a user-level service process.

VM images are split into fixed size data blocks. Each data block is identified by its unique fingerprint, which is calculated during deduplication process. Liquid represents a VM image via a sequence of fingerprints which refer to the data blocks inside the VM image.

The meta server maintains information of file system layout. This includes file system namespace, fingerprint of data blocks in VM images, mapping from fingerprints to data servers, and reference count for each data block. To ensure high availability, the meta server is mirrored to a hot backup shadow meta server.

The data servers are in charge of managing data blocks in VM images. They are organized in a distributed hash table (DHT) [29] fashion, and governed by the meta server. Each data server is assigned a range in the fingerprint space by the meta server. The meta server periodically checks the health of data servers, and issues data migration or replication instructions to them when necessary.

A Liquid client provides a POSIX compatible file system interface via the FUSE [4] toolkit. It acts as a transparent layer between hypervisors and the deduplicated data blocks stored in Liquid. The client is a crucial component, because it is responsible for providing deduplication on VM images, P2P sharing of data blocks, and features like fast cloning. When starting a new VM, client side of Liquid file system fetches VM image meta info and data blocks from the meta server, data servers and peer clients, and provides image content to hypervisors. After the shutting down of VMs, the client side uploads modified metadata to meta server, and pushes new data blocks to data servers, to make sure that the other client nodes can access the latest version of image files.

Liquid offers fault tolerance by mirroring the meta server, and by replication on stored data blocks. When the meta server crashes, the backup meta server will take over and ensure functionality of the whole system. Replicas of data blocks are stored across data servers, thus crashing a few data servers will not impair the whole system.

3.3 Deduplication in Liquid

3.3.1 Fixed Size Chunking

Liquid chooses fixed size chunking instead of variable size chunking. This decision is made based on the observation that most x86 OS use a block size of 4 KB for file systems on hard disks. Fixed size chunking applies well to this situation since all files stored in VM images will be aligned on disk block boundaries. Moreover, since OS and software application data are mostly read-only, they will not be modified once written into a VM image.

The main advantage of fixed size chunking is its simplicity. Storing data blocks would be easy if they have the same size, because mapping from file offset to data block could be done with simple calculations. Previous study [17] has shown that fixed size chunking for VM images performs well in measure of deduplication ratio.

3.3.2 Block Size Choice

Block size is a balancing factor which is very hard to choose, since it has great impact on both deduplication ratio and IO performance. Choosing a smaller block size will lead to higher deduplication ratio, because modifications on the VM images will result in smaller amount of additional data to be stored. On the other hand, smaller block size leads to more data blocks to be analyzed. When block size is too small, the sheer number of data blocks will incur significant management overhead, which impairs IO performance greatly. Moreover, smaller block size will result in more random seeks when accessing a VM image, which is also not tolerable.

Choosing block size smaller than 4 KB makes little sense because most OS align files on 4 KB boundaries. A smaller block size will not be likely to achieve higher deduplication ratio. A large block size is not preferable either, since it will reduce deduplication ratio, although the IO performance will be much better than a small block size.

Liquid is compiled with block size as a parameter. This makes it more adaptive to choose different block size under different situation. Based on our experience, it is advised to use a multiplication of 4 KB between 256 KB and 1 MB to achieve good balance between IO performance and deduplication ratio.

3.3.3 Optimizations on Fingerprint Calculation

Deduplication systems usually rely on comparison of data block fingerprints to check for redundancy. The fingerprint is a collision-resistant hash value calculated from data block contents. MD5 [26] and SHA-1 [12] are two cryptography hash functions frequently used for this purpose. The probability of fingerprint collision is extremely small, many orders of magnitude smaller than hardware error rates [25]. So we could safely assume that two data blocks are identical when they have the same fingerprint.

The calculation of fingerprint is relatively expensive. It is a major bottleneck for real time deduplication. To avoid such expensive fingerprint calculations while a VM image is being modified, Liquid delays fingerprint calculation for recently modified data blocks, runs deduplication lazily only when it is necessary.

The client side of Liquid file system maintains a *shared cache*, which contains recently accessed data blocks. Data blocks in a shared cache are read-only and shared among all VM images currently opened. When being requested for a data block, Liquid first tries to look it up in a shared cache. A cache miss will result in the requested data block being loaded into the shared cache. When the shared cache is filled up, cached data blocks are replaced using the least recently used (LRU) [9] policy. This layer of caching mechanism improves reading performance of VM images and ensures smooth operation of VMs.

Another portion of memory is used by the client side of Liquid file system as *private cache*, which contains data blocks that are only accessible from individual VM images. Private cache is used to hold modified data blocks, and delay fingerprint calculation on them. When a data block is modified, it is ejected from the shared cache if present, added to the private cache, and assigned a randomly generated *private fingerprint* instead of calculating a new fingerprint on-the-fly. This modified data block will then be referred to by the private fingerprint, until it is ejected from the private cache. The private fingerprint differs from normal fingerprint only by a bit flag, and part of it is generated from an increasing globally unique number, which guarantees that no collision will occur. The modified data block will be ejected from private cache when a hypervisor issues a POSIX `flush()` request, or the private cache becomes full and chooses to eject it based upon the LRU policy. Only then will the modified data block's fingerprint be calculated. This layer of caching mechanism improves writing performance of VM images and avoids repeated invalid fingerprint calculation to ensure the effectiveness of deduplication.

