Machine Bank: Own Your Virtual Personal Computer

Shuo Tang[1], Yu Chen[2], and Zheng Zhang[2]

[1]Tsinghua University
Dept. of Computer Sci. and Tech.
Beijing 100084, P. R. China
ts@mails.tsinghua.edu.cn

[2]Microsoft Research Asia
Beijing 100080, P. R. China
{ychen, zzhang}@microsoft.com

Abstract

In this paper, we report the design, implementation and experimental results of Machine Bank, a system engineered towards the popular shared-lab scenario, where users outnumber available PCs and may get different PCs in different sessions. Machine Bank allows users to preserve their entire working environment across sessions. Each client runs virtual machine, which is saved to and reinstantiated from a content-addressable backend storage. We carefully designed lightweight hooks at client side that implements caching and tracking logics to improve reinstantiation speed as well as to remove unnecessary network and disk traffic. Our detailed evaluation demonstrates that these techniques are effective, and the overall performance fits well with the shared-lab usage.

1. Introduction

Many institutions and enterprises face the problem that users outnumber available desktop PCs due to the cost, space and management constraint. Therefore, sharing PC is a very common practice. A common requirement under this scenario is to enable the users to preserve their working environments across work sessions.

For example, Microsoft Research Asia has many intern students. Due to the growth of personnel and lack of resources, part-time interns have to share computers in non-overlapping hours. They have two ways to ensure a work context across sessions. They can use Remote Desktop Connection [19] to work on dedicated servers, or copy their working files on the PCs to some data servers at the end of their work sessions. Both solutions have problems. The first requires dedicated and powerful servers, and the second is incapable of dealing with various software and personalized configurations across different PCs. For the time being, the first option is what the intern students typically use. Thus, the otherwise quite capable desktops in the lab are essentially downgraded into dumb terminals, whereas the group server(s) that the intern students associate with become overloaded. This situation is not unique. In Tsinghua University, the administrators of Central Computer Lab (open to all students) experience the similar problem when offering computers to students for their homework assignments. In fact, the current practice is to offer a clean system every time a user starts a session. As such there is no support of preserving a user's work environment. In what follows, we will refer to this usage scenario as *shared-lab*.

Virtual machine (VM) technologies such as Virtual PC [18] and VMWare [21] have brought the possibility to encapsulate the complete work environment of a user, and move it anywhere. The improvement of the hardware performance has also made the performance penalty of using the VM increasingly negligible. Furthermore, in the shared-lab scenario, network connectivity is often not only adequate, but also underutilized. Given that we can use commodity PCs to build large storage farm [10][15], an obvious architecture is to store a user's work environment into a storage vault when the user logs off, and then reinstantiate it to the user's next working PC (the destination PC) on-demand. However, there are a number of issues need to be addressed.

More often than not, the next PC is unpredictable. This results in several implications. First, when a user leaves a session, the entire environment must be backed up into a safe vault. The shared PC that the user just worked on may be trashed by the next user. Second, storing the entire VM image to the backend is wasteful. In fact, there are ample opportunities to optimize. For instance, even after personalization, the majority of the bits of a software package (e.g. Microsoft Office) do not differ across users in any significant way. Likewise, useless updates (e.g. temporary files) should not be committed to the backend storage. Third, in terms of performance, fast reinstantiation is critical. This means that we must reuse whatever data a previ-

1-4244-0910-1/07/$20.00 ©2007 IEEE.

ous user has left and optimize the performance when fetching data from backend storage otherwise.

Our solution to the situations is called the *Machine Bank*, analogy to the safebox image associated to a banking institute. The architecture combines lightweight hook at client side with a state-of-art content-addressable and highly reliable distributed storage system made of commodity PCs. Users' working environments at different sessions are encapsulated, versioned, and archived. In addressing the problems mentioned earlier, we employ a set of comprehensive techniques:

- On-demand fetching and caching mechanism to reduce VM reinstantiation latency. Unlike other solutions that have been proposed, we use a two-level cache architecture. The first level is private to a login session, and the second is a shared one across multiple sessions on a single PC. This enables maximum leverage across session boundary, even when sessions correspond to different users.
- Leveraging the content-addressable nature of the backend storage, we allow sharing across users and versions without imposing sophisticated metadata structure so that storage utilization does not explode with user population.
- We track disk updates to ensure that only useful updates are committed to the backend.

We have completed the full implementation at the client side and embedded a minimum set of functionalities at the backend server. Our evaluations suggest that our various optimizations are effective.

The remaining sections are organized as follows. In section 2 we analyze the main problem. Section 3 describes the design in detail and Section 4 provides implementation details. In Section 5 we provide experimental results. We cover related work in section 6, and we discuss future work and conclude in Section 7.

2. Migrating working environment

The working environment of a user consists of software (including OS, applications and their personalized settings) and the user's personal document/data. Virtual machine (VM) technology decouples the working environment from the underlying physical hardware and encapsulates the user's complete working environment into a VM.

The core of VM technology, Virtual Machine Monitor (VMM), runs upon underlying host machines (sometimes host operating systems) to provide a virtual hardware environment for conventional software [7]. Commercial VMMs such as Microsoft Virtual PC [18], VMware [21] all provide virtualized x86 platform. In our prototype system, we choose Microsoft Virtual PC as the platform to generate and migrate working environments. In Microsoft Virtual PC, a virtual machine is represented by a set of files including the VM configuration file, the virtual hard disk (VHD) file and sometimes a VM state file which contains the compressed runtime memory and hardware state. In the shared-lab scenario, the VM state file is not needed because the user logs off from one session completely. Since the content of these files changes over time, we collectively call the content of these files a *virtual machine instance* (VMI). By copying a user's VMI over a network, the VMM on another machine can reincarnate the VM to the state specified by the VMI, thus accomplishing the goal of moving the user's complete working environment.

