Efficient In-depth I/O tracing and its application for optimizing systems

Sushil Govindnarayan Mantri

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science and Applications

Ali R. Butt, Chair
Dennis Kafura
Pin Zhou

February 21, 2013
Blacksburg, Virginia, USA

Keywords: I/O Tracing, VM placement, File-system tracing

Efficient In-depth I/O tracing and its application for optimizing systems

Sushil Govindnarayan Mantri

(ABSTRACT)

Understanding user and system behavior is most vital for designing efficient systems. Most systems are designed with certain user workload in mind. However, such workloads evolve over time, or the underlying hardware assumptions change. Further, most modern systems are not built or deployed in isolation, they interact with other systems whose behavior might not be exactly understood. Thus in order to understand the performance of a system, it must be inspected closely while user workloads are running. Such close inspection must be done with minimum disturbance to the user workload. Thus tracing or collection of all the user and system generated events becomes an important approach in gaining comprehensive insight in user behavior.

As part of this work, we have three major contributions. We designed and implemented an in-depth block level I/O tracer, which would collect block level information like sector number, size of the I/O, actual contents of the I/O, along with certain file system information like filename, and offset in the file, for every I/O request. Next, to minimize the impact of the tracing to the running workload, we introduce and implement a sampling mechanism which traces fewer I/O requests. We validate that this sampling preserves certain I/O access patterns. Finally, as one of the application of our tracer, we use it as a crucial component of a system designed to do VM placements according to user workload.

Dedication

Dedicated to my parents, family, and friends for their encouragement and support.

Acknowledgments

Firstly, I would like to thank my advisor Dr.Ali R. Butt. Ali is an extremely capable academic, who easily recognizes the problems that should be studied and the questions that need to be asked. He motivated me to look at the problem from different perspectives. Besides providing me with input and ideas, he pushed me to keep on working throughout this thesis, and providing valuable guidance.

Next, I would like to thank Dr.Pin Zhou who has been our collaborator in this work. Her practical knowledge helped me understand many aspects which need to considered in designing efficient systems.

I would also like to thank Dr.Dennis Kafura for his insights and useful comments in pursuing this research and thesis.

I would also like to express my gratitude to Min Li, for her guidance and cooperation in this work. Her mentorship was really important for this project.

Finally, I would also like to thank my friends and colleagues at the Distributed Systems and Storage Laboratory (DSSL). We had quite a bit of fun at various lab meetings, paper presentations and lunch trips.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Contribution 3

1.3 Outline 4

2 Background **5**

2.1 Study of available traces 5

2.2 Different tracing approaches 6

2.3 Block trace 7

3 Design **8**

3.1 Blktrace 8

3.2 Modification 9

3.3 Sampling 12

3.3.1 Detail 12

3.3.2 Experiment 14

3.3.3 Evaluation 14

4 Application **15**

4.1 Introduction 15

4.2 Background 16

4.2.1 I/O Reduction Techniques 16

4.2.2 Virtual Machine Management in Virtualized Environment 18
4.3 System Design 19
4.3.1 Design Rationale 19
4.3.2 Terminology 20
4.3.3 Architecture Overview 21
4.3.4 Hierarchical Clustering in VM Manager 21
4.3.5 I/O Monitor 25
4.3.6 DHT Node Operation 25
4.3.7 Migration Execution 25
4.4 Evaluation 27
4.4.1 Methodology 28
4.4.2 Effectiveness of *SMIO* 30
4.4.3 Parameter sensitivity analysis 32
4.4.4 System Overhead and Scalability 34

5 Bibliography **36**

Chapter 1

Introduction

1.1 Motivation

Understanding user and system behavior is important for designing efficient systems. Most systems are designed with certain user workload in mind. However, such workloads evolve over time, or the underlying hardware assumptions change. Further, most modern systems are not built or deployed in isolation, they interact with other systems whose behavior might not be exactly understood. For example, different disk buffering strategies can have different performance impact on user's throughput. Also modern systems are complicated, be it due to a longer software stack, cache layer. A storage system can be specialized by designing to a special data access pattern; e.g., a storage system for streaming supports different usage patterns than a document repository. The better the access pattern is understood, the better the storage system design. Thus in order to understand the performance of a system, it must be inspected closely, while user workloads are running. Such close inspection must be done with minimum disturbance to the user workload. Thus tracing or collection of all the user and system generated events, becomes an important approach in gaining comprehensive insight in user behavior.

Trends in last couple of decades has shown an exponential increase in computing power of modern systems [22]. Similarly, there also have been substantial improvements in storage technologies. In physical hard drive space, whose use is still quite ubiquitous, these improvements have been mainly in terms of capacity. However, bandwidth and latency are still major bottlenecks in physical disks. As CPUs become faster, this gap is projected to get worse, thus making it a critical problem among research community. As a result, I/O performance and reducing I/O cycles have gained a lot of research traction. Consequently, file system and I/O improvement techniques are being published.

Many file system studies have been done using trace collection [24, 18, 19, 3, 13]. The original analysis of the 4.2BSD file system [18] motivated many of the design decisions of the log-

structured file system (LFS) [20]. In early 2000, many trace-based studies were expanded to include the increasingly dominant desktop systems of pc and mac and new workloads such as web servers. It is clear from the literature studies of trace-based systems that there are many interesting and important scenarios to consider when designing a file system, and that new workloads emerge as new applications and uses for file systems appear. Further, as the community of computer users has expanded there has been a substantial difference in the workloads seen by file servers, and that the research community must find ways to observe and measure workloads. Because of the increasing gap between processor speed and disk latency, file system performance is mainly dependent on the system disk. Like other computer systems, file systems provide good performance by optimizing for common usage patterns. However, these usage patterns vary both over time and across different user communities, and restudying them becomes important.

Uses Traces can be used to determine dynamic access patterns of file systems such as the rate of file creation, folder creation, the distribution of read and write operations, file sizes, file life, movement of the head of the hard disk between different access i.e seek time, the frequency of each file system operation, etc. The information collected through traces is useful to determine file system bottlenecks. One of the new uses of traces has been to generate replayable models. These replayable models are used offline using the traces, and help in identifying subtle bugs. It can also help identify typical usage patterns that can be optimized and provide valuable data for improving performance.

Although traces are mainly used for file system performance studies, they can be also be used for debugging and security. First, file system tracing is useful for debugging other file systems. A fine-grained tracing facility can allow a file system developer to locate bugs and points of failure. Second, file system tracing is useful for security and auditing. Monitoring file system operations can help detect intrusions and assess the damage. Tracing can be conducted file system wide or based on a suspected user, program, or process. Also, file system tracing has the potential for use in offline assesment to roll back and replay the traced operations, or to revert a file system to a state prior to an attack.

Problems Most of the previous studies focused only on studying the characteristics of file systems[18, 19, 5, 3] and thus a reusable infrastructure for tracing wasn't developed or well documented in research literature. Further after such studies were published, the traces werent always released or were difficult to obtain. Sometime, the traces excluded useful information for others conducting new studies; information not included was about the description of system or workload on which the traces were collected, some file system operations and their arguments, file paths, etc [2]. Only very few of the studies[11] actually released the contents of the actual file system operation i.e the data read or written. Such information wasnt the primary interest for file system studies, and only has recently become important for analyzing and understanding dedup related systems. Some studies which release, particular studies related to block level traces, the data hash the data using some hashing function. Changing hash function is not possible. However, there has been very little interest in including both file and block level information in traces i.e including file offset, file name along the with disk

offset, and actual io contents. Such information could be useful for a remote storage server backed disk images. For a file system, such information can be used for on-disk layout.

Traditional storage systems design was mainly influenced by the underlying hardware and user's access patterns. However dedup effectiveness depends on data content, its metadata properties as well as access patterns. Most datasets that have been used to evaluate deduplication systems are either not representative, on in some cases unavailable due to privacy reasons, preventing easy comparison of competing strategies. Understanding how both content and metadata evolve is critical to the realistic evaluation of deduplication systems. In recent years, virtualization is becoming pervasive. Particularly in enterprise environment, users have only a thin client at their disposal, whereas their desktops are running on virtual machines on shared hardware with shared storage system. This has further complicated the data access pattern seen by the storage server.

1.2 Contribution

1) Multi-layered trace - Most of the studies focused on trace collection at a single layer. File system studies using system calls mostly focus on instrumenting the system call layer in the kernel. Studies focusing on NFS based tracing focus on the request received and sent by the File server, and thus lack any information about the underlying block/disk layer accessess. Pure block level studies collect disk access and (sometimes) iocontents, however these dont contain any file level information. In this work, we develop a multi-layer tracing mechanism which collects information from both the file system and the block layer for any IO request sent to the disk. Such multi-layer tracing requires carefully implementing the locks.Both information can be used in designing the complete filesystem i.e both the in-memory optimization(like prefetching) and on-disk layout of file data and metadata.

2) Configurable low-overhead tracing - Most of the tracing studies dont collect the actual content due to overhead of collecting it. For IO intensive tasks this data could be big. We optimize our tracing with the sampling approach where it does on-depth tracing for fewer requests. We experimentally show that this approach does retain many read-write characteristics.

3) Extend the tracing - We extend our tracer to be used as a live IO monitoring process. We, in a joint work, design a system to reduce shared storage accesses by placing vms having similar workloads. Similarity in workload is based on the comparing the trace data collected by IO monitor in the VMs.

