Chinese Journal of Electronics
vol. 33, no. 1, pp. 313–322, January 2024
https://doi.org/10.23919/cje.2022.00.405

RESEARCH ARTICLE

Troy: Efficient Service Deployment for Windows Systems

Deyu ZHANG, Yu XIE, Mucong XU, En CHENG, Xiaoyan KUI, Bangwen HE, and Yunhao LI

School of Computer Science and Engineering, Central South University, Changsha, 410083, China

Corresponding author: Xiaoyan KUI, Email: xykui@csu.edu.cn

Manuscript Received November 25, 2022; Accepted March 20, 2023

Copyright © 2024 Chinese Institute of Electronics

Abstract — The modern university computer lab and kindergarden through 12th grade classrooms require a centralized solution to efficiently manage a large number of desktops. The existing solutions either bring virtualization overhead in runtime or requires loading a large image over 30 GB leading to an unacceptable network latency. In this work, we propose Troy which takes advantage of the differencing virtual hard disk techniques in Windows systems. As such, Troy only loads the modifications made on one machine to all other machines. Troy consists of two modules that are responsible to generate an initial image and merge a differencing image with its parent image, respectively. Specifically, we identify the key fields in the virtual hard disk image that links the differencing image and the parent image and find the modified blocks in the differencing images that should be used to replace the blocks in the parent image. We further design a lazy copy solution to reduce the I/O burden in image merging. We have implemented Troy on bare metal machines. The evaluation results show that the performance of Troy is comparable to the native implementation in Windows, without requiring the Windows environment.

Keywords — Service deployment, Virtual hard disk, File system merging, Windows system.

Citation — Deyu ZHANG, Yu XIE, Mucong XU, *et al.*, "Troy: Efficient Service Deployment for Windows Systems," *Chinese Journal of Electronics*, vol. 33, no. 1, pp. 313–322, 2024. doi: 10.23919/cje.2022.00.405.

I. Introduction

Efficient service deployment serves as the key to enabling easy system management on a large number of homogeneous desktops. This is especially important in a scenario where one administrator manages all desktops, such as kindergarden through 12th grade classrooms and university computing labs [1]. It calls for a centralized management solution, such that one operation on a device can be automatically applied to other devices. The operations include changing system settings, software installations, and uninstallations, etc. [2]–[4].

Currently, the solutions for centralized management of large-scale desktops are mainly cloud-based: 1) Virtual desktop infrastructure (VDI). It runs all user desktops on a centralized server. In this case, instead of deploying services on individual PCs, all users connect to a virtual machine hosted on the server [5]. 2) Virtual operating system infrastructure (VOI). It works in a diskless mode. The cloud server stores the image of a whole operating system and software. The server sends the image to the desktops upon request. Since all desktops use the same image, the deployment of new services can be realized by updating the image on the server. 3) Intelligent desktop virtualization (IDV) [6]. The server provision the virtual machines. The virtual desktops run on high-performance desktops in the same physical location as users. Given that VDI relies on the connectivity to the server to function, IDV requires server connectivity for only the initial setup and image loading. In such a sense, it can be deemed as an updated version of VDI. The existing VDI and IDV solutions use a virtual machine to support the deployment of new services. It brings the virtualization overhead in runtime, lowering the user experience. More importantly, they can hardly support virtualizing a specific hardware, such as graphic processing unit (GPU), due to the limitation of nested virtual machine (VM) [7]. Although VOI brings native performance, the loading of a large image, typically over 30 GB, leads to unacceptable loading latency limited by the network bandwidth.

Associate Editor: Prof. Chao SHEN, Xi'an Jiaotong University.

To obtain efficient service deployment for a large number of homogeneous devices, we expect native runtime performance while avoiding loading the over-large image. In such a way, the operations on one desktop can be applied to other ones by renewing the storage. The overlay file system provides a viable solution toward this goal. Existing works include BAOverlay and DADI. BAOverlay [8] exploits an asynchronous copy-on-write (CoW) mechanism for fast file updates and designs a new file format to compress the storage space. In DADI [9], the authors use a block-based layer where each layer corresponds to a set of file changes, which allows the image to be file system and platform agnostic. However, they are mainly designed for cloud servers, which only work for Linux systems. For desktop and laptop computers, Windows OSs are still mainstream which takes over 74% market share. Although Windows supports the virtual disk format virtual hard disk v2 (VHDX) to enjoy the overlay file-system functionality. The generation and merging of differencing images can only be achieved by a command Diskpart that demands a Windows environment. It heavily limits the scope of use of VHDX in service deployment, i.e., it requires another Windows system for image management. Even the smallest Windows PE system takes over 200 MB of disk space.

In this work, we design two techniques in Troy to efficiently generate and merge differencing VHDX images without relying on a Windows environment. For creation, Troy generates appropriate attributes, i.e., metadata, and writes them into a new differencing image. For merging, Troy loads and parses the necessary metadata and calculates the correspondence between the differencing and parent images. According to the correspondence, it reads the payload blocks or sectors and writes them to the appropriate location in the parent image. As such, the differencing image generated on the administrator can be applied to all other users, as shown in Figure 1.

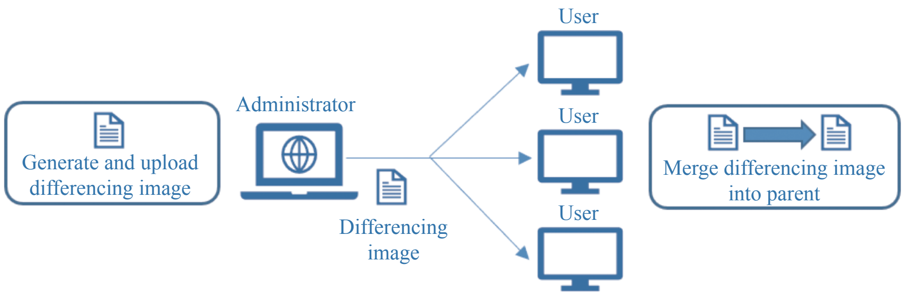

Generate and upload differencing image
Administrator
Differencing image
User
User
User
Merge differencing image into parent

Figure 1 System architecture.