As a working environment is encapsulated into a VMI, its size becomes a problem for both storage and migration. A typical VHD file is several gigabytes, and a VM state file amounts to hundreds of megabytes depending on the VM memory size. Therefore, the storage consumption and transmission time of VMIs without any optimization would be practically unacceptable.

To find potential optimizations, we conducted some preliminary investigations, showing that

1) The guest operating systems and other software only need a few megabytes of contents to start up and reach the ready state. For example, Windows XP starts in 30s ~ 1min and only needs to read about 75 MB disk content. The application software, such as Microsoft Office 2003 and Visual Studio 2003 needs even less.
2) IT supporting staff usually uses a disk image (e.g. ghost image) to replicate software on new machines.

The first observation means that a user can start to work when the VMI is partially available, and the rest can be fetched on-demand later. The second implies that systems of different users are evolved from the same start ing point and there is a significant amount of overlapping among different VMIs.

3. Design

Machine Bank is organized as a Client/Server architecture. The server side is BitVault[15], a reliable backend storage system responsible for storing VMIs of all users. The client side runs Virtual PC which instantiates users' working environments.

BitVault is a scalable distributed storage system to reliably store large amount of immutable data. BitVault is not a distributed file system. Instead, it gives a very large logical space (160bits) in which blobs of data can be checked in and out. The hash of the object is the handle for later checkout. In other words, BitVault is content-addressable. Two blobs with the same hash will only be stored once, as the collision probability is extremely small. The building block of BitVault is commodity PC which, when aggregated, can give extremely large storage capacity. Since commodity PCs are prone to failures, the core technologies of BitVault are to lower down the management overhead by automatically retaining reliability and

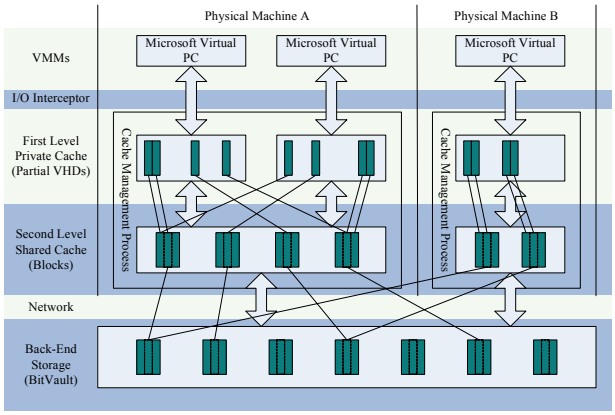

Figure 1: Cache architecture

availability goals. BitVault can continue to operate even when the system is handling failure or responding to reconfiguration (such as adding capacity by inserting a new PC online). One of our current installations uses 32 desktop PCs, with a total of 1TB of usable space. We should point out that for the purpose of Machine bank, *any* content-addressable storage system will work equally well.

The client-side takes care of on-demand fetching and sharing across different sessions. There are three mechanisms:

- VHDs are split into blocks and blocks are addressed and accessed by their content hash. This enables fetching partial VHDs as well as block sharing across VHDs of different users.
- A piece of code is inserted into the VMM to enable fetch-on-demand of VHD, decreasing the time required to instantiate a VM.
- VHD writes are cached locally and only useful updates are committed into the back-end storage after the VM is closed.

The VMIs of a user are versioned and stored in BitVault. When the user comes to a machine and begins to work, the VMI (usually the newest one) is retrieved and instantiated. Since the VHD is split into blocks, and only the needed blocks are fetched, instantiation latency is reduced. During a session, disk updates are tracked and cached locally. After a user finishes a session, the useful updates are identified to create a new VMI, which is committed into BitVault.

3.1. Overall architecture

The client side implements a Cache Management Process that bridges the interaction between Virtual PC and BitVault. It operates on a two-level cache, supporting VM instantiation via fetch-on-demand to VHDs and the creation and committing of new VMIs.

The first level cache (the private cache) imitates a private VHD mirror to the underlying VHD for a VM and handles I/O requests from VMM. All accesses to VHD issued by VMM are trapped by an I/O interceptor and forwarded to the private cache. The access unit in this cache could be the guest OS file system cluster. However, this requires a priori knowledge on the file system and disk analysis, because different VMs could use different cluster sizes. To keep it simple, we choose sector as the access unit, as is used in hard disks. The private cache is implemented using a sparse file, and there is a deterministic 1-1 mapping between the sectors of the VHD and the offset of the sparse file. The major function of the private cache is to cache VHD content updates, which will only be committed to BitVault at the end of session. As we will describe later, it is not efficient to simply commit all the updated sectors.

The second level cache (the shared cache) interfaces with BitVault to fetch demanded blocks and caches them on behalf of requests from the private cache. Different from the private cache, the access unit of the shared cache is a *block*, which consists of several consecutive sectors. As the access unit between client and BitVault, block also presents a tradeoff of optimizing VM instantiation performance. Larger block improves throughput, but at the expense of bringing more useless sectors. We will discuss this issue in detail in section 3.3.

