Towards Improved Provisioning and Utilization of Resources in Virtualized Environments

Thesis

Submitted in partial fulfillment of the requirements
for the degree of

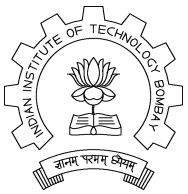
Doctor of Philosophy

by

Sujesha Sudevalayam
Roll No: 07305903

under the guidance of

Prof. Purushottam Kulkarni



Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Mumbai

Abstract

With widespread adoption of virtualization for hosting applications, service providers (like Amazon EC2 [1]) can facilitate better performance isolation, security, quicker deployment and elastic resource provisioning. Due to above benefits of virtualization, many hosting centers have transitioned from providing Hardware as a Service (HaaS) to Infrastructure as a Service (IaaS) instead. The primary difference between HaaS and IaaS is that the former involves use or leasing of physical hardware/machine whereas the latter involves leasing of virtual resources/machines.

When multiple virtual machines (VMs) are placed on a single physical machine (PM), they compete for various resources like CPU, memory, network and disk I/O and interact in many conflicting ways. In any given virtualized environment, the physical resources available can be broadly categorized into, (i) Resources allocated to the virtual machines—virtual CPU, memory, disk and (ii) Resources in the virtualized host—host CPU and cache. In this thesis, we address two important issues related to the management of both these types of resources more efficiently, towards the overall goal of optimizing the performance of virtualized applications.

The first problem of this thesis deals with managing the network usage of VMs and estimating CPU requirement on both the VM and its host PM. Since different tiers of an application require mutual network communication, *colocation* of communicating VMs on the same PM reduces physical network usage. *Network affinity* is the presence of network communication between a pair of VMs, and is *intra-PM* when the VMs are colocated, and *inter-PM* when they are dispersed onto different PMs. Thus, the nature of network affinity is *mutable* (i.e., changing between inter-PM and intra-PM) upon VM migrations. We make the case that since there is significant change in CPU resource usage when the VMs are colocated versus when they are dispersed, it is essential to capture such changes via a model, for server consolidation and VM placement decisions.

In our work, we explore the difference in CPU utilization due to *network affinity*, and propose models to estimate the changed CPU utilization. Specifically, we perform network benchmarking, which demonstrates effects of network affinity on CPU usage when VMs are colocated versus dispersed. Next, we develop VM *pair-wise* models that can estimate the "colocated" CPU usage, on being input their individual dispersed-case resource usages. We also build similar models to estimate the "dispersed" CPU usage based on the individual colocated-case resource usages. For the "colocation" and "dispersion" models, we first built models that predicted the total CPU usage upon migration—these CPU models use all resource (CPU, disk, mutable and immutable network) usage profiles as their input. However, these models had an error of around 4%. So, next we built enhanced models to predict only the difference in CPU usage—these models use only the *mutable* network traffic metrics as input, and have maximum error within 2%.

Finally, we demonstrated the application of *pair-wise* models to predict for multi-VM scenarios, with high accuracy.

The second problem of this thesis deals with managing the cache usage on a virtualized host to improve disk access performance of VMs. Due to increased permeation of virtualization-based systems, there is a lot of inherent content similarity in systems like email servers, web servers and file servers. Harnessing content similarity can help avoid duplicate disk I/O requests that fetch the same content repeatedly. In this work, we incorporate intelligent I/O redirection within the storage virtualization engine of the device to manage the underlying sector-based cache like a content-based cache.

We build a disk read-access optimization called DRIVE, that identifies content similarity across multiple blocks, and performs hint-based read I/O redirection. A metadata store is maintained and implicit caching hints are collected based on the VM's disk accesses. Using the hints, read I/O redirection is performed from within the VM's virtual block device, to manipulate the entire host-cache as a content-deduplicated cache. Our trace-based evaluation using a custom simulator, reveals that DRIVE achieves up to 20% better cache-hit ratios and reduces up to 80% disk reads. It also achieves up to 97% content deduplication in the host-cache.

Contents

List of Figures

List of Tables

Listings

List of Abbreviations

DAS: Direct-attached Storage

DMA: Direct Memory Access

Dom0: Domain zero in Xen (driver domain)

DomU: User Domain in Xen (guest domain)

DRIVE: Disk I/O Reduction in Virtualized Environments

FIFO: First In, First Out

HaaS: Hardware as a Service

IaaS: Infrastructure as a Service

iSCSI: Internet Small Computer Systems Interface

KVM: Kernel Virtual Machine

LIFO: Last In, First Out

LRO: Large Receive Offload

LVM: Logical Volume Manager

NAS: Network-attached Storage

NFS: Network File System

NIC: Network Interface Card

PM: Physical machine

SAN: Storage Area Network

SLA: Service Level Agreement

TCP: Transport Control Protocol

TSO: TCP Segmentation Offload

UDP: User Datagram Protocol

VM: Virtual machine

Chapter 1

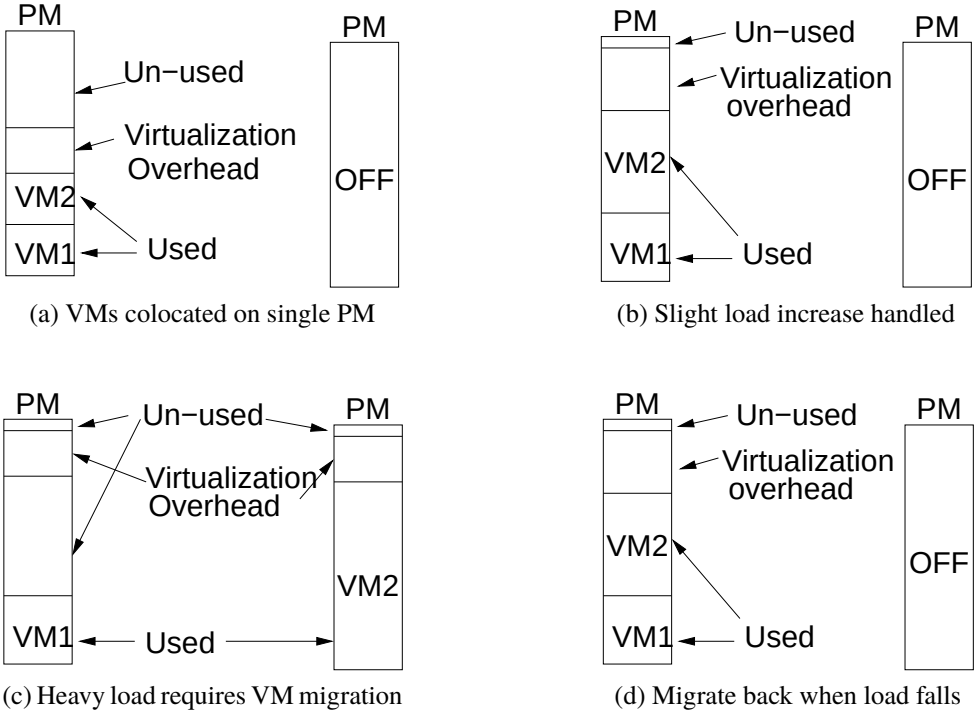
Introduction

With widespread adoption of virtualization for hosting applications, service providers (like Amazon EC2 [1]) can facilitate better performance isolation, security and elastic resource provisioning. A virtualization-based provisioning model is attractive for both providers—multiplex resources among several customers, and clients—*pay-per-use*, use and pay for only as much resource as required. Instances of both *public* [1] and *private* Clouds [2, 3] exist, which leverage virtual machines for flexible provisioning.

Several issues, some of which are—mapping of resource requirements from physical to virtual environments [4], placement policies for virtual machines [5], dynamic resource provisioning [6], runtime consolidation and migration [7], storage provisioning and access management [8] need to be addressed to provision applications in virtual execution environments. Further, these problems need to be addressed in the context of meeting service level agreements (SLAs) and resource guarantees [9], and simultaneously maximizing the resource multiplexing potential. *Server consolidation* and *dynamic resource provisioning* [5], [6], [7], [10] are virtual machine migration-enabled techniques aimed to reduce provider-side resource sprawl and to address elastic resource requirements, respectively.

Since application demands are expectedly continuously varying, resource requirements will also be correspondingly elastic [1]. To support this, virtualization-based services require automated and dynamic resource provisioning [7]. Specifically, if a physical machine faces an explosion of resource requirements, one or more of its virtual machines may need to be *migrated* to other physical machines for load balancing [11, 12] and meeting SLA guarantees [9]. Thus, *dynamic resource provisioning* is possible by scaling resources [13] on the same physical machine (when physical machine has sufficient resources to accommodate increased demands) or by migrating VM to another PM with sufficient resources (when source PM has insufficient resources).

It is widely acknowledged [14, 15] that the average utilization levels in a datacenter is around 20%, that is to say, the peak-to-average utilization ratios are very high. Typically, under periods of high load, a VM may be allocated to a single PM of its own, and when the load falls back



(a) VMs colocated on single PM

(b) Slight load increase handled

(c) Heavy load requires VM migration

(d) Migrate back when load falls

Fig. 1.1: Server consolidation and migration for dynamic resource provisioning

down, it may be moved back to another non-idle PM, such that the total resource utilization needs are fulfilled on the target PM, and no SLAs are breached of either the VM under consideration, or the VMs that were already executing on the target PM. This strategy, known as *Server Consolidation* [16, 17, 18, 19], allows under-utilized physical machines to be switched off, thereby saving power and cooling costs. This idea is illustrated briefly in Fig. 1.1.

Most web-based applications are multi-tiered and virtualization offers the possibility of hosting each of these tiers (e.g., the web-server tier, the application logic tier and the database server) on separate elastically provisionable virtual machines. Such differentiated hosting for various tiers is preferable as it enables independent resource management and administration of the different tiers. Additionally, due to elastic potential of resource allocation to virtual machines, differentiated hosting on virtual machines can help avoid resource wastage on under-utilized physical machines. When applications are instantiated in a virtual environment, the following major factors affect their performance—available network capacity, disk access bandwidth and virtualization overheads.

When multiple virtual machines are placed on a single physical machine, they compete for resources like CPU, memory, network and disk I/O and interact in many conflicting ways. In any virtualized environment, the resources available can be broadly categorized into two.

I. Resources allocated to the virtual machines. Every virtual machine has access to a set of resources, similar to those available on physical machines. For example, virtual CPU, VM page cache, virtual disk, etc. These are usually statically allocated in today's datacenters [1], but can

be better managed using dynamic provisioning techniques [7, 20].

II. Resources in the virtualized host. These are resources at the disposal of the virtualized host to support and enable the functioning of the hosted VMs. For example, apart from the virtual CPU allocated to the VMs, the hypervisor and/or Dom0 also need certain CPU allocation to handle the virtualization overheads [21]. Similarly, the virtualized host has a page cache which is used for managing the buffering of physical I/O access [22].

In this thesis, we address two important issues related to the management of both these types of resources more efficiently, towards the overall goal of optimizing the performance of virtualized applications.

1.1 Thesis contributions

This thesis addresses problems related to improved resource provisioning and utilization in virtualized environments. In this section, we concretely state our contributions.

1. Affinity-aware Modeling of CPU usage for Virtualized Applications. The first component of this thesis deals with managing the network resource usage of virtual machines and estimating the resulting CPU requirement on both the virtual machine and its host system. Since different tiers of an application require network communication with each other, placing communicating virtual machines on the same physical machine would reduce physical network usage. We define the presence of network traffic between a pair of VMs as their *network affinity*, and state that the nature of this network traffic is *mutable* (i.e., changing) based on whether the VMs are colocated or dispersed. In other words, the nature of the network traffic between a pair of VMs can change between being intra-PM and inter-PM depending on whether the VMs are hosted on the same physical machine or on different physical machines. We explore the effect of *mutable* network traffic on CPU usage of colocated and dispersed VMs. More specifically, the question is, *is the CPU utilization of mutually communicating VMs dependent on whether the communication happens intra-PM or inter-PM, and if so, how to estimate the target scenario's CPU utilization?* We make the case that there is significant change in CPU resource usage of communicating VMs when they are colocated versus when they are dispersed, and it is essential to capture such changes via a model, to assist in automated server consolidation and VM placement decisions.

We present benchmarking experiments which demonstrate impact due to network affinity on CPU usage of virtual machines and their hosts, when communicating VMs are colocated as compared to when they are dispersed. Motivated by these findings, we develop models that can estimate the "colocated" CPU resource usage when VMs transition from dispersed to colocated placements, and can estimate the "dispersed" CPU resource usage when VMs transition from

colocated to dispersed placements. These models predict CPU usage in target scenario (colocated/dispersed) based on resource usage profiles from source scenario (dispersed/colocated). First we built models to predict total CPU usage for target scenario, based on all resource usage profiles like CPU, disk, mutable network and immutable network usage. However, the maximum error with these predictions was found to be around 4 to 6% absolute CPU usage. Hence, based on our findings that CPU usage is affected only by *mutable* network traffic levels, we build models to predict the difference in (or differential) CPU usage based on only the mutable network traffic profiles. These models were much more accurate, with maximum error within 2%. Finally, we applied these pair-wise models to multi-VM scenarios using a multi-phase prediction methodology. This demonstrated that simple models built on the scale of two VMs could be successfully used to predict for multi-VM scenarios as well.

2. Using Implicit Caching Hints for Disk I/O Reduction in Virtualized Environments. This component deals with managing the cache resource usage on a virtualized host machine so as to improve the disk access performance of the virtual machines. Due to increased permeation of virtualization-based systems, there is a lot of inherent content similarity in systems like email servers, web servers and file servers. All this data resides on disk and is fetched by corresponding applications, as and when required. Typically, caches are addressed by block number (hence called *block-based* caches) and are not equipped to recognize content similarity across multiple blocks. Harnessing the content similarity can help avoid duplicate disk I/O requests that fetch the same content repeatedly. In this work, we incorporate intelligent I/O redirection within the storage virtualization engine of the device to manage the underlying block-based cache like a *content-deduplicated* cache.

We build a disk read-access optimization called DRIVE, that identifies content similarity across multiple blocks, and performs hint-based read I/O redirection to improve cache effectiveness, thus reducing the number of disk reads further. A metadata store is maintained based on the virtual machine's disk accesses and implicit caching hints are collected for future read I/O redirection. The read I/O redirection is performed from within the virtual block device in the virtualized system, to manipulate the entire host-cache as a content-deduplicated cache implicitly. Our trace-based evaluation using a custom simulator, reveals that DRIVE always performs equal to or better than the Vanilla system, achieving up to 20% better cache-hit ratios and reducing the number of disk reads by up to 80%. The results also indicate that our system is able to achieve up to 97% content deduplication in the host-cache.

1.2 Tools and deliverables

As part of the work in this thesis, we developed several tools and utilities, that we report here as "deliverables" of this thesis. For each of these tools, their motivation, requirements specification, design and implementation are discussed in the Appendix chapters.

1. `LoadGen`: This is a multi-threaded workload generator, that can be used to generate various types of worklaods like CPU, disk, network and mixed workloads at pre-specified levels. For details, refer to Appendix Chapter I.

2. `SimReplay`: This is a custom cache simulator, with extensions to look into content similarity while operating the cache. For details, refer to Appendix Chapter II.

3. `Preadwritedump`: This is an I/O trace logging and collection toolkit, consisting of different tools to perform I/O tracing in the Linux kernel and for collecting the logs from the kernel datastructures and writing into the persistent filesystem. For details, refer to Appendix Chapter III.

1.3 Thesis outline

The rest of this thesis is organized as follows. Chapter 2 presents brief background to cover the scope of this thesis. In Chapter 3, we present work related to building affinity-aware CPU estimation models for migrating VMs and their hosts. In Chapter 4, we present our I/O reduction system called DRIVE which improves the efficiency of host cache using deduplication-based I/O redirection. For further evaluation of DRIVE, we performed a detailed literature survey comprising over 100+ publications and 350+ datasets. So, in Chapter 5, we present the findings of our survey, which shows that there are no realistic I/O workload datasets or benchmarks available that captures content representation. In Chapter 6, we present some open directions and future work for this thesis, and Chapter 7 concludes.

Chapter 2

Background & Literature Review

In this chapter, we present a brief background review to cover the entire thesis scope. We present the basics of cloud computing, usage of virtualization to provide Infrastructure as a Service cloud computing, as well as the basic mechanisms of network and disk I/O virtualization.

2.1 Overview of cloud computing and virtualization

An organization which needs to host multiple services like e-mail servers, web servers, file downloading servers, e-commerce websites, etc may either opt to own, maintain and manage their own infrastructure (i.e., private clouds [2]), or alternatively, use the services provided by public hosting centers (i.e., public clouds [1]). In traditional datacenters (before virtualization), multiple servers would run as separate processes on the same machine and such mapping would exist from subsets of processes to a set of physical machines. Guaranteeing resource isolation could be possible only by hosting a single server on a single physical machine, but would cause resource under-utilization during periods of light loads.

For example, an auction website may have periods of heavy load during the day and be comparatively lightly-loaded during the night. Hence, it may be profitable to the hosting center provider to host multiple auction websites on the same machine during the night while assigning them separate machines during the day. However, such adjustments would need manual intervention or tedious automation to plan/set them up. Typically, tedious automation and/or manual intervention to provision resources based on workload levels, would be well avoided and traditional hosting centers would just provision the servers for peak load. However, such static allocation of resources to each server is wasteful and inefficient because the server is not expected to be handling peak load at all times [14, 15]. Also, with such static allocation, initial deployment of a server in the data-center will involve procurement and setup delays of the physical machines on which it is to be deployed [23].

Virtualization helps to avoid such pitfalls, by enabling dynamic resource allocation and quicker deployment of services. Instead of installing new hardware for deploying/scaling a server

in the data-center, virtualization allows transparent, on-demand deployment on a few processors in an existing multi-processor machine and avoids new machine procurement delays [23]. Virtualization allows accomodation of varying load-levels by on-demand resource allocation [24] and also helps reduce application downtime [20]. Due to above benefits of virtualization, many hosting centers have moved from providing Hardware as a Service (HaaS) to Infrastructure as a Service (IaaS) instead [1, 25]. The primary difference between HaaS and IaaS is that the former involves use or leasing of physical hardware/machine whereas the latter involves leasing of virtual resources/machines.

2.1.1 Virtualization

Virtualization is a technology that allows abstraction of network, storage and compute services, by providing a software middle-layer between the physical hardware and the applications that run on it. Hence, each physical machine (PM) can host multiple virtual machines (VMs), such that each virtual machine sees an abstraction of resources on which it executes. This makes it easy to start, stop, move or increase the number of virtual machines hosted on one or more physical machines.

To motivate the use of virtualization, let us consider a similar example. Suppose an organization needs to host fifty software applications—webservers, email servers, file servers and so on. Based on some back-of-the-envelope calculation of resource requirements, suppose twenty physical machines are procured for this purpose. In the absence of virtualization, a subset of servers each, would be hosted on each physical machine and each might be typically peak-provisioned [26]. The selection of which subset of servers can/should be hosted together on a single machine may be random or may depend on considerations like OS platform issues, licensing issues, hardware capacity, software/hardware version compatibilities, etc. Some subset of servers may even be grouped together if they have mutually exclusive resource requirements (say CPU-bound webservers and IO-bound database servers hosted together [27]) because there is a perceived high-level guarantee that neither one will encroach on the resource requirements of the other, thus providing approximate resource isolation. However, resource isolation guarantees can be provided only with kernel changes and/or real-time operating systems [28].

A user who invests in hosting services would prefer to get hard guarantees on resource and performance isolation instead of being subject to the vagaries of resource utilization of other colocated processes or applications. Moreover, in case of *public clouds*, the user would also prefer to be paying only for utilized resources and not for unused resources [26]. This is where virtualization offers a good solution. A virtual machine gives an entire machine-like resource environment with usability of any operating system. And thus, the user need only be charged for the resources being allocated to their virtual machine [1].

Virtualization allows each server process to be provided with its own resource environment and guarantees a fixed amount of resources such that its resource availability and performance

levels will not be affected by any other colocated process's resource usage. Such guarantees are possible owing to the virtualization middle-layer, or the Virtual Machine Monitor (VMM), which arbitrates communication back-and-forth between the applications and the hardware. Basically, each application gets its own (virtualized) environment, known as a *Virtual Container* or a *Virtual Machine* and it may be even unaware that its container is not a physical machine with access to real hardware i.e. virtualization is transparent to the applications/services themselves. Each virtual machine runs like a separate operating-system instance and has interfaces for direct or indirect access to the physical hardware. The OS[1] executing within the virtual machine is referred to as a *guest OS* while the original operating system on the physical machine which provides the virtualization support is referred to as the *host OS*.

Example instantiations of virtualization-based solutions are, (i) users/clients hosting their applications on a remote dataCenter and negotiating for service level agreements and (ii) enterprises hosting applications and services on a private self-managed cluster of machines, for internal and external access. In the latter example, both the user and the provider may be one and the same. Virtualization offers benefits to both users and providers of the datacenter. Traditionally, an end user would be burdened with planning, acquiring and deployment of infrastructure, and also regular maintenance [26]. The end-user would require to plan software updates and hardware upgrades as the system grows, and also manage low-level decisions to maintain performance requirements. However, with virtualization, the end user is responsible only to pay for the resources on-demand, while receiving guarantees on performance. The end-user can be agnostic of physical hardware issues. At the back-end, the service provider benefits by way of potential opportunities to effectively multiplex available resources [29], provide on-demand & scalable service [20] that is centrally managed, and cut down on expenses.

2.1.2 Virtualization techniques

There are various virtualization techniques: *full-virtualization* [30], *para-virtualization* [31], *OS-level virtualization* [32] and *hardware-assisted virtualization* [33]. Figure 2.1 shows the basic architecture of the virtualization framework. As seen in the figure, a host operating system, instrumented with the Virtual Machine Monitor (VMM) or Hypervisor, executes on the physical machine, and one or more VMs, containing guest operating systems, execute on top of the virtualization layer.

Full-virtualization: In this technique, the guest OS can run unmodified within the virtual machine. This is made possible by the use of Binary translation and Direct execution techniques [30]. *Binary translation* refers to the "on-the-fly substitution" of traditional guest OS instructions with a virtual sequence of instructions and, *Direct execution* strategy is adopted for executing user-level code. So the guest OS is completely decoupled from the underlying hardware by the virtualization layer. Binary translation is used in VMware's full-virtualization

[1] OS stands for Operating System

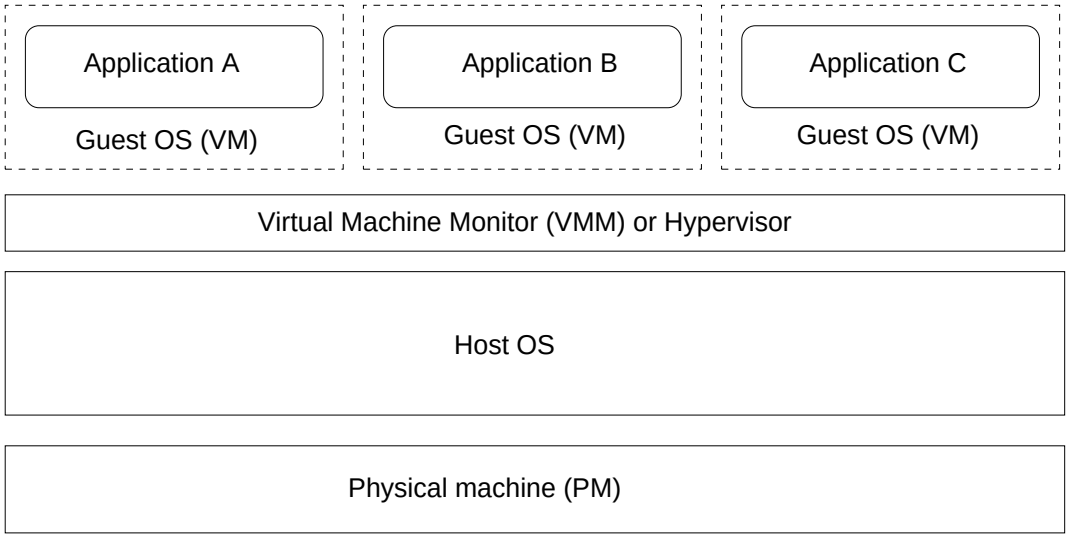


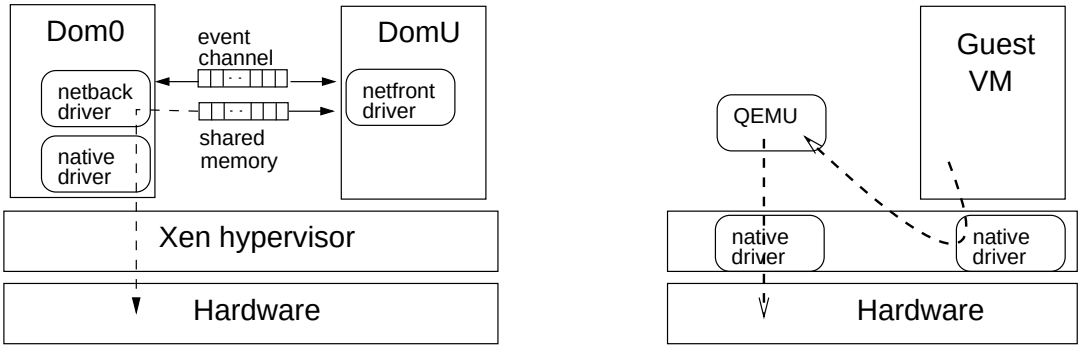
Fig. 2.1: Virtualization framework architecture: *A physical machine is virtualized using a VMM such that it is capable of hosting multiple virtual machines that have their own operating systems (OS) each. The VMM is thus, a middleware that handles and delegates the responsibilities related to virtualization of the physical machine.*

solution [30] due to challenges in virtualizing privileged operations, like I/O instructions. This is because if a guest OS is directly allowed to execute the privileged I/O instructions, it could alter the state of other guest OSes and compromise security.