1.3 Outline

Next we give a background study done to explore various tracing approaches used in the literature. We also classify them. In subsequent chapter, we give an overview of a block level event tracer called blktrace and show how we extended it to provide I/O contents and file level information. We discuss sampling and show overhead comparison. In the next chapter, we describe the problem where we use the tracer to help in better placement of VMs for IO optimization.

Chapter 2

Background

2.1 Study of available traces

Following are some of the background study done to understand tracing techniques:

1) Zhou - Berkley UNIX

Zhou implemented a tracing package for the DEC VAX 11/780 running UNIX 4.2BSD. They instrumented file operations and other system calls to log both the call and its parameters. Their traces were collected comprehensively and in a binary format and buffered in the kernel before flushing to the disk. The package uses a ring of buffers that are written asynchronously using a user-level daemon. The tracing system also switches between trace files so that primary storage can be freed by moving the traces to a tape. The overhead of tracing is reported up to 10 percent. However, it provides little flexibility in which calls to trace and was verbose and required tedious post-processing. Finally, not all I/O happens at system call level, e.g memory mapped I/O; such tracing at the system call level makes it hard to log those or to trace network-based file systems (NFS)

2) Baker - Distributed File System

Baker et al collected and analyzed the user-level file access patterns and caching behavior of the Sprite distributed file system. They compared their findings with the 1985 BSD study, by collecting file system activity served by four file servers over 8 days. They instrumented the kernel to trace file system calls and periodically transfer the data to a user-level logging process. Some of the file system operations like read, write, directory reads were not recorded to limit the performance hit on running processes.

3) Roselli - UNIX and NT

Roselli et al described the collection and analysis of file system level traces for various operating system, production uses, client and server architecture. HP-UX traces were collected

by using the auditing subsystem to record file system events. They did a comprehensive study of modern workloads, focusing on disk I/O aspects of tracing. They observed that process use memory mapped I/O more frequently than direct I/O. Windows NT traces were collected by interposing file system calls using a file system filter driver. One of the problems is, due to the nature of Windows NT paging, separating actual file system operations from the other VM activity is difficult and must be done during post-processing.

2.2 Different tracing approaches

Network based tracing: The idea of collecting traces by snooping on a broadcast network had emerged almost immediately after the introduction of client server protocols like RPC. NFS tracing was among the first in this kind, and began to appear after NFS clients were used in university departments. Advantages :

1. NFS is portable and widely adopted.
2. Does not require modification to neither the client nor the file server.

Disadvantages :

1. Much information about the underlying file system is hidden.
2. Client-side caching can affect the observed workload.
3. NFS calls and responses could be lost or reordered.
4. If a VM-image is served by the file-server, it is not easy to detect what file in the vm image is being accessed.

static tracing: This approach takes snapshot of the file system. It conduct file system studies by infering traces from examining file system metadata at several instants in time. They use these snapshots for understanding distributions of file attributes commonly stored in metadata, such as file size, last access time and modification time, file name and directory structure. This significantly reduces the complexity of trace collection. It has the benefit of not affecting the running workload, and isolation of the I/O generated by tracing itself.

kernel based tracing: As the name suggests this involves modifying the kernel to record the user and system events. One earliest approach that did that was the BSD paper of 1985. It requires careful implementation as to not affect the running workload. More efficient frameworks have been developed leveraging this approach.

system call tracing : System-call level traces often miss infor mation about how system call activity is translated into multiple actions in the lower layers of the OS. This is one of the easier way to do tracing.

2.3 Block trace

Linux kernel abstracts hard disk as a block level device, where writes/reads happens as a block or in a unit large than byte. Block level tracing is a common technique to understand closer interactions between disk and the block subsystem. It is a low overhead tracing. It works by trapping the requests in and out of the block susbsytem in linux kernel. On the fly events are generated as it is integrated with kernel event mechanism and outputs interesting statistics of measured event. This is particularly desirable for us - can trace contents, closer to the dedup layer and block device is virtualized. In the next section we will go in more details regarding its design.

Chapter 3

Design

3.1 Blktrace

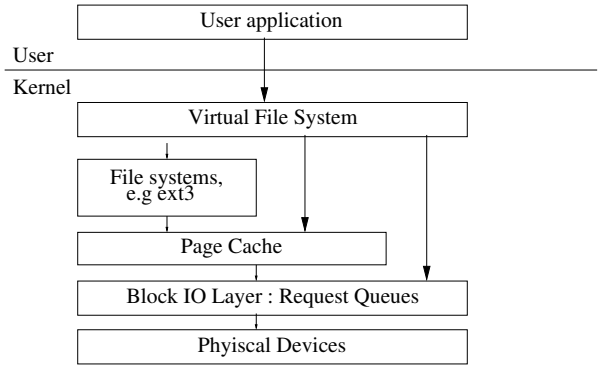

Figure 3.1: Block layer architecture.

Figure shows different layers that come into play for a IO request. User applications issue system call to read or write a particular file. For the linux kernel, VFS layer handles these calls. Advantages of VFS are it simplifies providing a unified filesystem tree even if there are multiple active filesystems. Thus VFS layer routes the request to appropriate file system. However, some request could be sent directly to the block device by mapping the device as a special file in /dev. Not all requests to the filesystem is sent to the disk. There is a critical component called page cache or buffer cache which temporally stores the recently used pages. However, if such request cannot be served by the page cache it is then sent to the block layer to get the data blocks from the underlying device. Block layer consists of block device driver, and request queues, and scheduling algorithm. The scheduling of io is important to hide the latency of modern hard disk drives.

Blktrace is a block layer IO tracing mechanism which provides detailed information about request queue operations to the user space. It is an open-source project primarily developed by Jens Axboe, from Oracle. It was released in October 2005, and is included in the linux mainline starting from kernel version 2.6.17. It has been very stable and used in many studies. A number of statistics was added to it to show various features. It provided a means to do closer inspection of the order of IO request to the, the time taken for the round trip, the efficiency of different block queue scheduling techniques. To ease this, a number of post-processing tools were also developed which output interesting stats and plots to help. The tool also is also highly configurable with options to tune all the various parameters like operation type, events to trap, etc. A number of factors we were looking were matched by this, and hence we extended this tool.

Overview : It has two components. kernel and user space.

Blktrace uses tracepoints mechanism for its kernel side implementation. Tracepoints are static probe points that are located in strategic points throughout the kernel. 'Probes' register/unregister with tracepoints via a callback mechanism. The 'probes' are strictly typed functions that are passed a unique set of parameters defined by each tracepoint.

From this simple callback mechanism, 'probes' can be used to profile, debug, and understand kernel behavior. There are a number of tools that provide a framework for using 'probes'. Tracepoint are exposed via the debugfs interface. Advantage of this mechanishm is that any particular probe can be activated or deactivated. These probes then write to a relayfs filesystem, which acts as a ring buffer, with the producer consumer analogy where the user thread continuosly reads from the buffer. It is efficient as being a per cpu thread and file.

3.2 Modification

One of the motivation for this project is to be able to collect different parameters that could be used as a hint by IO reduction mechanisms to detect similarity in IO. Thus filename and offset of a read and write become important. For example based on observing the filename access pattern of 2 vms, a VM placement manager could place those 2 vms on the same physical machine. An IO reduction mechanism like Seacache could benefit more from such placement. A study in the past from NFS also reveals similar finding

Write request coming from page cache or read request coming from the user or prefetching are translated to certain call from the VFS to the particular file system. We use ext3 for our purpose. For any write call, we look at the in-memory dentry structure to obtain the filename for the file. We maintain all necessary locks to ensure that the dentry is not cleaned from beneath. Similarly we obtain the offset for the write in the file. This offset can later be combined at the block layer to obtain the actual size of the request. To simplify this we modify the following path.

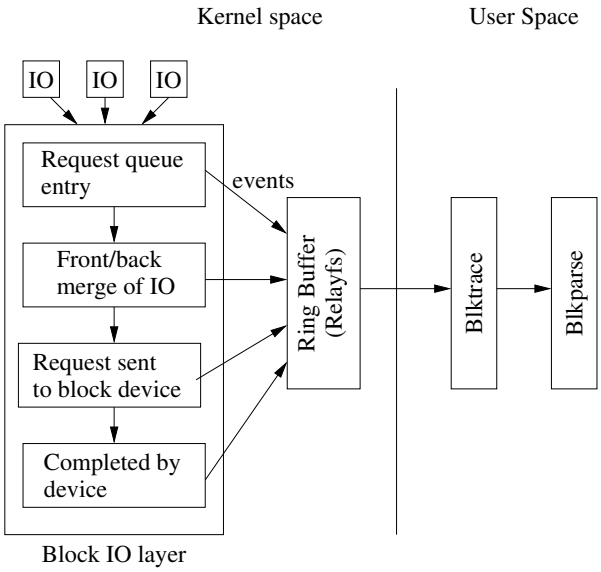

Figure 3.2: Blktrace architecture.

Life of a block request : where all could it be captured, what all does it capture

Algorithm 1 Pseudo Code for blktrace modifications.