In both BitVault and the shared cache, the blocks are identified and accessed by their content hash using SHA-1 [17]. The probability that two blocks map to the same 160-bit SHA-1 hash is negligible; the newest research shows that the complexity required for finding a collision in SHA-1 is 2^{63} [13], which is less than the error rate of a TCP connection or memory. Therefore, blocks from VHDs in different VMIs from different users can share storage space in BitVault. Moreover, this access mechanism also enables different VMI instantiations on the same client to share VHD content. After a user logs off from a client machine, the accessed blocks in the instantiation remain in the shared cache. Therefore, when the next user comes, the blocks shared between the previous instantiation and the new one could be fetched from the shared cache directly. As a result, the network consumption and instantiation latency are reduced. If the two users are actually the same person, or are using two VMs evolved from the same starting point, the saving can be significant.

3.2. Coordinating the private and shared cache

Since the private cache is addressed by sector offset and the shared cache is addressed by block content hash, there is a need to establish a fixed mapping between the two. For example, Virtual PC issues a read request for the 9th sector in VHD, and the private cache misses. Then we need to find the corresponding block in the shared cache using a hash. In order to decide the hash of the block con-

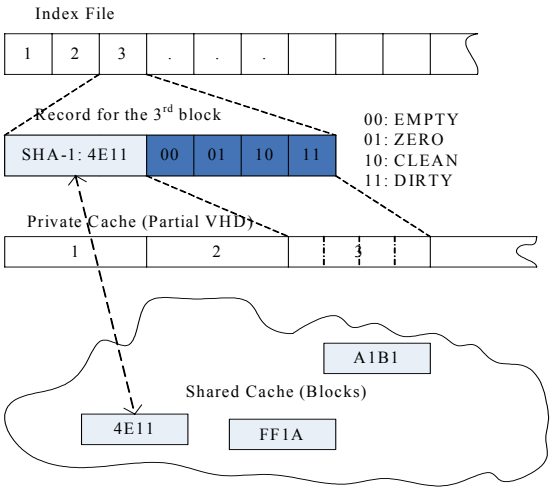

Figure 3: Data structure of the index file. The requested sector is the 1st of the 3rd block.

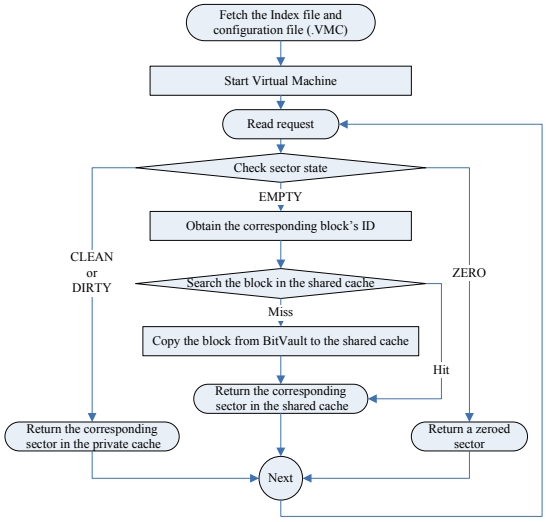

Figure 2: Instantiation procedure

taining the sector, a metadata file – VHD index is employed to bridge the gap.

As shown in Figure 3, a VHD index file is composed of a sequence of records, each corresponding to a block in the VHD. In each record, there is a field (block ID) storing the content hash of the block. Besides the content hash, we also allocate 2 bits for each sector in the block to store the sector state in the private cache. There are four states:

- EMPTY (00): The corresponding sector in the private cache does not contain valid data. Access on this sector will cause a miss in the private cache.
- CLEAN (10): The corresponding sector in the private cache contains data ready for read.
- DIRTY (11): The corresponding sector has been updated in this instantiation. Note that read on a sector in this state is also allowed.
- ZERO (01): Data contained in the corresponding sector is a sequence of zero. The reason of introducing this special state will be explained in 3.4.

Recall the example earlier in this section. As shown in Figure 3, the 9th sector is in the 3rd block and the sector status is EMPTY. So the content hash (0x4E11) is retrieved to search for the block, which is currently in the shared cache.

The size of the index file is proportional to the VHD size. In our prototype, a 4GB VHD generates a 3.1MB index file.[2] If the VHD is 40GB, the index file would be 31MB. The instantiation process (described next) will need to get the index before moving forward. For the shared-lab environment where network bandwidth is abundant, the bottleneck is the disk I/O (at one of the BitVault node). Assuming a 5MBps random I/O bandwidth, this will take roughly 6 seconds to retrieve. Thus, we believe this design is adequate for shared-lab scenario.

3.3. Instantiation

A user could have one or more instances stored in the BitVault. To reinstantiate his/her VMI on a client, the corresponding VM configuration file and VHD index file need to be retrieved first. During a VM instantiation, all VHD accesses issued by VMM are forwarded to the Cache Management Process. Figure 2 shows the procedure of handling the disk reads.

- If the requested sector is CLEAN or DIRTY, read it directly from the private cache.
- If the status is ZERO, return a zeroed buffer to the VMM.
- If the status is EMPTY, obtain the corresponding block's ID from index file and search in the shared cache. If the block is still not found, use the ID to copy the block from BitVault to the shared cache. Then copy the content from the shared cache to the read buffer directly of VMM and return. Note that we didn't fill the private cache to avoid extra local disk accesses. And the subsequent reads to the sector can be still fed from the shared cache.

As we mentioned earlier, we implement the private cache as a sparse file. The length of the file should be the same as the VHD size. The reason of not using a normal file to implement it is because many sectors will never be written. The EMPTY sectors are to be retrieved from BitVault by their content hash, and therefore should reside in the shared cache.