The contributions of this work are three-fold: 1) In-depth analysis on the rules for image generation and merging; 2) Propose the image generation and merging techniques to enable efficient service deployment; 3) Implement Troy and demonstrate that it achieves state-of-the-art performance on different hardware settings, including both hard disk drives (HDD) and solid state drives (SSD).

The rest of the paper is organized as follows. Section II reviews the related works in the efficient service deployment. Section III provides the background about the VHDX file format, especially the differencing images. Section IV and V give the design and implementation of Troy, respectively. In Section VI, we evaluate the performance of Troy on different hardware settings, in comparison to the native implementation in the Windows system. Section VII concludes the work.

II. Related Works

Numerous works study the overlay file system-based and virtual disk-based service deployment and migration [10]–[14]. To lower the maintenance cost in data centers, Oliveier *et al.* [10] proposed migrating the high-performance computing workloads from the x86 server machine to a few connected ARM embedded boards. To this end, they find the migration point to ensure a state of equivalence and translate the state between ISAs. Furthermore, they design a semantic migration scheme to allow the guest OS to extract the entire application state from the x86 server and a minimal architecture-independent subset of the kernel state. To enable live migration of containers, Ma *et al.* [11] synchronize the data in both storage and run-time memory to find the identical layers in the file system and reduce the memory image size, respectively. They also design parallel and pipelined processing to improve the process efficiency. Gotanda *et al.* [12] design a lazy layer pull scheme to reduce the start-up of a new or updated container. They modify the existing overlay file system to trace file accesses during container startup, such that the lazy pull of layers is allowed. Selfie [13] finds that the efficiency of virtual disks is largely compromised by random writing and keeping the data consistent. To address this issue, Selfie refines the format for storing data and metadata. It translates the block address to map a virtual disk space to the data space. Wu *et al.* [14] proposed a virtual computing and storage approach EVCS to execute the software for near-to-native performance based on an incremental virtual storage mechanism. Given that the works [10]–[13] are viable for efficient service deployment, they are designed for Linux-based containers, while for desktops, Window OSs are the mainstream. Although [14] claims that EVCS can be used for Windows systems, it does not propose any technical detail regarding image management, which is the key to efficient service deployment.

There are extensive works that optimize the resources scheduling and allocation in service deployment [15]–[22]. Wang *et al.* [15] design both offline and online optimal microservice coordination algorithms to reduce the overall service delay. The offline algorithm achieves the globally optimal performance but requires prior information such as computation request arrival and time-varying channel conditions. For online scheduling, they reformulate the problem using the markov decision process and solve it by reinforcement learning to achieve near-optimal performance. Akahoshi *et al.* [16] consider the deployment of virtual network services. They design an algorithm to minimize the deployment cost with VM merging and priority queuing. The cost includes both fixed and proportional costs. Hazra *et al.* [17] share the computing resources at edges to maximize the revenue of edge service providers and minimize the service delay and price. They first formulate a mixed integer non-linear programming problem, then transform the objective function into a Stackelberg game problem to coordinate the competition between servers. Xiang *et al.* [18] partition the applications into pieces and model the dependency between the pieces by a directed acyclic graph. Based on the model, they translate the energy consumption and application performance into a multi-objective optimization problem, which is solved by a heuristic algorithm. In [19], Bozorgchenani *et al.* exploit different service deployment models, such as SaaS, PaaS, and IaaS, among edge servers to serve end users. They model the problem as a size-constrained weighted set cover problem aiming at maximizing the number of satisfied end users while minimizing the service completion time. These works are orthogonal to the design in Troy in the sense that they can be used to optimize the performance based on image generation and merging in Troy.

III. Background

We design the efficient service deployment scheme designed based on the virtual disk format VHDX, which supports the Deployment of Windows OSs. Especially, it supports the differencing VHDX image type storing the blocks that have been modified compared to a parent virtual hard disk image. In this section, we introduce the mechanism of differencing VHDX in runtime and the layout of a VHDX image.

1. Differencing VHDX image

In case differencing VHDX is used in runtime, any modification to the file system, *i.e.,* a new write, is captured in the differencing VHDX image instead of its parent VHDX image through CoW. In CoW, if a unit of data is copied but not modified, the copy operation can be implemented by setting a reference to the original data. The copy is created only when the copied data is modified. By using CoW in the runtime of Windows systems based on differencing VHDXs image, it creates point-in-time snapshots of virtual disks for service deployment and backups. For a read operation to the virtual disk, the Windows system first check if the content can be found in the differencing VHDX image. If not, the read operation will be traversed toward the parent VHDX image. As such, a differencing VHDX image cannot work independently. It has to link to the parent VHDX image which stores the whole payload. The upper bound of a differencing image in size is that of its parent VHDX image. In case a parent image has a size 10 GB, the size of a differencing image created from the parent image will be constrained at 10 GB. It helps to determine the configuration of the hard disks.

The parent VHDX image has two types, *i.e.,* the fixed and dynamic types. The size of the VHDX image in fixed type does not change during runtime. For the dynamic type, as more write operations, the image file increases in size by allocating more space. To ease the configuration of the hard disk, we consider only the fixed-type VHDX image in this work.