Kernel:
For every IO request received at the file system
if IO request cannot be completed by page cache **then**
  copy filename, if available, for the request
  copy file offset, if available, for the request
  submit request to block layer
  On request entering the queue
  if request == WRITE **then**
    create a *blk_io_trace* struct, with fs and disk attributes, and IO contents
    write the struct on the ring buffer
  end if
  if request == READ **then**
    create a *blk_io_trace* struct, with only fs and disk attributes
    write the struct on the ring buffer
  end if
  wait till request is completed by the disk
  On request completed by disk
  if request == READ **then**
    create a *blk_io_trace* struct with only io contents
    write the struct on the ring buffer
  end if
end if

User space:
Read the ring buffer continuosly
For every struct
if request == WRITE **then**
  parse the struct and generate MD5 hash of the IO contents
end if
if request == READ **then**
  parse the struct and note the unique sequence number for this request
  wait till another struct with same sequence number is found
  merge the two structs, to obtain fs attributes and IO contents
end if

The *blk_io_trace* struct is written to a ring-buffer, which is exported to the user space as file, as explained above. The struct has a fixed size part and a variable size. Fields like filename and IO contents are variable. The user space can detect the actual end of the struct by looking at the size field in the fixed size part.

```
struct blk_io_trace {
        u32 magic;              /* MAGIC << 8 | version */
        u32 sequence;           /* event number */
        u64 time;               /* in microseconds */
        u64 sector;             /* disk offset */
        u32 bytes;              /* transfer length */
        u32 action;             /* what happened */
        u32 pid;                /* who did it */
        u32 device;             /* device number */
        u32 cpu;                /* on what cpu did it happen */
        u16 error;              /* completion error */
        u16 pdu_len;            /* length of data after this trace */
        // new attributes added
        u32 page_offset;        /* page offset within the file where this io op
        u8 filename_len;        /* length of the name of file */
        u32 iocontent_len;      /* length of the io data after this trace */
        // variable length filename and iocontent follows
};
```

3.3 Sampling

An in-depth tracing requires collecting all the data pertitent to an event. Thus for an IO request, it incurs the overhead of copying the entire contents of the IO request. This overhead might not be amenable to the current workload, particularly if the workload is IO-intensive i.e issues a lot of IOs. One mechanism we can employ is to trap only certain requests and collect their details. Thus we compromise completeness of the trace to be able to control the perfomance hit done by the monitoring. We use a uniform sampling to decide which requests to trap, the principal advantage is that we are still able to retain the original IO pattern. We experimentally show that certain properties like read/write request ratio, read/write data ratio is retained in the sampled trace.

3.3.1 Detail

We implement uniform sampling by introducing a sequence number. At the start of tracing, a sampling rate is decided. For every IO request trapped, we increment the sequence number. However if the sequence is only a certain multiple, we actually copy the request details and data. Thus for the non-sampled requests the overhead is only of incrementing the sequence number, which is very low. There are certain nuissance to this: i.e since there are many request sizes, it is quite possible that the average sampled requests are much bigger than the

average of whole corpus. For the applications, we wanted to test on this turned out to be work till a certain sampling rate. For most of the applications, the sampling rate will need to be tuned so that the actual sampling is done bearing that in mind.

Algorithm 2 Pseudo Code for sampling modifications.

```
Kernel:
For every IO request received at the file system
if IO request cannot be completed by page cache then
  copy filename, if available, for the request
  copy file offset, if available, for the request
  submit request to block layer
  On request entering the queue
  if request == WRITE and sequence number is 0 modulo sampling rate then
    create a blk_io_trace struct, with fs and disk attributes, and IO contents
    write the struct on the ring buffer
  end if
  if request == READ and sequence number is 0 modulo sampling rate then
    create a blk_io_trace struct, with only fs and disk attributes
    write the struct on the ring buffer
  end if
  wait till request is completed by the disk
  On request completed by disk
  if request == READ then
    create a blk_io_trace struct with only io contents
    write the struct on the ring buffer
  end if
end if
```

3.3.2 Experiment

To measure the effectiveness of the sampling we compared them against an unsampled data. We calculated the read to write ratio and read-write size ratio. Our results show that depending on the sampling rate, we can get very similar numbers between the two.

3.3.3 Evaluation

More details of the experiment setup in in subsequent chapter.

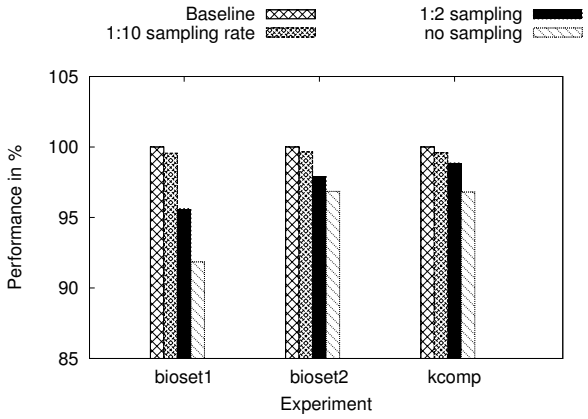

Figure 3.3: IO Monitoring Overhead with real applications.

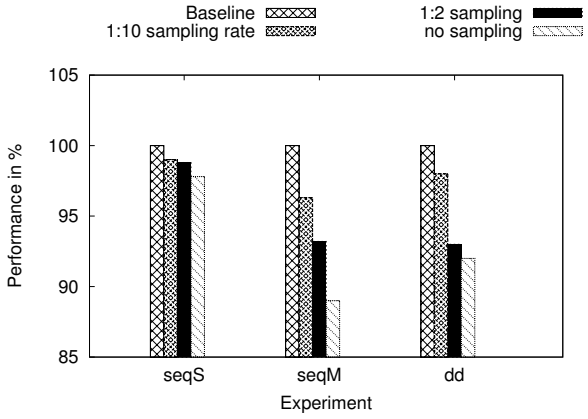

Figure 3.4: IO Monitoring Overhead with micro-benchmarks.

Chapter 4

Application

Traces can be used in a variety of ways. It can either be online, or offline. While the first one is useful for adapting your system based on user workload, the second can help in detailed analysis, debugging and security. These traces are analyzed, the user pattern and the system response is understood. If used online, these traces become input to the system being optimized. One novel way we have used the system is to use as a continuous monitoring process. The tracer continues to trace the IOs and send them over to a server, which analyzes the data access pattern. In the next sections, we will provide motivation for a problem where tracer is a crucial component, and describe our overall approach. We will provide brief background and short outline of the system, focusing only on the aspects related to the IO tracing.

4.1 Introduction

In recent years, virtualization has attracted significant attention due to its capability to act as the core enabling technology for virtual desktop environments (VDE) and cloud computing. VDE significantly eases lives of both administrators and users; it allows administrators to manage desktops in a centralized manner, and users access a desktop environment through thin-clients. The main advantage of VDE is it enables enterprises to drastically lower the hardware and operational costs. VDE also provides flexibility of dynamic workload management and secure remote access to an enterprise desktop environment.

Within the VDE, the storage infrastructure is typically realized using shared storage box, that offer management features preferred by system administrators [23]. However, the virtual desktop deployment suffers significant capital costs from this storage. In order to support the high scalability enabled by virtualization technology, the shared storage must also be scalable. Moreover, the shared storage should also be able to support peak load such as boot storms, login storms, virus scan storms. For example, boot and login storms usually happen

at 9 am on weekdays, virus scan storms happen at 3 am [14]. While the average IOps from light users are usually 8-10 IOps and from heavy end users is 14-20 IOps, the login process generates 90-100 IOps per end user on average, which is around 5 times higher than the load from a heavy end user [9].

Researchers and vendors have observed that there is a lot of duplicate data within VDEs, since the virtual images usually are created using the same golden images and the virtual desktops typically install the same set of applications such as anti-virus software and web browsers. Based on this observation, the I/O reduction techniques including dedup-box [8]; atlantis ILIO [7]; Capo [23]; seacache [14] have been proposed to reduce the duplicated I/O load from the shared storage system, and hence improve the storage efficiency.

The effectiveness of all the above-mentioned techniques depends on the amount of duplicated data accessed by VMs running on the same physical hosts. While VMs are usually placed and managed by a centralized VM manager, suboptimal VM placement can lead to reduced (or preclude) common data accesses by VMs on the same physical host, and thus results in less I/O reduction. For example, a naive VM placement algorithm that places virtual desktops belonging to employees from payroll department and virtual desktops belonging to employees from software development department indistinguishably reduces opportunity to detect common accesses and is not a good idea. In contrast, placing virtual desktops of payroll department on one set of physical hosts separately from those of software development department offers better reduction in I/O.

To best leverage the I/O reduction techniques, we propose *SMIO*, a virtual machine placement technique based on the I/O similarity among VMs. The novelty of *SMIO* is to detect I/O similarity among different virtual machines, utilize hierarchical clustering to produce a new I/O similarity aware VM placement scheme, and migrate the VMs correspondingly when benefits of such consolidation outweighs migration cost. We define I/O similarity as the ratio of common data block accesses to the total unique data block accesses among all virtual machines during an epoch. *SMIO* complements and improves the efficiency of I/O reduction techniques and reduces the I/O load on the shared storage system, which in turn reduces the cost of the shared storage system while sustaining higher performance.