The shared cache is organized as a file folder in the host OS. A block is saved as a file using the hex representation of is content hash as the file path. This naturally

[2] Each block contains 128 sectors (64KB). The reason of choosing 64KB as the block size will be explained in the section 3.3.

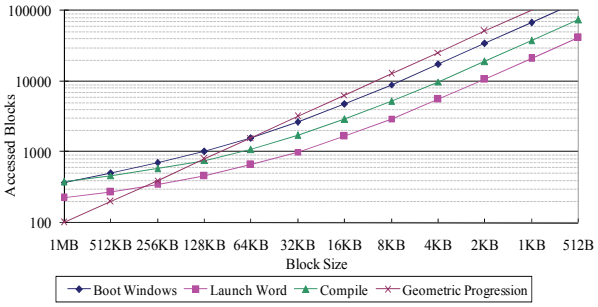

Figure 4: Relationship between Accessed-Blocks and BlockSize.

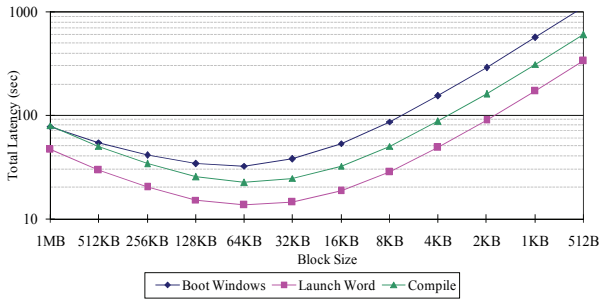

Figure 5: Relationship between total latency and BlockSize.

uses the file system directory structure to implement the hash lookup to store and retrieve a block. To avoid the performance penalty of putting too many files in a single folder, we break the hex string into segments of 2 characters long. Each segment represents a sub-folder except the last one. In this way, the number of files/sub-folders in each folder will not exceed 256 and the folder hierarchy contains at most 20 levels with SHA-1. For example, the block 4E11 will be placed as the file "d:\MBCache\4E\11". As our experiment results will show, this straightforward implementation delivers satisfactory performance.

The shared cache needs a replacement algorithm in case the storage utilization of the host machine becomes too high. A common cache replacement algorithm, such as LRU or LFU, can be employed, although in this prototype we have not implemented that. Since we do not copy content from the shared cache to the private cache when reading an EMPTY sector, the replacement algorithm may cause a block to be swapped in and out of the shared cache repeatedly, if the EMPTY sector is read sporadically but endlessly. Therefore, we implement a locking mechanism to prevent those blocks referenced by some EMPTY sectors in the private cache from being removed. When such removal becomes necessary, we will fill the related EMPTY sectors with the block content and change the state of those sectors to CLEAN.

On-demand fetching eliminates the waiting time before booting a VM, but also introduces extra latencies at VM instantiation. The latency is composed of two parts: a fixed delay and the block transferring delay. The fixed delay is an inherent property of the underlying network and BitVault (e.g. disk seek on a BitVault node and the message delay for sending requests to a BitVault node). The block transferring delay depends on block size and network bandwidth. Therefore, the total extra latency for a series of disk access is

$$\text{Total Latency} = AccessedBlocks \times \left(FixDelay + \frac{BlockSize}{BandWidth} \right)$$

While larger blocks require longer transferring time, they have a prefetching effect and serve better for sequential disk reads. For sequential reads, since the number of accessed blocks is inversely proportional to the block size, the total block transferring time is almost a constant for any block size. Thus, fewer accessed blocks lead to fewer fixed delay. On the other hand, for random reads, smaller blocks are desirable because using larger blocks is likely to fetch a large amount of unnecessary data.

To find the optimal point of block size, we conducted several experiments on Virtual PC to get some VHD access traces. The traces include booting Windows XP, launching Microsoft Office[3] and compiling a project using Visual Studio C++.Net. The first one is mainly composed of sequential reads, the second one has a lot of random reads, and the third one includes both random reads and writes.

We apply our on-demand fetching logic to the traces to get the values for *AccessedBlocks* on different *BlockSize*. According to [15], *FixDelay* is 8 milliseconds and *BandWidth* is 6 MBps (taking TCP slow start and concurrent accesses into account). Using the formula above, the total latency can be estimated.

Figure 4 clearly shows that the number of accessed blocks is inversely proportional to block size when it is smaller than 64KB. For blocks larger than 64KB, the number of accessed blocks is pushed up away from the inversely proportional curve, indicating the impact of fetching of unnecessary data. As shown in Figure 5, our estimations indicate that 64KB is the optimal tradeoff point of fixed delay and the block transferring delay for all traces.

Further analysis on the traces indicates that almost all the reads issued from VPC are for 64KB and 32KB blocks, revealing a possible pre-fetching behavior in VPC and/or Windows.[4]

[3] We launch Word, Excel and PowerPoint consecutively.

[4] According to *Windows Internals*, windows read 64KB for file reads by default when the running process does not specify other policies.

3.4. Commit

Creation and commit of new VMIs are important for users to preserve their working environments and results. Committing VMIs generally falls into two categories – initial commit and update commit.

The initial commit is for creating the first VMI from a VM and storing it into BitVault. It can be used to store a golden state into BitVault for users to start with. The initial commit scans through the VHD file of a VM, calculating content hash for each block and checking in block to BitVault using the content hash as their IDs. The index file is also created along this scanning processing, recording IDs for all blocks and setting flags of all sectors to be EMPTY. Then the index file and the VM configuration file are packed together and checked in to BitVault, representing the new VMI.