2. Layout of a VHDX image

The layout of a VHDX image is shown in Figure 2. Overall, a VHDX image consists of two kinds of data, *i.e.,* the metadata, and payload. The metadata stores the features that are necessary to manage a VHDX file and the properties of the payload, while the payload stores the data of the file systems. Since we focus on the creation and merging of VHDX images, the core issue is to generate and modify the metadata according to the specification of VHDX images. In the following, we introduce the functionality of the important metadata for image creation and merging, respectively.

Header is the first region that is examined when loading a VHDX image. It serves as a root of the VHDX data structure tree. It consists of the sub-header and region table. The sub-header consists of the unique identifiers that identify if the image has been modified, including both data writing and file opening. The region table lists the location of the block allocation table (BAT) and the metadata region. As such, it helps the system find the metadata of interests.

BAT translates the sequence number of a block to the offset in the virtual hard disk file. It consists of entries that determine the state and file offset of blocks. There are two kinds of blocks: sector bitmap blocks and payload blocks. They are aligned in an interleaved manner in the image. The sector bitmap block is the metadata for the payload blocks, in the way that it identifies whether the corresponding payload block has been modified.

The metadata region contains the metadata items such as virtual disk size, logical and physical sector ID, etc. The Log contains the updates of all metadata except the headers. These two regions will not be modified for image generation and merging.

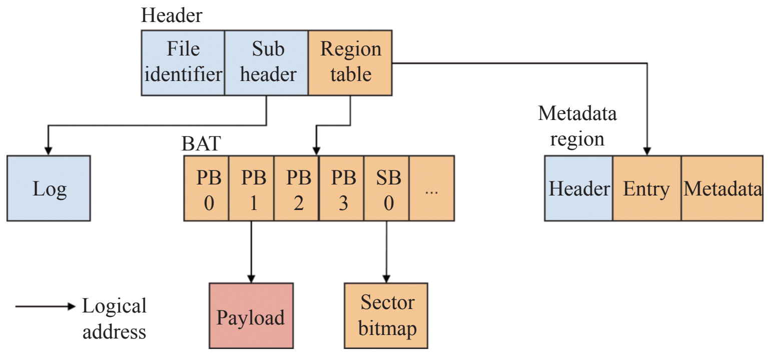


Figure 2 VHDX logical layout.

IV. Design of Troy

To enable efficient and automated service deployment, we design two modules in Troy, i.e., initial image generation module and differencing image merging module. The generation module generates an initial image that is linked to the parent image. To this end, we parse and analyze the initial image generated by diskpart to find out the structure and semantics of each field in the VHDX image. In runtime, all the modifications that are made in the parent image are saved in the initial image through CoW, turning the initial image into the differencing image. Then the merging module merges a differencing image with its parent image. The key is to find out which blocks are modified during runtime, then replace the corresponding blocks in the parent image. We find that the sector bitmap blocks record whether the payload blocks are modified during runtime, and one sector bitmap block is responsible for multiple payload blocks. To this end, we analyze the relationships between the sector bitmap blocks and the payload blocks. Furthermore, the intensive data reading and writing in image merging leads to a heavy I/O burden, which constitutes the performance bottleneck in terms of latency. To alleviate the I/O burden, we design a lazy block writing scheme that groups the block write operations into one I/O, to reduce the number of I/O operations. We give the design of the two modules in detail as below.

1. Initial image generation

This module creates an initial image that is filled with appropriate metadata to make it a differencing image for a specific parent image. To this end, it creates the required metadata including Header, Log, BAT, and Metadata region. There are two goals in generating an initial image. The first one is to minimize its size. The second one is to link the initial image to the specific parent image, such that it can serve as a differencing image. Toward the two goals, we develop three rules for initial image generation.

Rule 1: The size of a block should be carefully set to minimize the size of the initial image By analyzing the documentation of VHDX, we find that the metadata has to be aligned at a granularity of MB. For example, if the size of BAT is 2.4 MB, it should be assigned a 3 MB block in the storage. Since the metadata consists of four domains, i.e., the Header, Log, BAT, and Metadata region, we require at least 4 MB for the initial image. To constrain the size of the BAT to 1MB, we determine the size of each block by $b_s = 2^t$ bytes, where t is equal to:

$$t = \max(\lceil \log_2(i_s/B) \rceil, 20) \tag{1}$$

where B denotes the number of entries in the BAT. We use the ceiling operator to align the blocks at a granularity of MB.

Rule 2: The BAT and log domains can be set to zeros Since the initial image does not consist of any payload and operating history, we initialize the BAT and log domains to all zeros.

Rule 3: Fill in the corresponding fields in the gion domain to link the initial image to the parent image To enable the linkage, we run multiple experiments to find out the key fields. The fields can be divided into two categories. In the first category, the fields in the initial image and parent image should be the same, including virtual disk size, virtual disk id, logical sector size, and physical sector size. In the second category, the fields indicate the relative path of the parent image and the check code between the two images. For the relative path, the Relative_path value stores the relative path from the initial image to its parent image. For the check code, the parent_linkage value in the parent locator field must be equal to the value of the DataWriteGuid field in the Header of the parent image. The check code of the parent image changes upon modification. As such, the differencing image cannot be linked to a parent image that changes.

2. Differencing image merging

The differencing images need to be transfered to dozens or hundreds of user terminals through the network, so large images will occupy a lot of network bandwidth. We use compression and prune methods to reduce the size of the differencing image.

We use LZ4 compression algorithm to compress the differencing image. LZ4 is one of the fastest compression algorithms and has a high compression ratio. It can greatly reduce the size of the differencing image with limited overhead on decompression.

We also prune the differencing image according to its characteristics. We find that the payload block of the differencing image is identical to the corresponding part of the parent image by comparison. Such that prune the payload block to reduce the size of the image.