In order to speed up a deduplication process, Liquid uses multiple threads for fingerprint calculation. One of the concurrent threads calculates the fingerprint for one data block at one time, so multiple threads will process different data blocks currently. When ejecting modified data blocks from private cache, instead of ejecting a single element as in conventional LRU implementations, Liquid ejects multiple data blocks in one round. The ejected data blocks are appended into a queue, and analyzed by multiple fingerprint calculation threads. With this approach, we achieved linear speedup by increasing the number of fingerprint calculation threads, until the memory bandwidth is reached. Since fingerprint calculation is CPU intensive, and would probably contend with hypervisors, it is also possible to do fingerprint calculation on GPU instead of on CPU [18].

Based on our experience, 256 MB of shared cache and 256 MB of private cache would be sufficient for most cases. A group of four fingerprint calculation threads will provide good IO performance.

3.3.4 Storage for Data Blocks

Deduplication based on fixed size chunking leads to numerous data blocks to be stored. One solution is to store them directly into a local file system, as individual files. This approach will lead to additional management overhead in a local file system. Even though file systems such as ReiserFS [6] and XFS [7] have been designed to be friendly to small files, there will still be overhead on the frequent `open()`, `close()` syscalls and Linux kernel vnode layer. Moreover, most Linux file system implementations use linked lists to store meta data of files under a directory [21], so file look-up will have a time complexity of $O(n)$.

An alternative solution would be combining multiple small data blocks together into a single file, and managing them within the file. Storing data blocks into a database system would be one such solution, but the complex transaction logic of database systems incurs unnecessary overhead.

Liquid implements its own storage module for data blocks. Data blocks are split into groups according to their fingerprints. Three files are created for each group, i.e., an extent file containing all the data blocks' content, an index file mapping a fingerprint to corresponding data block's offset and reference count, and a bitmap file indicating if certain slot in the extent file is valid.

The bitmap file and index file are small enough to be loaded into memory. For example, with 256 KB data block size, and 1 TB size of unique data blocks, we will only have an index file size of 80 MB, and a bitmap file size of 512 KB. As shown in Fig. 2, look-up by fingerprint could get the block's address with hashing, and the whole time complexity is $O(1)$. Deleting a data block is as simple as flipping a bit flag in the bitmap file. When inserting a new data block, Liquid first seeks for unused slots by checking the bitmap file, and reuses any invalid slot. If no reusable slot is found, a new slot is appended to accommodate the data block. After a new data block is inserted, its corresponding bit flag in the bitmap file will be set, indicating the slot to be under use. Liquid borrows much from dynamic memory allocation algorithms, and uses a free list to track unused slots with $O(1)$ time complexity.

3.4 File System Layout

All file system meta data are stored on the meta server. Each VM image's meta data are stored in the meta server's

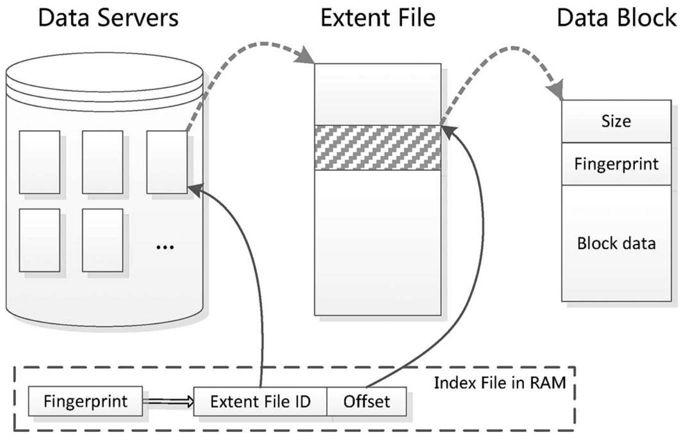


Fig. 2. Process of look-up by fingerprint.

local file system as individual files, and organized in a conventional file system tree. User-level applications are provided to fetch VM meta data files stored on the meta server, or to publish a newly created VM image by pushing its meta data to the meta server.

The client side of Liquid file system could cache portions of file system meta data for fast accesses. For most cases, only meta data of a single VM image or a subdirectory containing VM images will be cached on client side. Liquid follows the common practice in source code management systems, where a client modifies a local copy of portions of the whole project, and publishes the modification to others by committing to a central repository. A Liquid client fetches portions of file system meta data from the meta server, which contains directory listing, file attributes, and set of data block fingerprints needed by the VM image being accessed. After meta data are prepared, a Liquid client then fetches data blocks from data servers and from peer client nodes (see Section 3.5.2 for details). Note that only data blocks not present on local cache will be fetched. When a VM is stopped, modified meta data and data blocks will be pushed back to the meta server and data servers, which ensures that the modification on the VM image is visible to other client nodes.

Locking is provided to ensure the Assumption 1 given in Section 3.1 not being validated. In order to publish new versions of a VM image, a client needs to hold the *edit token* for that VM image. The edit token is transferred to a client node where the VM image is being modified, and returned to the meta server after modifications on the VM image are pushed back. Failure to acquire the edit token indicates that the VM image is currently being used by some VM running on a client node. As stated in Assumption 1, under this situation, the VM image cannot be used by another VM.