The update commit creates and commits new VMIs after a user finishes a session. As described previously, all updates to VHDs are intercepted and recorded in the private cache, and the index file marks all the written sectors as DIRTY. Consequently, the simplest way of update committing is to scan the index file for DIRTY sectors and create new blocks by merging the DIRTY and non-DIRTY sectors. If a modified block also has non-DIRTY sectors that are marked EMPTY, these sectors are first fetched from BitVault with a similar procedure described in 3.3. Then the new blocks are committed into BitVault using their content hashes and the corresponding records in the index file are updated (assigning the new hash value and setting all sectors' state to EMPTY). Finally, the index file and the VM configuration file are packed and checked into BitVault to represent a new VMI. If any error has stopped the index file from being checked in, then this is as if the VMI is not stored into BitVault. As long as the private cache's contents are not lost, the VMI can be created again.

A problem in the commit process is that not all sectors marked DIRTY are useful updates. In order to save the space consumption at storage side and reduce the committing time, we want to exclude unnecessary updates. For example, the page files of virtual memory bring significant amounts of DIRTY sectors, whereas they are essentially useless for the next instantiation after the guest operating systems are shut down. If we can identify those sectors, we can simply set their states to EMPTY. However, in general this is not possible. To solve this problem, we leverage a security feature in Windows, which clears the content of page files by writing zeros to it during OS shutdown. This is how the ZERO state is brought into the play: the sectors with zeros are marked with ZERO in the index file. And the private cache logic for writes becomes:

- If content written to a sector are all zeroes, skip disk accesses and mark the sector as ZERO in the index.
- If not, write directly to the private cache and mark the sector as DIRTY in the index.

At the commit time, the updated blocks full of zero sectors can be skipped. Since when accessed, a buffer of zero will be returned (refer to Section 3.3). This mechanism saves both commit traffic as well as instantiation traffic. Another benefit of the ZERO state is that the cost of writing zeros to the sectors is skipped too: we only update the index file.

For Linux systems, it is even simpler because the whole swap partition can be ignored through parsing the VHD partition table when committing.

Another problem relates to temporary files. Many user workloads generate large amount of files that will not be needed after shutdown. Compiling a big project and operating a Microsoft Office document are such examples. On deleting a file, most operating systems only modify the containing directory while leaving the disk sectors of the file unchanged. While this optimization saves disk I/O, it becomes a challenge for Machine Bank because the disk sectors occupied by the deleted files will also be committed unless we can identify them.

To solve this problem, we parse the cluster allocation table of the file system in the VHD and set the updated sectors (marked as DIRTY or ZERO) in unallocated clusters to CLEAN. Specifically, in NTFS, we check the $BITMAP file to discard the unallocated clusters in the disk. Thus, these blocks will not be committed to the backend storage.

3.5. Versioning of VM instance

As a user continuously creates and commits new VMIs in his/her daily work, an evolution history of the user's VM is generated. By browsing the history and reinstantiate a previously checked-in VMI, a user is able to perform "time travel". This capability is very useful, especially when permanent failures occur in the current VMI.

To accomplish this, we leverage the Catalog File (CF) feature in BitVault. With Catalog File, a user can add an

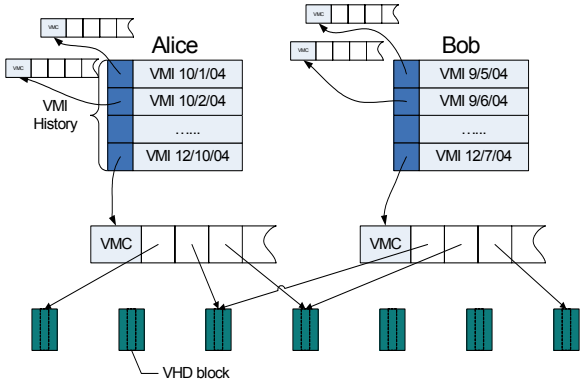

Figure 6: Logical structure of VM instances in BitVault.

optional description and a *catalog id* when checking in an object. BitVault will group the ids and descriptions of the objects with the same catalog id into a Catalog File. The Catalog File can be checked out using the catalog id as if it is a common object.

In Machine Bank, a VMI is represented by the vmc file (VM configuration file which is typically very small) and the VHD index file, since the VMI can be reinstantiated as long as the two files are available. So we call the package of these two files a VMI package. When committing a VMI, we store the current time as the description and then check in the VMI package to BitVault, using the user's id as the catalog id. As shown in Figure 6, pointers to VMIs are thus grouped into the catalog file keyed by the user id.

By default, the Machine Bank client automatically retrieves the latest VMI from the catalog file and starts the reinstantiation. Alternatively, the user can browse the catalog file and choose which VMI to reinstantiate from. This is how time-travel is accomplished. Figure 6 also demonstrates how storage sharing is achieved at block level across different users and different sessions: for any unique block, there is one and only one copy stored inside BitVault

4. Implementation

We implemented our prototype on Windows XP using C++. The system consists of three components: a hook library that intercepts accesses to VHD file from VPC, a daemon for interacting with BitVault, and the committer.