4.2 Background

4.2.1 I/O Reduction Techniques

Various I/O reduction techniques have been proposed to reduce the pressure on the shared storage system in virtualized environment and are complementary to *SMIO*. Capo [23] leverages the fact that most of VM disk images are the linked clones from a small set of "golden images" and uses a bit-map to eliminate duplicate read requests. It also utilizes the host-side cache to reduce the number of I/O requests to the shared storage system. The VM images

are cached in each host locally, the redundant reads/writes to the same block from virtual desktops residing on the same host are served by the host cache instead of the shared storage system. However, Capo cannot detect duplicate reads outside of golden images or handle duplicate writes.

Taking the idea a step further, SeaCache [14] integrates host-side cache with storage-side deduplication. It eliminates not only the I/Os from the same logical block from the same host, but also the I/Os from the same host to different logical blocks but with the same data contents. More specifically, if a data block is already stored in the content addressable host cache, the access to this data content from the same host will not be seen by the shared storage system.

Clearly, at each host, the more accesses to the same data contents, the bigger savings the IO reduction techniques like SeaCache can bring. The goal of our work is to maximize the power of such I/O reduction techniques by placing the VMs with similar I/Os on the same physical host.

SeaCache Overview

In this section, we discuss how I/O reduction techniques help improve the scalability and performance of storage system in virtualized environment and why the efficiency of I/O reduction technique depends on the I/O similarities of VM workloads on the same physical host. We focus on discussing SeaCache [14] which we use as the our underlying I/O reduction approach.

In this work, we use SeaCache [14] as the underlying I/O reduction approach. SeaCache proposed efficient data transfer protocol between VMs and storage systems, which is integrated with host side caching and storage side deduplication holistically. SeaCache reduces the I/O requests sent from VMs to storage system by detecting and removing redundant content transfers.

The system is composed of a deduplication engine in shared storage system, a content addressable cache [12] inside the hypervisor at each host side, a content sharing protocol between the hosts and the storage for I/O reduction and a cache-tracker in the storage system which keeps track of the host cache contents.

The content addressable cache maintains a mapping of data blocks and the corresponding hash value, which enables the hypervisor to detect duplicate reads, and store only one copy of data in the host hypervisor cache. Using content addressable cache makes the hypervisor cache much more efficient in that the same physical memory now can save much more data blocks.

The deduplicated storage system maintains a mapping of each data block and its hash value. It supports the hash value access of any data blocks stored in the system efficiently. The

basic protocol of read path first asks for hash value of the data from storage system before the data is actually read from storage. If there is already one copy of the data in the hypervisor cache, there is no need to get the data block from the storage server again. Otherwise, the read is performed the same as traditional read request. For the write path, before the data is written back to storage system, the hypervisor computes the hash value of the written data blocks, asks the storage system if it has the blocks stored. If so, there is no need to send back the actual data. Otherwise, the data blocks are written to the storage system as in traditional approach.

It is straight forward to see that the more data blocks with the same content accesses by VMs on the same physical host, the higher I/O reduction ratio will be achieved by a I/O reduction technique, e.g., SeaCache. This serves as a motivation for *SMIO* to place virtual machines with higher I/O workload similarities together and significantly improve the efficiency of I/O reduction techniques.

4.2.2 Virtual Machine Management in Virtualized Environment

The virtual machines (VM) management in virtualized environment is typically handled by a centralized VM manager [6]. The VM manager maintains the global information such as VM resource allocation information, the VM location information. It also periodically receives the heart beat messages from each hypervisor, which carries information about the CPU, IO , network utilization information, etc. Based on such dynamically collected information, the VM manager computes the new VM placement and migration plan. The VM placement manager optimizes the VM placement and migration scheme based on certain metrics, such as energy consumption, CPU consumption, and network traffic while attempting to minimize the migration overhead and reduce the number of migrations required.

The reason for dynamically adjusting the VM placement scheme is the changing nature of application workloads. For example, a VM that is inactive at one point might become active, leading to an overloaded physical host. The VM placement algorithm must be capable of alleviating the hot spots [26]. Moreover, the I/O access pattern may change overtime. The set of VMs that have similar I/O accesses at one point may become dramatically different later. If the optimization goal of the VM placement is to maximize I/O similarities in the same host, static VM placement scheme can not be employed given the changing workloads.

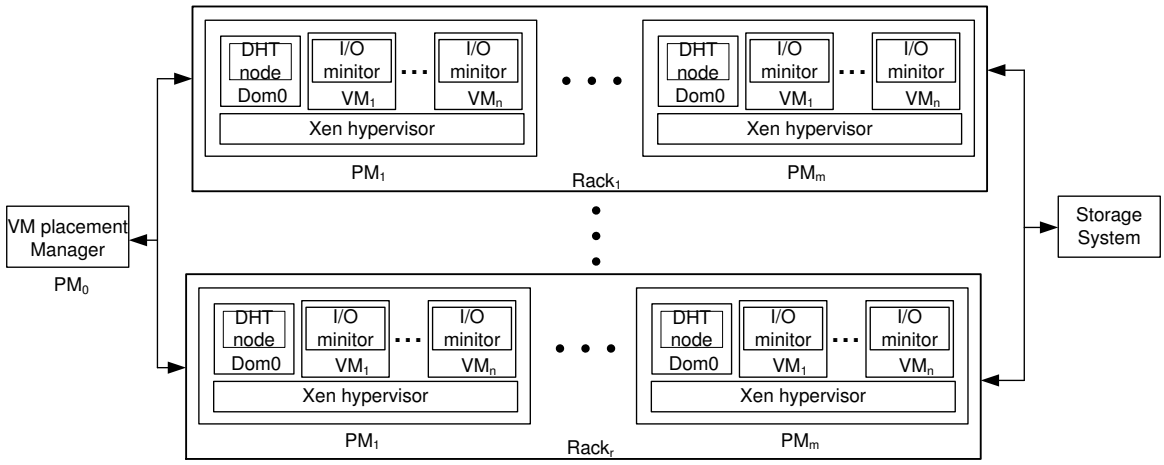

Figure 4.1: The overall system architecture.

4.3 System Design

4.3.1 Design Rationale

The goal of *SMIO* is to efficiently detect I/O similarities among different VMs, cluster the VMs with similar workloads together and place the clustered VMs on the same physical host. The system should try to do this only if the migration benefits exceed the migration cost. The system needs to periodically re-adapt to the workload changes by creating better VM placements when possible.

A basic approach is to periodically collect I/O access information from each VM and send that to a VM placement manager. Access information is collected by installing an I/O monitoring component on each VM. The monitor computes the hash value of each block being accessed by the VM using collision resistant hash mapping, such as SHA-1 [1]. It then regularly sends a list of these collected hash values to a centralized placement manager. The manager is capable of using this information and adapting to changes in workload properties, and ensuring that a placement are done on the latest workload.

The manager uses the gathered information to cluster VMs with similar accesses, which can then be assigned to a physical host together. The centralized approach faces scalalbility issues with the large number of hash values collected from thousands of VMs in a large cluster. The network capacity of the centralized manager will easily become the bottleneck, thus impacting the overall system performance. To address the bottleneck, we design a layered approach, where individual hosts collect local I/O access information, process it, and only report a summary to the placement manager. However, the approach still requires a central manager to collect the global information from all hosts to make proper VM clustering decisions. We design *SMIO* to address the above challenges. Specifically, we address the

following design goals from our system:

- Scalability: the VM placement manager should be able to scale and generate a solution for thousands of VMs in seconds, and the solution time should grow very slowly if at all.
- Low bandwidth consumption: the communication between the VM placement manager and the hosts should be minimal.
- Low overhead: the work performed at hosts and VMs for collection and processing of the I/Os should minimally impact the performance of the VMs.
- Adaptablity: the VM placement manager should be able to adapt dynamically and reconfigure the placement topology to better suit to the detected I/O workload changes.

4.3.2 Terminology

The overall architecture of *SMIO* is shown in Figure 4.1. Here, we introduce the terminology that we have used.

- Cluster: A cluster is a composite strucutre consisting of one or more VMs. A cluster can also contain a group of other clusters.
- Cluster size: the number of VMs within the cluster.
- I/O similarity: For any two clusters i, j, during a certain time interval, their I/O similarity is defined as the ratio of the number of common unique blocks accessed by both clusters (α_{ij}) to the number of total unique blocks accessed by both the clusters (β_{ij}).
- Data sharing matrix M_{DS_k}: It is defined as

$$\begin{bmatrix} - & (\alpha_{12}, \beta_{12}) & \cdots & (\alpha_{1n}, \beta_{1n}) \\ \vdots & & \ddots & \vdots \\ & \cdots & & (\alpha_{(n-1)n}, \beta_{(n-1)n}) \end{bmatrix}$$

  , where n refers to the number of clusters. M_{DS_k} represents in DHT of node k, the number of common unique blocks accessed by cluster i, j and the number of total unique blocks accessed by both cluster i, j, under a distinct hash value range taken in charged by host k.