3.5 Communication among Components

3.5.1 Heartbeat Protocol

The meta server in Liquid is in charge of managing all data servers. It exchanges a regular heartbeat message with each data server, in order to keep an up to date vision of their health status.

The meta server exchanges heartbeat messages with data servers in a round-robin fashion. This approach will be slow to detect failed data servers when there are many data servers. To speedup failure detection, whenever a data server or client encounters connection problem with another data server, it will send an error signal to the meta server. A dedicated background daemon thread will immediately send a heartbeat message to the problematic data server and determines if it is alive. This mechanism ensures that failures are detected and handled at an early stage. The round-robin approach is still necessary since it could detect failed data servers even if no one is communicating with them.

3.5.2 P2P Data Block Sharing

One advantage of Liquid is its P2P data block sharing scheme. Liquid alleviates burden on data servers by sharing data blocks among all client nodes in a peer-to-peer fashion, eliminating network IO bottlenecks. However, existing peer-to-peer distribution protocol such as BitTorrent [2] will not perform well in this case because of the sheer number of data blocks and corresponding meta data tracking overhead. Moreover, existing protocols do not utilize deduplication info available for VM images. Liquid implements its own peer-to-peer distribution protocol with inspiration from BitTorrent protocol. Each client node or a data server is a valid data block provider, and publishes a Bloom filter [3] where the fingerprints of all their data blocks are compacted into.

A Bloom filter uses an array of m bits, all set to 0 initially, to represent the existence information of n fingerprints. k different hash functions must also be defined, each of which maps a fingerprint to one of the m array positions randomly with a uniform distribution. When adding a new fingerprint into the Bloom filter, the k hash functions are used to map the new fingerprint into k bits in the bit vector, which will be set as 1 to indicate its existence. To query for a fingerprint, we feed it to each of the k hash functions to get k array positions. If any of the bits at these positions is 0, the element is definitely not in the set; otherwise, if all are 1, then either the element is in the set, or the bits have been to 1 by chance during the insertion of other elements. An example is given in Fig. 3, representing the set x, y, z. The olid, dashed, and dotted arrows show the positions in the bit array that each set element is mapped to. The element w is not in the set x, y, z, because it hashes to one bit-array position containing 0.

Bloom filters have *false positives*. According to [3], the probability of false positive is given as

$$\left(1-\left(1-\frac{1}{m}\right)^{kn}\right)^{k} \simeq \left(1-e^{-\frac{kn}{m}}\right)^{k}. \tag{1}$$

For the case of 256 KB block size, 40 GB unique data blocks, 4 hash functions in Bloom filter, and a constraint of less than 1 percent false positive rate, we could calculate that the Bloom filter should contain at least around 1, 724, 000 bits, which is about 210 KB in size. In practice, we choose a Bloom filter size of 256 KB.

Each client maintains connections to a set of peer clients tracked by the meta server, and periodically updates its copy of peer clients' Bloom filters. When fetching a data block by its fingerprint, it checks existence of the fingerprint among peer clients' Bloom filters in a random order, and tries to fetch the data block from a peer if its Bloom filter contains the requested fingerprint. If the data block is not found among peers, the client will go back to fetch the data block from data servers. Checking peers and data servers in a random order eliminates the possibility of

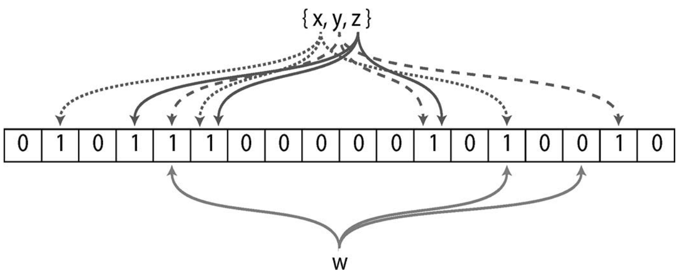


Fig. 3. Example of a Bloom filter [3].

turning them into hot spots, and brings little additional cost because of its simplicity.

3.5.3 On-Demand Data Block Fetching

Liquid uses the copy-on-read technique to bring data blocks from data servers and peer clients to local cache on demand as they are being accessed by a VM. This technique allows booting a VM even if the data blocks in the VM image have not been all fetched into local cache, which brings significant speedup for VM boot up process. Moreover, since only the accessed portion of data blocks are fetched, network bandwidth consumption is kept at a low rate, which is way more efficient than the approach of fetching all data blocks into local cache and then booting the VM.

However, on-demand fetching has a relatively low read performance compared with fully cached VM images. This is because a cache miss will result in an expensive RPC call for fetching the data block being requested. This incurs several times longer IO delay compared with local disk IO. However, this problem will not impair IO performance greatly, since only the first access to such data blocks will affect IO performance. As the frequently used data blocks are fetched, IO performance will return to the normal level.

3.6 Fast Cloning for VM Images

The common practice for creating a new VM is by copying from a template VM image. Most VM images are large, with sizes of several GB. Copying such large images byte-by-byte would be time consuming. Liquid provides an efficient solution to address this problem by means of fast cloning for VM images.