Para-virtualization tries to address the concern of I/O virtualization another way, by making changes to the guest OS such that privileged I/O instructions cause traps to the hypervisor and are executed by it on behalf of the guest OS. For example, request for an I/O operation by the guest OS will be made in the form of a function call that does not actually perform the I/O operation, but merely requests the hypervisor to perform it. The hypervisor will execute the I/O after checking for requisite permissions, and will intimate the guest OS when task is over.

OS-assisted virtualization: In this technique, guest operating systems are processes that are allocated different name-spaces such that they seem to be separate machines altogether. However, in OS-level virtualization, the same host OS kernel also supports the guest processes. The advantage of full-virtualization and paravirtualization over OS-level virtualization is that they can support heterogeneous operating system distributions as guest OSes, like Linux, BSD and Windows XP [23]. Although, paravirtualization is an OS-assisted form of virtualization where all privileged instructions need to be executed by the virtualization layer, however, the difference is that para-virtualization can support different guest operating systems, while OS-level virtualization cannot.

Hardware-assisted virtualization: In this technique, the hardware is enhanced with virtualization awareness such that the CPU itself traps the privileged/sensitive instructions and emulates the instructions in hardware instead of software, hence obviating the need for binary translation or paravirtualization. Hardware-assisted virtualization is also a form of full virtualization since



(a) Xen's driver domain model

(b) KVM's direct I/O model

Fig. 2.2: Different I/O virtualization architectures—Xen and KVM

the guest OS can remain ignorant of the virtualization involved [34].

Based on the above types, there are several virtualization technologies like Xen [23], Vmware [30], KVM [33] and OpenVZ [35]. Xen is an example of para-virtualization, KVM and VMware are examples of full-virtualization and OpenVZ is an example of OS-level virtualization.

2.2 Basics of I/O virtualization

All regular instructions involving CPU computation within the virtual machines are executed similarly to the physical machine case, since no switch between kernel and user modes is required during execution of CPU instructions. However, in case of I/O operations, not only context switches between kernel and user modes are required, but also permissions and security are of paramount importance. In other words, I/O operations have a higher access level than CPU operations, and hence it is necessary for the Hypervisor to intercept and arbitrate the I/O operations requested by the virtual machines. In this section, we present a brief background on I/O virtualization techniques, towards setting up the background for work done in this thesis.

2.2.1 Network I/O virtualization

Xen [23] is a para-virtualization based technology while KVM [33] is full-virtualization based. Fig. 2.2(a) shows the Xen virtualization architecture depicting Dom0—which is the privileged management/driver domain, and DomUs—which are the guest virtual machines. All network I/O operations of the guest VMs are arbitrated by Dom0 via a shared memory interface termed Tx and Rx I/O rings. An event channel notification mechanism is used to notify events to the domains, for example, notification to DomU regarding a received packet or notification to Dom0 for a packet to be transmitted from DomU. The `netfront` and `netback` drivers, in DomU and Dom0 respectively, coordinate the data exchange between the domains, and the native driver in Dom0 coordinates exchange with the physical network interface.

The I/O architecture used by Xen with para-virtualization is referred to as the Driver do-

main I/O model. On the other hand, KVM[36] is an example of hardware-assisted virtualization (also full-virtualization), and uses another model for I/O, wherein virtual machines are run as processes on the host and I/O processing is done by the QEMU [37] emulator in userspace. In this architecture, there is no separate driver domain to handle privileged instructions, hence it is called the Direct I/O model [38].

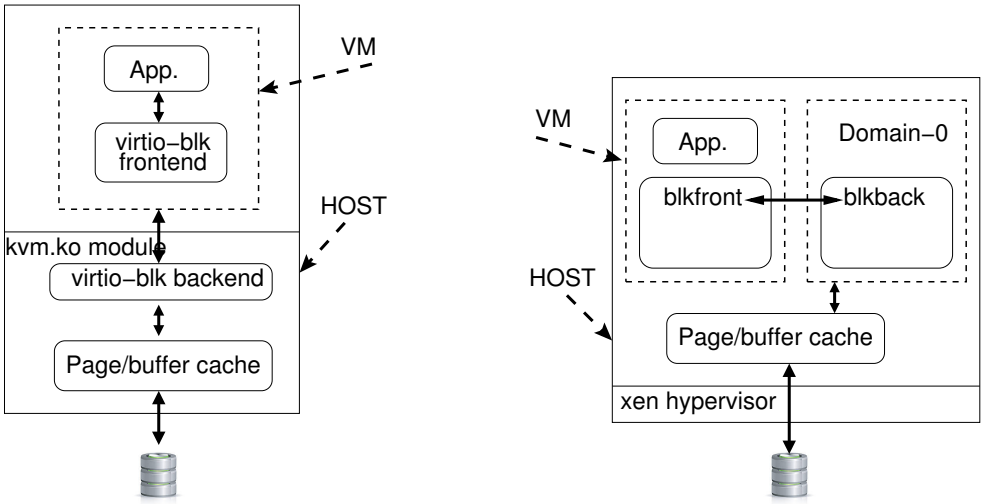
Due to arbitration of DomU's I/O access by Dom0, network activity of guest VMs results in additional CPU utilization at Dom0. However, since dispersed VMs communicate by accessing the physical network interface as opposed to colocated VMs that communicate "locally", network communication between colocated and dispersed VMs results in different CPU overheads. In the first component of our thesis, we study the Dom0 CPU overheads with colocation and dispersion of virtual machines and build resource estimation models based on these findings.

In the above description, both Xen and KVM technologies are shown to have different I/O architectures, and the first component of our thesis deals with both architectures to demonstrate the difference in CPU overheads for communicating VMs. However, subsequently in 2008, an I/O processing framework known as `virtio` was proposed in [39], which is a generic, modular and pluggable platform that can be used transparently with any hypervisor. The motivation of this framework was to make hypervisor development independent from the development of virtualization-based I/O drivers [39]. This new framework shaped the design of our disk I/O redirection system in the second component of this thesis, as explained next.

2.2.2 Disk I/O virtualization

A virtual machine's storage is called a virtual disk, can be either an image file or a block device [8], and can be located on either a local disk attached to the physical machine, or network-attached as well. When the virtual disk is located on a locally attached disk on the physical machine, it is referred to as Direct-attached storage (DAS), whereas Network-attached storage (NAS) and Storage Area Networks (SAN) are examples of storage that are accessible over the network. Moreover, the difference between the virtual disk being a file or a block device is that in case of a file, the virtual machine's "disk-access requests" are translated into file-access requests by the Hypervisor, and then once again converted into block layer requests at the host physical machine. This double-indirection is avoided in the case where the virtual disk is itself a block device, and accessible using block layer semantics, for eg. iSCSI SAN device [40] or LVM-configured storage volumes [41] on the physical machine.

Irrespective of the type of storage used, the access to virtual disk is performed via virtual block drivers within the VM. As mentioned above, a new and emerging framework for virtual I/O device drivers is `virtio` [39]. The basic concept of `virtio` is a split-driver architecture, i.e., a pair of `frontend` and `backend` drivers that communicate with each other using a ring-buffer mechanism. The virtual machine hosts the `frontend` driver and forwards the request



(a) KVM (b) Xen

Fig. 2.3: Split-driver architecture for virtual block devices in KVM and Xen virtualization.

to the corresponding `backend` driver hosted in the hypervisor or VMM. Virtio is developed in Hypervisor-agnostic fashion, with the only constraint that Hypervisor and the guest OS are able to interact with virtio using appropriate ring-buffer handling mechanisms. Hence, virtio can be used with a variety of Hypervisors like Xen, KVM and lguest virtualization solutions [39].

The de-coupling of the virtual block device driver into `frontend` and `backend` drivers makes `virtio` a generic virtual I/O mechanism, which can work on multiple hypervisors and platforms. Currently, `virtio` is used as a high-performance I/O mechanism in KVM [33] virtualization technology. The paravirtualized block driver in Xen [31] architecture also follows the split-driver paradigm, wherein a privileged VM (Domain-0) hosts the `backend` driver. The split-driver architecture for virtual block devices in Xen and KVM is illustrated in Fig. 2.3. The communication between the `frontend` and `backend` block drivers is accomplished via a ring buffer transport mechanism, wherein each read request is described in a descriptor placed into the ring buffers. Each read request descriptor includes the block ID/address (to be read) and a buffer (into which the data is to be copied). In the second component of our thesis, we present a hint-based read I/O redirection method positioned within the `frontend` driver in a virtual machine, which can manipulate the downstream host cache in a content-deduplicated fashion to improve its caching efficiency.

In this chapter, we presented a brief background review of the area of cloud computing via virtualization, toward the scope of this thesis. Specifically, the discussion of network I/O virtualization is relevant to the first component of this thesis while the discussion on disk I/O virtualization corresponds to the second component. In the rest of this thesis, we present more detailed literature review and background details within each chapter as well, where appropriate.

Chapter 3

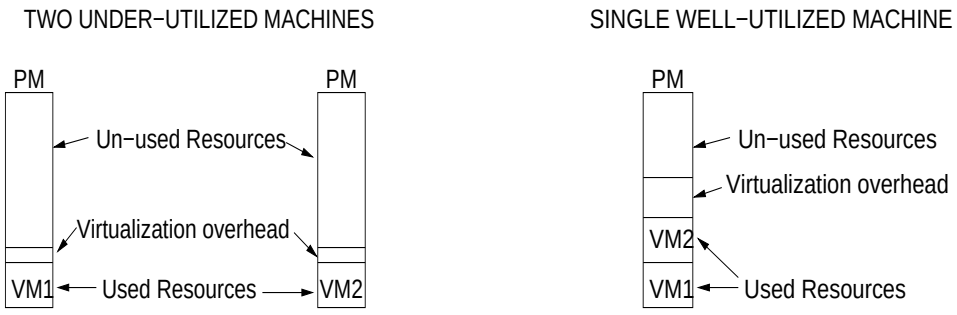
Affinity-aware Modeling of CPU usage for Virtualized Applications

3.1 Introduction

In traditional datacenters, the practice is to either follow a dedicated model and provision servers for peak loads, or share resources in a best-effort manner that offers no guarantees [42]. For example, an auction website may be hosted on a single PM in order to guarantee that it has enough resources to face peak loads. However, this results in under-utilization of resources and wastage of power during low loads, say, when the auction website is idle during night. Alternatively, co-hosting the auction website with another service in simply a best-effort manner will be insufficient when either of the co-hosted services is bombarded with heavy load. Resource utilization can be maximized while still ensuring performance guarantees through dynamic resource provisioning [7, 29], by adopting virtualization and consolidating/co-hosting multiple services (like auction websites and email servers) as VMs on the same physical machine [11].

Several to-be-virtualized applications have data dependencies between each other or among their components. Example instantiations are, (i) a multi-tier web-based application has exchange of data between its components—web-server, application logic server and the database, and (ii) heavily parallelized applications [43] which have distributed computing tasks with mutual data dependencies. *Server consolidation* refers to mapping a set of applications/servers to instances hosted within virtual machines, in such a way as to maximize the utility of the available infrastructure [12]. Thus, server consolidation can facilitate sharing of available resources among the tiers of a single application or across tiers of multiple applications, using dynamic resource allocation and virtual machine migration techniques [7, 44].

An example consolidation scenario is shown in Figure 3.1, where two virtualized server instances can be moved to or co-hosted on a single physical machine such that only one physical machine is sufficient to host both services and the other machine which becomes idle can be powered off. We can see that in the “Before Consolidation” case, each physical machine is exe-



(a) Before Consolidation (b) After Consolidation

Fig. 3.1: Server Consolidation, via Virtualization, for efficient resource multiplexing.

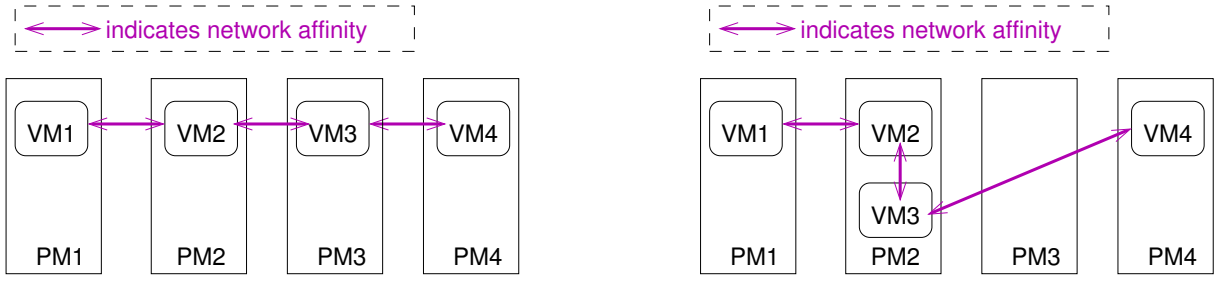
cuting a single virtual machine each, which are quite lightly loaded. Here, power is being wasted in keeping both physical machines ON though under-utilized. In the "After Consolidation" case, both VMs have been moved to one of the physical machines. If the physical machine can accommodate both VMs and the corresponding virtualization overheads, the other resultant unused PM can be switched OFF. Such consolidation and powering off of unnecessary resources results in better resource utilization and saves copious amounts of power (otherwise used for supplying power to machines as well as for cooling) in the data-center.

Colocated and dispersed VMs: When applications are instantiated in a virtual environment, two major factors affect their performance—available network capacity and virtualization-related CPU overhead, and these factors vary based on whether the communicating virtual machines are located on the same host or on different hosts [45]. *Colocated* virtual machines (VMs hosted on the same PM) incur different virtualization overheads for mutual network communication as opposed to *dispersed* VMs (VMs placed on different PMs). It is claimed in [45] that transitioning between colocated and dispersed placements for communicating VMs can result in a change in their CPU requirements. However, empirical quantification is lacking.

Network affinity: In general, VMs are said to have *network-affinity* for each other if they have network communication between them [45, 46]. When the VMs are colocated, the network traffic between them is *intra-PM*, whereas when they are dispersed, it is *inter-PM* network traffic. Since migration of a VM can potentially change the nature of network traffic between *intra-PM* and *inter-PM* depending on the source and destination hosts for migration, we define this as the *mutable* nature of network affinity, as explained next.

Mutable nature of network affinity: Given a set of VMs, some of which may have network communication with one another, every "communicating pair" of VMs is said to have network affinity for each other. A migration of any one VM can, but need not necessarily, cause network traffic between a VM pair to change from *intra-PM* to *inter-PM*, or from *inter-PM* to *intra-PM*. For example, suppose there are 4 VMs hosted on 4 PMs, as illustrated in Fig. 3.2(a). The VM pairs that have network affinity are, (i)VM1 $\leftrightarrow$ VM2, (ii)VM2 $\leftrightarrow$ VM3 and, (iii)VM3 $\leftrightarrow$ VM4.

In the initial state (refer Fig. 3.2(a)), all VMs are hosted on different PMs (i.e. dispersed), hence the network traffic in each case is *inter-PM*. Suppose VM3 is now migrated from PM3 to



(a) Initial configuration

(b) VM3 migrates to PM2

Fig. 3.2: Example to demonstrate *mutable* and *immutable* network affinity.

PM2, it gets colocated with VM2, resulting in the configuration shown in Fig. 3.2(b). After the migration, the nature of traffic between VM2 and VM3 has changed from *inter-PM* to *intra-PM*, whereas VM2 continues to have *inter-PM* network affinity with VM1. Thus, migration of VM2 caused its network traffic with VM3 to change nature, while its traffic with VM1 remained unchanged. We differentiate these two types of network traffic as *mutable* and *immutable* network traffic. Basically, for a given VM migration, *mutable* network traffic is that which changes nature between *inter-PM* and *intra-PM*, and *immutable* network traffic is that whose nature does not change due to the migration.

We present benchmarking experiments which demonstrate impact due to network affinity on CPU usage of virtual machines and their hosts, when communicating VMs are colocated as compared to when they are dispersed. Motivated by the benchmark findings, we develop models that can estimate the "colocated" CPU resource usage when VMs transition from dispersed to colocated placements, and can estimate the "dispersed" CPU resource usage when VMs transition from colocated to dispersed placements. These models predict CPU usage in target scenario (colocated/dispersed) based on resource usage profiles from source scenario (dispersed/colocated). First we built models to predict total CPU usage for target scenario, based on all resource usage profiles like CPU, disk, mutable network and immutable network usage. However, the maximum error with these predictions was found to be around 4 to 6% absolute CPU usage. Hence, based on our findings that CPU usage is affected only by *mutable* network traffic levels, we build models to predict the difference in CPU usage based on only the mutable network traffic profiles. These models were much more accurate, with maximum error within 2%. Finally, we applied these pair-wise models to multi-VM scenarios using a multi-phase prediction methodology. This demonstrated that simple models built on the scale of two VMs could be successfully used to predict for multi-VM scenarios as well. Our contributions are,

1. *Event profiling* of intra-PM and inter-PM network communication paths in Xen, using `Xenoprof` [47].

2. *Benchmark* CPU resource requirements of VMs with different levels of network-affinity in colocated and dispersed configurations.

3. Perform the above benchmarking step for both Xen and KVM virtualization environments, which shows that a linear relationship between network usage and the resulting CPU overhead exists in both.

4. Develop *pair-wise affinity-aware* models to predict *total* as well as *differential* CPU usage, when a pair of VMs move between dispersed and colocated configurations.

5. Apply above pair-wise models to *multi-VM scenarios*, where a single migrating VM has more than one neighboring VMs with which it exhibits network-affinity, both on source & target PMs.

6. Present a *comprehensive evaluation* using synthetic workloads & benchmark applications, of the pair-models as well as application to multi-VM scenarios.

The rest of this chapter is structured as follows. Section 3.2 presents background regarding network virtualization in Xen and KVM, and problem statement is presented in Section 3.3. Section 3.4 presents empirical benchmarking of the effects of relative placement on CPU usage of communicating VMs, in both Xen and KVM virtualization environments. Section 3.5 presents our approach to build affinity-aware models to predict CPU usage for Xen DomU and Dom0 and Section 3.6 presents evaluation of the models. Section 3.7 presents related work in juxtaposition with our work and in Section 3.8, we present our ideas for future work. Section 3.9 concludes the chapter.

3.2 Background

This section recalls the concept of network I/O virtualization in Xen and KVM (previously presented in detail in Section 2.2), and supplements it with an event profiling study of network virtualization in Xen.

3.2.1 Recalling the basics of network I/O virtualization

As mentioned in Chapter 2, the Xen virtualization technology has the concept of a split-driver architecture to deal with network I/O virtualization—consisting of a `frontend` and `backend` driver. Of these, the `backend` driver is responsible to communicate with the native network drivers and get the I/O performed on behalf of the virtual machines.

In case of virtual machines that are colocated on a single physical machine, the native driver does not need to be invoked at all whereas if the VMs are hosted on different physical machines, physical network communication is essential. This difference manifests as different CPU overheads for the virtual machines and the physical hosts concerned. In this section, we present an event profiling study of the CPU overheads in both scenarios.

3.2.2 Profiling study of network I/O virtualization

To further motivate the study of differences in communication between colocated and dispersed virtual machines, we performed a detailed profiling-based study of Xen's networking architecture and implementation [48, 49, 50].

A common optimization in the colocated case communication is that packet check-summing (both calculation and verification) is not performed since it is assumed that memory copying (performed in colocated case) is quite reliable as opposed to physical network transmission (corresponding to dispersed case). Additionally, when Xen-based VMs are colocated, they are connected via a layer 2 software bridge and hence a packet transmitted from one VM to another colocated VM is locally delivered on the bridge itself. On the other hand, when communicating VMs are on different physical machines, a packet transmission by one VM is forwarded over the software bridge, DMA-copied into the network interface card's (NIC) buffer, and placed on the network link by the NIC. This transmitted packet is then received on the destination host's NIC, copied into a kernel buffer for further processing and interrupt sent to destination host's driver domain (Dom0). Upon subsequent scheduling, the received packet is inspected by Dom0 to determine the destination VM, packet delivery scheduled and destination VM notified of incoming packet. Thus, end-to-end communication path in dispersed case is comparatively longer than the path in colocated case.

Using the tool `Xenoprof` [47], we performed event monitoring for network transmission between a pair of Xen-based VMs in both colocated and dispersed scenarios. `Xenoprof` performs statistical profiling of applications by using non-maskable interrupts when a performance counter overflows, to sample the function under execution. Since it performs statistical sampling, higher number of samples of a specific function call can imply either that a single invocation of the function had a long execution time compared to other functions, or that the function call had higher number of invocations as compared to others, or both.

To perform the profiling study, TCP network traffic of 50Mbps was generated from one VM to the other, wherein VM1 sent requests and VM2 served the requests by transmitting back the requested number of bytes. We define the transmitting and receiving VMs in terms of the direction of data traffic, i.e., the VM sending requests receives data responses, hence is the *receiver*, whereas the VM receiving requests sends data responses and is the *transmitter*. Thus, in our example, VM2 is the transmitter (of requested data) and VM1 the receiver (of requested data). We refer to Dom0 on VM2's host as the transmitting Dom0 and the Dom0 on VM1's host as the receiving Dom0. However, it should be noted that neither of the VMs are pure transmitters or pure receivers—because VM1 (i) sends requests, (ii) receives responses and (iii) sends TCP acknowledgements whereas VM2 (i) receives requests and (ii) sends responses.

We perform monitoring on both the transmitting Dom0 (*Disp-Tx*) and the receiving Dom0 (*Disp-Rx*) in the dispersed case. Meanwhile in the colocated case, we perform monitoring on the sole Dom0 instance (*Colo*) which performs both transmit and receive processing. Our aim is to

Table 3.1: Normalized number of samples reported by Xenoprof

Function Call	Normalized num of samples		
	Disp-Tx	Disp-Rx	Colo
e1000	0.87	1.00	0.01
gnttab_copy	0.06	0.56	1.00
bridge	0.84	1.00	0.74
change_page_attr	0.00	1.00	0.00
x86_emulate	0.00	1.00	0.00
do_mmuext_op	0.00	1.00	0.01
ptwr_do_page_fault	0.00	1.00	0.00
get_page_from_l1e	0.00	1.00	0.00
xen_tlb_flush	0.00	1.00	0.00
All	0.44	1.00	0.49

empirically observe the difference in CPU processing required in the three cases —(i) *Disp-Tx* (short for Dispersed-Transmit), (ii) *Disp-Rx* (short for Dispersed-Receive) and (iii) *Colo* (short for Colocated). We consider the 10 most sampled function calls in each case. Since we are interested in the differences and not the similarities, hence we discard those calls which are common to all three lists. We compute the union of all three lists and select those function calls that exhibit some distinctive feature of network flow. The sample counts of these calls are enumerated in Table 3.1 for all the three cases. The number of samples is represented in a normalized format, wherein for every function call, the number of calls in each of the three cases is normalized w.r.t the case which has the highest number of samples. For example, the function `change_page_attr` has highest number of samples in *Disp-Rx* case and almost negligible numbers in the other cases. The normalized number is shown correct to 2 decimal places, so very low numbers get automatically rounded off to 0, thus further simplifying our analysis. Thus, the function calls that are listed as having 0.0 samples are those which have very few samples in the `Xenoprof` output.