- Global I/O similarity/ migration cost matrix M_{SC}: It is defined as

$$\begin{bmatrix} - & \gamma_{12} & \cdots & \gamma_{1n} \\ \vdots & & \ddots & \vdots \\ & \cdots & & \gamma_{(n-1)n} \end{bmatrix}$$

, where $\gamma_{ij} = \sum_k \alpha_{ij} / \sum_k \beta_{ij} - mcost_{ij}/mcost_{max}$. $mcost$ is the migration cost of cluster i, j, while $mcost_{max}$ is the maximum migration cost within all the cluster pairs. M_{SC} indicates the I/O similarity and migration cost between any two clusters $(i, j)/$. *The higher γ two clusters have, the higher overall benefit, namely high I/O reduction with low migration overhead, can be achieved by migrating the two clusters to the same host.*

4.3.3 Architecture Overview

The architecture overview of *SMIO* is shown in Figure 4.1, where it runs on the Xen platform [4]. The targeted environment comprises of a shared storage system for persistent data storage and hosts organized in racks. The shared storage system eliminates the need to migrate the VM disk image files during migration, and only requires moving the in-memory VM state. Each host supports a number of VMs and has a DHT node running in the most privileged VM (Dom0). Each VM has an I/O monitor running in its guest OS. The I/O monitor traps the application I/O accesses at block level, computes the hash values and sends it to the hosts corresponding DHT node periodically. Each DHT node is responsible for a distinct hash range. A VM placement manager runs on a dedicated host, which implements most of the intelligence of *SMIO*. The Xen hypervisor on every host receives instruction from VM placement manager for VM placement and migration. The VM placement manager collects information from DHT nodes in each host and uses hierarchical clustering [25] to generate a VM placement scheme. Hierarchical clustering is a widely used data analysis tool, which successively merges similar groups of points to create clusters of similar items. Compared with k-means [10] or k-medoids [21], hierarchical clustering does not require specification of the number of clusters k, which is an unknown in our environment.

4.3.4 Hierarchical Clustering in VM Manager

We adopt a bottom-up approach for hierarchical clustering. Each VM starts as a cluster with only itself as a member, then merges with other VMs (clusters) are performed successively until the algorithm can no longer find a suitable cluster to merge with based on the defined clustering criteria. The criteria factors in I/O similarity between the clusters to be merged and the cost of migrating the associated VMs. This is critical, as while merging clusters with high I/O similarity are preferred, the resulting migration overhead may negate the benefits. Such cases may arise for example when the two candidate clusters are far apart in terms of network distance. Once a suitable clustering plan is determined, a VM migration executor

Algorithm 3 The hierarchical clustering algorithm used in *SMIO*.

VM manager:
Epoch current_epoch;
On every t_1 minutes:
n=0;
while true **do**
  send to all DHT nodes: getDSMatrix(n, current_epoch, null);
  Gather all M_{DS} for all DHT node;
  Calculate global data sharing matrix by $\sum M_{DS_i}$;
  Calculate M_{sc};
  Sort the γ entries decreasingly;
  Group the cluster pairs which can fit into a physical host and have highest value of γ;
  if new schemes generated == false **then**
    break;
  end if
  n++;
  Broadcast the new scheme to all DHT nodes;
end while
Send the generated plan to VM migration executor.
Start a new epoch by increasing current_epoch by one and send to all DHT nodes;

DHT node:
Hashtable _ht_dht;
Epoch current_epoch;
Onreceive getDSMatrix(n, current_epoch, null) from VM manager:
Send message M_{DS} to VM manager;
Onreceive the new scheme:
Update the M_{DS} for new cluster (i, j), delete column j, row j, update column i, row i, based on the definition.
Onreceive start new epoch e' from VM manager:
Clean up the hashtable _ht_dht and the data sharing matrix M_{DS};
Current_epoch=e';
Onreceive list of block hashes from a I/O monior:
Update the data sharing matrix M_{DS};
Merge the list of block hashes into _ht_dht;

I/O monitor:
Hashtable _ht_iotrace;
Sampling I/O accesses, store it in Hashtable _ht_iotrace;
On every t_2 sec, sends calculated block hashes to DHT nodes, cleans up _ht_iotrace;

generates a migration plan aimed at minimizing the number of migrations, resulting network traffic, and migration time.

In order to cluster the VMs more effectively, *SMIO* needs to estimate the benefits gained and the migration cost incurred by grouping the VMs/clusters with similar I/O workloads. The benefit is quantified by the current I/O similarity $\sum_k \alpha_{ij} / \sum_k \beta_{ij}$ of the two clusters i, j. Calculation of the migration cost needs careful consideration. Under a shared storage infrastructure that does not require migrating the VM disk image files, the migration cost of grouping two clusters mainly depends on the allocated memories of clusters and the network distance between them. Network distance here refers to the hops required to transfer the in-memory data from one host to another. The larger the allocated memory clusters have and the longer the distance, the higher network traffic they would incur, thus leading to higher migration cost. The reason is that a typical live VM migration involves copying the memory pages from the source host to the destination host across the network. Thus, we estimate the migration cost of grouping two clusters as the smaller of the allocated memories size among the two clusters times the network distance between the two clusters. The allocated memory size of a cluster is the sum of the allocated memory size of its children and the network distance is the minimum network distance between any child pairs from the two clusters. To make the two factors comparable, we choose to normalize the migration cost by the maximum migration cost within each iteration within the hierarchical clustering. The detailed criteria will be explained later in the section.

Note that it is possible that some of the VMs do not have similarity with other VMs. In this case, other orthogonal placement algorithms [16, 26] can be used to determine the placement of these VMs, since I/O similarity does not have impact for such cases. Similarly, for VMs that are launched without any prior collected I/O information, such placement algorithms can be used to do the initial placement until the *SMIO* VM manager processes and suggests a I/O similarity-based clustering scheme. The detailed algorithm is illustrated in Algorithm 3.

In *SMIO*, the VM manager and each DHT node maintain an epoch number *current_epoch* as the local variable that is used to synchronize between VM manager and all DHT nodes. If a DHT node is out of sync, it will be excluded in the current epoch. The DHT node's epoch number is then updated to join the next new scheme calculation. At every $t1$ minutes, the VM manager start a new epoch by increasing the epoch number by 1 and launch a new round of hierarchical clustering.

Each DHT node maintains a hash table *_ht_dht* and a data sharing matrix M_{DS}. The keys in *_ht_dht* are block hashes (all falls in the specific range), and the value for each key is a list of VMs which accessed this block during the current epoch. Note that, the keys in the hash tables belonging to different DHTs do not have any overlap. It receives list of hash values from I/O monitors periodically.

At the end of each epoch, the VM manager launches the hierarchical clustering algorithm to generate a new placement scheme. The clustering criteria in *SMIO* is designed to capture both the similarity and the migration cost. With the purpose of obtaining the criteria for the

system, The VM manager first gathers data similarity matrices M_{DS} from all DHT nodes that have the same epoch number with iteration number 0. It then calculates the global data sharing matrix by summing up all the gathered data similarity matrices, followed by calculating the global I/O similarity migration cost matrix M_{SC}. Next, the manager sorts the γ values of cluster pairs in a decreasing order, and groups the cluster pairs into a new cluster, which can fit into a physical host and have γ greater than the threshold th_{sim}. The algorithm will not group two clusters together as a new cluster if the two clusters cannot fit into a physical host due to resource constraints. This makes sure that the cluster generated by the manager does not exceed the capacity of a physical host. If there are new clusters generated, the VM manager broadcasts the new generated grouping scheme to all DHT nodes. Upon receiving a new grouping scheme, the DHT nodes update the M_{ds} for each new cluster (i, j) by deleting column j, row j, updated column i, row i as described in Section 4.3.6. The VM manager then proceeds to the next iteration for the current epoch.

If no more new clusters are generated, the VM manager sends the placement plan to VM migration executor, starts and sends out a new epoch number $current\_epoch + 1$ and terminates the current algorithm iteration. On receiving the new epoch number, DHT nodes clean up their hashtables and their data sharing matrices M_{DS}.

Figure 4.2 shows an example execution of the hierarchical clustering algorithm. Initially, $VM1$ to $VM6$ each are separate clusters. After gathering the data sharing similarity matrices and calculating the global I/O similarity matrix the VM manager determines to group $(VM1, VM3)$ and $(VM2, VM5)$ into new clusters $C7$ and $C8$, which have the first two highest I/O similarity ratio greater than threshold th_{sim}. $VM4$ and $VM6$ can not be paired because the I/O / similarity ratio is lower than th_{sim}. In the second iteration, the VM manager groups $C7, VM6$ into cluster $C10$, $C8, VM4$ intro cluster $C9$. In the third iteration, VM manager finishes the current algorithm because the $C9$ and $C10$ cannot be further merged due to the fact that no physical hosts can fit cluster $(C9, C10)$.

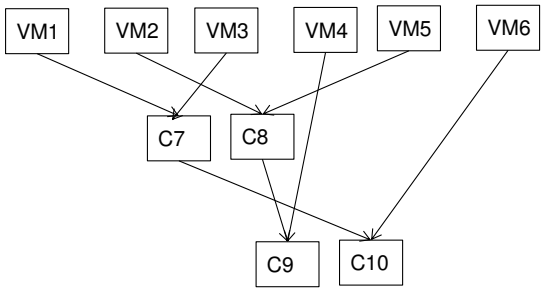

Figure 4.2: An example execution of hierarchical clustering.

4.3.5 I/O Monitor

The I/O monitor inside each VM traps the block level data, calculates the hash value of sampled blocks, stores the block hashes as keys in its hashtable *_ht_iotrace*. The value for each hash key is simply null. We use hash table to quickly identify the redundant I/O within a single VM and report only unique blocks because popular blocks within a VM should not affect the similarity between VMs.