The VM disk images, as seen by Liquid, are represented by a meta data file containing references to data blocks. Simply by copying the meta data file and updating reference counting in data block storage, we could achieve cloning of a VM image. Since the underlying data block storage is a content addressable storage [25], the cloned VM image is by nature a copy-on-write product, which means that modification on the cloned image will not affect the original image. Due to the small sizes of meta data, VM images could be cloned in several milliseconds in the users' view.

3.7 Fault Tolerance

Liquid provides fault tolerance through data replication, data migration, and hot backup of the meta server.

Data blocks are stored in two replicas, in case some data server crashes. When the meta server detects failure of a data server, it immediately contacts other data servers containing replicated data blocks, and sends instructions to create new replicas. Most data blocks have a replication count of more than two, since they also exist on multiple client nodes. Even if all data servers crash, those blocks are still available through the P2P block sharing protocol.

When a data server is planned to be offline, its data blocks could be migrated to other data servers. This is done by simple copying the extent files to other data servers, and merging with existing extent files on destination data servers. Taking a data server offline with planning enables Liquid to handle re-replication work gracefully.

The meta server is a critical component in Liquid. It achieves high availability with a hot backup shadow meta server. Every meta data mutation is performed on both meta servers to ensure a consistent file system view. The shadow meta server exchanges heartbeat messages with the meta server periodically. When the primary meta server is failing, the shadow meta server will take over and operate in read-only mode until an administrator sets up a shadow meta server for it. The other components will discover the failure of the origin meta server, and switch to work with the new meta server.

3.8 Garbage Collection

Liquid uses reference counting extensively to track usage of each data block, and removes unused garbage data block when running out of space.

For client side of Liquid file system, the data block storage contains reference counting of each data block. When a data block's reference count drops to zero, it is considered to be a garbage data block. Garbage data blocks are not immediately removed, since they might be used again some time later. Those garbage data blocks will only be removed when the client cache is nearly full, and the extent file containing data blocks will be compacted to reduce storage consumption.

For data servers, they are not responsible for collecting reference counting of data blocks. The reference counting of all data blocks is maintained by the meta server, and it periodically issues garbage collection requests to data servers. Based on the data server's fingerprint range, the meta server will generate a Bloom filter containing all the valid data blocks inside the range. The Bloom filter is then randomly sent to one of the replicas, and an offline garbage collection is executed based on data block membership in the Bloom filter.

4 Performance Evaluation

4.1 Experiment Environment

To evaluate Liquid's design, we conducted a few experiments on a set of 8 blades connected by 1 GB Ethernet. Each

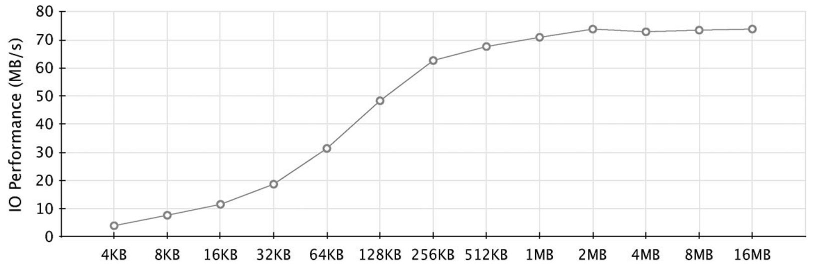


Fig. 4. IO performance under different data block size.

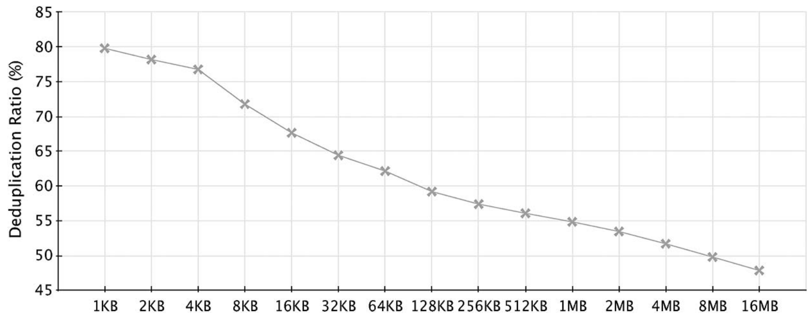


Fig. 5. Deduplication ratio under different data block size.

blade has 4 Xeon X5660 CPUs, 24 GB DDR3 memory, and a 500 GB hard drive (Hitachi HDS721050CLA362). The blades are running Ubuntu 11.04 with Linux kernel version 2.6.38. The benchmarks include PostMark [19], Bonnie++ [11], Linux booting, and Linux kernel compilation.

4.2 Deduplication Block Size Choice

Data block size has a direct impact on disk IO performance and deduplication ratio. A proper choice of data block size needs to balance these two factors in order to achieve the best result.

Fig. 4 provides an analysis of IO performance under different data block sizes. The statistic results are obtained by reading data blocks stored in client side of Liquid file system's local cache. A smaller data block size results in more frequent random access operations, and in turn degrades IO performance. With the increase of data block size, IO performance gradually improves, and stabilizes after it reaches 256 KB.