Observations from profiling study. From Table 3.1, we make the following observations,

- `e1000` (the native network driver) is used only in the dispersed case whereas in the colocated case, network packets are passed from source to destination over the bridge without using the native driver.

- `gnttab_copy` is a page copy mechanism involving grant tables. Hence, it is used significantly in *Disp-Rx* for copying packets from Dom0 memory to the receiving DomU and the highest in *Colo* case where Dom0 copies the packet from the transmitting DomU to the receiving DomU. It is not used much in *Disp-Tx* because of scatter/gather wherein the data to be transmitted is collected by DMA device directly from DomU memory.

- `bridge` is used approximately equally in all three cases. This is because packet delivery

in colocated case needs a one-shot traversal of the bridge as compared to traversing the bridge on both transmitting and receiving ends in the dispersed case.

- `change_page_attr`, `ptwr_do_page_fault`, `get_page_from_l1e`, `xen_tlb_flush`, `x86_emulate` & `do_mmuext_op` are related to the copying of received packets from the network buffer to Dom0 memory, wherein Dom0 has to acquire free pages, request for copying of packets to those free pages, and facilitate guest TLB updates. These calls, being specific to dispersed receive flow, are absent in colocated case network flow.

Thus, depending on whether the VMs are colocated (causing intra-PM network traffic), or dispersed (causing inter-PM network traffic), the network communication between them follows different data-paths, in-turn incurring different overheads. These differences in CPU overheads were observed empirically with both Xen and KVM environments (refer Section 3.4).

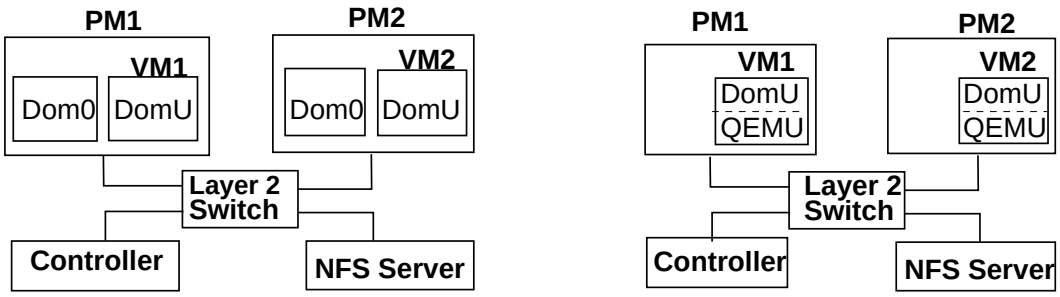
3.3 Problem definition

We are interested in the problem of estimating the CPU utilization of a virtual machine, based on its location relative to its communicating set of virtual machines. Additionally, the virtual machine should be able to continue execution of its tasks to meet specified service level objectives. Since CPU is the primary resource affected due to handling network I/O operations in colocated and dispersed scenarios, the scope of this work is restricted to predicting CPU resource usage.

We consider the following scenario for our problem. A set of applications, each application having several mutually communicating components or tiers, are provisioned as VMs in a cluster of inter-connected PMs. Thus, each VM has network activity, disk activity and CPU utilization. Since disk partitions may be network-attached (NFS-mounted or SAN or NAS appliance), disk activity of guest VMs may also manifest as network traffic at their host PM. In this setting, proactively or reactively, a decision process may decide to move/migrate a subset of VMs to meet dynamic resource requirements or to consolidate VMs on fewer PMs, and a vital input to this decision process is the resource requirements of the VM on the target machine after migration.

Given a set of virtual machines and their current resource utilization levels, our aim is to predict CPU resource required by the virtual machine on target host after migration. Additionally, the virtual machine should be able to support the *same load level* as on the source physical machine. Memory is assumed to be not a bottleneck in our placement configurations, and the VMs are assumed to maintain their intrinsic resource utilization levels towards maintaining the SLA guarantees.

In [45], there is allusion to the possibility of change in resource requirements upon a change in the hosting scenario of two communicating VMs—however, empirical quantification is lacking. As part of our work, we perform a detailed benchmarking exercise to empirically demonstrate that there is indeed a difference in CPU utilization of communicating VMs when their hosting



(a) Xen setup (b) KVM setup

Fig. 3.3: Setup for benchmarking, profiling and model evaluation.

scenario changes between *colocated* and *dispersed*. The difference in CPU usage for intra-PM (due to VM colocation) and inter-PM (due to VM dispersion) network communication forms the motivation for affinity-aware CPU usage modeling. Benchmarking for both Xen and KVM virtualization environments is presented in the next section and towards CPU requirement prediction, we build models for both the colocation and dispersion scenarios, which we describe in detail in the following sections.

3.4 Benchmarking with colocated and dispersed provisioning

In this section, we study the implications of communication among VMs in colocated and dispersed placement scenarios. Although the study reveals that there are differences in CPU utilization based on the placement scenarios, however the exact CPU utilization levels are incidental to (i) the operating system versions, (ii) the virtualization technology used and its version, (iii) the physical machines used and their configurations.

3.4.1 Experimental setup

Fig. 3.3 shows the experimental setup that we used. The setup used for benchmarking with Xen virtualization technology is as shown in Fig.3.3(a) wherein Dom0 is the privileged domain as described earlier in Section 2.2. In case of KVM platform, there is no management domain since it does not follow the driver domain I/O model. Hence, the setup for KVM benchmarking is a slightly modified version, as depicted in Fig. 3.3(b).

As shown in the setup of Fig. 3.3, two PMs (both with same configurations) host the VMs. Each PM is connected via a Layer-2 Switch to an NFS server which hosts disk images associated with the VMs. Thus, all disk read/write operations are NFS-read/write operations which generate network traffic at the host. The Controller is responsible for coordination of load generation and resource-usage measurements on the VMs. Load generation is done using an automated script residing at the Controller, that invokes a custom application program (called `LoadGen`) at each VM. For logging of resource utilization, we adopt the "black-box" approach [7], i.e, we monitor VM's resource usage by measuring only at the hosts (PMs) and not inside the VMs.

Resource utilization logging is done on the host, using utilities like `sar` [51], `Xentop` [52] (for Xen), `top` [53] (for KVM) and `iptables` [54].

The two physical machines hosting the VMs are Intel Core 2 Quad (Q9550) machines with 2.83 GHz cores. Xen version is 3.2 with Linux kernel 2.6.24-26 and KVM version is kvm-62 having QEMU PC emulator version 0.9.1. Both Controller and NFS server (not virtualized, hence common to both Xen and KVM setups) are Intel Core 2 (E7400) machines with 2.60 GHz cores. The Layer-2 Switch and all network links of the machines operate at 100 Mbps.

3.4.2 Workload generation

As part of our experimental evaluation, we generate different types of workloads for benchmarking and model building. Workloads are generated using a generic client-server setup, wherein a *client* (the controller machine) remotely connects to the *servers* (each PM or Dom0, and VM or DomU). The workload generation tool resides on each such "server" and complies with the load generation requests received from the "client" machine. Referring to Fig. 3.3, the workload generation requests are sent by the "Controller" and VM1/VM2 execute benchmarks to generate the requested resource utilization levels.

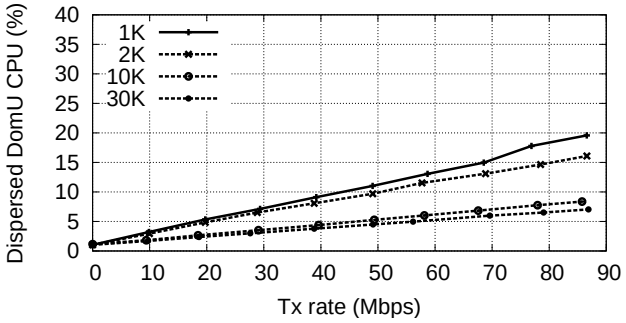
Though more detail regarding the design, implementation and usage of the load generation tool (called `LoadGen`) is presented in Appendix I, here we mention the different workloads briefly. The different types of workloads generated by `LoadGen` are,

- **CPU-intensive workloads.** CPU intensive workloads are generated by having a worker thread calculate a Fibonacci series with varying periodicity. If T is the average time for a round of Fibonacci series calculation, CPU load of $X\%$ is generated by having the worker thread perform computations for $X \times T$ milliseconds (*active period*) and sleep for $(100 - X) \times T$ milliseconds (*sleep period*).

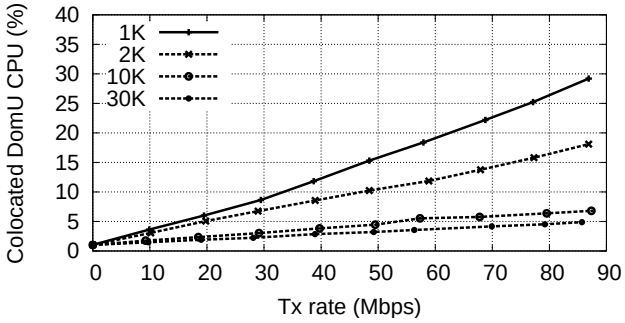
- **Mutable and immutable network-intensive workloads.** We generate various levels of network traffic between a pair of VMs by using a TCP-based custom application that sends a string of bytes on a TCP socket with different periodicity. For network workloads, we assume the maximum available capacity to be 100 Mbps and vary the load on each VM by steps of 10 Mbps, from 10 to 90 Mbps.

- **Disk read & write workloads.** We generate disk read (or write) workload by reading (or writing) files of $4kB$ size, with varying intervals to achieve different read access rates ranging from 0 to 1500 blocks/second. Translating into Mbps, these rates range from 0 to 5120 Mbps.

- **Combination workloads.** Combination or mixed workloads are generated using the same procedures as described above, with a multi-threaded process executing different workloads simultaneously.

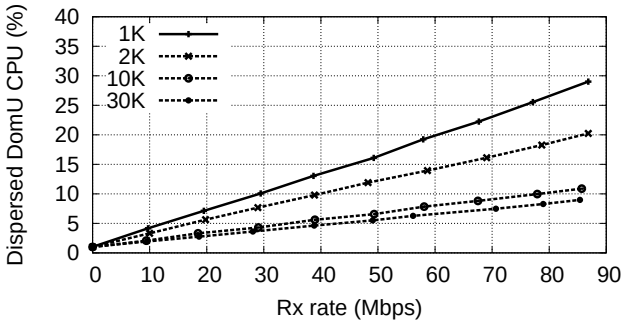


(a) Dispersed DomU CPU utilization for Tx

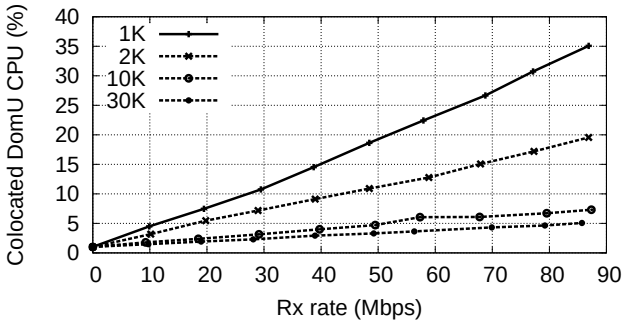


(b) Colocated DomU CPU utilization for Tx

Fig. 3.4: CPU utilization due to mutable transmit traffic in dispersed and colocated scenarios with different segment sizes (in Xen setup).



(a) Dispersed DomU CPU utilization for Rx



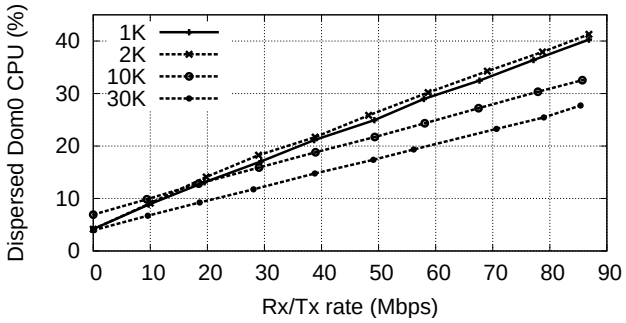
(b) Colocated DomU CPU utilization for Rx

Fig. 3.5: CPU utilization due to mutable receive traffic in dispersed and colocated scenarios with different segment sizes (in Xen setup).

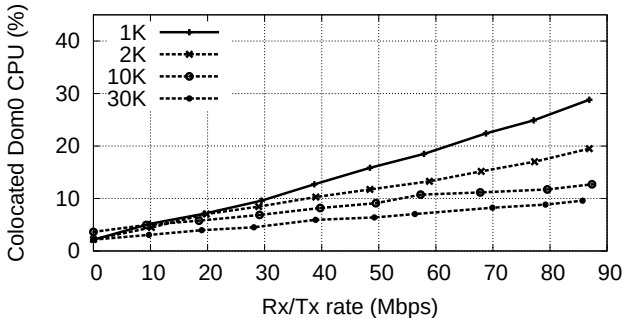
As part of the experimental setup, we ensured that for each experiment, all combinations of workloads over all VMs do not saturate capacity of any resource, i.e., CPU utilization and I/O utilization levels are always less than 100% in all experiments.

3.4.3 Effect of colocation on CPU usage

In this section, we empirically observe the effect of colocation on CPU resource usages of both Dom0 and DomU. By design, we generate the "same" load (type and amount) for each experiment in both the configurations—dispersed and colocated—and observe the differences in actual resource utilization levels. We are interested in addressing the following questions, (i) For mutable network workloads, is there a decrease in CPU usage when VMs are colocated, as compared to when they are dispersed? (ii) For pure CPU-intensive workloads, is the colocated Dom0 CPU usage a simple summation of the individual (or dispersed) Dom0 usages, (iii) For disk read & write workloads, and immutable network workloads, is the resultant colocated CPU usage a summation of usages in dispersed scenarios?



(a) Dispersed summation Dom0 CPU utilization for Rx/Tx



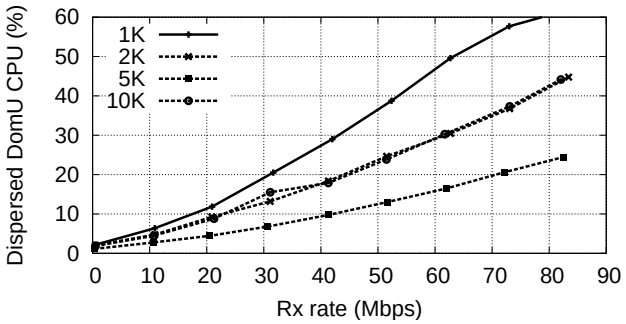
(b) Colocated Dom0 CPU utilization for Rx/Tx

Fig. 3.6: CPU utilization for mutable network traffic with different segment sizes (in Xen setup).

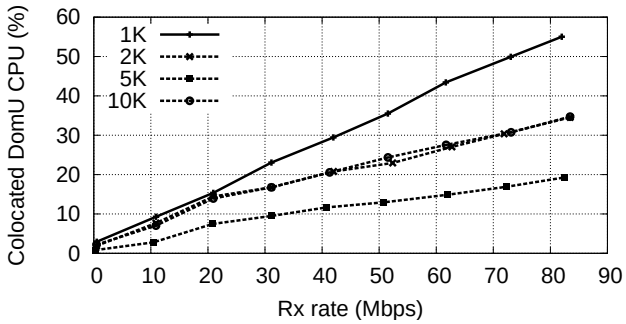
I. Impact of mutable network traffic in Xen. One of our claims (and as also discussed in [45], [46]), is that colocated provisioning can result in changes in resource usage for mutually communicating VMs. In this experiment, VM1 and VM2 act as a Tx/Rx pair to transmit and receive data at different rates, such that VM1 is the transmitting DomU and VM2 is the receiving DomU. To study the implications of mutable network-affinity between VMs, the experiment is conducted in both colocated and dispersed scenarios, and CPU utilization for DomU (and Dom0 in Xen) is measured. In our setup, we observed that TSO (TCP Segmentation Offload) feature was enabled whereas the complementary feature LRO (Large Receive Offload) at receiving end was not functional. For the sake of uniformity at both transmitting and receiving ends, we disabled TSO for our experiments.

Fig. 3.4(a) and 3.4(b) plot the transmitting DomU's CPU utilization for varying usage of network bandwidth in Xen setup. Each line represents the use of a different application-level segment size (sizes are mentioned in the legend). As described in Section 3.2, inter-PM and intra-PM communication have different execution paths, hence resulting in different CPU utilization. As can be seen, for higher segment sizes, increase in network bandwidth usage results in higher CPU savings upon colocation. However, the opposite result is observed for smaller segment sizes. The bold lines in the graph represent those segment sizes for which colocated CPU usage is higher than dispersed whereas the dotted lines indicate those having colocated CPU usage as lower than dispersed. For example, for the segment size of 30KB at 80Mbps network utilization, CPU utilization is 8% for dispersed and 4.8% for colocated, thus indicating a drop of approximately 3% absolute CPU when transitioning into colocated. On the other other hand, for the segment size of 1KB at 85Mbps, CPU utilization is 29% for dispersed and 35% for the colocated scenario, which indicates a 6% increase while moving into colocated placement. A similar result was observed for the receiving DomU as well, as depicted in Fig. 3.5.

Fig. 3.6 shows Dom0 CPU utilization levels for two communicating VMs, with 3.6(a) showing the summation of the two dispersed Dom0 CPU utilization and 3.6(b) showing the colocated Dom0 utilization. We can see that in all cases, the summation utilization is significantly greater than the colocated Dom0 utilization. For example, for the segment size of 30KB at 80Mbps net-

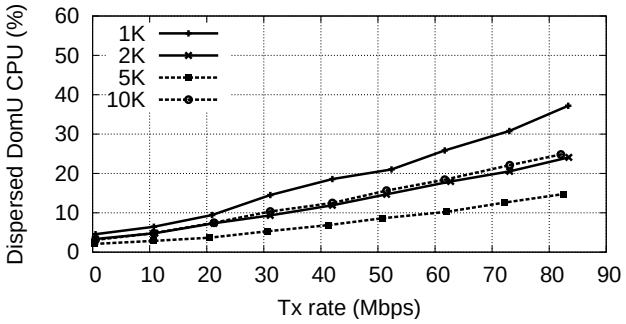


(a) Dispersed Guest VM CPU utilization for Rx

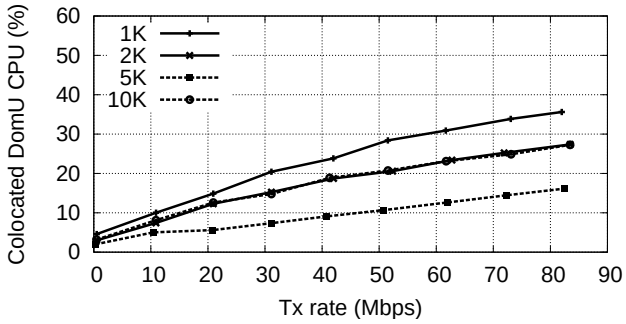


(b) Colocated Guest VM CPU utilization for Rx

Fig. 3.7: CPU utilization for mutable receive traffic with different segment sizes (in KVM setup).



(a) Dispersed Guest VM CPU utilization for Tx



(b) Colocated Guest VM CPU utilization for Tx

Fig. 3.8: CPU utilization for mutable transmit traffic with different segment sizes (in KVM setup).

work utilization, the difference in absolute Dom0 CPU usage between dispersed (summation of utilization at both Dom0's in the dispersed scenario) and colocated (single Dom0 CPU) scenarios is 16% and similarly for 2KB segment size, this difference is around 20% absolute CPU. The above observations suggest that not only the bit-rate of transmission, but also the segment size affects CPU utilization in both colocated and dispersed scenarios, and we need to consider both these factors while modeling CPU usage. Further, we also observe that in all above experiments, CPU usage varies linearly with respect to network utilization.

II. Impact of mutable network traffic in KVM. Similar to the benchmarking for Xen presented above, we performed an empirical study of colocation effects in KVM as well. As mentioned earlier, KVM does not have the concept of a privileged domain to arbitrate I/O access amongst the VMs. Instead, each VM is similar to a process and invokes the QEMU emulator to access the I/O devices. Thus, the overhead related to I/O processing (and the resulting effects due to dispersed and colocated network communication) would be reflected in the VM (DomU) CPU usage itself. Fig. 3.7 and Fig. 3.8 show CPU utilization of the DomUs in dispersed and colocated scenarios in KVM setup. The experiment setting is such that VM1 requests varying network rates in the range 10 to 80Mbps for each segment size while VM2 requests a fixed rate of

Table 3.2: Percentage CPU usage for *immutable* Rx.

Immutable Receive (Mbps)	**Percentage CPU utilization**	
	Dispersed case $VM_1, VM_2, \sum Dom0_i$	**Colocated case** $VM_1, VM_2, Dom0$
<20, 50>	4, 7, 18	4, 7, 14
<20, 70>	4, 9, 21	4, 9, 17
<40, 10>	6, 2, 15	6, 2, 11
<40, 30>	7, 5, 18	7, 5, 15
<40, 50>	7, 7, 21	7, 7, 18
<60, 10>	8, 2, 18	8, 2, 14

10Mbps. Thus, though both VMs are transmitting and receiving data, the variations in network rate at a given segment size setting are caused only by VM1's request-rate.

Fig. 3.7 shows that when receive traffic at VM1 is 80Mbps for a segment size of 10KB, DomU CPU usage is 44% absolute CPU in dispersed case and drops to 35% in colocated case whereas for segment size of 1KB, CPU usage drops from 62% in dispersed case to 55% in colocated case. Similarly, Fig. 3.8 shows CPU usage plots for VM2 transmitting 80Mbps using various segment sizes. Thus, we observe that CPU usage of transmitting and receiving DomUs change between colocated & dispersed scenarios. Additionally, we also observe that CPU utilization is approximately linear with respect to network utilization in above experiments.

III. Other workloads. Table 3.2 shows colocated benchmarking results for immutable *receive* traffic—VMs receiving network packets from "dispersed" (i.e. hosted on different PMs) transmitters. The table shows CPU usage of VMs in dispersed and colocated cases for Dom0 and DomU. The left most column shows tuples of the form $< x, y >$ where x is the the receive rate at VM1 and y is the receive rate at VM2. The second column shows the CPU utilization of VM1, VM2 and the summation of the CPU utilization of the two Dom0 instances in dispersed scenario. The third column shows the CPU utilization of VM1, VM2 and the single Dom0 instance in the colocated scenario. Observe that DomU CPU utilization stay similar in both dispersed & colocated cases. The colocated Dom0 CPU utilization is consistently (4%) less than the summation of the Dom0 utilization levels in the dispersed case.

Similar observation was made for other workloads—immutable transmit traffic, CPU workloads, disk read and disk write workloads. The Dom0 utilization of 4% is the same as its utilization under idle load, hence the above observation implies that for all other workloads except mutable traffic, colocation results in saving the CPU overhead related to an extra Dom0 instance.

IV. Summary of benchmarking From the benchmarking phase, we have following take-aways,

- When the nature of mutable network traffic changes between intra-PM and inter-PM, there is a change in guest VM CPU utilization in both Xen and KVM environments.

- In case of Xen Dom0, change in nature of network-affinity results in significant change (up to 25% absolute CPU) when comparing colocated Dom0 against the summation of dispersed Dom0 utilization.

- With increase in network utilization, the increase in CPU utilization is linear in both Xen and KVM environments.

- For all workloads, other than mutable network traffic, colocation results in similar DomU CPU usages in both colocated and dispersed cases.

- For all workloads, other than mutable network traffic, the summation of CPU usage of the two Dom0 instances in dispersed case differs from the colocated CPU usage by a constant amount (say 4% absolute CPU).

Based on the results of benchmarking, we conclude that the relation between increase in network utilization and CPU utilization is linear for both Xen and KVM environments. [1] As a result, both the "total" as well as the "differences" in CPU utilization can be captured using linear estimation models.