Every t_2 second, the I/O monitor periodically sends the hashes of unique blocks accessed by the VM during $(current - t_2, current]$, i.e., the collected hash keys, to corresponding DHT nodes. The keys are sent to the DHT nodes based on hash ranges and not directly to the VM manager. This distributes the network traffic across participating physical hosts, thus avoiding saturating the network bandwidth to the VM manager.

Our implementation of the I/O monitoring process is done by modifying an existing I/O tracer for linux kernel, called blktrace, to also record the I/O content. The kernel space component of the tracer transfers the I/O events onto the userspace one, which among other things computes the fingerprint and pass that to the DHT node periodically. We would also like to mention that we chose to implement this in the VM itself, rather than doing in Dom0, as it was easy to modify an existing tool (blktrace). I/O monitoring and hash value computation can also be done in Dom0, which would be less intrusive to the VM users. That might also have lower overhead than in-the-VM approach used currently in *SMIO*.

In order to make I/O monitoring lightweight, *SMIO* samples the I/O accesses with uniform distribution [17] and only compute the hash value of sampled data blocks. The sampling greatly reduces the monitoring overhead in terms of CPU and memory utilization.

4.3.6 DHT Node Operation

DHT nodes work cooperatively with the VM manager to implement the hierarchical clustering algorithm. Each DHT node is in charge of a distinct range of block hash values. Assuming the block hash values are uniformly distributed, the work will be evenly distributed among the DHT nodes. DHT nodes helps in offloading the computation and network traffic from the VM manager by grouping and summarizing the data sharing information between VMs before sending to the VM manager. This greatly increases the scalability of *SMIO*.

4.3.7 Migration Execution

Once a new clustering scheme is generated, the migration executor is responsible for computing a migration plan. This plan specifies the host for each cluster. Since the hierarchical clustering may generate more clusters than the number of hosts, multiple clusters may share a single host.

One policy in the migration plan is to place an entire cluster rather than part of it in a host if possible, since the purpose of this work is to put VMs with similar IOs together. Under this condition, we try to minimize the migration cost. Computing a migration plan that minimize the migration overhead is NP-hard, because the wellknown NP-hard multi-dimensional knapsack problem can be reduced to it. Therefore, we design a greedy heuristic algorithm to determine the migration plan.

The main idea of this greedy algorithm is that initially each host has zero VMs or clusters assigned. Then the algorithm picks a cluster i and assigns it to a host j, based on the benefit of the cluster i can bring and the migration cost to place cluster i to host j, if the resource requirement does not exceed the physical limit. Next, we describe how to pick a cluster and a host in details.

The benefit of cluster i is μ_i. μ_i represents the number of block accesses can be saved in current epoch if its member VMs are placed together compared with each member VM is placed on different hosts. If each VM is placed separately, the unique blocks accessed by each VM will be requested from the storage server once. The subsequence accesses to an unique block from the same VM will be satisfied by the host cache. If VM 1 accesses block 1 at the first time, block 1 will be requested from storage. After that, the accesses to block 1 from VM 1 will hit the cache, but the accesses to block 1 from VM 2 will still go to storage since VM 2 is in a different host. Thus, the total accesses to the storage server from all separately placed VMs will be the sum of the unique block accesses from each VM. If the VMs in a cluster are placed together, only the unique blocks accessed by the cluster will be requested from the storage server once. The subsequence accesses to an unique block from the same VM or different VMs in this cluster will be satisfied by the host cache. For example, VM 1 is the first one in the cluster to access block 1, and the request to block 1 goes to storage server. After that, the access to block 1 from VM 1 or other VMs in the cluster will not go to storage server. Thus, the total accesses to the storage server from all VMs in the same host will be the total number of unique blocks accessed by the cluster. Therefore, the benefit μ_i is defined as the sum of the unique block accesses from each VM within cluster i subtracted by the total number of unique blocks accessed by cluster i.

The migration cost of assigning cluster i to host j is υ_{ij}. It is the total cost of moving all its VMs which are not in host j to host j. The cost of moving a VM from host l to host j is defined as the memory size of the VM multiples the network distance between host l and j. To put the resource constraints into the picture, if host j does not have adequate resources to fit cluster i, the migration cost is set to ∞.

Combining both factors, we define the benefits-costs metrics of assigning cluster i to host j as:$\tau_{ij} = \mu_i * S_{block} - a * \upsilon_{ij}$, where S_{block} is the storage block size, a is a parameter used to adjust the weight between benefit and migration cost. For clusters with cluster size greater than one, τ values smaller than zero means the benefit is less than the migration cost, the corresponding assignment is unqualified; if the cluster in its entirety are residing on host j, then $\tau = \mu * S_{block} >= 0$. Clusters with cluster size one have $\mu = 0$ and $\upsilon = 0$ thus $\tau = 0$ if

they are on host j, $\tau < 0$ if not . Next, we describe the algorithm in

To decide to pick which cluster and place it to which host, the algorithm maintain a sort list of τ values in decreasing order. Unqualified assignment with negative τ for clusters with cluster size greater than one is not in the sorted τ list. the algorithm picks the highest τ, assigns the corresponding cluster i to host j. The move is feasible because if host j does not have sufficient CPU, network, memory resources to fit cluster i, the τ value is $-\infty$. The affected τs are updated to reflect the placement decision just made. Specifically, the τ_{kj} are set to $-\infty$ thus removed from τ list if any unassigned clusters k can not fit on to host j. The τ_{ik}s of cluster i are deleted from τ list for each host k. The algorithm repeats the process until not positive τ values in the list. At this point, the unassigned clusters with cluster size greater than one will not be clustered. These clusters are then break into clusters with size one, the corresponding τ values are updated and inserted into the τ list. The process is repeated until all clusters are placed.

After the new placement topology is generated, the migration executor needs to make sure that the new calculated placement topology actually outperforms the current placement topology before the actual migration. This is done by comparing the total benefit (amount of accesses saved) by changing from current placement to new placement with the total migration cost of this change. For the comparison, we compute μs for each physical host of new placement topology, the μ's for each physical host of current placement topology, the migration cost υs for each VM needed migration. The benefit-cost metrics Φ here is $\sum_{i \in P}(\mu_i - \mu'_i) * S_{block} - a * \sum_{i \in Mset} \upsilon_i$, where P is the set of physical hosts, $Mset$ is the set of VMs required migration.The cluster for μ_i of host i is the VMs assigned or residing on host i. If $\Phi > 0$, then the new calculated placement topology have high possibility to yield better performance than the current solution after migration. Otherwise, the current migration plan is abandoned without changing the placement topology at this round.

If the algorithm decide to execute the migration plan. The output of the algorithm is a list of clusters and the new destination host for each; the migration is triggered after the algorithm is terminated. Note that it is common to have clusters with cluster size one stay on the same host according to the computed migration plan.

4.4 Evaluation

We use trace driven simulations to evaluate the effectiveness of *SMIO*. In this section, we first describe the traces we collected, then we give details of the simulator, followed by a description of the experiments conducted.

Table 4.1: Workload characteristics.

Workload	Training center	Biobench1	Biobench2	TestDev
Similarity	medium	medium	varied	strong
Duration	2.5 hours	1.5 hours	46 min	36min
Read	28.9G	419G	407.7G	47.9G
write	30.5G	29G	10.2G	16.3G
# of requests	201K	495K	456K	4.1M
# of clients (VMs)	280	12	12	8

4.4.1 Methodology

Workloads

We collected and used four different traces for our experiments, which are summarized in Table 4.1. The traces are classified into different similairty level, namely strong, medium and varied, based on the I/O accesses they exhibit. Strong similarity means different clients (VMs) have higher possiblity to access the same data contents in a relatively small time frame, whereas varied similarity means different clients have lower possiblity to access the same data contents or access the same data contents at very different time. Within the trace file, for each I/O access we collect the type of I/O (read/write command), the timestamp, the IP address of hosts, the file name, the offset and the size of the I/O and a list of hash values computed from actual data.

Training center traces: Running VMs within a training center is another common usecase that presents similar I/O workloads. For example, in a TOEFL English Test training center, all the classes have same time durations of typically 45 minutes. The first class usually begins at 8am with 4 classes packed in the morning. Within a class section, VMs owned by each student is likely to show similar I/O workloads. for example, VMs in a listening test section are going to retrieve the same audio file as students are instructed by the teacher to listen to a particular content. VMs in a speaking test section retrieve same spoken instructions but write different I/O contents to the shared storage as the audio recorded from different students will be different. We collect the traces of a total of 280 students within 5 listening sections and 2 speaking sections. Each section comprises of 40 students. The VMs from the listening sections present strong similarity correlation, while the VMs from the speaking section show weak similarity correlation. The 7 sections begin at the same time with a total duration of 2.5 hours.