During runtime, the windows system saves the modifications on the parent image into the initial image, bringing up the differencing image. By merging the differencing image into the parent image, we can apply all the modifications to other systems without manual intervention. To this end, we analyze the structure of the differencing image in-depth. We show the layout of BAT in Figure 3. Importantly, the sector bitmaps serve as metadata for the payload blocks. Specifically, each bit in the sector bitmaps identifies whether a sector in the payload blocks has been modified in runtime. The payload blocks and sector bitmaps are aligned in an interleaved way in the image. Each block has two parts. For a payload block, the first part identify whether it has been modified, while for a sector bitmap, it identifies if any payload blocks it corresponds to has been modified. The second part in both kinds of blocks gives the offset used by the system to locate the block in the storage.

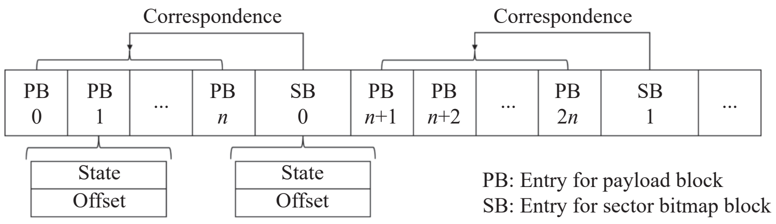


Figure 3 BAT Example.

In the following, we first identify the correspondence between the sector bitmaps and payload blocks. Then check the modification of payload blocks at the granularity of a sector, and matches the blocks in the differencing image to the parent image. At last, We replace the corresponding content in the parent image with the modified content in the differencing image. Since the replacement incurs an intensive I/O burden, we integrate the I/O of sectors into a large chunk, to reduce the I/O times. In the following, we detail the design of the differencing image merging in a step-by-step way.

Step 1 Determine the correspondences between sector bitmaps and the payload blocks in the BAT of the differencing image, according to the block size and sector size.

$$c\_chunk\_ratio = 2^{23} \times S/B \tag{2}$$

where c_chunk_ratio is the proportion of payload blocks and sector bitmap blocks of the differencing image. S denotes the sector size of the differencing image. B denotes the payload block size of the differencing image.

Step 2 Locate the sectors that have been modified in all the payload blocks.

The sector bitmap block consists of two fields, i.e., the state field and the offset field. The State field indicates whether the corresponding PBs have been modified. It has three possible values, i.e., Not_Present, Partially_Present, and Present. Not_Present state indicates that this sector bitmap block's contents are undefined and that the block is not allocated in the image. Partially_Present state indicates that partial blocks are modified. The Present state indicates that the sector bitmap block contents have been modified as a whole.

For blocks in Partially_Present state, we need to determine the modified blocks from the differencing image according to the corresponding sector bitmap blocks. In the differencing image, the size of a sector is usually 512 bytes. If a payload block's size is 32 MB, then it consists of 216 sectors. For each sector, there is a corresponding bit in the corresponding sector bitmap block. If the bit is 1, the sector should be read from the differencing image. If the bit is 0, the sector should be read from the parent base fixed image.

Step 3 Find the matching between the payload blocks in the differencing image and parent image, and their locations.

There is a correspondence between the payload blocks of the differencing image and the payload blocks of the parent image, determined in the BAT. There is a sequence correspondence between the payload block entries in their block allocation tables. For example, for a parent image with a block size of 16MB and a differencing image based on it with a block size of 4MB, there is a one-to-four relationship. That is, payload block 0 of the block allocation table of the basic fixed image corresponds to payload blocks 0, 1, 2, and 3 of the block allocation table of the differencing image. The corresponding relationship is shown in Figure 4.

Step 4 Replace the original sectors in the parent image by the modified sectors in the differencing image, while minimizing the I/O burden.

Troy reads the modified payload block or sectors from the differencing image, then write it to the corresponding location of the parent image. For partially modified payload blocks, we can write each modified sector to the corresponding location of the parent image one by one. However, it results in a large number of write operations. We combine the writes of the continuous modified sectors into one write operation, thus reducing

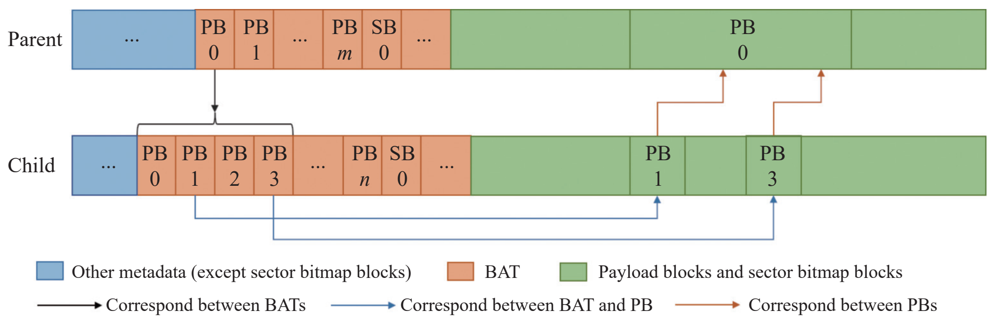

Parent
Child
...
PB 0
PB 1
PB m
SB 0
PB 2
PB 3
PB n
Other metadata (except sector bitmap blocks)
BAT
Payload blocks and sector bitmap blocks
Correspond between BATs
Correspond between BAT and PB
Correspond between PBs

Figure 4 Block correspondence between the parent image and the differencing image.

the number of write operations.

Further, a similar optimization can be adopted for the fully modified payload block. For a continuous fully modified payload, we can write each modified payload block to the corresponding location of the parent image one by one, resulting in a large number of write operations. In the parent image, the positions of the continuous payload in the BAT are also continuous in the physical disk, so there is a condition for completing a write operation across multiple payload blocks. We can combine the writes of continuously modified payloads in the child image into one write operation to reduce the number of write operations.