On the other hand, smaller data block size has the benefit of better redundancy. Fig. 5 presents deduplication ratio under different data block size. The results are obtained by running deduplication on a set of 183 VM images totaling 2. 31 TB. This collection of VM images includes OS of Microsoft Windows, Ubuntu, RedHat, Fedora, CentOS, and openSUSE. The number of each OS image is shown in Table 1. The applications such as Microsoft Office, MATLAB, Apache, Hadoop, MPI, etc. are installed in these OS images randomly. A disk cluster size of 4 KB is used in all VM, so deduplication data block size smaller than 4 KB will not bring more advantages. For data block size larger than 4 KB, the deduplication ratio drops quickly.

Based on our experience, data block size in the range 256 KB $\sim$ 1 MB achieves moderate IO performance and deduplication ratio, satisfying common applications.

4.3 VM Benchmark

In order to illustrate the difference in IO performance under different storage policies, we conducted a few experiments inside VM.

TABLE 1
Number of Each OS Image

OS type	Number
Mircosoft Windows	75
Ubuntu	32
RedHat	22
Fedora	21
CentOS	21
openSUSE	12

The first benchmark is against VM disk IO performance. Bonnie++ is an open-source benchmark suite that is aimed at performing a number of simple tests of hard drive and file system IO performance. PostMark is also a common benchmark for file system developed by NetApp. We ran Bonnie++ and PostMark benchmarks on native disk in host machine, raw format VM image, qcow2 format VM image, and raw format image stored in Liquid with different data block size in range 16 KB $\sim$ 2 MB. The VM images are created with a size of 50 GB, formatted as an ext4 file system. The VM is assigned 2 GB memory and 1 vCPU, running Ubuntu 10.10 with Linux kernel 2.6.35. The results are shown in Figs. 6 and 7.

Native IO on host machine has the best performance. It is the criterion to evaluate other storage policies. The raw image format provides good read and write performance. For qcow2 format images and raw images stored in Liquid, write performance is degraded due to the frequent need to allocate new data block. However, Liquid writes a little faster than qcow2 because of its caching mechanism, even if it is running expensive deduplication process concurrently. Read performance sees weaker impact compared to that of write, because the data being accessed is likely to be cached by OS. A moderate data block size (256 KB, 512 KB) will result in efficient use of cache, while large data block size causes lower cache hit rate, and small data block size results in additional overhead due to frequent random seeks.

In addition to Bonnie++ and PostMark benchmark, we also conducted a set of comprehensive benchmarks based on VM Booting and Linux source code compilation. This benchmark set is both CPU and IO intensive and is representative for most use cases. The Linux kernel source code used in these benchmarks is version 3.0.4. Benchmark results are shown in Fig. 8, with time normalized according to the longest sample.

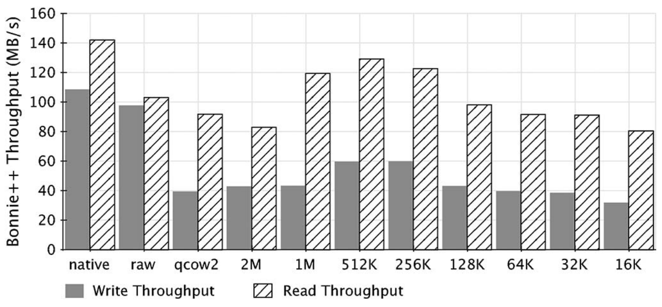


Fig. 6. Bonnie++ benchmark result in VM.

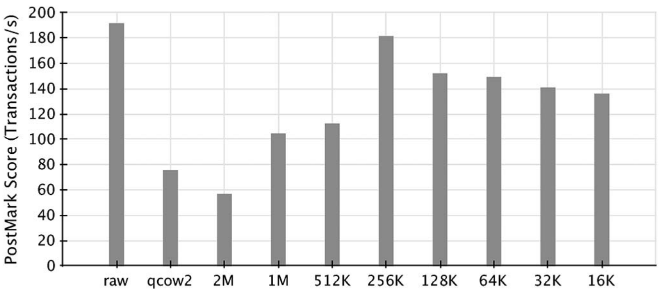


Fig. 7. PostMark benchmark result in VM.

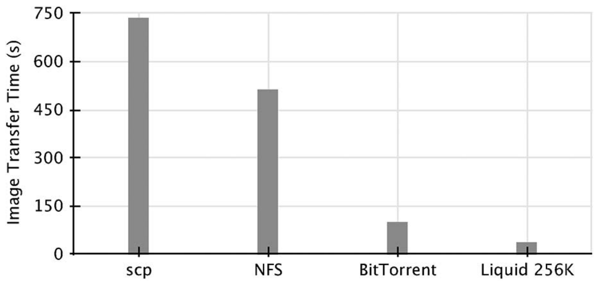


Fig. 9. Time used to transfer a VM image.

For VM booting benchmark, raw image format provides the best result, followed by qcow2 format. Booting a VM stored on Liquid takes longer time, because the cache layer has not gotten enough data blocks to accelerate IO performance. Smaller data blocks result in longer boot phase, because more random access requests will be issued to fetch all the data blocks needed during VM boot phase.