3.4.4 Feasibility of Generic models

In order to build a model that can be generically used to predict CPU utilization of DomUs, even though it is trained on only one or couple of VMs, it is necessary that VMs behave similarly under similar loading conditions. For example, suppose a CPU load of $x\%$ is requested, but creates $x + 5\%$ CPU load on one VM. It is essential that the same load when requested on another homogeneous VM, results in similar resource utilization levels. Thus, we are interested in verifying the following two hypotheses—(i) Given similar configurations, CPU utilization for similar load levels on different VMs match up, (ii) Given similar platforms, CPU utilization for similar loads on same VM but different PMs match up.

To this end, we performed several repeatable experiments on two VMs that were placed in a dispersed manner and compared whether similar loads result in similar CPU utilization levels on both VMs and their Dom0s. We performed such experiments for all load types—CPU intensive, mutable traffic, immutable traffic and disk-intensive—and as expected, we observed that both the above hypotheses about CPU utilization on different VMs and PMs held true.

In this section, we established the basic benchmarking results that will be applied to develop a generic CPU estimation model. By generic, we imply that a single model would be able to estimate CPU utilization for any given application, without requiring re-training on the specific

[1] These results are applicable only when both the network and CPU resources have spare capacity, e.g., if CPU utilization is already close to 100%, increasing the network rate is either not possible or results in no corresponding change in CPU utilization. All further observations, modeling and inferences are based on this premise.

Table 3.3: Metrics considered per load type on each DomU (for predicting total CPU).

Disk	**Mutable network traffic**	**Immutable network traffic**	**CPU**
Read (bytes/s) Write (bytes/s) Read (blocks/s) Write (blocks/s)	Rx (Kbps) Tx (Kbps)	Rx (Kbps) Tx (Kbps)	User (%) System (%) Iowait (%)

application itself. In the next section, we present our approach to develop the generic model and use it to estimate the CPU usage when a pair of VMs are colocated or dispersed.

3.5 Linear regression modeling for CPU requirement estimation

This section presents the core idea of our model generation and usage methodology. We run a set of benchmarks, also referred to as micro-benchmarks, in both scenarios—dispersed and colocated—which exercise the utilization levels of VMs along different axes—CPU, mutable and immutable network traffic, disk read and write operations.

Colocation and Dispersion models: We wish to develop pair-wise CPU estimation models that can predict total CPU requirement in target scenario based on source scenario's resource usages. Specifically, using resource usage measurements in the dispersed scenario, the "colocation" model predicts CPU utilization for the VM pair in the colocated scenario, and based on the resource usage in the colocated deployments, while the "dispersion" model predicts the CPU utilization of VMs in the dispersed case.

Our benchmarking revealed that only the mutable network usage causes change in CPU usage, upon change in VM placement. Hence, we build models to use two approaches of prediction:- (i) Predict *total* CPU requirement based on multiple resource usage profiles—CPU, disk and network, (ii) Predict *differential* CPU requirement based only on mutable network traffic metrics—later, take summation of prediction with the source scenario's CPU usage to estimate the total CPU requirement. In this section, we explain both approaches in detail.

3.5.1 Approach 1: Prediction of total CPU requirement

Core Idea: Using the profiling data from execution of micro-benchmarks and strengthened by our conclusions in Section 3.4, we believe that a generic linear model to estimate *total* (dispersed or colocated) CPU resource usage is realizable. The total CPU requirement of a virtual machine accounts for all its activities, including usage of all other resources. Hence, the model

for predicting total CPU requirement has all resource metrics as its parameters: (i) 4 metrics for mutable and immutable transmit and receive network rates (in Kbps), (ii) 3 CPU metrics of `iowait`, `system` and `user` CPU (in %), and (iii) 4 metrics among disk read/write rates in blocks/second and bytes/second. These metrics are tabulated in Table 3.3. Since the correlation of all resource usages to CPU usage is linear, we employ linear regression methods to build the models for CPU estimation.

Micro-benchmark Profiling: The idea is to capture behaviour of resource usage in all possible conditions for both dispersed and colocated scenarios. So, the micro-benchmarks should span the full range of resource utilization levels. The workload micro-benchmarks are generated using the workload generation procedures described previously (in Section 3.4.1). For CPU micro-benchmarking, we split the CPU load on each VM into $nine$ different intensities ranging from 10% to 90%. For network loads, we vary the load on each VM by steps of 10 Mbps, from 10 to 90 Mbps. For disk read and write micro-benchmarking, we vary the disk read/write rate from 0 to 1500 blocks/second on each VM.

The set of inputs to train/build the models also includes combination workload benchmarks, which have CPU, network and disk loads executed simultaneously, with different combinations of utilization levels. However, to conduct exhaustive experiments that cover the entire combination input set is not possible, since there is an exponential number of cases in the input space of combinational load. Thus, we adopted a workaround of choosing a set of random input workload types. For each combination benchmark, we sample a target utilization level, for each workload type, from a pre-defined range—CPU utilization from 10% to 90%, network rate from 10 Mbps to 90 Mbps and disk read/write rate of 0 - 1500 blocks/second. Each sample for a workload type is chosen uniformly at random. This is intended to keep the sample points uniformly distributed throughout the available sample space.

Overall, for the model building, we used 956 sample points in total, consisting of 200 points for CPU workload, 96 points for mutable network usage, 182 points for immutable network usage, 162 points for disk read workload, 158 points for disk write workload and 158 combinational workloads.

Multi-Linear Regression Modeling: Since the correlation of various resource usage metrics in the dispersed (colocated) case to CPU utilization level in the colocated (dispersed) case emerged as approximately linear, we employ linear regression methods to build the models for CPU estimation. Using values from the collected profiling data, the colocated CPU usage is represented as a linear function of the individually profiled resource metrics in the corresponding dispersed scenario (i.e. dispersed scenario stressed with the same workload), and similarly dispersed CPU usage is represented as a linear function of the colocated resource usage metrics.

The relation between estimated CPU and resource parameters is shown in Eqn. (3.1).

$$\text{CPU}^{i}_{estimated} = C_0 + C_1 \times M^i_1 + C_2 \times M^i_2 + ... + M^i_m \tag{3.1}$$

where, i is an iterator over each experiment or sample point to be considered in the modeling, m is the number of parameters being considered, M^i_j is the value of parameter M_j collected in the benchmark experiment number i, and $\text{CPU}^{i}_{estimated}$ is the CPU usage either after dispersion or after colocation of VMs. There are 11 metrics in all to be considered, as discussed earlier in Table 3.3. Both the colocation and dispersion DomU models have all these 11 parameters. The Dom0 *colocation* model is built with the metrics of both DomUs, and hence has 22 parameters. The Dom0 *dispersion* model depends on only one DomU's metrics at a time, and hence has only 11 parameters. This is because the dispersed Dom0's CPU usage intuitively depends on the resource usage of only its own hosted DomU and not on the other DomU of the VM pair.

3.5.2 Approach 2: Prediction of differential CPU requirement

Core idea: The difference in CPU usage upon transition between colocated and dispersed placements of a VM pair, depends only on the mutable network affinity between them. Hence, the idea here is to predict only the difference, and sum it with the original scenario's CPU usage, to obtain the total CPU requirement. Note that differential Dom0 CPU is the difference in CPU usage between that incurred for the single (colocated) Dom0 and the summation of both (dispersed) Dom0. The model parameters are bit-rates (in Mbps) and packet- rates (in packets per second) for both transmission (Tx) and reception (Rx), thus four parameters in all: (i) Rx Mbps, (ii) Rx pkts/s, (iii) Tx Mbps, and (iv) Tx pkts/s.

Modeling of differential CPU utilization: Since we predict only the difference in CPU usage, the total CPU requirement of the migrating VM on the target can be computed as the sum of the predicted value and the observed CPU usage on the source PM. Formally, Eqn. (3.2) captures the general notion of change in CPU usage (ΔCPU) while transitioning between colocated and dispersed scenarios.

$$\Delta\text{CPU} = \text{CPU}^{scenario1} - \text{CPU}^{scenario2} \tag{3.2}$$

where, scenario$\{1|2\}$ = $\{colocated|dispersed\}$. In case of **Dom0's CPU usage**, colocated Dom0 CPU usage is incurred for hosting both VMs whereas dispersed Dom0 usage levels are on behalf of a single VM each. During transition from dispersed (*disp* for short) to colocated (*colo* for short), the change in Dom0 CPU is defined as the difference in CPU usage between that incurred for the single (colocated) Dom0 and the summation of both (dispersed) Dom0, as illustrated in Eqn. (3.3).

$$\Delta\text{CPU}^{colo}_{Dom0} = \text{CPU}^{colo}_{Dom0} - \sum_{i=1,2} \text{CPU}^{disp}_{Dom0} \tag{3.3}$$

Similarly, during transition from colocated to dispersed, change in CPU usage for each Dom0 instance is,

$$\Delta\text{CPU}_{Dom0}^{disp} = \text{CPU}_{Dom0}^{disp} - \frac{\text{CPU}_{Dom0}^{colo}}{2} \tag{3.4}$$

Here we use $\text{CPU}_{Dom0}^{colo}/2$ since colocated Dom0 CPU usage is incurred on behalf of both VMs whereas we wish to predict dispersed Dom0 CPU usage incurred on behalf of only a single VM each. So, we use $\text{CPU}_{Dom0}^{colo}/2$ as an approximation of CPU usage incurred for a single VM within the colocated scenario. In case of **DomU's CPU usage**, we define change in CPU as shown in Eqn. (3.5).

$$\Delta\text{CPU}_{DomU} = \text{CPU}_{DomU}^{disp} - \text{CPU}_{DomU}^{colo} \tag{3.5}$$

Based on Eqn. (3.5), the difference needs to be added to or subtracted from the observed CPU usage depending upon whether the VM's migration results in dispersion or colocation, respectively. Thus, DomU's colocation and dispersion models are one and the same, since the *magnitude* of the difference in CPU usage is the same in both directions.

Using values from the collected profiling data, difference in CPU usage is represented as a linear function of the individually profiled network usage metrics in dispersed scenario (i.e. dispersed scenario stressed with the same workload). The relation between estimated CPU and resource parameters is the same as shown earlier in Eqn. 3.1, except that the value being estimated is the difference in CPU usage and not the total. In the case of DomU model, the CPU usage estimate is expectedly a function of its own network usage rates and uses all these four metrics in its model. Similarly, in the case of Dom0 colocation model, colocated Dom0 CPU usage accounts for both the colocated DomUs and hence should intuitively depend on four metrics each of both the DomUs, i.e. eight metrics in total. However, given a VM pair with only mutual network communication, the received traffic by one VM is the transmitted traffic by the other, hence the metrics of one VM are in fact duplicates of the metrics of the other, albeit in a different order. We discard the redundant columns, therefore even Dom0 model has only four parameters. Thus, each model can be represented as

$$\Delta\text{CPU}_{est} = C_0 + C_1\text{MutRx}_{Kbps} + C_2\text{MutRx}_{p/s} + C_3\text{MutTx}_{Kbps} + C_4\text{MutTx}_{p/s} \tag{3.6}$$

where, "Mut" stands for mutable network traffic, superscripts Rx & Tx stands for receive & transmit, and subscripts Kbps & p/s represent the units Kbps and packets/second, respectively.

To solve for the $m+1$ coefficients C_k, a stepwise linear regression approach can help to select the "best" set of parameters. The stepwise approach ensures that only significant parameters that influence the predicted value are part of the model [55]. For our implementation, we use the Robustbase package [56], part of the R statistical computing environment, for robust and stepwise linear regression. The linear regression solver, takes as input M_j^i values for the various workloads and outputs values of C_k that best predict CPU utilization. The set of coefficients

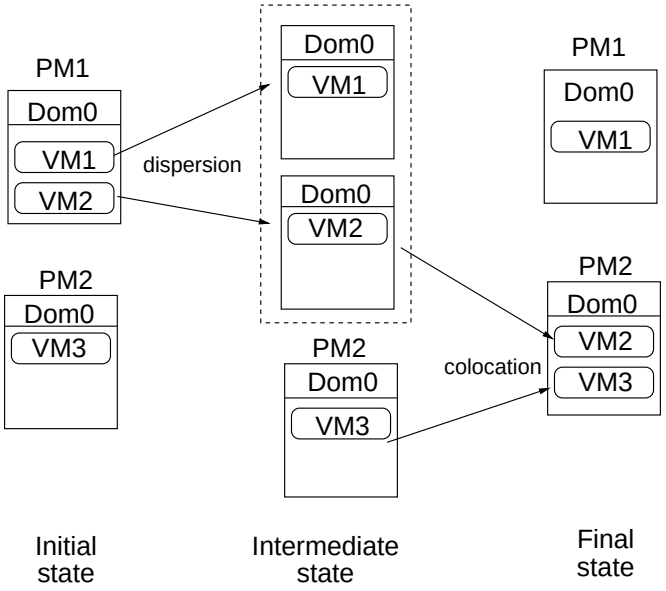


Fig. 3.9: Combined transition for $VM2$

thus derived is the model that describes the relation between the k parameters and the total or differential CPU usage.

3.5.3 Applying the pair-wise models to multi-VM scenarios

In order to apply the pair-wise models to situations with multiple VMs hosting different applications that may have different number of communicating tiers, the first step needed is to consider that in a multi-VM scenario, a single VM migration step can result in not only dispersion w.r.t one (or a few) VM(s), but also in colocation w.r.t other VM(s). Fig. 3.9 illustrates the case where a single VM is being dispersed from one VM and colocated with another. Suppose $VM1$, $VM2$ and $VM3$ host different tiers of a 3-tier application. As can be seen, one VM migration can result in that VM ($VM2$ in Fig. 3.9) being dispersed from one VM ($VM1$) on the source PM and being colocated with another VM ($VM3$) on the target PM.

In the general multi-VM scenario, any particular VM (say VM_x) may have multiple communicating neighbors, each of them belonging to exactly one of three sets, (i) *affinity-at-source* (ii) *affinity-at-target*, and (iii) *affinity-at-other*. Migration of VM_x will cause it to become dispersed from its *affinity-at-source* set and colocated with its *affinity-at-target* set. We refer to this dual step of being dispersed from *affinity-at-source* set and being colocated with *affinity-at-target* set, as a *combined transition*. In this case, the CPU utilization of all VMs in the *affinity-at-source* & *affinity-at-target* sets and the migrating VM itself are expected to change. In fact, the *colocation* and the *dispersion* transitions can be viewed as special cases of the *combined* transition, starring only two VMs, with the dispersion transition being the case where dispersion set on source PM is empty while the dispersion transition is the case with an empty set on target PM. Our aim is to predict the resource usage of all affected VMs and PMs.

Consider the combined transition illustrated in Fig. 3.9. It is straight-forward to apply the

DomU *colocation* and *dispersion* models to $VM1$ and $VM3$ to predict their resultant CPU usages. Similarly, $PM1$'s Dom0 CPU usage can also be predicted by applying Dom0 dispersion model. However, prediction of $VM2$'s and $PM2$'s resultant CPU usage after this combined transition needs a multi-phase prediction methodology. The basic idea is to first apply the *dispersion model* to the measured metrics corresponding to network-affinity level between $VM2$ to/fro the *affinity-at-source* set, and then use this intermediate estimate to make final prediction based upon network-affinity level between $VM2$ to/fro the *affinity-at-target* set. Intuitively, VMs in the *affinity-at-other* set do not affect CPU requirement of the migrating $VM2$ because the nature of network-affinity between them stays the same (*immutable*) both before and after migration.

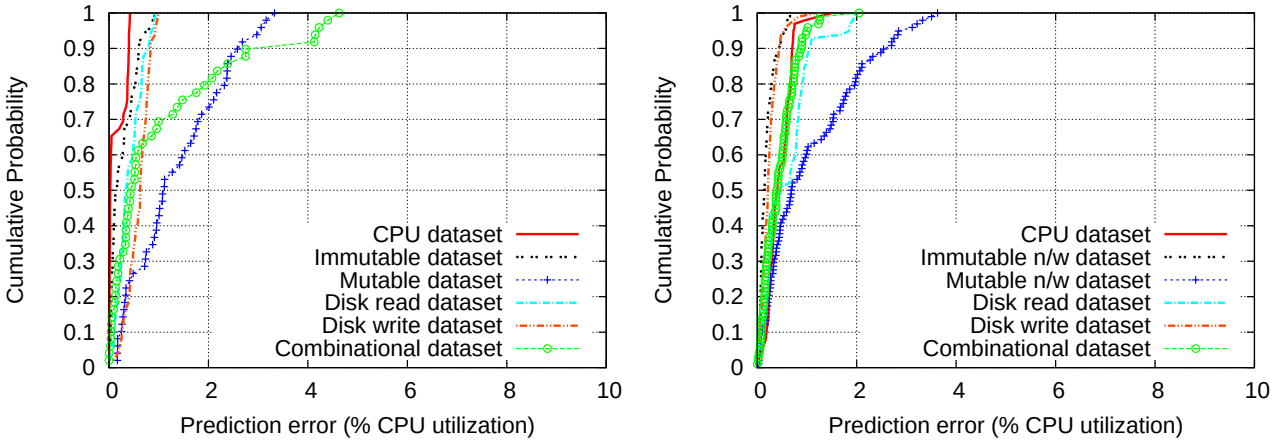
In this section, we described our model-building process and described the synthetic training datasets used to learn this model. We also described the process of applying the pair-wise prediction models to multi-VM scenarios. Though the models are built with synthetic data-sets in which TCP data-rates are generated by manipulating the application level segment size and the periodicity of transmission, the application scenarios for the models are not expected to have such controlled behaviour. In particular, though the applications are expected to use TCP for communication, the application-level segment sizes and the inter-request times (also called think-times) would be significantly more random. In the next section, we experimentally evaluate the models on randomized synthetic data as well as on benchmark application data. We also apply the model to multi-VM scenarios and evaluate prediction effectiveness.

3.6 Experimental evaluation

In the previous section, we described the motivation and methodology of developing models to predict colocated and dispersed CPU utilization. After the models are built, we recompute or "predict" the colocated CPU value for the training set itself, using the generated model coefficients. This is a sanity check to validate model correctness. The results showed that well-fitting models were built, with less than 1% to 2% error for all workloads. However, the real test is whether the models are able to successfully predict the average CPU usage well, when applied to unseen data. We present model evaluation on both synthetic data-sets and benchmark application data in this section.

3.6.1 Model evaluation with synthetic data-sets

In this sub-section, we present our findings when the generated models were applied to "unseen" datasets. By unseen, we mean that these datasets are not a part of the input set for model creation. The setup used for evaluation with synthetic data-sets is the same as the one that was presented in Section 3.4.1, for the benchmarking experiments. We present the evaluation of the



(a) Dom0 CPU in colocated scenario (colocation model) (b) Dom0 CPU in dispersed scenario (dispersion model)

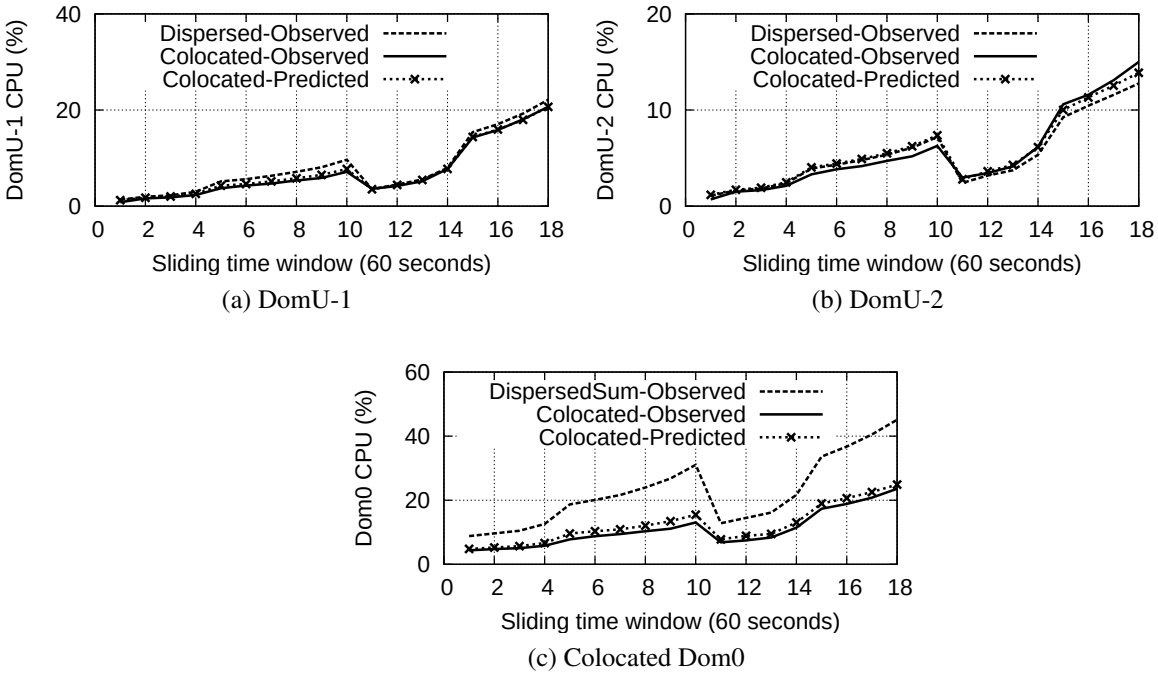
Fig. 3.10: Prediction error CDF for Dom0 CPU estimation.

two approaches separately—predicting total CPU usage, and predicting differential CPU usage.

For models predicting total CPU usage: The training data input for the models was generated by generating resource utilization at pre-defined discrete levels, however, the derived models need to be applied to "unseen" resource usage profiles in order to judge their adequacy. For this reason, the CPU workload for testing consists of 100 randomly picked values in the range 1% to 100%, the mutable and immutable Rx/Tx test workloads are chosen randomly from the range 10 Mbps to 90 Mbps, the disk read/write rates are randomly chosen from the range 0 to 1500 blocks/second, and the combinational workloads had randomly chosen values for each parameter (CPU, mutable network traffic, immutable network traffic, and disk) from these same ranges.

Fig. 3.10 plots the CDFs of error when Dom0 *colocation* and *dispersion* models were applied to all the six unseen datasets—cpu, mutable & immutable network traffic, disk read, disk write and combinational loads. It can be observed that all the CDFs seem to stick to the left end of the graph, however, the 90th percentile error is around 3% absolute CPU and maximum error is around 4%. We plotted similar error CDFs for colocated and dispersed DomU CPU estimation. In all cases, we found that the 90th percentile absolute error was within 3%. We do not present all the CDFs here, for sake of brevity; they can be found in the technical report at [57]. In case of Dom0 models, since the savings due to co-location is significant (as concluded from the benchmarking results), an accuracy with 3%error might suffice for a good prediction. However, in case of DomU models, the savings themselves being marginally lower (between 0 to 10%), a 3% error could imply higher relative error in comparison.

For models predicting differential CPU usage: In order to generate synthetic workload which closely emulates a real application scenario, we spawn multiple client processes, each sleeping for fixed intervals of time (emulating average think-time) and generating request for transmission of varying number of bytes or segment sizes. Thus, the artificial constraint of each client process requesting only a single segment size has been discarded. We generate randomly different



(a) DomU-1 (b) DomU-2

(c) Colocated Dom0

Fig. 3.11: Estimating colocated CPU utilization for synthetic data-set

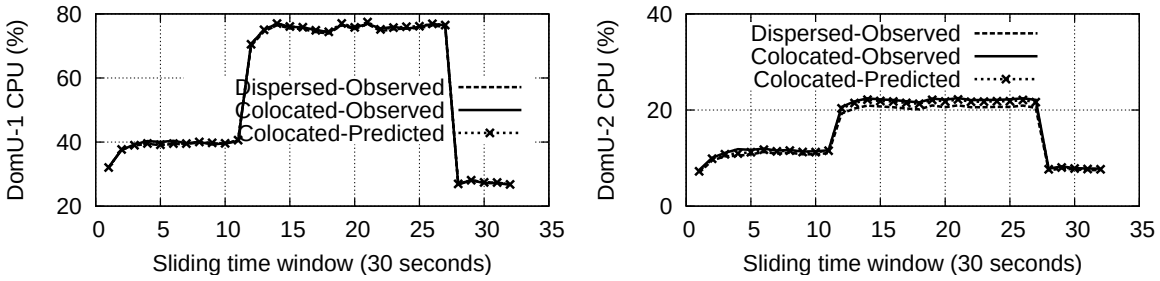
load levels using different number of client threads every minute, each generating requests for different segment sizes and monitor resource usage utilization over a period of 18 minutes. As expected, the resultant network traffic rates and CPU utilization also vary.