V. Implementation

We implement Troy in Linux with 4523 lines of C code. We use the memalign, malloc, free, memset and memcpy functions in C standard library to manage memory access. We use the bswapseriesmacro in GCC to convert byte order according to the byte order of CPU. We use the implementation of the CRC32 algorithm in C to calculate the checksum.

We use the glib library function UTF8_to_UTF16 to provide a UTF16 encoding function. We use the Glib library function random_int to provide the UUID generation function. We use open, lseek, read, and write of the Linux system calls to provide a file access function. We use the strcmp, strlen, memcmp, strdup functions in C standard library to manage strings and validate the integrity and valid relationship of differencing images.

VI. Evaluations

In this section, we evaluate the performance of the key components of Troy. We evaluate Troy on two bare metal machines. One is equipped with AMD Ryzen 7 5800H 3.20 GHz CPU, WDC PC SN730 SSD, and 16 GB of memory; The other is equipped with Intel Core i7 7700HQ 2.8 GHz CPU, HGST HTS721010A9E630 HDD, and 16 GB memory.

Resource Adjustment On the machine with an SSD disk, we use the Cgroup mechanism of Linux to adjust and limit system resources. Linux Cgroups provide the ability to control and count the resources of a group of processes, including CPU, memory, storage, network, etc. Cgroups can easily limit the resource occupation of a process and can monitor the monitoring and statistical information of the process in real-time. We use Cgroup to limit the CPU performance, available memory size, and hard disk I/O speed to assist our evaluation.

Parent Image and Differencing Image We use 2 parent images. Both of them are with Windows 10 system. The first is used in generation evaluation, with a size of 40 GB. The second is used in merge evaluation, with a size of 30 GB. We use three differencing images with different sizes to evaluate the merging module. They are generated in two ways. The first is installing softwares. The second is changing system settings. The details are shown in Table 1.

Table 1 Differencing image generation

Image size	Operations
2.57 GB	Install WeChat; Change user group config
4.93 GB	Install QQ, TencentVideo, Feishu; Change security config
8.76 GB	Install IQIYI, bilibili, Epic, Vmware Workstation

Baseline The baseline for us to compare Troy is Diskpart, which is a management tool that manages the physical disk and virtual disk in Windows systems.

Metrics We evaluate the performance of Troy in terms of image size, and merging latency. For initial image generation, we use image size as the metric. For image merge, we use merging latency in seconds as the metric.

Figure 5 shows the impact of different block size settings on the initial size of the child image when the size of the parent image is 40 GB. When the block size is set to a small value, the size of the generated initial image increases. Troy sets the block size of the child image according to the image size of the parent image so that the size of the generated child image is always 4 MB.

1. Merge function evaluation

We then evaluate the performance of the merge function of Troy. We evaluate the impact of various performance factors on the performance of image merging in Troy. The parent size is 30 GB.

1) Disk type

We evaluate the latency required for Troy and Diskpart to merge differencing images of different sizes in

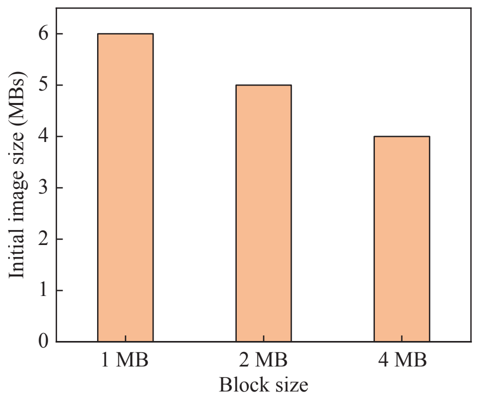


Figure 5 Initial image size for different block size.

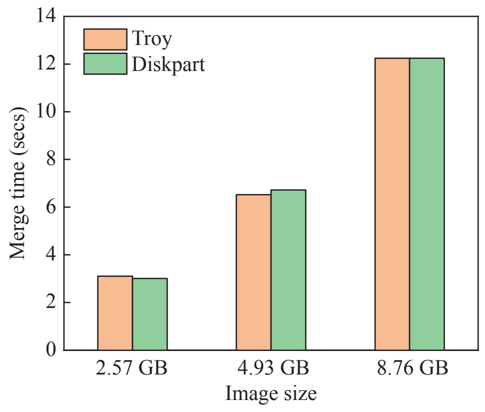


Figure 6 Merging latency on the bare machine with SSD.

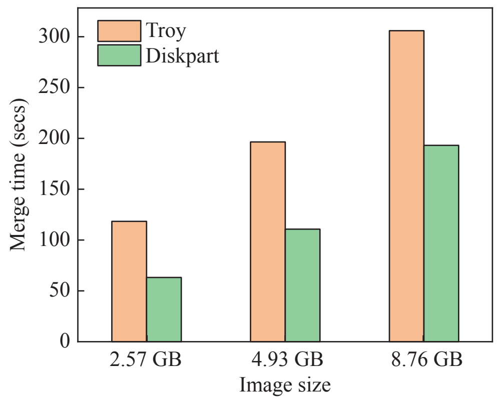


Figure 7 Merging latency on the bare machine with HDD.

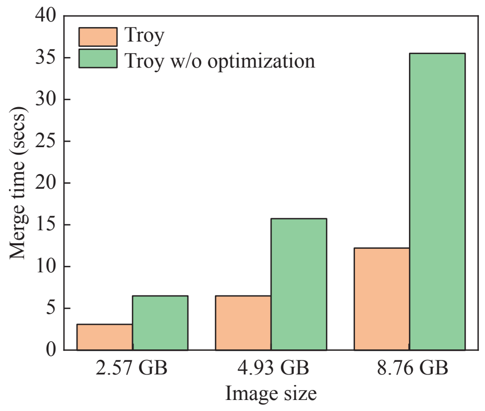


Figure 8 Merging latency with or w/o Optimization on the bare machine with SSD.

bare metal machines configured with 16 GB memory and SSD and bare metal machines configured with 16GB memory and HDD, respectively. The results are shown in Figures 6 and 7.