Bioinformatics benchmarking traces: These traces capture a typical scenario within scientific research centers such as national labs or university labs where users perform bio-

informatics related research. Users in this case usually focus on the research of a particular DNA and protein, and run search queries against corresponding databases. We use Blastto demonstrate such a usecase and collect the traces. Blast is a widely used DNA/protein sequence searching application. In our setup, three databases are used: NR with size of $17G$, NT with size of $14G$, and HTGS with size of $6G$. Our bio-benchmark 1 (biobench1) has 4 clients running on queries against each database with a total of 12 clients. Clients searching against the same database run queries with different parameters representing the case that users are tweaking the parameters. Our bio-benchmark 2 (biobench2) has a similar setup except that clients searching against the same database run a set of 4 queries in different orders representing the cases that users are collaborating on research of same types of protein or nucleotide sequences. The similarity between VMs are varied in these traces.

Test and development traces: A typical scenario is enterprise level test development environment where users usually continuously 1) develop/ edit codes, 2) compile codes, 3) install builds, and 4) conduct QA activities. Within an enterprise, different departments might be responsible for developing and testing different products and different teams within a department might be responsible for developing and testing different features of the same product. The group of VMs that test different features of the same product will typically exhibit strong similarities since the majority of code base is the same. We setup such a test-dev environment and collect the traces. More specifically, there were four VMs for developing and testing linux kernel version 2.6.32.15 and four VMs for developing and testing Xen 4.2. The 8 VMs read $47.9G$ data and write $16.3G$ data in total.

Simulator Design

Trace driven based simulation allows us to explore a variety of configuration spaces and the scalability of our system. We developed our simulator based on the one used in SeaCache. SeaCache is a simulator that simulates I/O behavior of VMs and hypervisors under a share storage architecture that is responsible for computing the average I/O latency and I/O reduction between VMs and storage server. Our simulator implements all the components shown in Figure 4.1 except the I/O monitor because we collected the traces offline. Particularly, the simulator consists of a DHT node, hierarchical clustering component and migration execution component. The hierarchical clustering component takes traces and configured system parameters as input and generates clustering schemes that are fed into the migration execution components. The migration execution component then computes the actual placement of clusters, and executes the migration plan by instructing the SeaCache simulator to change the placement of VMs.

We assume each VM has the same cache size of $1G$, and the storage server cache size is $4G$. The parameters used in the following experiments are: threshold th_{sim} of 0.2, decision interval of 90 seconds, and no sampling unless mentioned otherwise.

4.4.2 Effectiveness of *SMIO*

In our first set of experiments, we use our simulator to show the effectiveness of our system *SMIO* by comparing with First Fit Decreasing (FFD) [15] placement, the best and worst placement technique under test development, training center and biobench workloads. FFD is a greedy approximation algorithm designed for multi-dimensional bin packing problem, which attempts to place the VMs in the first host that can accommodate the VM. The order of hosts are sorted according to network architecture initially but fixed in all the algorithm runs. Particularly, hosts within a rack are neighbors in the host list. The placement of VMs is processed in the arrival order. The best/worst placement is the best/worst placement policy that yields the best/worst performance under different traces, which in general consumes the least/most I/O bandwidth between storage server and hosts. We obtained the best/worst placement manually to the best of our knowledge of the traces used.

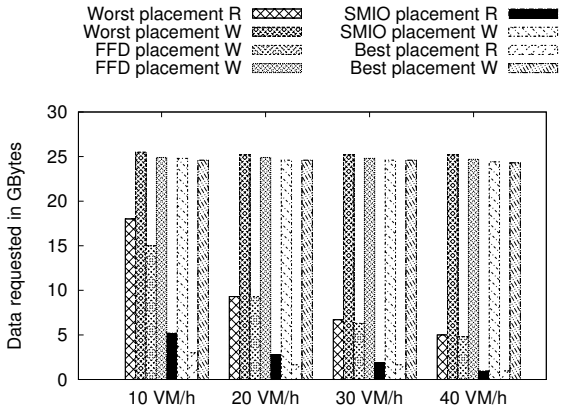

Figure 4.3: I/O bandwidth consumption under training center trace.

Training center trace Figure 4.3 shows how much data is transferred between storage server and 280 VMs for training data center trace under different number of VMs per host. The four groups of bars in the graph are 10 VMs/host with 28 hosts in total, 20 VMs/host with 14 hosts in total, 30 VMs/host with 10 hosts in total and 40 VMs/hosts with 7 hosts in total. With each group, the I/O bandwidth between hosts and storage servers are illustrated under FFD placement, *SMIO* placement and best placement policy. We observed that the different placement policies significantly impact the I/O bandwidth consumption between hosts and the storage server. As we can see, *SMIO* can effectively detect the similarity between VMs belonging to different sections and yields low I/O bandwidth consumption comparable to best placement in all cases. On average, the read path performance of FFD is 4.9 times worse than *SMIO*, while write path is only 2.1% worse. The reason is that the students in each listening section of training center trace listen to same materials most of the time during the section. This leads to high similarity of read workloads within each

listening section. On the other hand, the students in speaking section listen to instructions intermittently and record their speeches most of the time during the section, which results in nearly zero similarity for the write path I/O traffic. As the number of VMs per hosts increases, the I/O bandwidth consumptions are all reduced for all placement techniques.

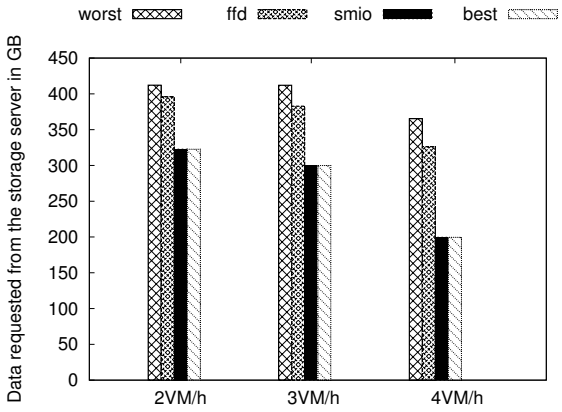

Figure 4.4: I/O bandwidth consumption under biobench1 trace.

Biobench traces Figure 4.4 and Figure 4.5 show the I/O bandwidth consumption for biobench1 and biobench2 traces under different placement policies and different number of VMs per host, namely 12 VMs distributed evenly on 3 hosts, 4 hosts and 6 hosts. In both traces, *SMIO* achieves almost the same I/O consumptions as the best placement policy. In biobench1, the I/O consumptions of worst placement policy is 1.27, 1.37 and 1.87 times worse than *SMIO* for the three number of hosts considered, while the FFD placement is 1.22, 1.27 and 1.63 times worse than *SMIO*. In biobench2, the I/O consumptions of worst placement policy is 1.52, 1.51 and 1.97 times worse than *SMIO* for the considered scenarios, while the FFD placement is 1.35, 1.31 and 1.95 times worse than *SMIO* for the corresponding scenarios. It is observed that *SMIO* again effectively detects the similarity between VMs working on different data sets and groups the VMs correspondingly. Here we do not show the I/O consumption of read and write path separately because the traces are read-intensive with negligible write traffic.

Test development trace Figure 4.6 shows similar results that *SMIO* effectively detects the I/O similarity and achieve the performance as the best placement. To see how the migration scheme is helping save the I/O consumptions, we plot the I/O consumption over time using test development trace. In the simulator, we record the I/O and print out the value every 20 seconds. Figure 4.7 illustrates that after monitoring the I/Os for the first 5 minutes, *SMIO* decides to migrate VMs for new placement and the I/O seen by storage server are consistently less than the I/O consumption before migration. Moreover, *SMIO*

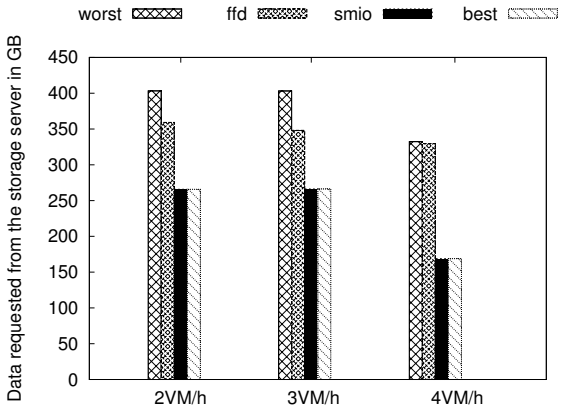

Figure 4.5: I/O bandwidth consumption under biobench2 trace.

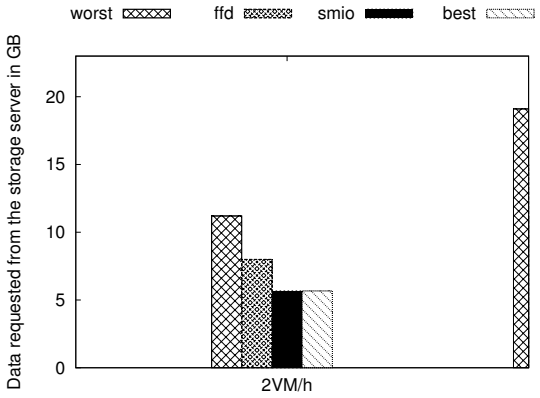

Figure 4.6: I/O bandwidth consumption under test development trace.

helps to reduce the peak bandwidth requirement by 33%. In total, *SMIO* reduces the I/O consumption by 74% compared to the base case.