Fig. 3.11 shows colocated CPU usage, predicted versus actual, for both DomUs and Dom0. We can see that as dispersed CPU utilization changes due to change in network usage, predicted colocated utilization also varies similarly. The maximum error in DomU prediction is within 1% absolute CPU usage for both DomUs (VM1 and VM2) and the maximum error in colocated Dom0 prediction is within 2.4% absolute CPU. Since difference in CPU utilization for DomU is quite low and therefore less interesting, we focus on Dom0 models here onwards.

3.6.2 Model evaluation with application benchmarks

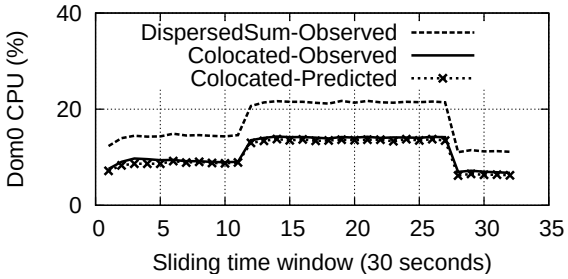
The above evaluation with synthetic workloads demonstrated that predicting the differential CPU usage, which considers only the mutable network usage metrics (including both network rate and segment size metrics) give higher accuracy predictions. Hence, we use that approach for the rest of the evaluation in this chapter.

For evaluating our prediction models with an application benchmark, we chose RUBiS [58]. RUBiS emulates an auction website like eBay, and allows simulation of various users/clients who engage in different tasks like user registration, item registration, browsing items, bidding for & buying items and so on. RUBiS is a two-tier application, consisting of a web tier and a database tier that communicate with each other for servicing each user request. Each tier is hosted in a separate VM. We generate repeatable RUBiS workloads in both dispersed and



(a) Web tier VM

(b) DB tier VM



(c) Colocated Dom0

Fig. 3.12: Estimating colocated CPU utilization for RUBiS

colocated cases, and compare average resource utilization levels (predicted versus measured) over 30-second intervals. The RUBiS workload consisted of 800 clients generating site browsing load simultaneously, and requests being fired at determined times. As done in [4], we also consider open-loop applications, where the inter-arrival times of the sequence of requests is the same across the dispersed and colocated scenarios. This is the case when response times are far lesser than the minimum think-time. The think-times are randomly chosen from a negative exponential distribution with a mean of 7 seconds, and minimum think-time of 5 seconds.

Fig. 3.12 plots predicted and measured CPU utilization for the web tier VM (Fig. 3.12(a)), the DB tier VM (Fig. 3.12(b)) and the Dom0 CPU utilization (Fig. 3.12(c)) after colocation. As seen in the figure, the RUBiS workload consists of 3 phases. It starts with an up-ramp phase which lasts till the 10th interval and then starts climbing to steady state. Prediction is able to follow the climb well. The steady state lasts till the 27th interval and then begins the ramp-down phase. Prediction follows the drop also pretty accurately. The maximum error in CPU usage prediction is less than 2% for both Dom0 and DomU models. Similar graphs were plotted for *dispersion* model as well and maximum error is within 2% absolute CPU utilization. Thus, prediction accuracies for both Dom0 and DomU models are equally high. indicating that the models can be successfully applied to an application without training on that specific application's resource utilization profiles themselves.

To demonstrate the extent of applicability of these models, we conducted extensive experiments with various number of clients in RUBiS and measured maximum error in each case. This is tabulated in Table 3.4, where number of clients is increased from 500 to 3000. We can see

Table 3.4: Model accuracy with varying load

No. of clients	Maximum net-affinity (Mbps)	Max error (% CPU)		
		DomU	Dom0 colo	Dom0 disp
500	6.8	1.73	1.33	0.90
1000	13.4	1.50	0.98	0.82
1500	19.8	1.42	0.69	0.90
2000	26.4	0.90	1.08	1.21
2500	32.6	1.03	1.29	1.27
3000	39.4	1.36	1.50	1.54

Table 3.5: Model accuracy with varying network-affinity levels

No. of items per page	Maximum net-affinity (Mbps)	Max error (% CPU)		
		DomU	Dom0 colo	Dom0 disp
5	4.8	1.51	1.55	1.06
10	7.7	0.64	1.00	0.81
15	10.5	1.47	1.44	0.86
20	13.4	1.5	0.98	0.82
25	16.2	1.51	0.98	0.82
35	21.5	1.27	1.02	0.87
45	28	0.62	1.11	1.00

that in each case, maximum error is within 2% absolute CPU utilization.

Another configurable parameter in RUBiS setup is the "number of items to be displayed per page" for any given browse or search request. Within the RUBiS implementation, this parameter dictates the network-affinity level between the web tier and the database tier. At an abstract level, the amount of network data being sent per request is similar to the concept of segment size considered earlier. Thus, this parameter indirectly influences the range of segment sizes that would be requested. The default value for this parameter in RUBiS was 20, and though this parameter would change very infrequently (or not at all) in a production web service, we use this parameter as a knob to simulate other web services which may have different network-affinity levels with different segment sizes, flowing amongst its various tiers. For the Xen-based RUBiS setup, we fixed number of clients to 2000 for this experiment and varied *NumItems* from 5 to 45. Table 3.5 lists maximum prediction error observed in each case—all within 2%.

3.6.3 Estimating CPU usage for "combined" transitions

In order to evaluate our prediction models on this combined transition, we setup a three-tier application. Instead of using an existing three-tier application, we use a simpler alternative of having an extra proxy tier in the previous setup such that all client requests are sent to this redirecting proxy, which forwards them on to the web-server and also relays back the responses received from the web-server back to the client. Fig. 3.13 is a pictorial representation of our

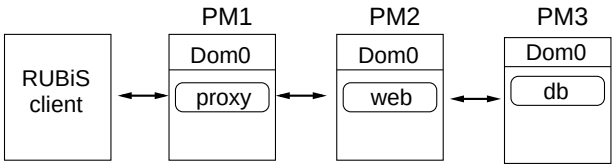


Fig. 3.13: RUBiS 3-tier setup with proxy, webserver and database

three-tier setup, where we use Muffin[59] proxy as the first tier of our test application.

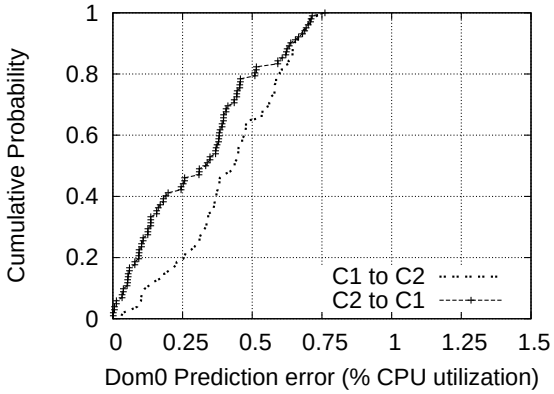


Fig. 3.14: Error CDF of Dom0 CPU estimation for C1 to C2 transitions

Prediction for two configurations: The RUBiS application run is then performed in two configurations—(i) *C1*: with proxy server (VM1) and web-server (VM2) colocated on PM1 while database server (VM3) hosted alone on PM2, and (ii) *C2*: with VM1 hosted alone on PM1 while VM2 and VM3 are colocated on PM2. Resource usage monitoring is performed on both PMs, as before. Fig.3.14 shows the CDF of prediction error considering error in prediction for all three PMs, during transition from *C1* to *C2* and vice-versa. Maximum error is within 1% absolute CPU usage for both transitions.

Prediction over series of migration steps: The above experiment demonstrated that *colocation* and *dispersion* models can be used as building blocks to do multi-phase CPU usage predic-

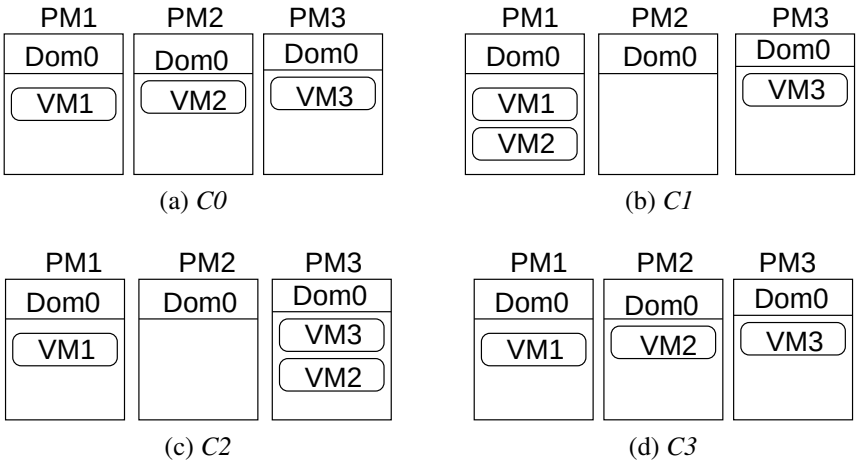


Fig. 3.15: Different placements due to series of VM migration steps.

tion for combined transitions. To extend this further, we consider a series of VM migration steps for this three-tier application such that each VM migration step results in a different VM placement and we apply *colocation* and *dispersion* prediction models appropriately to derive CPU usage prediction for each step. The series of VM migration steps being considered are (Fig.3.15 shows configurations): (a) *C0*: all 3 VMs on different PMs each (b) *C1*: VM2 migrates into PM1 and is colocated with VM1 (c) *C2*: VM2 migrates into PM3 and is colocated with VM2 (d) *C3*: VM2 migrates into PM2, same as initial configuration. For each of the above configurations, we measure CPU utilization incurred for a RUBiS run with 1000 clients. To test our models, we perform prediction of CPU usage for the three Dom0 instances (corresponding to PM1, PM2 and PM3) upon transition from one configuration to the next (e.g., *C0*→*C1*, *C1*→*C2* and *C2*→*C3*). Maximum error in prediction for each step is listed in Table 3.6.

Table 3.6: Error in Dom0 prediction over series of VM migrations.

Transition	**Maximum error in Dom0 prediction (% absolute CPU)**		
	PM1	**PM2**	**PM3**
C0→*C1*	0.75	NA	NA
C1→*C2*	1.99	NA	0.85
C2→*C3*	NA	0.51	0.43

Table 3.6 shows that for every transition step, Dom0 CPU prediction models perform good prediction and maximum error is within 2% absolute CPU utilization. The entry NA in any particular column implies that prediction is not performed for that PM during that step. This could be due to one of two reasons: (i) Due to the VM migration step, the PM has become idle, or (ii) During the VM migration step, the PM is not the source or destination of migration. Thus, we have demonstrated that simple CPU usage prediction models built on the scale of two VMs can be extended to apply to multi-VM scenarios as well.

3.7 Implications of affinity-aware CPU estimation

The work in this paper is focussed on effects of relative location (colocation or dispersion) on the CPU utilization of communicating virtual machines. Though most server consolidation and load balancing algorithms [10, 7, 5] acknowledge that VMs may have elastic resource requirements to satisfy dynamically varying load levels, it is generally accepted that resource requirement to support a single load level is the same on all homogeneous physical machines. In this work, we present an empirical study to quantify the effect of colocation on CPU utilization of mutually communicating VMs, and demonstrate that a single resource configuration to handle a certain load level is at-best not efficient and at-worst short of requirements. Thus, our work is supplementary to many server consolidation and load balancing approaches proposed in literature so far, and hence we use this section to chronicle the related work in these areas.

3.7.1 Server consolidation and load balancing

During periods of under-utilization, multiple virtual machines can be migrated onto a single physical machine, such a step is referred as *Server Consolidation*. On the other hand, when load experienced on a single physical machine increases such that resource requirements exceed resource capacity, *Hotspot Mitigation* is performed by migrating out some of the virtual machines. Thus, both the problems of server consolidation and hotspot mitigation are enabled by virtual machine migration. Given a set of VM configurations, the general problems of server consolidation and hotspot mitigation are viewed as bin-packing problems, which are NP-Hard. Trace-based approaches [10] have been adopted to predict workload and perform consolidation, whereas online monitoring and reactive approaches [7, 5] are used to address problems of load balancing and hotspot mitigation.

In all above efforts, VM configurations per load-level are assumed to be constant, implying that it is considered that a VM that requires x % of a resource on a PM would need the same amount of that resource on another homogeneous PM. Also, in most cases, CPU resource utilization levels determine whether the PM is heavily or under-loaded. If the VMs under consideration are mutually communicating, then this "network affinity" causes changes in the CPU usage profiles depending on the neighborhood set of colocated VMs and their communication patterns. Thus, in case of communicating VMs (or application tiers), it is essential to be "affinity-aware" while determining VM configurations, and the configurations need to be recomputed each time, instead of being assumed constant.

3.7.2 Affinity-aware provisioning

It has been demonstrated in [45] that transfer time between two communicating VMs can be reduced drastically (as much as 90%) by placing both on the same host. This points to opportunities for server consolidation based on *network affinity*. This may not only improve response or transfer times but also reduce the load on network resources, since the inter-VM communication between VMs co-hosted on a single physical machine is more akin to IPC rather than a network transfer. The effect of network-affinity on CPU usage has been alluded to in past research. In [45], the authors claim that decrease in physical network usage due to colocation will eventually translate to an increase in another resource dimension, say CPU. However, no empirical studies have been presented to quantify it so far, and hence the benchmarking study presented in this paper is especially relevant.

A non-intrusive black-box approach for identifying mutually communicating groups of VMs or *VM ensembles*, called Net-Cohort, is presented in [60]. This work advocates monitoring only guest-level network statistics using hypervisor-tools and building an N$\times$N correlation matrix to "infer" network communication rather than "observe" it by explicit monitoring of network traffic between all VM pairs. Using hierarchical clustering algorithm, VMs are grouped into ensembles based on correlation in their resource usage. After this initial identification of potential mutually

communicating VMs, packet sniffing is performed only on those VMs to identify the actual communication dependencies. This work is complementary to our work, in that, it can provide the affinity relations between various VMs which are required for affinity-aware CPU utilization estimation at datacenter scale.

A decentralized affinity-aware migration technique, which takes into consideration the network transmission traffic between each VM pair and tries to minimize communication overhead in the entire cluster of VMs, is presented in [46]. The *bartering algorithm* presented in [46] reflects the basic assumption of similar CPU requirements irrespective of VM placement. However, our work suggests that this assumption may not hold in real scenarios and network affinity-aware CPU requirement estimation is essential.

3.7.3 Estimation of virtualized CPU usage

A set of micro-benchmarks are used to profile the CPU usage on a given hardware platform, using regression modeling techniques in [4]. The aim is to determine the virtualization overheads of an application before it is placed in a virtualization environment, so that it is not accidentally deployed to a physical machine with insufficient resources. This is useful during initial placement of applications into the virtualized domain, also referred to as VM Sizing.

We borrow the idea of micro-benchmark profiling from this paper, however, we apply it to solve the problem of estimating colocated (virtualized) CPU usage given two dispersed (virtualized) resource usage profiles. Additionally, consider the scenario that the VMs to be transitioned from physical to virtual are mutually communicating VMs or form various tiers of a single application. Depending on whether communicating VMs are colocated or dispersed upon transition from physical to virtual domain, their virtualized CPU overhead and CPU usages would be different, and thus our work can help extend the work in [4].

3.8 Open directions

3.8.1 Benchmarking of 1 Gbps link network usage

In our experiments, we used network links with capacity 100 Mbps for network traffic and found that CPU usage had a linear correlation with network usage. This was the basis for using a linear regression modeling technique for CPU estimation. However, in order for our approach to be valid for 1 Gbps links, this linear correlation should hold in a setup with traffic up to 1 Gbps as well. To this end, we performed a benchmarking exercise as before (with Xen), but with a 1 Gbps link, to answer the following questions: (i) Does the linear correlation of CPU to network usage still hold with 1 Gbps links? (ii) Does using different capacity links between communicating VMs result in different CPU utilization at the VMs (and Dom0 for Xen)?

Fig. 3.16 shows CPU utilization at Dom0 for network-affinity levels ranging from 100 to 900 Mbps, in both dispersed and colocated scenarios for segment size of 5KB. As can be seen, the

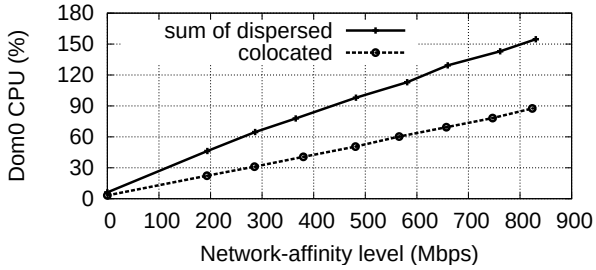


Fig. 3.16: Dom0 Utilization over a 1Gbps link

CPU utilization is linearly correlated with network rate in both the cases. Thus, we conclude that linear regression modeling approach can be applied for affinity-aware CPU estimation even in deployments with 1 Gbps links. However, the CPU utilization incurred for network rates $\leq$ 100 Mbps over a 1 Gbps link are not the same as those observed during benchmarking with 100 Mbps links. This implies that network link bandwidth is an important determinant of CPU utilization incurred at the transmitter and receiver. So, if the cloud deployment has networks links or switches of heterogeneous capacities, different CPU estimation models would have to be developed for each type of link.

3.8.2 Effect of colocation for data-centric applications

The work in this paper is targeted towards multi-tiered service-oriented applications such that the user issues a request and waits for a certain time (think-time) before firing the next request. In such a scenario, where think-time intervals are relatively large (as compared to response transmission time), requests are issued at long enough intervals such that observed network-affinity levels in colocated and dispersed scenarios are the same. However, for data-centric applications where tiers exchange large amounts of data, the transmission durations are significant and hence observed network-affinity levels in colocated and dispersed placements are not same (for observation intervals smaller than data transmission durations). For example, with an application that sends bulk data in an as-soon-as-possible manner, the completion time may be sooner or later depending on whether the participating VMs are colocated or dispersed and hence the network bandwidth available in each scenario. An enhanced model that estimates the network-affinity level and resulting CPU utilization based on placement scenarios would be required.

3.8.3 Capacity planning for virtualized services

An important aspect of migration of services from physical to virtual environments (P2V) is estimation of resources needed to meet SLA requirements [61, 62, 63]. As shown in our work, colocation and dispersion of communicating tiers of an application can result in differing CPU utilization for both DomUs and Dom0. Thus, estimating CPU usage of two tiers individually for the P2V transition will neglect the effects of affinity on their CPU usage.

Resource requirement in virtualized environment depends on type of application being mi-

grated from physical to virtual. In [4], linear regression models have been built to estimate virtual CPU usage from given physical resource usage profiles. However, this has been done only for one tier (web server tier) of the application and not for the other (database tier). Thus, it does not consider both cases—colocation and dispersion—of location of the two tiers, and instead addresses only the dispersed placement by default. In our work, we have demonstrated that virtualized CPU usage of both communicating tiers are dependent on their relative placement also. Thus, during the P2V transition, virtual CPU usage estimation should also consider whether or not the communicating VMs are intended to be colocated. Our idea is that for every VM that has network communication with any other VM, the P2V CPU estimation models presented in [4] can be used to predict its dispersed virtual CPU usage, and then our *colocation* model can be applied to this estimation to predict the final virtual CPU usage.

3.9 Conclusions

In this work, we performed benchmarking, to quantify the effects of network affinity on CPU usage when communicating VMs are colocated versus dispersed. Next, we developed VM *pair-wise* models that can estimate "colocated" CPU usage, on being input their individual dispersed-case resource usages, and to estimate "dispersed" CPU usage based on colocated-case resource usages. For the "colocation" and "dispersion" models, we first built models that predicted the total CPU usage upon migration—these CPU models use all resource (CPU, disk, mutable and immutable network) usage profiles as their input. However, these models had an error of around 4%. So, next we built enhanced models to predict only the differential CPU usage—these models use only the *mutable* network traffic metrics as input, and have maximum error within 2%. Finally, we demonstrated the application of *pair-wise* models to predict for multi-VM scenarios, with high accuracy. This proves that CPU usage prediction models built on the scale of two VMs can be used for prediction in multi-VM scenarios as well.

Chapter 4

DRIVE: Using Implicit Caching Hints to Achieve Disk I/O Reduction in Virtualized Environments

4.1 Introduction

Duplicate content—memory pages and disk blocks—is a common occurrence in virtualization-based hosting of applications, where virtual machines (VMs) are instantiated from the same image template [64] or execute similar software environments. A study of content similarity amongst 525 virtual images from a production remote desktop environment [65], reported that 30% blocks were found to repeat at least twice and 12% blocks were found to repeat 5 times. A follow-up study [66] reported median of pairwise similarity across virtual machine images to be around 48%.

Content similarity within a single virtual machine image is referred as *intra-VM* similarity, whereas across multiple images, it is referred as *inter-VM* similarity. A study in [67] has reported that, in general, intra-VM similarity is significant, with up to 90% of total similarity observed among VM disk content being intra-VM. The work in [68] studied the degree of content similarity in application workloads (web, mail and file system) hosted in individual virtual machines, and reported that the amount of unique content accessed is lesser than the unique number of blocks accessed, implying that there are multiple blocks having identical content within each application workload.

Consider the system illustrated in Fig. 4.1. When an application or service (example, mail server or web server) is deployed within a VM instantiated on a host, the virtual disk corresponding to the VM is present on the host's storage. The storage may be a *local* disk or a network-attached *remote* disk. As discussed above, virtualized systems have inherent content similarity among the data content that resides on disk, and this data is fetched from disk by corresponding applications. Disk access times are usually in the order of a few milliseconds[69, 70], whereas in

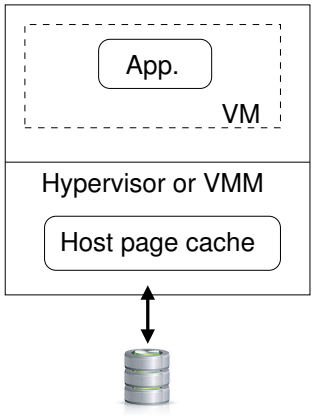


Fig. 4.1: Typical Virtualized System Under Consideration

comparison, host cache access timings are orders of magnitudes lower[71, 72]. Hence, improving caching effectiveness can, in general, help improve storage access performance and overall application performance.

Due to the inherent similarity present in virtualized systems, the effectiveness of the host's page-cache and overall application performance are impacted by two major factors: (i) *duplicate I/O* problem [68], i.e., multiple blocks containing the same content being fetched from disk, and (ii) *duplicate content* problem [72, 73], i.e., multiple blocks in cache have same content. Both the duplicate I/O problem and duplicate content problem can be addressed by actively maintaining metadata regarding the content similarity among blocks inserted into, and evicted from, the host page cache. However, actively tracking the insertion and eviction of blocks in page cache would require invasive changes in the host kernel.

Harnessing content similarity to avoid duplicate disk I/O requests is referred as I/O deduplication. To harness content similarity across different blocks, [68] suggests the use of a *content-based cache* which can be looked up by content also. The proposed approach in [68] is to maintain a *content-based cache* and build metadata regarding blocks inserted therein. A content-based cache[1] can be referenced by content (unlike a block-based cache), and hence retains only a single copy of each content. It is used to intercept disk read requests and deliver content directly, thus enabling I/O deduplication [68]. However, since only a fraction of total page cache space can be reserved as an explicit content-cache, the remaining page cache still faces the duplicate content problem (illustrated subsequently in Section 4.3).

In this chapter, we present a read I/O redirection technique positioned within the VM's virtual block device, such that both problems are addressed. If a block of content is present in the host's page-cache, it need not get fetched from disk, i.e., I/O reduction is achieved. This, in turn, implies that multiple blocks with the same content are (almost) never stored into the host's page-cache, i.e., a content-deduplicated page-cache is achieved. Thus, our work is a deduplication-inspired technique for effective host cache management.

[1]The terms *content-cache* and *content-based cache* are used interchangeably. Similarly for *block-cache* and *block-based cache*.