According to the experimental results, it can be reasonably inferred that the I/O speed of the hard disk is the most important factor affecting the merging performance of Troy. At the same time, in the bare metal machine configured with SSD hard disk, Troy achieves the same performance as Diskpart, but on the bare metal machine configured with HDD, Troy is about half slower than Diskpart. The implementation of the merge function of Troy does not fully utilize the I/O performance of the HDD.

2) I/O operation optimization

To verify the impact of our I/O operation optimization on the performance of the image merging module, we evaluate the performance with and without I/O operation optimization on bare metal machines with SSD and HDD, respectively. The results are shown in Figures 8 and 9.

The results show that the lazy I/O can effectively reduce the latency. On bare metal machines with SSD, the lazy I/O achieves 2 to 3 times performance boost. On the bare metal machine with HDD, the performance boost achieves 5 times. For HDD, the latency of a disk I/O operation consists of the non-transfer time (seek time and rotation delay) and the transfer time. If the number of I/Os increases, even if the total amount of data transfer remains the same, the increase of non-transfer time will also cause the total latency to increase. For SSD, every I/O operation needs to be processed by the controller. If the number of I/O operations increases, the increase in controller processing time will also lead to an increase in the total latency. The lazy I/O optimization can reduce the number of I/O operations, thus optimizing the performance of the image merging module.

3) Compression and prune methods

To verify the impact of our compression and prune methods on the size of the differencing image, we evaluate the size of the differencing image with and without Compression and prune methods. The results are shown in Figure 10. The results show that the LZ4 compression algorithm can effectively reduce the size of the image. The compression ratio achieves about 50% or more. The effect of prune method is relatively less. We suppose that the prune method is better for images in which users have done uninstall and remove operation.

4) I/O speed

To further verify the impact of I/O speed on the performance of the merge function, we use the Cgroup mechanism of Linux on the bare metal machine with SSD configuration to limit the hard disk I/O speed and test the performance at different I/O speeds. The results

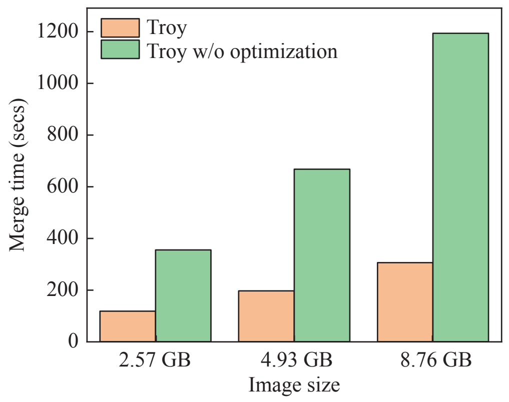


Figure 9 Merging latency with or w/o Optimization on the bare machine with HDD.

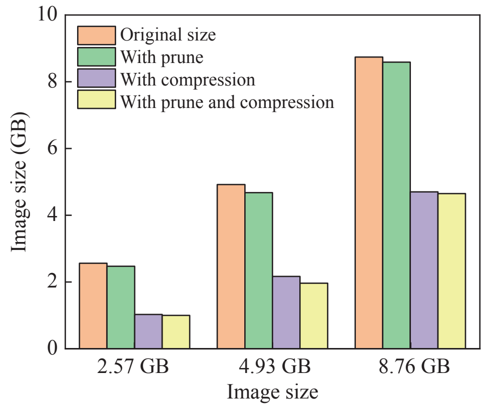


Figure 10 Image size with prune and compression.

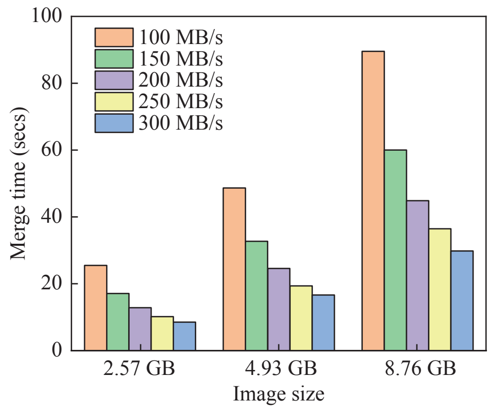


Figure 11 Merging time on bare machine with different I/O speed.

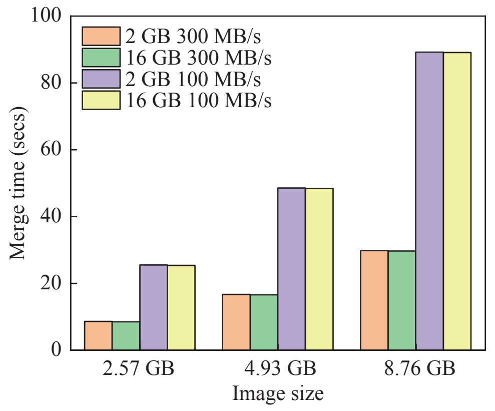


Figure 12 Merging time with different memory size and limited I/O speed.

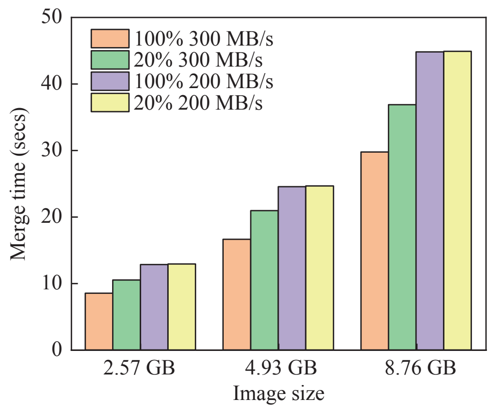


Figure 13 Merging time with different CPU performance and limited I/O speed.

are shown in Figure 11. We can see that the faster the I/O speed is, the shorter the merging latency is, and the faster the merging speed is, the merging speed is proportional to the hard disk I/O speed.