4.4.3 Parameter sensitivity analysis

The performance of *SMIO* depends on selection of various thresholds and system parameters, such as th_{sim}, t_1 and t_2, which needs to be chosen carefully. Lower benefit-costs threshold th_{sim} indicates that more clusters would be paired up within each iteration, which results in fewer iterations and faster algorithm convergence. However, it may miss better pair-up opportunities. For example, cluster $3, 4$ might be paired up in n_{th} iteration with lower th_{sim}, which misses the opportunity to pair cluster 3 with cluster $1, 2$ in $n + 1_{th}$ iteration. On the other hand, if th_{sim} is set too high, it would increase the number of clusters with

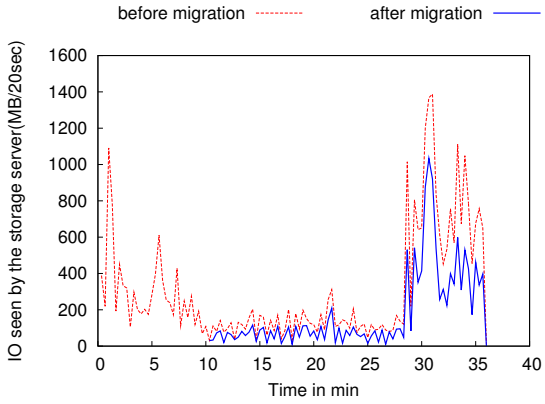

Figure 4.7: I/O bandwidth consumption under test development trace.

size 1 and deteriorate the quality of generated cluster scheme. In terms of t_1, if it is set too low, unnecessary resources will be wasted. To generate a new VM placement plan, network bandwidth will be consumed for VM manager to communicate with DHT nodes not to mention the computing cycle and memory space utilized by Algorithm 3. If t_1 is set too high, the potential I/O optimization opportunities will be missed. On the other hand, interval t_2 decides the length of the package delivered to corresponding DHT nodes. Small t_2 would lead to small packages with larger network package header overhead, whereas large t_2 would lead to big package which would get large memory overhead since I/O monitors have to buffer them before sending to DHT nodes.

To quantify the quality of a generated cluster scheme, we use the Φ as discussed in Section 4.3.7. The experiments in this section are conducted with 40 VMs per host on 7 hosts. The results show a general guideline for picking th_{sim}, t_1 and t_2.

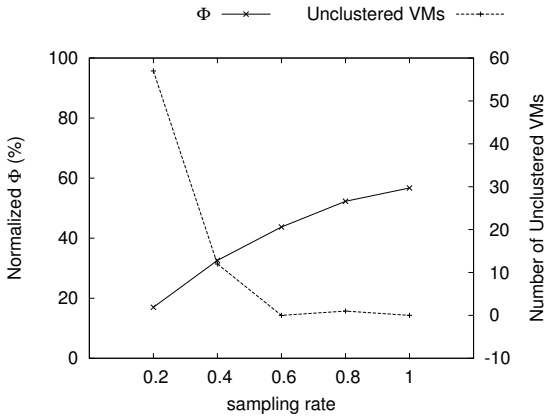

Figure 4.8: Sampling rate and similarity score.

Impact of sampling rate

The next experiment demonstrates the relationship between sampling rate of I/O monitor and the similarity score and the number of clusters with size 1 under hierarchical clustering. As expected, Figure 4.8 shows that the similarity score positively relates to the sampling rate, the higher sampling rate yields higher similarity score. This provides a trade-off for users to adjust. The sampling rate can be dynamically increased when the host has idle system resources and decreased when the VMs use up the system resources. On the other hand, the number of clusters with size one keeps decreasing as the sampling rate keeps increasing until to 0.6, which suggests a sweet configuration spot under this setup.

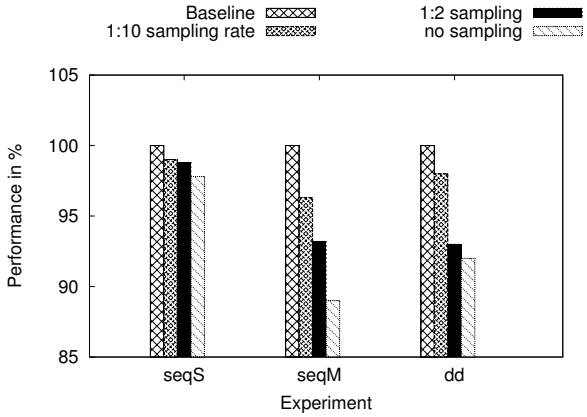

Figure 4.9: Overhead of the I/O Monitoring component.

4.4.4 System Overhead and Scalability

Monitoring overhead. In this experiment, we measure the overhead of our I/O Monitoring component under different sampling rates, including without monitoring (baseline), 1 : 10 sampling rate, 1 : 2 sampling rate, and monitoring all I/Os (no sampling). The performance (speed) is normalized to the baseline performance. Since our monitoring component kicks in only when there are active I/Os happening to the disk, we test some I/O intensive and I/O-CPU intensive applications. The first experiment is sequential read of a 5GB file (seqS) under various sampling rates. The experiment shows that even with no sampling, i.e., tracing all the I/Os, the observed read speed decreases by less than 3%. The same is not true for the multi-threaded (2 threads) version of this experiment (seqM), where we see how higher sampling rate helps to keep the performance hit in check. For our third experiment, we run the unix utility `dd` to copy a 10GB file. Again, as this is more "I/O-intensive" than the first experiment, we see a more decreased transfer speed (in terms of MB/s). Finally, we run the task of Linux kernel compilation, which is both computation and I/O intensive task. We

observe that . These results show that the overhead of the I/O monitoring can be kept low with an appropriate sampling rate, thus *SMIO* offers a feasible and practical approach to managing VMs.

Chapter 5

Bibliography

[1] F. 180-1. *Secure Hash Standard*. U.S. N.I.S.T. National Technical Information Service, Springfield, VA, Apr. 1995.

[2] A. Aranya, C. P. Wright, and E. Zadok. Tracefs: a file system to trace them all. In *In Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004*, pages 129–143. USENIX Association, 2004.

[3] M. G. Baker, J. H. Hartmart, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a distributed file system, 1991.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[5] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer. Passive nfs tracing of email and research workloads, 2003.

[6] F. Hermenier, X. Lorca, J.-M. Menaud, G. Muller, and J. Lawall. Entropy: a consolidation manager for clusters. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 41–50, New York, NY, USA, 2009. ACM.

[7] http://www.atlantiscomputing.com/. Atlantis computing, June 2006.

[8] http://www.riverbed.com. Riverbed steelhead family overview, June 2005.

[9] http://www.unidesk.com/blog/local-vs-centralized-storage models. Local storage vs. centralized storage models, Aug. 2012.

[10] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu. An efficient k-means clustering algorithm: Analysis and implementation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 24(7):881–892, July 2002.

[11] R. Koller and R. Rangaswami. I/o deduplication: Utilizing content similarity to improve i/o performance.

[12] R. Koller and R. Rangaswami. I/o deduplication: Utilizing content similarity to improve i/o performance. *Trans. Storage*, 6(3):13:1–13:26, Sept. 2010.

[13] A. Leung, S. Pasupathy, G. Goodson, and E. L. Miller. Measurement and analysis of large-scale network file system workloads. In *Proceedings of the 2008 USENIX Technical Conference*, June 2008.

[14] M. Li, S. Gaonkar, A. R. Butt, D. Kenchammana, and K. Voruganti. Cooperative storage-level de-duplication for i/o reduction in virtualized data centers. In *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2012 IEEE 20th International Symposium on*, aug 2012.

[15] S. Martello and P. Toth. *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, Inc., New York, NY, USA, 1990.

[16] X. Meng, V. Pappas, and L. Zhang. Improving the scalability of data center networks with traffic-aware virtual machine placement. In *INFOCOM, 2010 Proceedings IEEE*, pages 1 –9, march 2010.

[17] G. P. Moore, David S.and McCabe. *Introduction to the practice of statistics*. W.H. Freeman Company, Feb. 2005.

[18] J. Ousterhout, H. D. Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson. A trace-driven analysis of the unix 3.2 bsd file system. pages 15–24, 1985.

[19] D. Roselli and T. E. Anderson. A comparison of file system workloads. In *In Proceedings of the 2000 USENIX Annual Technical Conference*, pages 41–54. USENIX Association, 2000.

[20] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, Feb. 1992.

[21] L. ROUSSEEUW. Clustering by means of medoids. *Statistical data analysis based on the L1-norm and related methods*, page 405, 1987.

[22] R. R. Schaller. Moore's law: past, present, and future. *IEEE Spectr.*, 34(6):52–59, June 1997.

[23] M. Shamma, D. T. Meyer, J. Wires, M. Ivanova, N. C. Hutchinson, and A. Warfield. Capo: recapitulating storage for virtual desktops. In *Proceedings of the 9th USENIX conference on File and stroage technologies*, FAST'11, pages 3–3, Berkeley, CA, USA, 2011. USENIX Association.

[24] W. Vogels. File system usage in windows nt 4.0. In *ACM Symposium on Operating System Principles (Kiawah Island Resort*, pages 93–109. ACM, 1999.

[25] J. H. Ward. Hierarchical grouping to optimize an objective function. *Journal of the American Statistical Association*, 58(301):236–244, 1963.

[26] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif. Black-box and gray-box strategies for virtual machine migration. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation*, NSDI'07, pages 17–17, Berkeley, CA, USA, 2007. USENIX Association.