The aim of this work is to achieve disk I/O reduction without actively tracking the state of the page-cache, as well as without maintaining any explicit content cache. We use a custom simulator to capture the potential and benefits of the above approach. The contributions of this work are,

1. Simulation-based analysis of IODEDUP [68] system to show that it is sub-optimal in terms of achieving the goal of I/O deduplication.

2. Design and implementation of the DRIVE module that tackles both the duplicate content problem and the duplicate I/O problem

3. Extensive trace-based evaluation of DRIVE with prototype implementation within a custom simulator.

4.2 Background & related work

This section presents an overview of existing techniques for I/O reduction in virtualized data centers, then presents other page cache management techniques and their shortcomings in virtualized environment.

4.2.1 Virtualization-based I/O reduction techniques

Reduction of disk read accesses can be achieved by better host cache usage (for example, use as a content-cache [68]), better cache management (insertion and eviction) policies [74], and content-based deduplication techniques [75].

Prior work [76, 75, 77] to achieve I/O reduction in virtualized data centers focusses on content-based deduplication techniques. SeaCache[76] deploys a de-duplication system on the shared storage server in a virtualized data center, and designs efficient cache eviction algorithms to aggregate distributed host caches as a unified cache. Thus, it can be used complementarily with host cache management techniques. VMAR[75] builds a deduplication map at VM instantiation time, and redirects read requests for I/O reduction. Since a long-running application will write many blocks and subsequently read them again, the deduplication map should ideally be updated for higher efficiency. However, VMAR's choice to not gather the deduplication map dynamically, implies that many deduplication opportunities may be lost in a long-running application. Capo[77] expoits the fact that all VMs are instantiated from a relatively small number of "golden images" and generates a bitmap, which is then used to eliminate duplicate read requests. However, it can not detect duplicate requests for data outside of golden images, for example, customized VMs and workloads of virtualized applications.

In our work, we build metadata regarding virtual disk data while the virtual machine (and its hosted application) is executing, and hence the resulting I/O deduplication can detect duplicates in customized VM workloads as well.

4.2.2 Cache management techniques for I/O reduction

Memory deduplication [78, 72, 73, 79, 80] refers to storing only one copy of duplicate content in memory to save memory space and improve cache effectiveness. In I/O deduplication [68], the aim is to avoid disk I/O requests that fetch duplicate content into cache. Hence, though both memory deduplication and I/O deduplication are basically cache management techniques[76], the difference is that memory deduplication is performed after multiple blocks with duplicate content have been fetched into memory, whereas I/O deduplication seeks to avoid block fetches altogether if the same content is already present in memory. In our work, we seek to improve host cache management via I/O redirection & deduplication strategies.

Memory deduplication techniques: Previous work in [78, 79] has shown the possibility of saving memory using deduplication techniques, so as to improve the caching performance. However, the content similarity identification in these approaches is done via periodic scanning of guest VM memory. Memory scanning is beneficial for memory deduplication only when done at a high scan-rate, however, this results in higher CPU overheads. Moreover, because of their overcommitment of memory to guest VMs, the Hypervisor has to resort to paging of VM memory in order to ensure that all VM's contents are accommodated in memory, and this can lead to degraded performance.

The work in [72] develops a memory-deduplication system called Satori, in which multiple blocks in cache having same content are mapped to a single page, which is marked as *read-only*. Due to the *read-only* setting of shared pages, an attempt to write to it will cause a page fault, called *copy-on-write-fault* and the Hypervisor will handle the fault by allocating and new page frame with the page contents copied. Also, the page table mappings need to be updated to ensure that the faulting VM uses the newly allocated page henceforth.

In our work, we build content-similarity metadata by dynamic interception of read and write requests within the virtual disk driver in the VM. Hence, there is no scanning of memory and related overheads are avoided. Moreover, an update to the metadata is a "local" update as far as the VM's virtual disk driver is concerned, hence costly page faults and any related traps to Hypervisor are also unnecessary in our system. Thus, our system can achieve efficient cache usage with lower overheads than existing memory deduplication systems.

I/O deduplication techniques: The work in [68] demonstrates that a content-cache is better than a block-cache for read-only workloads, due to the inherent content similarity. However, it does not present any study of the effects of the above split-cache approach on overall cache effectiveness. Specifically, write requests attain a higher number of cache hits in a block-cache rather than a content-cache [68] whereas read requests attain more cache hits in a content-cache. Since IODEDUP has both a *block-cache* and a *content-cache*, it is essential to explore the holistic effects of reserving a fraction of the block-cache to be used as a content-cache. This is crucial because content-cache benefits may be read/write request-mix dependent, as explained next.

The split-cache approach of [68] introduces two problems: (i) cache inclusiveness problem,

i.e., same block present in both the block-cache and the content-cache, and (ii) content-cache sizing problem, i.e., determining what portion of the host's cache should be maintained as a content-cache for maximum performance benefits. To demonstrate these problems, we present results from a simulation experiment using the traces available at [81]. This study is presented in the next section.

4.2.3 Revisiting disk I/O virtualization

The de-coupling of the front-end and back-end virtual block drivers enables trapping and redirection of guest disk I/O requests before they encounter the host page cache. In our work, we exploit this de-coupling of drivers and introduce an I/O redirection mechanism such that the host cache in a virtualized system is implicitly manipulated in a content-deduplicated fashion without needing any invasive changes in functioning. In the next section, we present the design of our I/O reduction & deduplication system called DRIVE (Disk I/O Reduction in Virtualized Environments).

4.3 Analysis of existing I/O deduplication technique: IODEDUP

In the area of I/O deduplication for a host operating system's block device, the sole prior work so far is [68]. The work in [68] demonstrates that a content-based cache is strictly better than a block-based cache for most real workloads, due to the inherent content similarity present in them. Three mechanisms are used to achieve host-based I/O deduplication and I/O latency reduction in [68]—(i) A content-based cache, to store deduplicated content and respond to read requests directly instead of fetching from disk, (ii) A dynamic replica retriever, to optionally redirect the remaining read requests such that overall disk access latencies are lowered, (iii) A selective duplicator, to track frequently accessed content and dynamically create more replicas for use by dynamic replica retriever. Fig. 4.2 presents the system architecture of the I/O Deduplication system (henceforth referred as IODEDUP), wherein the storage stack's block layer is shown as being augmented with the above-listed additional functionality, labeled as the I/O Deduplication layer in totality. Although the work in [68] implements three mechanisms as described above, only the content-cache mechanism eliminates duplicate I/O requests, and hence performs I/O deduplication. Thus, we regard only the *content-cache* mechanism of IODEDUP as "prior art" in the area of I/O deduplication.

4.3.1 Motivation for in-depth analysis

Our work is motivated by a few observations from the work presented in [68]. First, it demonstrates that a content-based cache is strictly better than a block-based cache of the same size.

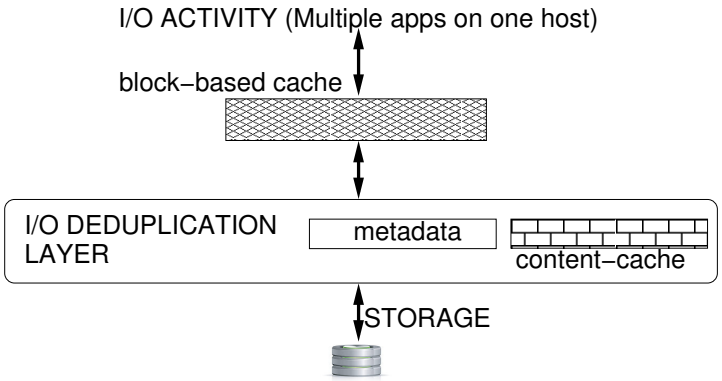


Fig. 4.2: System Architecture of IODEDUP

However, since IODEDUP has both a *block-cache* (i.e., page/buffer cache) and a *content-cache*, it is essential to explore the holistic effects of subtracting from the existing block-cache size while creating the new content-cache. Second, since the *content-based* cache basically takes away a finite amount of memory space available to the buffer/page cache, a study is needed to learn the optimal content-cache size per workload. This is especially important because content cache usage/benefits may be application/workload or request-mix specific. Based on above observations, we claim that a split-cache approach of sub-dividing available memory space into block-based and content-based caches, is sub-optimal. More specifically, our claims are the following.

- As the evaluation of IODEDUP shows [68], a content-based cache succeeds in improving cache hit ratio for a read-only request stream, but performs sub-optimally with a read/write request stream. This was attributed [68] to the journaling process writing varying content to same numbered sectors repeatedly.
- The split-cache approach has a sweetspot-finding dilemma, wherein the content-cache needs to be correctly-sized to allow better cache utilization.

4.3.2 Simulation setup and workloads

To support our claims, we built a custom simulator (refer Appendix chapter II for simulator details) with prototype implementations of the Vanilla and IODEDUP [68] systems. The block-cache is looked up based on block numbers as usual, whereas the content-cache can be looked up based on both block number and content. IODEDUP maintains metadata containing MD5 [82] hashes of content, to identify similarity.

A high-level overview of the working of Vanilla and IODEDUP [68] systems is as follows. In Vanilla system, a read request's block number is first looked up into block-cache. Upon hit, content is served whereas upon miss, the block is fetched from disk. In I/O Deduplication system, the original disk fetch path due to block-cache miss, is intercepted and metadata is used to lookup the block in the content-based cache. Upon a hit in content-cache, the content is returned straight-away to callee and thus, the disk fetch is *averted*. However, upon a content-cache miss, a disk fetch becomes inevitable, and the fetched content is finger-printed by a hash function like MD5

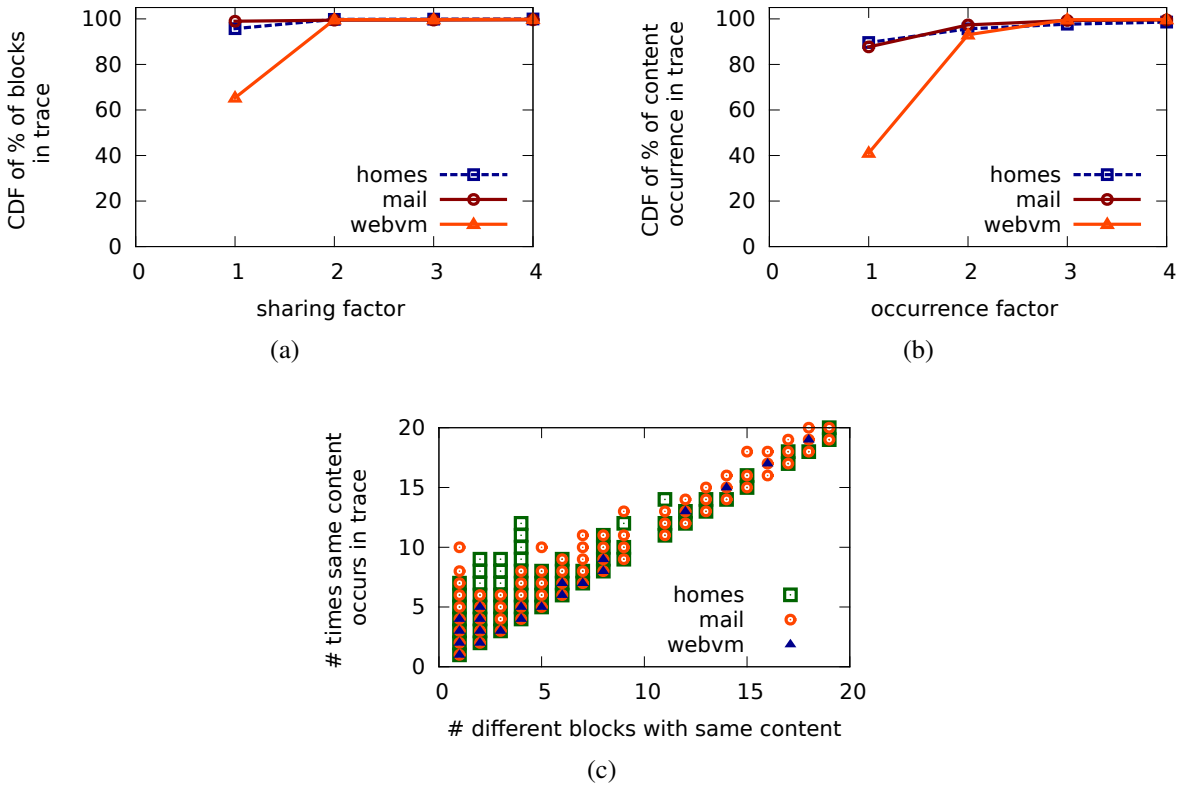


Fig. 4.3: Study of content similarity in *webvm*, *homes* and *mail* trace workloads.

Table 4.1: Summary statistics of traces used for evaluation

Trace Name	**Num of read requests**	**Num of write requests**	**Max sharing factor**	**Max occurrence factor**	**Write intensivity factor**
homes	4,052,176	17,110,222	5904	5905	4.2
mail	2,375,409	18,007,471	57015	57015	7.6
webvm	3,116,456	11,177,702	124	125	3.6

and metadata is updated. For our exploration of IODEDUP system's effectiveness, we chose six content-cache size settings as 10%, 20%, 30%, 50%, 70% and 90% of the total memory size, and remaining as the size of block-cache. We replay each trace with a total memory size of 1 GB or 512 MB, as specified in each experiment respectively.

Evaluation in [68] is performed with traces over 21 days from three production systems at FIU Computer Science Department: (i) *webvm*, traces from two virtualized web-servers hosting webmail proxy and course management system, (ii) *mail*, traces from an email server, and (iii) *homes*, traces from a file server. From the above, we consider the entire available *webvm* and *homes* trace, and one day's trace for *mail* workload, for our evaluation.

4.3.3 Study of similarity in workloads under consideration

To study the content similarity in the workloads under consideration, we identify the two factors that contribute to the similarity: (i) content similarity in the data on disk (irrespective of workload), and (ii) access pattern of these similar blocks in the workload. Basically, even if all blocks on disk have duplicates, the I/O workload may still not have any duplicates if the access is such that identical content blocks are never accessed in the same workload. On the other hand, even if only a small percentage (say 10%) of blocks have duplicates, the I/O workload can still have a high deduplication potential if most of the blocks being accessed in the workload are from this small set of duplicated blocks. To distinguish these two factors, we define the following:-

1. *Sharing factor* refers to different blocks (i.e. block addresses) "sharing" identical content.

2. *Occurrence factor* refers to occurrences of identical content in the workload, where "occurrence" of identical content can be due to the same block or due to different blocks having identical content.

Since we do not have access to the actual underlying data for these traces, the block addresses present in the traces are our only clue to the content similarity present in the dataset. Thus, we compute "sharing factor" as an approximation of the static content similarity in the data. Further, although presence of static content similarity is necessary for I/O deduplication efforts to be useful, it is not a sufficient condition. This is because no I/O deduplication can occur if those identical blocks are not actually accessed by any workloads. Thus, we compute "occurrence factor" as an indicator of identical content access in the workloads.

Fig. 4.3(a) plots a CDF of the percentage of *sharing*. We can see that in *homes* and *mail* traces, more than 95% content is present in only one block, whereas in *webvm* trace, around 35% content is present in two blocks. Fig. 4.3(b) plots a CDF of the percentage of occurrences of identical content, where "occurrence" of identical content can be due to the same block or due to different blocks sharing the same content. It is observed that, in *webvm* workload, 45% content occur twice, whereas *homes* and *mail* workloads have only 6-10% content occurring twice. The difference between "sharing" factor and "occurrence" factor is that, when a single block address occurs multiple times in the trace, it is counted as only one when computing the "sharing" factor of that block's content whereas every occurrence contributes to the "occurrence" factor of that content.

In Fig. 4.3(c), we present a scatter-plot depicting for every observed sharing factor, what are the various number of occurrences observed in the trace for that content. Each point in the scatter-plot represents one or more chunks of unique content with sharing factor as depicted on x-axis and corresponding number of occurrences on y-axis.

Although the x-axis in Fig. 4.3(a) and 4.3(b) is cropped to value of 4, the maximum sharing and occurrence factors are much higher, and are listed in Table 4.1. The table presents an overall

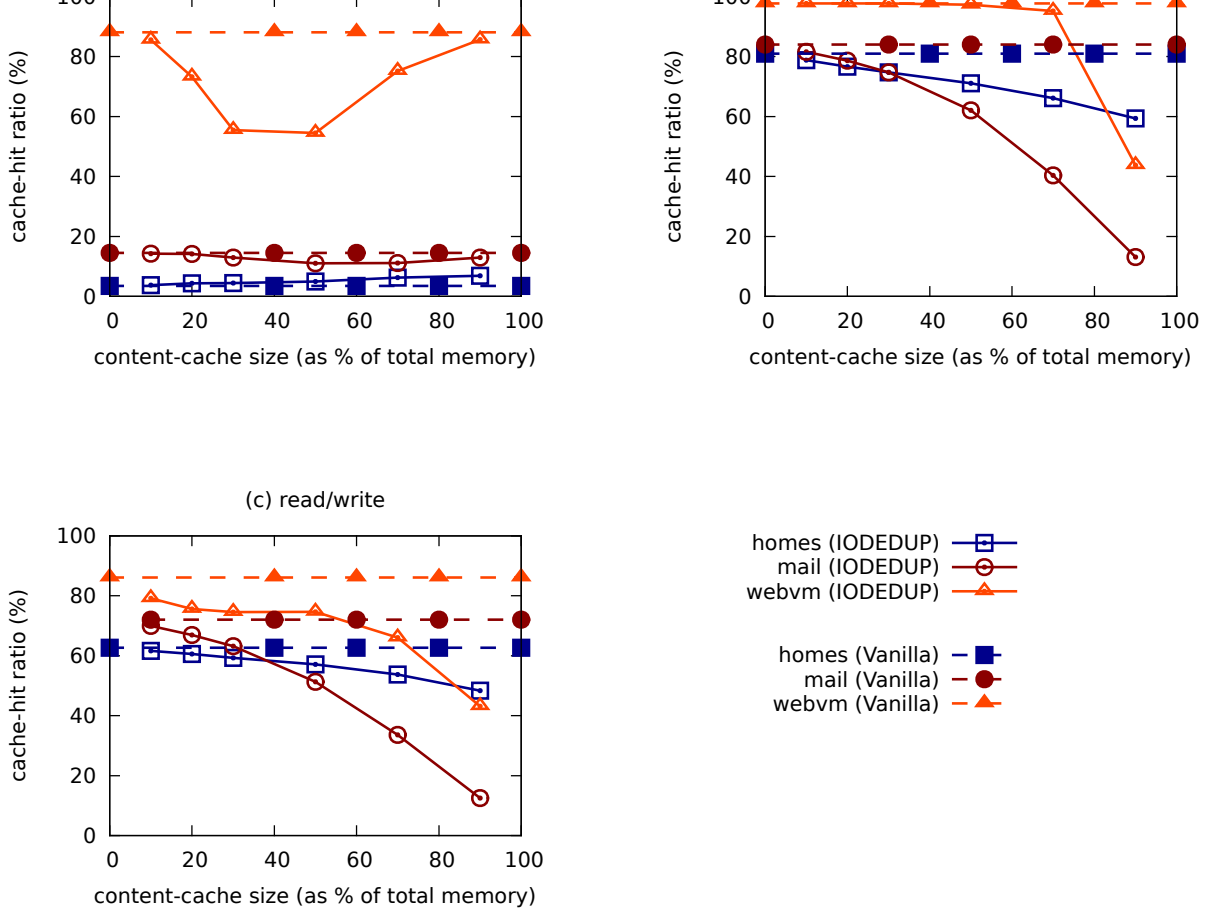


Fig. 4.4: Cache-hit ratios for IODEDUP upon *webvm*, *homes* and *mail* workloads. The total cache size is 1 GB.

summary of the traces, listing number of read and write requests, and maximum *sharing*, *occurrence* and *write-intensivity* factors of each trace. We define *write-intensivity* factor as the ratio of number of blocks written to number of blocks read. Since we perform read I/O redirection and do not optimize writes, this factor will impact effectiveness of cache, as explored later.

The above study indicates that *webvm* workload is a better candidate to benefit from I/O redirection & deduplication techniques. However, we use all three workloads to perform our experiments. We wish to demonstrate that if a workload has "any" significant level of duplication, I/O deduplication can be beneficial *if and only if* performed effectively. By this, we intend to emphasize that although the I/O deduplication work [68] seems to perform well for read-only workloads, it still fails to perform well under realistic workloads like read-write workload and under practical conditions where a block cache is irreplaceable/unavoidable. In such realistic scenarios, we claim that our algorithm works far better as demonstrated in following sections.

4.3.4 Comparison of cache-hit ratios

Fig. 4.4(a) shows the cache hit ratio for read-only request traces. For the *homes* and *mail* workload, at all settings of content-cache size, IODEDUP has higher cache-hit ratio than Vanilla. Also, the cache-hit ratio increases as the % of content-cache size increases. However, for the *webvm* trace, IODEDUP performs worse than Vanilla at all settings. This is concluded to be because 1 GB total cache size is already sufficient enough for Vanilla system to cache all relevant blocks, hence achieving a high cache-hit ratio of almost 90%.

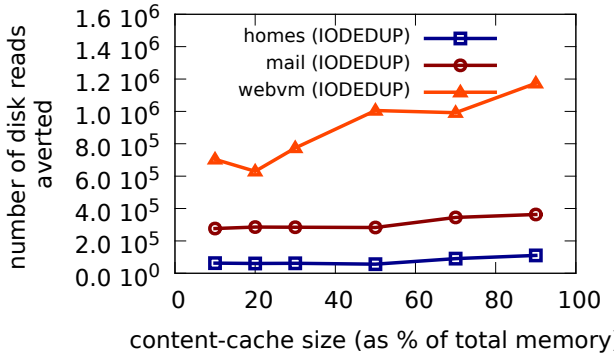


Fig. 4.5: Disk reads averted for IODEDUP upon *webvm*, *homes* and *mail* workloads. The total cache size used is 1 GB.

Fig. 4.4(b) shows the cache hit ratio for write-only request traces. Disk accesses with Vanilla block-cache has a higher number of cache hits than IODEDUP split-cache. This is due to varying content being written repeatedly to the same locations by journalling processes. For example, non-identical content written to the same block location intermittently, results in cache-hit in the vanilla block-cache (due to block number match) but causes cache-miss in the IODEDUP content-cache (due to content mismatch). This is also the reason why, as the content-cache size increases and block-cache size decreases, the cache hit ratio for write request streams decreases. This is true for all three workloads considered, and the sweetspot for highest cache-hit ratios can be considered to be at 100 MB for all three.

Fig. 4.4(c) shows the cache hit ratios for traces considering both reads and writes. As can be seen, the read-only and write-only curves seem to merge in an additive fashion to form the curves for the read/write traces for all three workloads. For example, in spite of very low cache-hit ratio in the read-only case, the *homes* and *mail* workloads have significantly improved cache-hit ratios in the read/write replay. This is attributed to the higher number of cache-hits incurred by the write requests (as seen in Fig. 4.4(b)).

From above experiments, we see that for each request-mix type, IODEDUP has a different optimal setting for content-cache size – 90% (900MB) for read-only, 10% (100 MB) for write-only, and 10% (100 MB) for read/write. These settings result in the highest cache hit ratios for the three workloads using IODEDUP system. Since real workloads would be read/write workloads, we consider 10% (100 MB) as the sweetspot setting for the rest of the evaluation in this thesis.

4.3.5 Comparison of number of disk reads averted

Next, let us take a look at the metric of "disk reads averted", i.e., disk reads avoided owing to cache hits. The difference between the two metrics of *cache hit ratio* and *disk reads averted* is that even though the cache experiences churn due to both read and write requests, the former metric captures all cache-hits whereas the latter captures the cache-hits occuring on account of read requests only.

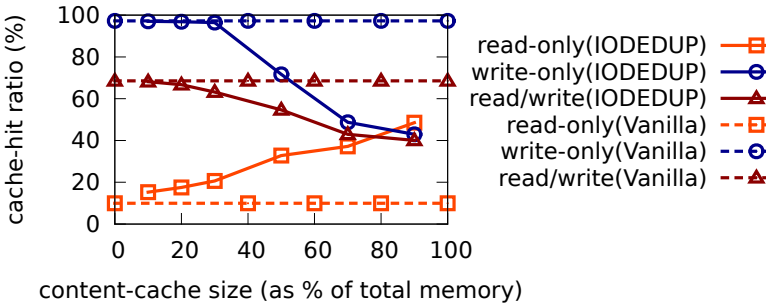


Fig. 4.6: Cache-hit ratios for IODEDUP upon *webvm* workload. Total cache size 512 MB.