5) Memory size

We evaluate the performance impact of different memory sizes on the merge function of Troy on the bare metal machine configured with SSD. We limit the memory to 4 GB and 16 GB based on the I/O speed of 100 MB/s and 300 MB/s. The results are shown in Figure 12. It can be seen that memory size has little impact on the performance of Troy and is not the main performance factor.

6) CPU performance

We evaluate the performance impact of different CPU performances on the merge function of Troy on the bare metal machine configured with SSD. Based on 100 MB/s and 300 MB/s I/O speed, we limit the CPU to use 100% performance and only 20% performance. The results are shown in Figure 13.

The experimental results show that when the I/O speed is 100 MB/s, the change in CPU performance does not affect the merging speed. When the I/O speed is 300 MB/s, the change in CPU performance has a significant impact on the merging speed. The reason is that when the I/O speed is slow, the CPU needs to issue and process fewer I/O requests per unit time, while when the I/O speed is fast, the CPU needs to issue and process more I/O requests per unit time.

7) Summary

According to the above experimental results and evaluation, the main factor affecting the performance of the merge function of Troy is I/O speed, which is I/O intensive. When the I/O speed is slow, the CPU performance will have a small impact, while when the I/O speed is fast, the CPU performance will have a significant impact. Memory size has little impact on performance. For end users, latency is one of the important indicators, so better hard disks and CPUs can bring a better user experience.

VII. Conclusion

We propose Troy to enable efficient and automated service deployment for Windows systems. Troy enables the administrator to automatically deploy services on a large number of machines. Troy consists of two modules, i.e., the image generation module and the image merging module. In the image generation module, Troy generates an empty differencing image that is minimized in size to link to a specific parent image. In the image merging module, we analyze the correspondence between the sector bitmap blocks and the payload blocks to locate the sectors that have been modified in a runtime. Furthermore, we find the matching between the payload blocks in the differencing and parent image, such that replace the blocks in the parent image with the modified blocks in the differencing image. We thoroughly evaluate the performance of Troy in terms of latency. The results show that Troy achieves comparable merging latency in comparison to the native implementation in Windows systems.

Acknowledgement

This work was supported in part by the National Key R&D Program of China (Grant No. 2022YFF0604504), the National Science Foundation of China (Grant Nos. 62172439, U22A2034, and 62177047), the Major Project of Natural Science Foundation of Hunan Province (Grant No. 2021JC0004), the National Natural Science Foundation of Hunan Province (Grant No. 2023JJ20076), the Central South University Research Programme of Advanced Interdisciplinary Studies (Grant No. 2023QYJC020), and the Central South University Innovation-Driven Research Programme (Grant No. 2023CXQD061).

References

[1] Y. J. Tang and X. X. Ding, "Application research of desktop virtualization technology based on VOI in computer room management of colleges and universities," *Journal of Physics: Conference Series*, vol. 1345, article no. 062055, 2019.

[2] X. Q. Shi, "The construction of language laboratory based on VOI technology," *Journal of Physics: Conference Series*, vol. 1952, article no. 042030, 2021.

[3] D. Y. Zhang, L. Tan, J. Ren, *et al.*, "Near-optimal and truthful online auction for computation offloading in green edge-computing systems," *IEEE Transactions on Mobile Computing*, vol. 19, no. 4, pp. 880–893, 2020.

[4] J. R. Zhang, H. Yang, J. Ren, *et al.*, "MobiDepth: Real-time depth estimation using on-device dual cameras," in *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking*, Sydney, Australia, pp. 528–541, 2022.

[5] C. T. Yang, J. C. Liu, J. Y. Lee, *et al.*, "The implementation of a virtual desktop infrastructure with GPU accelerated on OpenStack," in *2018 15th International Symposium on Pervasive Systems, Algorithms and Networks*, Yichang, China, pp. 366–370, 2018.

[6] H. Xia, "Research and Application of Cloud Computing and Big Data Technology in Intelligent Desktop Virtualization System," *2022 IEEE 2nd International Conference on Data Science and Computer Application (ICDSCA)*, Dalian, China, pp. 517-521, 2022

[7] J. T. Lim and J. Nieh, "Optimizing nested virtualization performance using direct virtual hardware," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, Lausanne, Switzerland, pp. 557–574, 2020.

[8] Y. Sun, J. X. Lei, S. Shin, *et al.*, "Baoverlay: A block-accessible overlay file system for fast and efficient container storage," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, Virtual Event, New York, USA, pp. 90–104, 2020.

[9] H. B. Li, Y. F. Yuan, R. Du, *et al.*, "DADI: Block-Level image service for agile and elastic application deployment," in *Proceedings of 2020 USENIX Annual Technical Conference*, pp. 727–740, 2020.

[10] P. Olivier, A. K. M. F. Mehrab, S. Lankes, *et al.*, "HEXO: Offloading HPC compute-intensive workloads on low-cost, low-power embedded systems," in *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, Phoenix, AZ, USA, pp. 85–96, 2019.

[11] L. L. Ma, S. H. Yi, N. Carter, *et al.*, "Efficient live migration of edge services leveraging container layered storage," *IEEE Transactions on Mobile Computing*, vol. 18, no. 9, pp. 2020–2033, 2019.