Untaring and compiling Linux kernel source code shows that Liquid has better performance than qcow2 image format. The caching layer inside Liquid which holds modified data blocks contributed to this result. These two benchmarks generate a lot of small files, which in turn create a lot of new data blocks. Liquid caches these data blocks in memory instead of immediately writing to local disk, in case subsequent writings are issued to these data blocks. New data blocks are written to disk in batch when memory cache becomes full. This mechanism avoids frequent IO request to a local disk, and thus guarantees better performance for VM.

4.4 Network Transfer

To evaluate the P2P data block sharing protocol implemented in Liquid, we record the total time used to transfer an 8 GB VM image from one node to other seven nodes. The VM image being transferred is a newly installed Ubuntu 10.10 instance. A moderate data block size of 256 KB is used by Liquid for this benchmark. Results are shown in Fig. 9.

Direct copying by the `scp` utility takes the longest time, with the source node being a hot spot and its network bandwidth saturated. NFS faces the same problem, but has better performance than `scp` due to its implementation optimization. BitTorrent protocol eliminates hot spots by spreading network burden across all nodes, and brings significant speedup compared to `scp` and NFS. However, the redundancy is not utilized by BitTorrent protocol, and all the redundant zero data blocks are all transferred through a network. Liquid client avoids fetching redundant data blocks multiple times, and achieves much better distribution speed than plain BitTorrent protocol.

VM boot benchmark is used to evaluate on-demand fetching, with results shown in Fig. 10. Compared with normal VM booting where all data blocks inside the VM image have already been fetched to local cache, VM booting with on-demand fetching takes several times longer duration. This is caused by the additional overhead to request missing data blocks from other nodes, which incurs a longer delay than local disk IO delay. As data block size decreases, local cache miss increases and leads to more frequent IO through a network, thus results in a longer VM boot phase. However, the whole VM boot time (the downloading image time and the VM boot time) has been shortened while the data block size is between 512 k and 2 M. In fact, the on-demand fetching scheme also speeds up the virtual machine startup speed.

5 Related Work

Currently, there are a lot of papers, which focus on distribute file systems. GFS [16], HDFS [28], and OpenStack [24] provide high availability by replicating stored data into multiple chunk servers. In Liquid, every client is also a replica storing data blocks frequently used. This indicates that Liquid has high fault tolerance capability. Even if all data servers are down, a client would still be able to fetch data block from peer clients. Moreover, compared with these systems, our Liquid has reduced storage consumption by 44 percent at 512 KB data block size, eliminating the

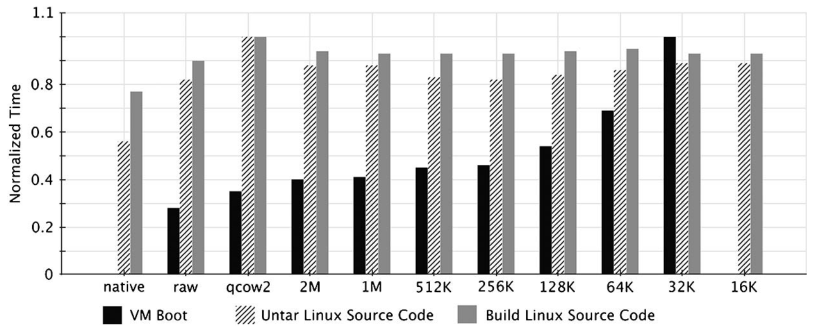


Fig. 8. Linux benchmark result in VM.

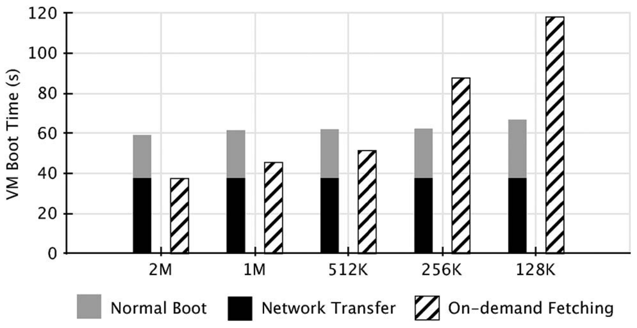


Fig. 10. VM boot time with on-demand fetching.

influence of back-up copy. Meanwhile, the I/O performance loss is just less than 10 percent.

Lustre [27] is a parallel and distributed file system, generally used for cluster computing. The architecture of Lustre file system is similar to Liquid, but there are several important differences. Liquid file system has reduced storage consumption by using deduplication technology, and solved the bottleneck problem of metadata server owing to our P2P data block sharing scheme among all client nodes.

Amazon's Dynamo [13] used DHT [29] to organize its content. Data on the DHT are split into several virtual nodes, and are migrated for load balance in unit of virtual nodes. Liquid follows this approach by splitting data blocks into shards according to their fingerprint, and management of data in unit of shards.

In addition to the common distributed storage systems, HYDRAstor [14] and MAD2 [31] propose effective distributed architectures for deduplication. The former uses distributed hash table to distribute data, and the latter uses Bloom Filter Array as a quick index to quickly identify non-duplicate incoming data. However, both of them focus on scalable secondary storage, which is not suitable for VM images storage. LiveDFS [23] enables deduplication storage of VM images in an open-source cloud; however, it only focuses on deduplication on a single storage partition, which is difficult to handle the problems in distribute systems.