Fig. 4.5 shows the number of disk reads averted in each of the traces in presence of both reads and writes. In all three workloads, when content-cache size increases from 10% to 90% of the total size, the number of disk reads averted almost doubles for all three workloads. Judging by number of disk reads averted as a measure of performance, a setting of 90% for content-cache size seems to be the sweetspot for all three read/write traces. However, this is contrary to our earlier conclusion that 10% is the sweetspot setting.

4.3.6 Effect of excess memory pressure

We claimed above that if the total memory size is already enough to sustain most block requests, an explicit content-based cache does not add much value and instead degrades the performance (refer *webvm* workload in Fig. 4.4(a)). To substantiate this claim, we performed trace replay for *webvm* workload with a total memory size setting of only 512 MB. The cache-hit ratios for read-only, write-only and read/write replays are illustrated in Fig. 4.6. We can see that the read-only trace exhibits lowest performance at a content-cache setting of 10% and the highest at 90%. However, the performance of write-only trace and read/write trace varies inversely as compared to read-only trace, achieving the lowest performance at a content-cache setting of 90% and the highest at 10%. Since Vanilla does not perform as well here as with the 1 GB setting, IODEDUP succeeds in improving the performance by almost 4× for the read-only trace at a content-cache setting of 90%. However, note that performance of IODEDUP for both write-only and read-/write traces still remain below par, up to 55% and 42% worse than Vanilla, respectively.

The above analysis shows that the dual metrics of *cache-hit ratios* and *disk reads averted* have different behaviour in relation to the content-cache sizing of IODEDUP system. Specifically, for a total memory size of 1 GB, highest cache-hit ratios are achieved at a setting of 10% (refer Fig. 4.4) and highest number of disk reads are averted at a setting of 90% (refer Fig. 4.5). Moreover, the split-cache approach of IODEDUP results in duplicate content across the block-cache and content-cache, hence degrading the overall performance (refer Fig. 4.4). We conclude that the split-cache approach performs sub-optimally, and establish the necessity for an I/O dedupli-

cation solution that can perform effectively even in presence of write-intensive workloads.

4.4 DRIVE system requirements & design

Since our aim is to achieve content-deduplication of the block-cache, we build an I/O redirection system positioned above the block-cache which uses implicit caching hints to perform the redirection. In this section, we first present the requirements of such an I/O redirection system and then present our design of the DRIVE (Disk I/O Reduction in Virtualized Environments) system that meets the specified requirements.

4.4.1 System requirements

To perform I/O redirection, metadata needs to be maintained regarding the similarity of content present in read and write requests. Due to inherent content similarity in the workloads, multiple blocks (i.e. blocks with distinct block ID) have identical content, and can be represented by a single abstract entity, called a a *deduplicated block*. The requirements of an efficient I/O reduction system can be stated as follows,

1. Fingerprinting mechanism to identify content similarity across different blocks.

2. Data-structures to store metadata for similar blocks.

3. Maintaining implicit caching hints within metadata to aid future I/O redirection.

4. Interception of block read *request* path for metadata lookup and I/O redirection, if present.

5. Interception of block read *return* path for metdata update, if not previously present.

6. Interception of block write *request* path for metadata invalidation.

7. (Optional) Interception of block write *return* path for metadata update.

The above interceptions of block read path may result in longer flow paths, however, the increased number of cache-hits are expected to outweigh the effect of increased latencies. Additionally, the interceptions of block write path should not be too "costly" if the underlying cache is in *write-back mode*. However, if the cache is in *write-through mode*, the extra latency due to metadata update may be considerably negligible. In the rest of this section, we present the design of the DRIVE system which addresses all the requirements discussed above.

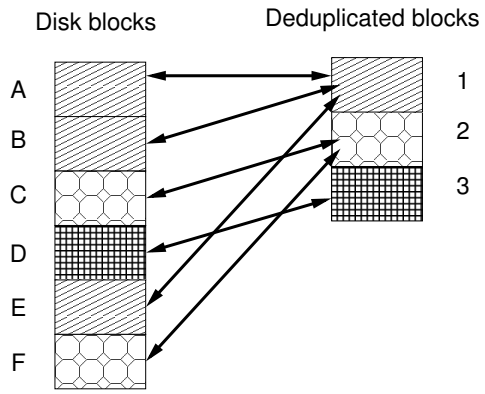


Fig. 4.7: Semantics of metadata store in DRIVE system: *each block points to a unique deduplicated block, and each deduplicated block reverse maps to multiple actual blocks*

4.4.2 System design

The DRIVE system has three main components: (i) Metadata maintenance, whereby metadata is maintained to indicate the similarity among blocks being accessed in every read and write request, (ii) Maintaining implicit hints regarding host-cache state, wherein the blocks which have been recently fetched are noted as *implicit hints* that they are more likely to still be in cache when requested next, and (iii) Hint-based read I/O redirection, wherein previously stored *implicit hints* are used to substitute, if possible, the block ID present in the read request with another block ID of duplicate content. Next, we explain each of these aspects in more detail.

Metadata maintenance

The metadata store contains information regarding identical blocks, where every block is represented by a block ID, and multiple identical blocks (with distinct block IDs) are mapped to a unique abstract block of content called a *deduplicated block*. A *deduplicated block* is associated with, and identified by, a fingerprint of the content. Thus, each block maps to a unique deduplicated block and every deduplicated block can potentially reverse map to multiple blocks in the system. This is depicted in Fig. 4.7.

Metadata can either be built by an *a-priori* scan of the entire file-system/disk or, at *runtime* as and when read/write requests are serviced. The advantage of building metadata a-priori is that when a particular block is requested, its metadata may already be available, whereas in case of runtime metadata updates, mapping information for every block can be known only after the block has been fetched from disk the first time. On the other hand, an advantage of runtime metadata updates is that, metadata needs to be stored only for those blocks that are in the workload, hence saving space, as compared to an a-priori scan that will build metadata of the entire disk space. We choose to build metadata at runtime in the DRIVE system because in most workloads, only a small fraction of the entire file-system is actually accessed from the disk [68].

Metadata is to be built/updated for every new block of content, to ensure that it is up-to-

date for customized VMs and long-running applications. In case of read and write requests, block content is encountered in two ways: (i) every block of data written to cache for later flushing to disk, (ii) every read request for an as yet unseen block (and hence) fetched from disk. In the case of read requests, we compare the content fingerprint with existing fingerprints to determine whether it is "new" content or "duplicate" of previously seen content, and update metadata accordingly. In case of write requests, we invalidate existing mappings for those blocks, so that stale metadata is not used for the next read request redirection.

Maintaining hints regarding host-cache state

When a read request from the front-end driver is serviced by the back-end driver, it is obvious that the physical block corresponding to the requested block ID has been recently brought into the host-cache. Thus, if the delivered content is found to be identical to any other previously requested block, it is not only noted so in the metadata, but also the most recently fetched block ID is appointed as the *leader*. This appointment as *leader* is done to aid redirection of future I/O requests that request the same content, and is a *hint* regarding the state of host-cache. Since we do not wish to actively track the host-cache state (i.e., by trapping and intercepting each insertion and eviction within host-cache), these are merely hints and do not guarantee a cache-hit upon I/O redirection. However, since the hint indicates the most recent block ID with identical content having been fetched into host-cache, picking that block ID for redirection is our best chance of encountering a cache-hit to fetch the desired content.

Read I/O redirection using implicit hints

The DRIVE system uses metadata to achieve non-invasive cache manipulation by redirection of read I/O requests. Every block read request is intercepted and metadata looked up to retrieve the corresponding deduplicated block ID. If the deduplicated block reverse maps to multiple blocks, it indicates that the requested block has identical content as each of the reverse-mapped blocks. Thus, fetching any of those blocks would suffice to retrieve the content being requested, instead of fetching anew from disk. Note that, among the reverse mapped blocks, we have flagged the most recently fetched block ID as *leader*, as described above. Hence, after interception of the read I/O request, we replace the original block ID with the *leader* block ID.

A simple example to elaborate on the above redirection approach, is visually illustrated in Fig. 4.8. Suppose blocks A, B and E have identical content, and hence are mapped to deduplicated ID 1 (refer Fig. 4.7), and E is appointed *leader* since it was the most recently fetched among the three. Whenever next, read requests for A, B or E are received, redirect the request to always read block E such that the number of times that block E or its duplicates are requested, governs the popularity of block E in the cache. Note that, blocks A, B and C will be in cache the first time that they are fetched but once the caching policy eventually evicts them, they will never enter cache again, unless writes are done to them. This implies that the cache is operated

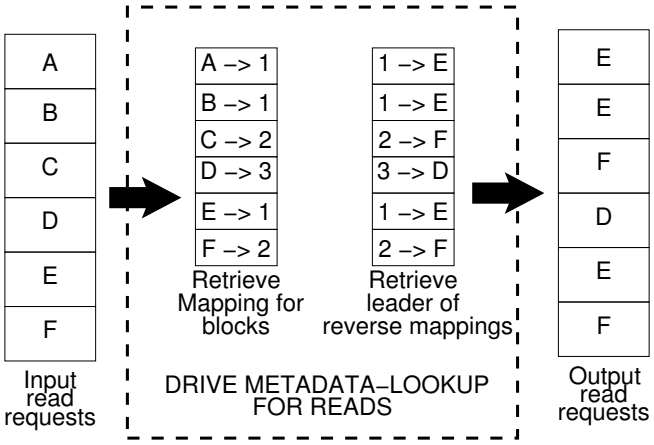


Fig. 4.8: Example of read request redirection in DRIVE system

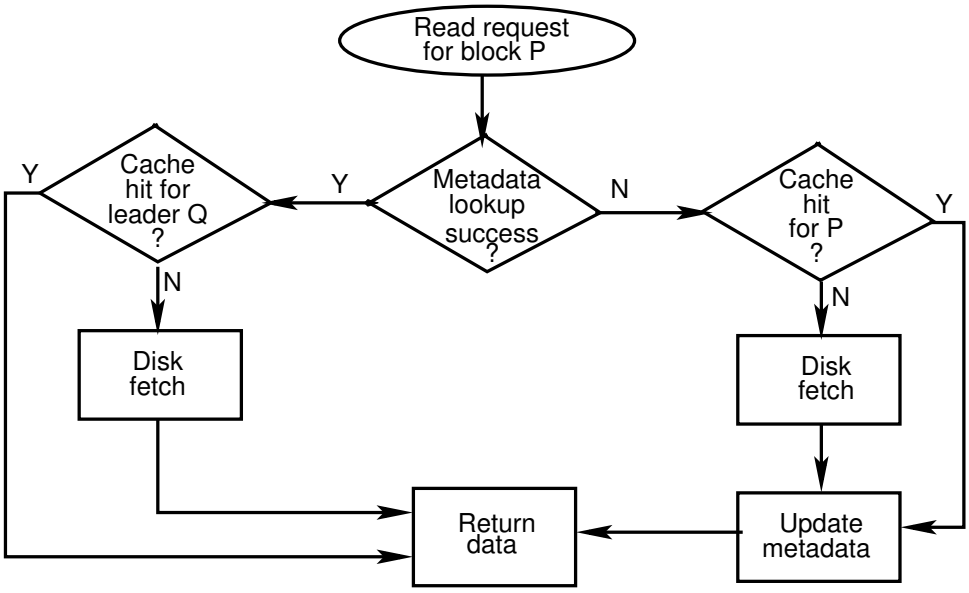


Fig. 4.9: Flow path(s) for read requests in DRIVE system: *P is the requested block ID and Q is the corresponding "leader" block ID.*

in an almost fully content-deduplicated fashion, hence improving its effectiveness in reducing the number of disk reads.

The above example assumes that all blocks being requested, already have metadata available. However, in case of runtime mapping, the first occurrence of every block will result in a metadata lookup failure and a mandatory disk fetch. Thus, a read request can have multiple flow paths: (i) metadata lookup failure, resulting in fetch from disk, (ii) metadata lookup success, but cache miss, resulting in fetch from disk, and (iii) metadata lookup success, and cache hit. If we maintain metadata store also as a limited-size cache, we would encounter two additional flow paths: (iv) metadata lookup failure, and cache miss (if previously non-leader block), resulting in fetch from disk, and (v) metadata lookup failure, and cache hit (if previously leader block). Fig. 4.9 depicts these multiple flows, wherein an incoming request for block P is looked up in metadata store and if successful, redirected to block Q. It follows that, Q equals P, *iff* P is itself

the *leader* in the corresponding reverse mappings.

4.5 DRIVE system implementation

In the previous section, we described the design of DRIVE system, including the rationale for various design decisions. In this section, we discuss its implementation and associated overheads.

4.5.1 Core idea

To identify intra-VM similarity, the DRIVE module is to be deployed within the front-end driver of the VM. When a read request for block ID (say P) is received at the front-end driver, metadata is looked up to find the associated *leader* block ID (say Q). If metadata is available, the frontend driver constructs a read request descriptor for block Q instead of block P, and forwards it to the backend driver for I/O. After receiving response from the back-end driver, the front-end driver delivers the content buffer back to requesting application. In case the metadata for block P was not available above, the frontend driver forwards the request as is. When response is received by the frontend driver for a block request that previously had no metadata, the received content is fingerprinted and compared with the hash-table to determine whether the content is new or duplicate, and metadata updated accordingly.

4.5.2 Metadata store

As discussed earlier, a *deduplicated block* is an abstract block which represents a block of unique content in the system. Since metadata lookup is done in both read and write paths, we conceived data structures to keep lookup times low. Fig. 4.10 shows a summary view of data structures used to implement metadata store in our implementation. It consists of an array containing entries for every block being addressed, a hash-table containing entries for every deduplicated block, and an array of pointers indexed by deduplicated block ID, pointing to the corresponding entry in the hash-table. Each entry in the hash-table corresponds to one deduplicated block, has a list of reverse mapped block IDs, as well as the associated content fingerprint. The list of reverse mapped block IDs in the deduplicated block entry indicates that each of those blocks has the same content as represented by the deduplicated block.

Duplicate content identification is accomplished using the content fingerprint as a key to lookup and find the corresponding hash-table entry. The fingerprint technique, used for duplicate content identification, can be MD5[82], SHA-1[83] or SHA-2[84], or any other technique which can be assumed to be hash collision-resistant. Usage of MD5 or SHA fingerprints to assess content similarity is established practice in existing work [65, 68, 85, 86, 87] and it is known

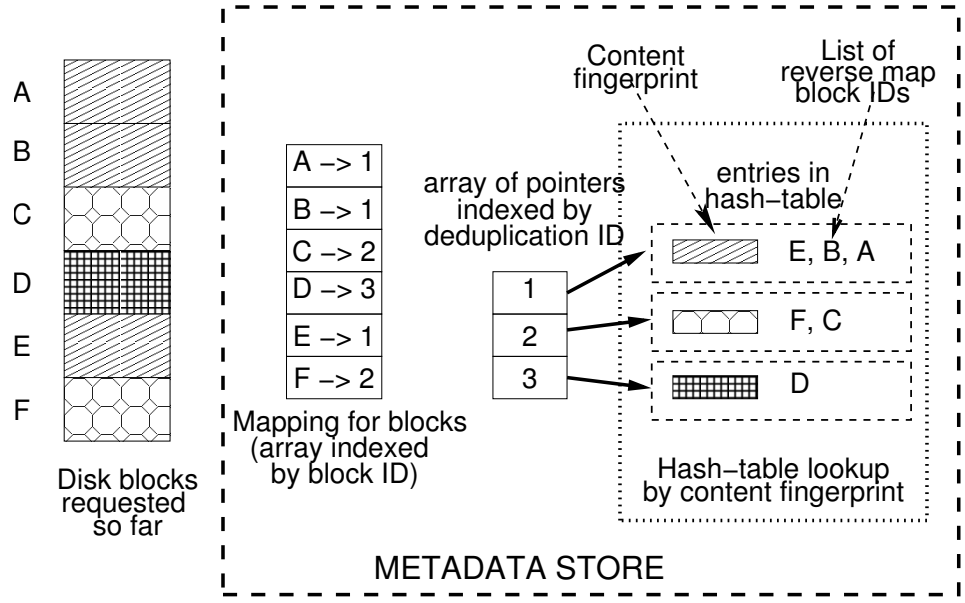


Fig. 4.10: Implementation of metadata store: *this figure shows the population of various data structures for the example introduced earlier in Fig.* 4.7.

that probability of (error caused by) a hash-collision in SHA fingerprinting is much lower than an arbitrary hardware error [88]. It is argued [88] that "compare-by-hash" is not a satisfactory technique to establish block content similarity, with suggestion to use combination of hash comparison with other methods like Rabin fingerprinting [89] to ensure correctness. We use MD5 fingerprints in our prototype implementation, however any collision-resistant fingerprinting module can be plugged into our implementation and would not have significant impact on the results presented in this work.

4.5.3 Metadata update and implicit hints about host-cache state

In the case where metadata was not previously available, and a content buffer has been received from the back-end driver, the DRIVE module within the front-end driver performs fingerprint comparison into the hash-table to determine whether the content is unique or a duplicate. If the content is unique, a new deduplication ID is assigned (say X) and a mapping from block ID P to X is registered, an entry for the new deduplication ID is made into the hash-table with the content's fingerprint as hash key, and the original block ID P is initialized as the only member of the list of reverse mappings in that hash-table entry. On the other hand, if the content is a duplicate, a hash-table entry already exists, so the block ID P is added to the associated list of reverse mappings.

Within the list of reverse mappings associated with each deduplicated block entry, the most recently fetched duplicate block (*leader*), is to be used for I/O redirection. To implement this, the list of reverse mappings is maintained in Last-in-First-Out (LIFO) order, and block IDs are added to the list as and when duplicate blocks are encountered. If the *leader* block content changes due to a write request, it is removed from the list of reverse mappings and the next most

recently fetched block ID is considered the new *leader*. Continuing with the example of Fig. 4.8, if block E (a *leader*) is written, metadata for deduplicated block 1 would change, hence picking a new leader, i.e. the next most recently fetched block B.

4.5.4 CPU overhead

Hashing of data, and hash-table lookups during read and write operations, result in resource overhead. As compared to Vanilla system, the DRIVE system adds CPU overhead in following scenarios: (i) Every read request requires a metadata lookup for I/O redirection, (ii) In case of non-existent metadata for a read request, metadata update is required after data fetch, and (iii) Every write request requires metadata lookup for invalidation. We analyze CPU overhead incurred for the two main cases of "metadata lookup" and "metadata update".

Metadata lookup consists of the following three steps. The *first step* uses block ID as an index into an array to get the corresponding deduplication ID. If metadata for a block is non-existent or dirty, the array lookup will fail, however if metadata exists, the *second step* uses the deduplication ID to index into another array to get the deduplication entry. The *third step* is to read the *head* of the list of reverse-mappings to get the *leader* block ID for I/O redirection. All the above three steps require constant time, hence metadata lookup times are O(1).

Metadata update operation can be of two types—creating new metadata for unique blocks, or modifying existing metadata for duplicate blocks. The dominating component of the metadata update operation is MD5 hash computation which is known to take around 100,000 CPU cycles[68] or 33μs on a 3GHz machine. The computed MD5 hash is then used in a hash-table lookup to locate a hash-table bucket. The hash-table size and the uniformity of entry distribution across buckets would determine the number of entries in each bucket. An empty hash-table bucket signifies that a new entry is to be created. However, if the bucket is non-empty, comparison of MD5 hash to the fingerprint stored within each entry is done until exact match is found. Higher the number of entries per bucket, longer this match-lookup time; however given enough number of buckets, hash-table lookup time would be O(1).

4.5.5 Memory overhead

In our implementation, we use an array indexed by block ID, another array of pointers indexed by deduplication ID and a hash-table containing deduplication entries. Assuming a 500GB file system, the first array would contain 500GB/4KB = 125,000,000 number of entries of 4 bytes each, which equals 500MB of space which is too huge. However, it is observed that only a small portion of the file system is accessed over long periods of time [68], hence we can instead use a hash-table implementation to store only those block IDs which have been requested so far. Thus, using a million-entry hash-table for block ID lookup, we would need only 4MB of space. However, this change would result in the metadata lookup time increasing slightly.

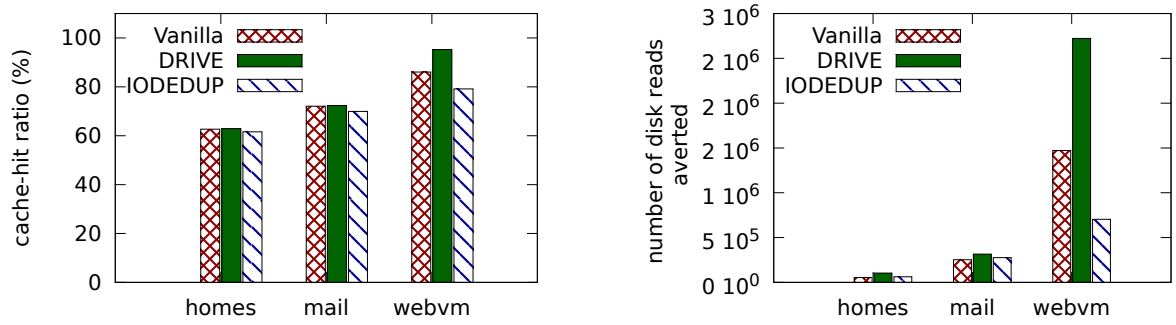
Next, we consider the size of the array of pointers indexed by deduplication ID. This structure would have only as many entries as there are deduplicated (or unique) content encountered during read operations. Assuming that 4 blocks have similar content on average (as reported in [68]), 4 million blocks read would result in 1 million deduplication indices. With 8 bytes per pointer, this results in 8MB of space. Third, we consider the memory used by the hash-table. Each deduplication entry is associated with an MD5 fingerprint of 16 bytes and a list of reverse-mappings. Due to content similarity of 4 assumed above, every deduplication block entry would have a list of size 4, i.e. 16 bytes. Thus, a single deduplication entry would amount to 32 bytes, and a million such entries would require 32MB. Thus, the total memory overhead incurred by DRIVE is 44MB, which is approximately 4.4% of 1GB.

Note that a trade-off exists between metadata lookup time and metadata space usage. However, studying this trade-off is not within current scope, and is left for future work. Nevertheless, we indeed make a humble attempt to minimize space usage to the extent possible while achieving constant or O(1) lookup timings.

4.6 Experimental evaluation

In this section, we present experimental evaluation to demonstrate benefits of the DRIVE-based I/O access mechanism. We also present comparison with the Vanilla and IODEDUP based I/O access mechanisms. Our evaluation employs a trace replay method along-with a custom simulator, `SimReplay`. For this, we need block-level traces for multiple virtual machine disk I/O activity. With due credit to Ricardo Koller et. al. [68, 81], we borrow traces made available online at [81] and use them for our evaluation.

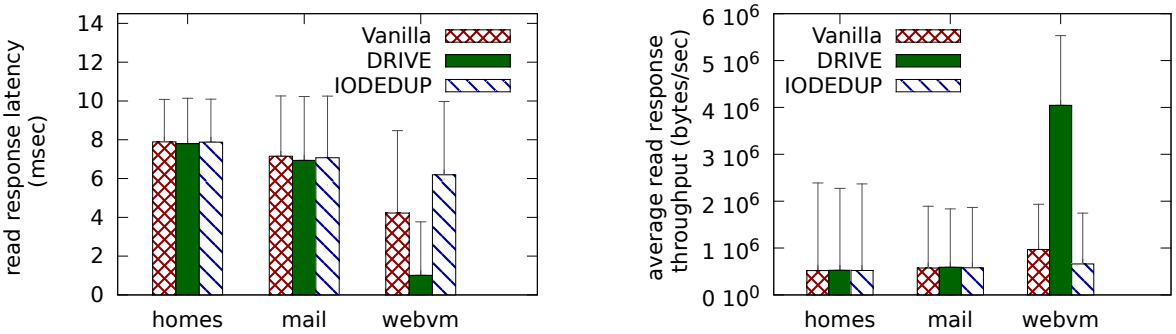
The trace data set consists of disk read and write request traces over 21 days from three production systems at FIU Computer Science Department: (i) *webvm*, traces from two virtualized web-servers hosting webmail proxy and course management system, (ii) *mail*, traces from an email server, and (iii) *homes*, traces from a file server. These are the same traces that were used earlier in the evaluation of IODEDUP system (in Section 4.3). From the above, we consider the entire available *webvm* and *homes* trace, and one day's trace for *mail* workload, for our evaluation. The primary components of every record in the trace file are:- (i) the block number to be read or written, and (ii) content (MD5 hash of content) read/written. Requests from the traces are replayed one after the other, and measures like cache hits, cache misses, disk fetches are recorded within our custom simulator `SimReplay`. First, we present a brief description of our custom simulator, and then present our trace-based evaluation results.