The hook library intercepts four Windows APIs, CreateFile, ReadFile, WriteFile and CloseHandle, which correspond to fopen, fread, fwrite and fclose in C runtime, respectively.[5] In Windows, these four functions are implemented in the kernel module Kernel32.dll. Modules in a process invoke the functions by looking up their Image Address Tables for the function addresses [20]. According to this behavior, the hook library modifies the entries of the four functions in Image Address Tables of all modules (except the hook library itself) in the VPC process. Therefore, calls to these functions are intercepted without modifying VPC source code and Windows kernel. When any of the four API functions is invoked, the hook library checks whether it is for the VHD file. If it is, the cache logic described above is used to handle the file request. If not, the request is directed to the original addresses in Kernel32.dll. Since Virtual PC does not use file mapping to access VHD files, we just ignored CreateFileMapping and MapViewOfFile / UnmapViewOfFile APIs, which provide similar functionalities as mmap().

The daemon is built upon the BitVault interface and is in charge of checking out blocks from BitVault to the shared cache. The communication between daemon and hook library is via IPC using Windows share memory and semaphore.

Finally, the committer is used for both initial committing and updates committing.

[5] VPC only uses these four APIs to access VHD and VMC files.

5. Experimental Results

We conducted some experiments to evaluate the efficiency of our solution. Client PCs used in the experiments run Microsoft Windows XP and Virtual PC. Their hardware configurations are 2.8GHz Pentium4 CPU with 1GB memory. Another four machines with 120G SATA hard disks, 3GHz Pentium4 CPU and 512MB memory run BitVault servers to provide back-end storage. All client and server machines are connected with two AT-8324SX 100Mb switches stacked together using 100Mb NICs. We configured the VMs to have 128MB memory (Windows automatically allocated 768MB page file) and 4 GB VHD. The software in VM includes Microsoft Windows XP with SP1, Office 2003 and Visual Studio C++.net. We committed the VHD into BitVault using block size of 64KB (the optimal one in Figure 5).

As the BitVault has provided the reliable storage to the users, we conduct the following experiments to evaluate 1) on-demand fetching and corresponding caching mechanism to reduce migration latency in section 5.1, and 2) cost of committing with the help of access-by-hash and useful update detection policies in section 5.2.

5.1. Instantiation efficiency

We collected data of reinstantiation by revisiting the preliminary analysis experiments of booting up Windows XP, launching Microsoft Office and compiling a project using Visual Studio C++.Net. These are typical behaviors of intern students.

For comparison, we test our prototype in four different conditions and collect the corresponding latencies.

- Empty Cache: the instance is started with both caches empty, meaning all blocks must be fetched on-demand from BitVault;
- Primed Cache: the shared cache has all needed sectors for the reinstantiation, i.e. the latency measured in this condition excludes the on-demand fetching latency;
- Local VHD: tasks are performed on a VM stored completely on local disk.;
- Remote VHD: the VHD is stored in a mounted network drive. To imitate the write back cache as our private cache, we turn on the undo disk functionality in Virtual PC, which caches disk writes to a local log file.

The latencies of Primed Cache indicate the best possible performance using Machine Bank. The difference be-

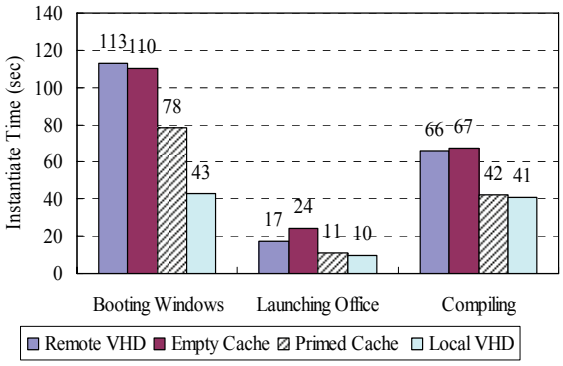

Figure 8: Elapsed time of instantiation for different task.

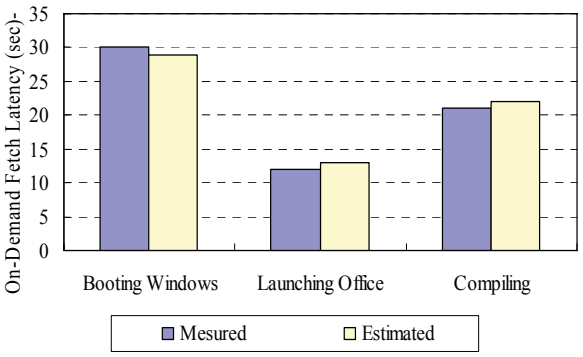

Figure 7: Latencies of on-demand fetching for different tasks.

tween Primed Cache and Local VHD represents the overhead of our cache logic. The latencies of Empty Cache are the worst case performance. The difference between Empty Cache and Primed Cache reveals the cost of fetching blocks from BitVault.

As shown in Figure 8, for Launching Office and Compiling, the latencies of primed cache and local VHD are statistically identical. This means that the cost of pure cache logic is negligible. The gap in Booting Windows is caused by a strange behavior of Virtual PC, in which our trace showed repeatedly file enumerations in the directory that contains the VHD file in VM startup.[6]

The latencies of Empty Cache almost equals to the ones of Remote VHD, meaning BitVault provides a similar access performance as the Windows network file system. Since the shared cache is usually not empty, the real performance that a user experiences in daily use is between the Empty Cache and the Primed Cache. By adding prefetching algorithms to our cache logic, we expect the instantiation performance could be improved even further.

Figure 7 plots the time stalled while waiting for the blocks to arrive from BitVault. The figure also draws the estimated stall time, using the traces and the formula described in Section 3.3. If the user log in session lasts several hours, a half-minute longer booting does not appear as a huge overhead. Likewise, if the user uses application (such as Office and compiling projects) throughout the session, the 20 seconds delay will be paid only once. Thus, we conclude that the performance is adequate for the shared-lab scenario.