6 Conclusion

We have presented Liquid, which is a deduplication file system with good IO performance and a rich set of features. Liquid provides good IO performance while doing deduplication work in the meantime. This is achieved by caching frequently accessed data blocks in memory cache, and only run deduplication algorithms when it is necessary. By organizing data blocks into large lumps, Liquid avoids additional disk operations incurred by local file system. Liquid supports instant VM image cloning by copy-on-write technique, and provides on-demand fetching through network, which enables fast VM deployment. P2P technique is used to accelerate sharing of data blocks, and makes the system highly scalable. Periodically exchanged Bloom filter of data block fingerprints enables accurate tracking with little network bandwidth consumption.

Deduplication on VM images is proved to be highly effective. However, special care should be taken to achieve high IO performance. For VM images, parts of an image are frequently modified, because the OS and applications in VM are generating temporary files. Caching such blocks will avoid running expensive deduplication algorithms frequently, thus improves IO performance.

Making the system scalable by means of P2P technique is challenging because of the sheer number of data blocks to be tracked. By compacting data block fingerprints into Bloom filters, the management overhead and meta data transferred over network could be greatly reduced.

Acknowledgment

The authors are grateful to three anonymous reviewers for their valuable and constructive comments. This work is supported by National Basic Research (973) Program of China (2011CB302505), Natural Science Foundation of China (60963005, 61170210), National High-Tech R&D (863) Program of China (2012AA012600, 2011AA01A203), Chinese Special Project of Science and Technology (2012ZX01039001). Part of the work was performed while K. Li was visiting Tsinghua National Laboratory for Information Science and Technology, Tsinghua University, in the summer of 2012 and the winter of 2013 as an Intellectual Ventures endowed visiting chair professor.

References

[1] Amazon Machine Image, Sept. 2001. [Online]. Available: http://en.wikipedia.org/wiki/Amazon_Machine_Image
[2] Bittorrent (Protocol), Sept. 2011. [Online]. Available: http://en.wikipedia.org/wiki/BitTorrent_(protocol)
[3] Bloom Filter, Sept. 2011. [Online]. Available: http://en.wikipedia.org/wiki/Bloom_filter
[4] Filesystem in Userspace, Sept. 2011. [Online]. Available: http://fuse.sourceforge.net/
[5] Rabin Fingerprint, Sept. 2011. [Online]. Available: http://en.wikipedia.org/wiki/Rabin_fingerprint
[6] Reiserfs, Sept. 2011. [Online]. Available: http://en.wikipedia.org/wiki/ReiserFS
[7] Xfs: A High-Performance Journaling Filesystem, Sept. 2011. [Online]. Available: http://oss.sgi.com/projects/xfs/
[8] Data Deduplication, Sept. 2013. [Online]. Available: http://en.wikipedia.org/wiki/Data_deduplication
[9] A.V. Aho, P.J. Denning, and J.D. Ullman, "Principles of Optimal Page Replacement," *J. ACM*, vol. 18, no. 1, pp. 80-93, Jan. 1971.
[10] A.T. Clements, I. Ahmad, M. Vilayannur, and J. Li, "Decentralized Deduplication in San Cluster File Systems," in *Proc. Conf. USENIX Annu. Techn. Conf.*, 2009, p. 8, USENIX Association.
[11] R. Coker, *Bonnie++*, 2001. [Online]. Available: http://www.coker.com.au/bonnie++/
[12] D. Eastlake, 3rd, *Us Secure Hash Algorithm 1 (sha1)*, Sept. 2001. [Online]. Available: http://tools.ietf.org/html/rfc3174
[13] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's Highly Available Key-Value Store," in *Proc. 21st ACM SIGOPS SOSP*, New York, NY, USA, 2007, vol. 41, pp. 205-220.
[14] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki, "Hydrastor: A Scalable Secondary Storage," in *Proc. 7th Conf. File Storage Technol.*, 2009, pp. 197-210.
[15] A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, and I. Stoica, "Above the Clouds: A Berkeley View of Cloud Computing," Dept. Elect. Eng. Comput. Sci., Univ. California, Berkeley, CA, USA, Rep. UCB/EECS 28, 2009.
[16] S. Ghemawat, H. Gobioff, and S.T. Leung, "The Google File System," in *Proc. 19th ACM SOSP*, New York, NY, USA, Oct. 2003, vol. 37, pp. 29-43.
[17] K. Jin and E.L. Miller, "The Effectiveness of Deduplication on Virtual Machine Disk Images," in *Proc. SYSTOR, Israeli Exp. Syst. Conf.*, New York, NY, USA, 2009, pp. 1-12.

[18] M. Juric, *Notes: Cuda md5 Hashing Experiments*, May 2008. [Online]. Available: http://majuric.org/software/cudamd5/

[19] J. Katcher, ''Postmark: A New File System Benchmark,'' Network Applicance Inc., Sunnyvale, CA, USA, Technical Report TR3022, Oct. 1997.

[20] A. Liguori and E. Hensbergen, ''Experiences with Content Addressable Storage and Virtual Disks,'' in *Proc. WIOV08*, San Diego, CA, USA, 2008, p. 5.