[12] S. Gotanda and T. Shinagawa, "Short paper: Highly compatible fast container startup with lazy layer pull," in *Proceedings of 2021 IEEE International Conference on Cloud Engineering (IC2E)*, San Francisco, CA, USA, pp. 53–59, 2021.

[13] X. B. Wu, Z. L. Shao, and S. Jiang, "*Selfie*: Co-locating metadata and data to enable fast virtual block devices," in *Proceedings of the 8th ACM International Systems and Storage Conference*, Haifa, Israel, article no.2, 2015.

[14] M. Wu, L. Zhou, and F. J. Huang, "EVCS: An edge-assisted virtual computing and storage approach for heterogeneous desktop deployment," in *Proceedings of the 2022 IEEE 8th Intl Conference on Big Data Security on Cloud (BigDataSecurity), IEEE Intl Conference on High Performance and Smart Computing, (HPSC) and IEEE Intl Conference on Intelligent Data and Security (IDS)*, Jinan, China, pp. 107–112, 2022.

[15] S. G. Wang, Y. Guo, N. Zhang, *et al.*, "Delay-aware microservice coordination in mobile edge computing: A reinforcement learning approach," *IEEE Transactions on Mobile Computing*, vol. 20, no. 3, pp. 939–951, 2021.

[16] K. Akahoshi, F. J. He, and E. Oki, "Service deployment model based on virtual network function resizing," *IEEE Transactions on Network and Service Management*, vol. 20, no. 1, pp. 547–562, 2023.

[17] A. Hazra, M. Adhikari, T. Amgoth, *et al.*, "Stackelberg game for service deployment of IoT-enabled applications in 6g-aware fog networks," *IEEE Internet of Things Journal*, vol. 8, no. 7, pp. 5185–5193, 2021.

[18] Z. Z. Xiang, Y. H. Zheng, M. Z. He, *et al.*, "Energy-effective artificial internet-of-things application deployment in edge-cloud systems," *Peer-to-Peer Networking and Applications*, vol. 15, no. 2, pp. 1029–1044, 2022.

[19] A. Bozorgchenani, D. Tarchi, and W. Cerroni, "On-demand service deployment strategies for fog-as-a-service scenarios," *IEEE Communications Letters*, vol. 25, no. 5, pp. 1500–1504, 2021.

[20] Y. M. Zhang, X. L. Lan, J. Ren, *et al.*, "Efficient computing resource sharing for mobile edge-cloud computing networks," *IEEE/ACM Transactions on Networking*, vol. 28, no. 3, pp. 1227–1240, 2020.

[21] F. C. Jia, D. Y. Zhang, T. Cao, *et al.*, "CODL: Efficient CPU-GPU co-execution for deep learning inference on mobile devices," in *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, Portland, Oregon, pp. 209–221, 2022.

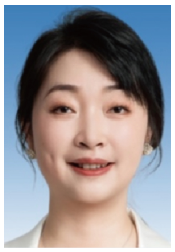
[22] S. Yue, J. Ren, N. Qiao, *et al.*, "TODG: Distributed task offloading with delay guarantees for edge computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 7, pp. 1650–1665, 2022.

Deyu ZHANG received the B.S. degree in communication engineering from PLA Information Engineering University, Zhengzhou, China, in 2005, and the M.S. degree in communication engineering and Ph.D. degree in computer science from Central South University, Changsha, China, in 2012 and 2016, respectively. He is currently an Associate Professor with the School of Computer Science and Technology. His research interests include mobile system optimization, edge computing, and stochastic optimization. He has published more than 50 papers in prestigious conferences and journals, such as ACM MobiCom, MobiSys, SenSys, CSUR and IEEE TMC, JSAC. He was a Visiting Scholar with the Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, ON, Canada, from 2014 to 2016. He visited Microsoft Research Asia between 2019 to 2020. Prof. Zhang has served as the Co-Chair of the SECIoT1 workshop and the Guest Editor for a special issue of the IEEE Internet of Things Journal and PPNA. He is an Associate Editor for KSII Transactions on Internet and Information. He is a Member of the ACM, IEEE, and CCF.
(Email: zdy876@csu.edu.cn)

Yu XIE received the B.E. degree from University of Electronic Science and Technology of China, in 2017. He is currently pursuing the M.A. degree at Computer Department, Central South University. His research interests include storage system and virtualization.
(Email: xie_yu@csu.edu.cn)

Mucong XU received the B.E. degree from Hunan Institute of Technology, in 2016. He is currently pursuing the M.A. degree at Computer Department, Central South University. His current research interests include stream computing and machine learning.
(Email: mucongxu@126.com)

En CHENG received the B.E. degree from Hainan University, in 2020. He is currently pursuing the M.A. degree at Computer Department, Central South University. His current research interests include computer vision and stream loading.
(Email: chengen@csu.edu.cn)



Xiaoyan KUI is a Professor in the School of Computer Science and Engineering at Central South University, China. She received the B.S., M.S., and Ph.D. degrees, all in computer science, from Central South University, China, in 2003, 2008, and 2012, respectively. Her research interests include information visualization, data visualization, visual analytics, mobile computing and vehicular network. She has published more than 40 technique papers in international journals and conferences. Prof. Kui has served as the Guest Editor for a special issue of the Electronics. She is a Member of the CCF. (Email: xykui@csu.edu.cn)

Bangwen HE received B.E. degree from School of Computer Science and Engineering, Central South University, China, in 2021. He is currently pursuing the M.A. degree at Central South University, Changsha, China. His current research interests include mobile neural network acceleration and edge computing.
(Email: hebangwen@csu.edu.cn)

Yunhao LI received the B.E. degree from Central South University of China in 2022. He is currently pursuing the M.E. degree at Computer Department, Central South University. His current research interest is bootstrap procedure of operating systems.
(Email: liyunhaocsu@csu.edu.cn)