5.2. Commit cost

As described in section 3.4, the committing phase includes the initial commit and the update commit. In the initial commit, we commit the 4GB VHD into BitVault using 64KB blocks. The OS and the software installed in the VM consumed 3.63GB of the VHD space. After initial commit, a total of 47391 unique blocks are generated, amounting to 2.89GB. The difference is caused by excluding the 768MB page file. Since we enable clear page file policy mentioned in section 3.4, the disk sectors occupied by page file are set to ZERO and blocks full of ZERO sectors are omitted in committing.[7]

Meanwhile, an index file of 3.1MB is created, in which the 160bit content hash for each block occupies 1.1MB and the rest are for sectors' state. Because initial committing only assigns EMPTY or ZERO to each sector (DIRTY or CLEAN are used only within a session), we use a gzip library [6] to compress the index file to around 1.4MB. Thus, the index itself is very small.

For update commit, we carried out two experiments to validate the efficiency of detecting useful updates. In the first experiment, we started Windows XP, compiled a project, and deleted all the generated files except the exe file. Then, the VM was shut down and committed. The compilation generated a total of 55MB files, in which the exe file is 1.4MB. In the second experiment, we started Windows, created and saved a Word document of 1.5MB, then shut down and commit the VM. During document editing, Word created 5MB temporary files, which was deleted after Word was closed. For each experiment, we collected the total number of dirty sectors and the number of useful dirty sectors after applying P1 (introducing ZERO and clearing page file) and P2 (discarding unallocated clusters).

[6] We conducted experiments that change the extension of local VHD files and force the VM configuration file to use relative path point to local VHD files. In both cases, Booting Windows got slow down.

[7] The unallocated disk sectors are also ZERO. When creating a new fix VHD, Virtual PC sets the initial content of the VHD file to zero.

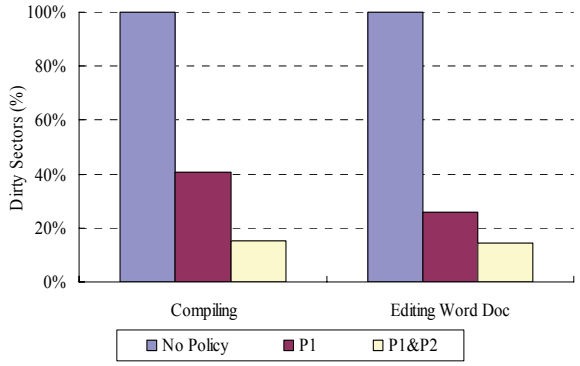

Figure 9: Percentage of reduction of dirty sectors after applying policies.

In the experiments, we recorded a total of 459170 (224.2MB) and 421448 (205.8MB) dirty sectors for Compiling and Editing Word Doc, respectively. As shown in Figure 9, P1 in Compiling discards around 60% of the dirty sectors and P2 further extends the reduction to around 85%. Noted that 224.2MB*(85%-60%) = 56MB, which is very close to the amount of files that are deleted. In Editing Word Doc, we also get a reduction of more than 80% using both P1 and P2. Because Word creates relatively small number of temporary files, the effect of P2 is limited.

Although the dirty sectors left after applying P1 and P2 are no more than 20% of the total, the absolute value is still exceeds our expectation. The number of dirty sectors left is 69290 (33.8MB) in Compiling and 61429 (30.0MB) in Editing Word Doc. The additional dirty sectors come from disk writes on NTFS system files and other files such as registry hive files and application configuration files. We analyze the result of Editing Word Doc experiment and find that the dirty sectors belong to 208 files/folders. In Table 1, we summarize the top 10 "dirty" files/folders.

The "\Doc1.doc" is the word document we produced in the experiment. To our surprise, the virtual memory page file "\pagefile.sys" still contributes more than 12MB non-zero dirty data, although the clear page file option is enabled. Other updated files include NTFS system files ($MFT and $LogFile), OS system files/folders and registry hive files (NTUSER.DAT) that contains user's personal settings. These files as a whole actually represent the evolution of the user's working environment.

Table 1: Top 10 "dirty" files in editing Word doc experiment

file path	# of dirty sectors
\pagefile.sys	25200
$LogFile	7632
\Documents and Settings \Administrator\Local Settings \Application Data\IconCache.db	7561
$MFT	4487
\Doc1.doc	3325
\WINDOWS\system32\config\system	3142
\WINDOWS\system32\wbem\Repository \FS\OBJECTS.DATA	2334
\WINDOWS\system32\config\software	1457
\Documents and Settings \Administrator\NTUSER.DAT	1211
\WINDOWS\system32 (folder)	584

6. Related work

In recent years, virtual machine monitor technology revives with Disco [3], Denali [14] projects and commercial products, such as Microsoft Virtual PC [18] and VMWare [21], long after the upsurge of research into virtual machines at hardware level [7] in 1970s. This brought many new and interesting applications. For instance, Chen and Noble [4], Kozuch and Satyanarayana et al [8][12] have independently come up with the idea of using virtual machines for user mobility. Awadallah and Rosenblum proposed the vMatrix architecture to achieve server multiplexing using VMWare x86 VMMs [1][2]. Zhao et al utilized the facility of VMMs to provide execution environments across distributed resources in grid computing [16].