[21] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, ''The New ext4 Filesystem: CURRENT STATUS and Future Plans,'' in *Proc. Linux Symp.*, 2007, vol. 2, pp. 21-33, Citeseer.

[22] M. McLoughlin, *The qcow2 Image Format*, Sept. 2011. [Online]. Available: http://people.gnome.org/markmc/qcow-image-format.html

[23] C. Ng, M. Ma, T. Wong, P. Lee, and J. Lui, ''Live Deduplication Storage of Virtual Machine Images in an Open-Source Cloud,'' in *Proc. Middleware*, 2011, pp. 81-100.

[24] K. Pepple, *Deploying OpenStack*. Sebastopol, CA, USA: O'Reilly Media, 2011.

[25] S. Quinlan and S. Dorward, ''Venti: A New Approach to Archival Storage,'' in *Proc. FAST Conf. File Storage Technol.*, 2002, vol. 4, p. 7.

[26] R. Rivest, *The md5 Message-Digest Algorithm*, Apr. 1992. [Online]. Available: http://tools.ietf.org/html/rfc1321

[27] P. Schwan, ''Lustre: Building a File System for 1000-Node Clusters,'' in *Proc. Linux Symp.*, 2003, vol. 2003, pp. 400-407.

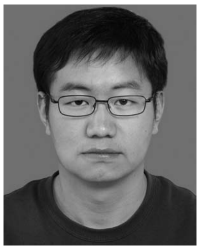
[28] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, ''The Hadoop Distributed File System,'' in *Proc. IEEE 26th Symp. MSST*, 2010, pp. 1-10.

[29] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan, ''Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications,'' in *Proc. Conf. Appl., Technol., Architectures, Protocols Comput. Commun. (SIGCOMM)*, New York, NY, USA, Oct. 2001, vol. 31, pp. 149-160.

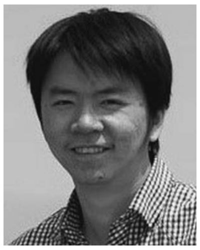
[30] C. Tang, ''Fvd: A High-Performance Virtual Machine Image Format for Cloud,'' in *Proc. USENIX Conf. USENIX Annu. Tech. Conf.*, 2011, p. 18.

[31] J. Wei, H. Jiang, K. Zhou, and D. Feng, ''Mad2: A Scalable High-Throughput Exact Deduplication Approach for Network Backup Services,'' in *Proc. IEEE 26th Symp. MSST*, 2010, pp. 1-14.

[32] B. Zhu, K. Li, and H. Patterson, ''Avoiding the Disk Bottleneck in the Data Domain Deduplication File System,'' in *Proc. 6th USENIX Conf. FAST*, Berkeley, CA, USA, 2008, pp. 269-282.



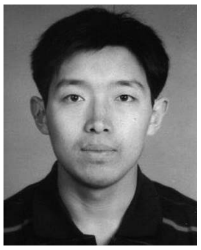
Xun Zhao received the BE degree from Harbin Institute of Technology, Harbin, China, in 2009. He is a PhD candidate in the Department of Computer Science and Technology, Tsinghua University, Beijing, China. His research interests include cloud computing, virtual machine scheduling, and distributed file systems. He has developed an Urgent Scheduler for Xen virtual machine. His current work primarily involves VM scheduling on multicore machines.



Yang Zhang received the MS degree in computer science and technology from Tsinghua University, Beijing, China, in 2012. Currently, he is a PhD student in computer science at New York University, New York. His research interests include distributed systems, virtualization, and cloud computing.

Yongwei Wu received the PhD degree in applied mathematics from the Chinese Academy of Sciences, Beijing, China, in 2002. Currently, he is a Full Professor of computer science and technology at Tsinghua University, Beijing, China. His research interests include distributed processing, virtualization, and cloud computing. Dr. Wu has published over 80 research publications and has received two Best Paper Awards. Currently, he is on the editorial board of the *International Journal of Networked and Distributed Computing* and *Communication of China Computer Federation*. He is a member of the IEEE.

Kang Chen received the PhD degree in computer science and technology from Tsinghua University, Beijing, China, in 2004. Currently, he is an Associate Professor of computer science and technology at Tsinghua University. His research interests include parallel computing, distributed processing, and cloud computing.



Jinlei Jiang received the PhD degree in computer science and technology from Tsinghua University, Beijing, China, in 2004. Currently, he is an Associate Professor of computer science and technology at Tsinghua University. His research interests include distributed processing and cloud computing. Dr. Jiang is a member of the IEEE.

Keqin Li is a SUNY Distinguished Professor of computer science and an Intellectual Ventures endowed Visiting Chair Professor at Tsinghua University, Beijing, China. His research interests are mainly in design and analysis of algorithms, parallel and distributed computing, and computer networking. Dr. Li has published over 290 research publications and has received several Best Paper Awards for his highest quality work. He is currently or has served on the editorial board of *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Computers*, *Journal of Parallel and Distributed Computing*, *International Journal of Parallel, Emergent, and Distributed Systems*, *International Journal of High Performance Computing and Networking*, and *Optimization Letters*. He is a Senior Member of the IEEE.