Instead of focusing on advanced use such as moving the work environment among multiple machines owned by the same user, we concentrate on the popular scenario of shared laboratory. In this case, it is unpredictable where the user will start the next work session and techniques such as fast-migration [5] does not apply. While we do not need to concern ourselves with the issue of copying memory state, there are challenges on how to instantiate a user session and launch the application faster, and how to maximize the storage utilization at the backend server. These two problems are also addressed in the Collective system [11], which is the most closely related to our work.

Instead of using a single cache with small sectors, we carefully analyzed the applications and institute a second level cache with larger cache granularity. We use secure hash as the handler to address a block in a content-addressable reliable storage, and encode the hashes in a per-user per-session index file. This makes blocks across sessions and users are sharable, without the need of a sophisticated hierarchy as is done in the Collective system. Fetching and caching this index file at the client means retrieving missing block takes one round trip to the backend, as opposed to multiple rounds otherwise. Finally, to make sure that useless blocks waste neither bandwidth nor storage space, we skip them before committing to the server. Our evaluations have shown that these techniques are effective. In a recent work [9], Partho Nath et al. also gave detailed evaluation about the impact of chunk size to backend storage consumption, network utilization and privacy. But we focused more on ways of reducing net-

work latency of VM reinstantiation and discarding "useless updates".

7. Conclusions and future work

In this paper, we report the design and implementation of Machine Bank, which enables backup, fast migration and re-instantiations of *Virtual Machine Instances*. Machine Bank is layered on top of a content-addressable data retention platform BitVault and employs a two-level cache structure to enable 1) fetch-on-demand of data to reduce the time required to instantiate a VM, 2) copy-on-write to commit only differences of VM state and 3) access-by-hash to allow data shared among different VM instances. Our result shows that the block size of 64KB yields the best instantiation performance. Moreover, with our update detection mechanism, a good fraction (more than 80% in our experiments) of disk updates are identified as useless and discarded in the experiments.

Our future work centers mostly on the management and administration of the Machine bank, and there are issues at both the client and the server side. On the client side, our current prototype does not encrypt the index file, opening a possibility of compromising user privacy. On the server side, we will need to address the issue of removing all VMIs that the users are no longer interested in.

References

[1] A. A. Awadallah and M. Rosenblum. The vMatrix: A network of virtual machine monitors for dynamic content distribution. In *Seventh International Workshop on Web Content Caching and Distribution*, August 2002.

[2] A. A. Awadallah and M. Rosenblum, The vMatrix: Server Switching, IEEE 10th International Workshop on Future Trends in Distributed Computing Systems (IEEE FTDCS 2004), May 2004.

[3] E. Bugnion, S. Devine, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems*, 15(4):412–447, November 1997.

[4] P. M. Chen and B. D. Noble. When virtual is better than real. In *Proceedings of the 8th IEEE Workshop on Hot Topics on Operating Systems*, May 2001.

[5] C. Clark, K. Fraser and S. Hand. Live Migration of Virtual Machines, In *Proc. 2nd Symposium on Networked Systems Design and Implementation*, May 2005

[6] P. Deutsch. Zlib compressed data format specification version 3.3, May 1996.

[7] R. P. Goldberg. Survey of virtual machine research. *Computer*, 7(6):34–45, June 1974.

[8] M. Kozuch and M. Satyanarayanan. Internet Suspend/Resume. In *Proc. IEEE Workshop Mobile Computing Systems and Applications,* IEEE Press, 2002, pp. 40-46.

[9] Partho Nath, Michael Kozuch, David O'Hallaron, M. Satyanarayanan, N. Tolia, and Matt Toups, Design Tradeoffs in Applying Content Addressable Storage to Enterprise-scale Systems Based on Virtual Machines. *USENIX '06*, Boston, MA, May, 2006.

[10] Y. Saito, S. Frolund, A. Veitch, A. Merchant and S. Spence. FAB: Building distributed enterprise disk arrays from commodity components. In *11th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2004

[11] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M.Rosenblum. Optimizing the migration of virtual computers. In *Proc. of the 5th Symposium on Operating Systems Design and Implementation (OSDI-02)*, December 2002.

[12] M. Satyanarayanan, Michael Kozuch, Casey Helfrich, and David R. O'Hallaron, Towards Seamless Mobility on Pervasive Hardware, *Pervasive & Mobile Computing,* vol 1, num 2, pp 157-189, June, 2005.

[13] X. Wang, A. Yao, and F. Yao, New Collision search for SHA-1, *Rump Session, Crypto'05*, August 2005.

[14] A. Whitaker, M. Shaw, and S. Gribble. Denali: Lightweight virtual machines for distributed and networked applications. Technical report, University of Washington, February 2001.

[15] Z. Zhang, Q. Lian, S.D. Lin, W. Chen, Y. Chen, C. Jin, BitVault: a Highly Reliable Distributed Data Retention Platform, Technical report, MSR-TR-2005-179.

[16] M. Zhao and R. J. Figueiredo. Distributed File System Support for Virtual Machines in Grid Computing, In *Proceedings of HPDC-13*, 06/2004

[17] FIPS 180-1. Announcement of weakness in the secure hash standard. Technical report, National Institute of Standards and Technology (NIST), April 1994.

[18] http://www.microsoft.com/windows/virtualpc/default.mspx

[19] Get Started Using Remote Desktop http://www.microsoft.com/windowsxp/using/mobility/getstarted/remoteintro.mspx

[20] Inside Windows: An In-Depth Look into the Win32 Portable Executable File Format http://msdn.microsoft.com/msdnmag/issues/02/02/PE/default.aspx

[21] http://www.vmware.com