(a) Cache-hit performance

(b) Disk read reduction

Fig. 4.11: Comparison based on *measured metrics* for read/write traces



(a) Read response latency

(b) Read response throughput

Fig. 4.12: Comparison based on *derived metrics* for read/write traces

4.6.1 Simulator extension for evaluation of DRIVE

To evaluate the potential of our approach with respect to the Vanilla and IODEDUP systems, we extend our custom simulator, `SimReplay`. For simulation of DRIVE, we perform I/O deduplication and redirection in the virtual block address space itself, i.e., before using the V2P map to map from virtual address space into physical block address space. More details presented in Appendix chapter II.

4.6.2 Evaluating performance with read/write workloads

In this experiment, we replay both read and write requests of the traces in our simulation module, and count the number of cache-hits incurred. Note that, the number of cache hits incurred exclusively for read requests is equivalent to the number of disk reads reduced. With this in mind, we report two variations of the cache hits metric: (i) cache hit ratio, and (ii) number of disk reads reduced. The difference between these two metrics is that even though the cache experiences churn due to both read and write requests, the former metric reflects all cache-hits whereas the latter metric reflects the cache-hits occurring on account of read requests only. We use a total memory size of 1GB, and consider 10% as content-cache size for IODEDUP in all further experiments.

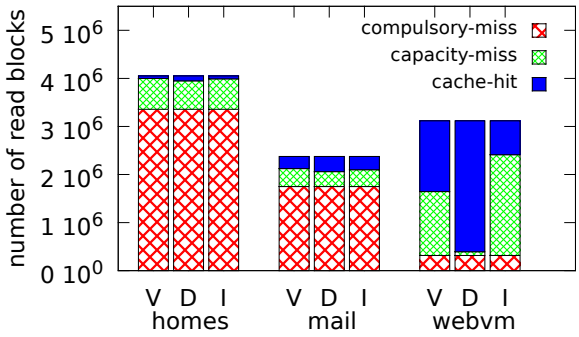


Fig. 4.13: Classification of read responses in Vanilla Vs IODEDUP Vs DRIVE for the *homes*, *mail* and *webvm* read/write traces.

Fig. 4.11(a) presents the cache hit ratio per workload for Vanilla, IODEDUP and DRIVE. Whereas IODEDUP performs slightly worse on occasion, DRIVE always performs equal or better than Vanilla. For example, with *webvm* workload, DRIVE has higher cache-hit ratios than both Vanilla and IODEDUP, 10% and 20% better, respectively. Fig. 4.11(b) presents the number of "disk reads reduced" and it shows that DRIVE performs significantly better at this metric, with nearly 85% improvement over Vanilla, and a 2.8× improvement over IODEDUP. As observed earlier, the *webvm* trace has the highest amount of content deduplication, and hence benefits the most from our I/O reduction techniques. Thus, given any disk access pattern with significant content similarity across blocks, our system can harness the similarity to improve host cache performance.

Fig. 4.12(a) shows the average of read response latencies for each trace, with the standard deviation plotted as yerrorbars. Mean cache-read and disk-read latencies are assumed as 230ns and 8ms, respectively [90], and individual request cache-read and disk-read latencies were sampled from normal distributions. Owing to the greater reduction in number of disk reads achieved by DRIVE, the average read response latency is lower than Vanilla and IODEDUP. Consequently, the average read response throughput is higher using DRIVE, as shown in Fig. 4.12(b).

In the operation of any cache, the first read request for every block would certainly result in a cache-miss. Such misses are referred as *compulsory misses* (or *primary misses*), and their number is equal to the number of different block addresses accessed in the trace. Remaining requests in the trace would be serviced from either cache (if present) or disk (if evicted from cache). Those blocks that are fetched from disk, after having been evicted from cache at least once, are referred as *capacity misses* (or *secondary misses*). Our performance comparison is based on reduction in number of capacity misses, and increase in number of cache hits.

Fig. 4.13 presents a classification of the total number of read requests into *compulsory misses*, *capacity misses* and *cache hits*, for each workload under each system: V for Vanilla, C for DRIVE and I for IODEDUP. We can see that both the *homes* and *mail* workloads have a huge number of compulsory misses, whereas the *webvm* workload has significantly fewer. In terms of capacity misses for the *webvm* workload, DRIVE succeeds in reducing the number down to 5% compared

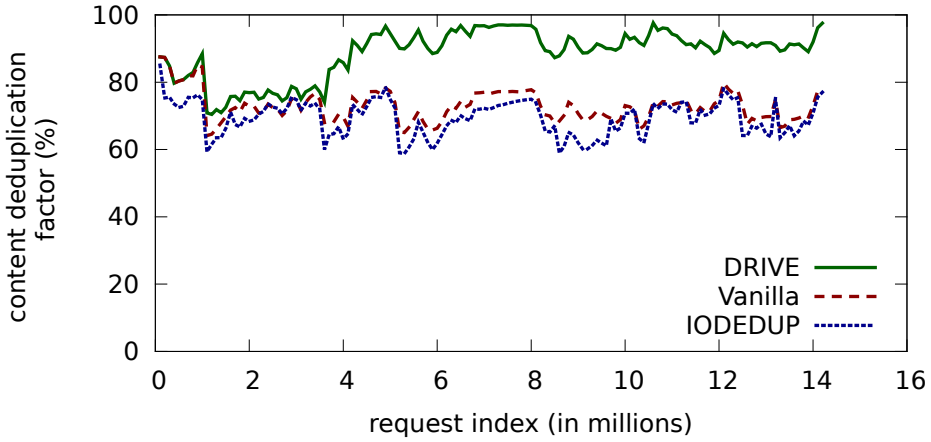


Fig. 4.14: Content deduplication factor of page cache upon *webvm* trace.

to those incurred by the Vanilla system. For the *homes* and *mail* workloads, DRIVE enables only marginal performance improvement. This is because, the large number of compulsory misses in these workloads implies that metadata is not available for many of the blocks being requested. Note that, most compulsory misses occur during *cache warm-up* phase of any system. Once the cache warm-up phase is over, a majority of the read requests would be for blocks that have been previously read or written. Hence, performance of DRIVE system will only improve.

4.6.3 Content deduplication in host cache

We claim that the DRIVE system effectively manipulates the host page-cache as a content-deduplicated cache. We define *content deduplication factor* as the ratio of number of unique content blocks to the total number of blocks in cache. Thus, higher the content deduplication factor, better is the efficiency of the cache. Fig. 4.14 shows the variation in content deduplication factor, as each request of the *webvm* trace is replayed. It can be seen that the DRIVE system achieves significantly higher content deduplication factor than both the Vanilla and IODEDUP systems. Note that, 100% content deduplication of page cache can not be achieved because of the compulsory misses incurred during cache-warmup, as well as the cache dirtying due to write requests. However, DRIVE attains a high content deduplication factor of up to 97%, and this is the reason for the significant performance improvement achieved by DRIVE.

4.6.4 Identifying similarity in multiple virtual machines

To simulate the scenario of multi-VM workloads being executed on a virtualized host, we aggregated the two three-week long traces of *homes* and *webvm* workloads in timestamp order. We performed a read/write replay of the aggregate trace and compare cache-hit ratios, disk reads reduced and read response latencies for Vanilla, DRIVE and IODEDUP.

The performance metrics are tabulated in Table 4.2 and show that although the cache-hit ra-

Table 4.2: Performance for aggregated trace replay

Scheme	Cache-hit ratio (%)	Disk reads reduced(%)	Avg. read response latency (msec)
Vanilla	61.2	1.6	7.9
DRIVE	67.6	18.5	6.5
IODEDUP	62.4	4.3	7.7

tios are comparable in all three schemes (DRIVE is highest nevertheless), there is a huge margin in percentage of disk reads reduced. DRIVE system averts 18% disk reads as compared to 1.6% of Vanilla and 4.3% of IODEDUP. Decrease in number of disk reads results in a lower average read response latency, with 6.5ms for DRIVE as compared to 7.9ms and 7.7ms for Vanilla and IODEDUP, respectively. Fig. 4.15 plots the read response throughput per hour, assuming continuous replay of read requests in the aggregated trace. We can see that DRIVE produces more number of responses per hour in the 4th, 5th, 6th hours and so on, hence resulting in an earlier completion than Vanilla or IODEDUP replays. Notably, in hour 13, DRIVE finishes almost double the number of read requests as Vanilla and IODEDUP.

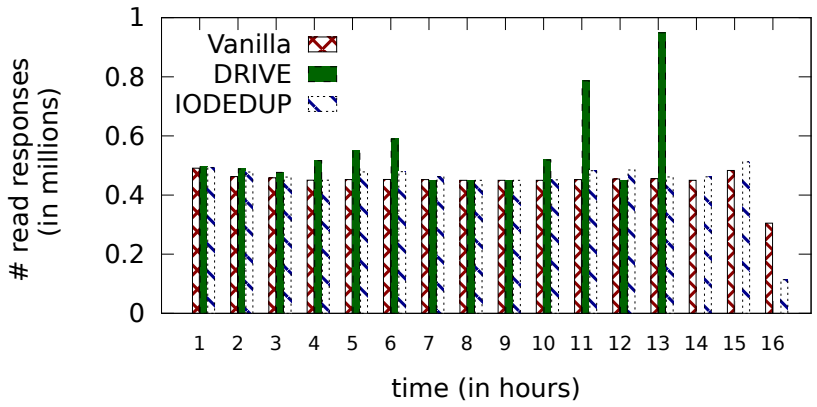


Fig. 4.15: Read response throughput for the aggregated trace.

4.6.5 Evaluating impact of write-intensivity factor

The presence of write requests demands continuous metadata updates and also causes cache churn, which results in lowered cache hit performance for read requests. Hence, it is important to study effectiveness of cache manipulation with write-intensive workloads as well.

In this experiment, we generated trace-snippets with various write-intensivity levels using the *homes* workload as base trace, and probabilistically sampling write requests. The write-intensivity of the original *homes* trace is 4.2, and we generated trace-snippets with varying write-intensivity levels like 0.25, 0.5, 1, 2 and 3, and performed trace-based replay. As seen in Fig. 4.16, as the write-intensivity level increases from 0.25 to 4.2, the difference in number of capacity misses between Vanilla and DRIVE increases. This implies that even as the write-intensivity factor increases, DRIVE is able to convert higher number of capacity misses into

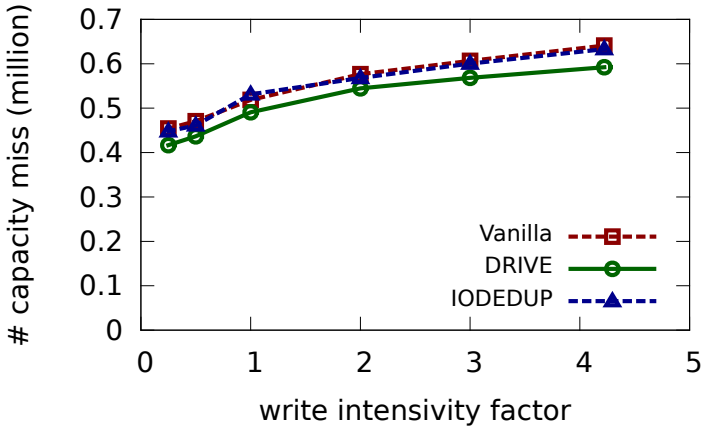


Fig. 4.16: Impact of write-intensivity factor for the *homes* workload.

cache-hits. Thus, DRIVE performs well even with write-intensive workloads.

4.7 Detailed quantification of overheads

In the previous section, we evaluated the incorporation of intelligent I/O redirection within the virtual block driver of a VM to manipulate the underlying block-based cache as a content-based cache. The quantitative results presented were based on the number of cache hits measured in our custom simulator, and the average disk access and memory access times were input variables. The CPU overheads due to metadata lookup and metadata update operations is analyzed qualitatively in Section 4.5.4, and indicated that they were constant or O(1). First, we present quantification of the CPU overheads and an in-depth comparative study of the CPU overheads incurred by DRIVE and IODEDUP systems. Later, we present a comparative study of the cost and benefits of maintaining metadata stores in DRIVE versus IODEDUP.

4.7.1 CPU overheads in DRIVE versus IODEDUP

Although the DRIVE system succeeds in improving both the cache-hit ratio as well as the number of disk reads averted, it also introduces CPU overhead during the metadata lookup required for every read request. The IODEDUP system also incurs similar overheads, but since the content-cache and associated metadata are positioned downstream of the block-cache in case of IODEDUP, therefore CPU overheads are incurred only for those read requests that are not already satisfied by the block-cache. This implies that whereas IODEDUP incurs CPU overhead only for a subset of the read requests, DRIVE incurs similar overheads for entire set of read requests. On the other hand, notice that IODEDUP overhead includes content-cache manipulation (insertion, lookup, eviction) as well, apart from metadata manipulation which is common to both DRIVE and IODEDUP.

Similar conditions exist in case of write requests as well. Although the aim of I/O Deduplication is to optimize only read request performance and not write, metadata update is still

Table 4.3: Latency parameters in simulation

Sr. No.	Component	Latency
1	*Block-cache lookup*	83ns
2	*Block-cache update*	83ns
3	*Metadata lookup*	100ns
4	*Content-cache lookup*	100ns
5	*Metadata update*	33μs
6	*Content-cache update*	100ns
7	*Metadata invalidate*	100ns
8	*Disk read*	13.7ms
9	*Disk write (write-back)*	83ns
10	*Disk write (write-through)*	15ms

performed for every write request. In case the cache is a write-through cache, both DRIVE and IODEDUP can perform metadata update inline in the write-path itself, since its impact on write performance would be indistinguishable. However, if the cache is a write-back cache, IODEDUP performs metadata update only for those blocks that are flushed from block-cache to disk, whereas DRIVE performs it for every write request. Similar to case of read requests, IODEDUP overhead in write-path also includes content-cache manipulation whereas DRIVE overhead includes only metadata update overhead.

The above varying CPU overheads cause varying read response latencies per read request, resulting in varying throughput. Thus, in this section, we take a deeper look at the interplay of CPU overheads and their impact on I/O Deduplication performance.

Description of latency parameters

The various simulation timing parameters used for the evaluation are presented in Table 4.3. The parameters include disk read time, disk write time, block-cache lookup time and block-cache update time which are the default parameters in the Vanilla system. The CPU overheads are the ones highlighted in gray like metadata lookup time, metadata invalidate time, metadata update time, content-cache lookup time and content-cache update time. The values for disk and cache lookup/update timings are retrieved from literature [90, 91]. However, the latency values for the CPU overhead parameter are informal estimates, since their true values can be measured only in an actual implemented system.

Effect on read access performance due to write request handling

In the DRIVE system discussed so far, a write request involves merely the *invalidation* of existing deduplication metadata. However, technically it is possible to *update* the metadata as well, for every write request. With this in mind, we present the following discussion and our rationale for choosing to only invalidate the metadata during write requests and not update it. We explore the impact of our choice on the achievable improvement in read access performance, as well.

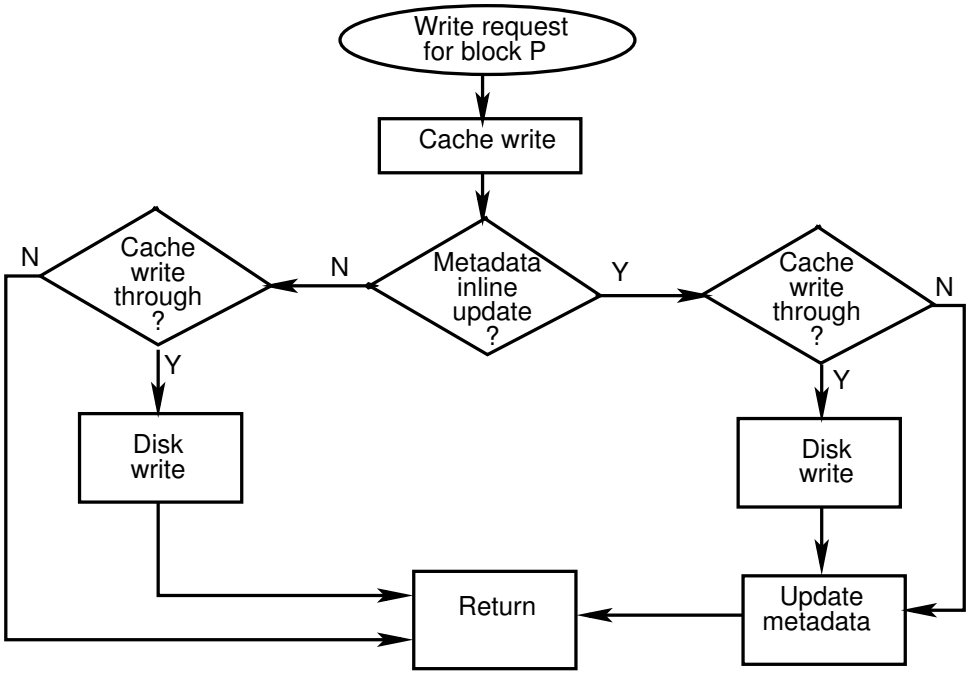


Fig. 4.17: Flow path(s) for write requests in DRIVE system: *Differences due to cache mode (write-back, write-through) and metadata update mode for writes (updated, not-updated) shown.*

Fig. 4.17 depicts the different write paths which govern user perceived write response latency. A write request can be executed in the following four ways, depending upon whether the block-cache is write-through or write-back, and whether metadata update is to be performed or not:- (i) Invalidate metadata, write to block-cache, return, (ii) Invalidate metadata, write to block-cache, write to disk, return, (iii) Invalidate metadata, write to block-cache, write to disk, update metadata, return, and (iv) Invalidate metadata, write to block-cache, update metadata, return.

If the block-cache is a *write-through* cache, the additional latency due to metadata update (order of nanoseconds) is insignificant relative to disk write latencies (order of milliseconds). Hence, in case of a write-through cache, the impact of I/O deduplication overheads on user perceived response for write requests would be indistinguishable. However, in case of a *write-back* cache, user perceived write-performance is dictated by the length of the write-path till the cache, hence write-path overhead needs to be kept as low as possible. Note that this concern does not apply to IODEDUP since its location is downstream from the block-cache and hence needs to handle write requests only when they are eventually flushed to disk from block-cache. However, DRIVE is located above the block-cache and has to perform metadata management *inline* for every write request.

Alternatively, since optimizing writes is not the aim of this work, we can still achieve correctness by only invalidating existing mappings for the blocks being written, *without updating* the metadata. Metadata invalidation (i.e. marking dirty) ensures that we do not use stale metadata for redirecting subsequent read requests. This strategy will result in lowered overhead for write requests, but at the cost of potential performance loss due to dirty metadata resulting in mandatory disk fetches. Next, we perform evaluation of DRIVE with and without metadata updates

upon writes, and present results to quantify its effects on performance, relative to IODEDUP and Vanilla systems.

Impact on per-request latencies due to metadata updates for writes

Since DRIVE system seeks to optimize only reads and not writes, the content of write requests is used only to invalidate existing metadata so that stale metadata is not used for (incorrect) redirection of subsequent read requests. An optional task is to update the metadata according to the new content encountered in write requests, after having marked the metadata dirty. If the cache is in *write-back* mode, the write path extends only till the block-cache, whereas if the cache is in *write-through* mode, the write path extends all the way till disk. Hence, if metadata update is performed *inline*, it will add to the write response latency. This metadata-update-upon-write operation (called MeU) is optional because it does not affect the correctness of redirection. However, since it leaves metadata dirty, subsequent read operation may lose optimization opportunity, hence resulting in fewer cache-hits. Fewer cache-hits (more disk fetches) will result in lowered throughput and on the other hand, non updation of metadata upon write requests will incur per-write lower latencies. We perform study of CPU overheads under following categories:-

1. Per-request read response latencies when metadata is updated upon writes (MeU)

2. Per-request read response latencies when metadata is not updated upon writes (MeNU)

3. Comparison of write response latency when metadata updated (MeU) and not (MeNU)

Per-request read response latencies in case MeU: In Table 4.4, we list the various components involved in the read path case-by-case, assuming that metadata update upon writes (MeU) is being performed. In case of Vanilla, there are only two cases for every read request: block found in cache, i.e., `cacheR` or block read from disk, i.e., `diskR`. In case of DRIVE, a `diskR` can be caused due to absent metadata, i.e., `metadatamiss` or block previously evicted from cache, i.e., `capacitymiss`, whereas a `cacheR` can happen after a `metadatahit`. In case of IODEDUP, a `cacheR` can be due to hit in block-cache, i.e., `blockhit` or due to hit in content-cache, i.e., `contenthit`, and a `diskR` may be inevitable inspite of a `metadatahit`, if a `contentmiss` occurs. The highlighted components are the overhead components, i.e. they are extra components present in either DRIVE or IODEDUP and are not present in Vanilla system.

Per-request read response latencies in case MeNU: If metadata is not updated upon write requests (MeNU) in DRIVE, upcoming read requests to the same block will encounter dirty metadata, i.e. `metadatamiss`, and may subsequently result in a `blockhit` or `blockmiss`. Table 4.5 lists the components involved in the read path in case of MeNU. Here, a `cacheR` can occur in two ways: (i) `metadatahit` followed by `blockhit` and (ii) `metadatamiss` followed by `blockhit`, and parallelly for `diskR` as well.

Table 4.4: Steps involved in Read-path for Vanilla, DRIVE and IODEDUP with metadata updated upon writes

	Vanilla		**DRIVE (with MeU)**			**IODEDUP (with MeU)**			
			cacheR	diskR		cacheR		diskR	
Step	cacheR	diskR	block hit	metadata miss	block miss	block hit	content hit	metadata miss	content miss
Metadata lookup			✓	✓	✓		✓	✓	✓
Block-cache lookup	✓	✓	✓	✓	✓	✓	✓	✓	✓
Content-cache lookup							✓		✓
Disk read		✓		✓	✓			✓	✓
Block-cache update		✓		✓	✓		✓	✓	✓
Metadata update				✓				✓	
Content-cache update								✓	✓
Total read latency	83ns	∼13.7ms	183ns	∼13.7ms	∼13.7ms	83ns	383ns	∼13.7ms	∼13.7ms

Table 4.5: Components in Read-path latency for Vanilla, DRIVE and IODEDUP with metadata NOT updated upon writes

	Vanilla		**DRIVE (with MeNU)**				**IODEDUP (with MeNU)**			
			cacheR		diskR		cacheR		diskR	
Component	cacheR	diskR	metadata hit block hit	metadata miss block hit	metadata miss block miss	metadata hit block miss	block hit	content hit	metadata miss	content miss
Metadata lookup			✓	✓	✓	✓		✓	✓	✓
Block-cache lookup	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Content-cache lookup								✓		✓
Disk read		✓			✓	✓			✓	✓
Block-cache update		✓			✓	✓		✓	✓	✓
Metadata update				✓	✓				✓	
Content-cache update									✓	✓
Total latency per request	83ns	∼13.7ms	183ns	33μs	∼13.7ms	∼13.7ms	83ns	383ns	∼13.7ms	∼13.7ms

Table 4.6: Write-path latency for Vanilla, DRIVE and IODEDUP with MeU and MeNU

Write-through or Write-back	System	Metadata Update?	Latency per request	Throughput (million per hour)
Write-through	Vanilla	-	∼15ms	0.216
Write-through	DRIVE	MeU	∼15ms	0.216
Write-through	DRIVE	MeNU	∼15ms	0.216
Write-through	IODEDUP	-	∼15ms	0.216
Write-back	Vanilla	-	100ns	36000
Write-back	DRIVE	MeU	∼33.33μs	108
Write-back	DRIVE	MeNU	200ns	18000
Write-back	IODEDUP	-	100ns	36000

Comparison of write latency when MeU versus MeNU: Metadata update during write requests will add to write response latency, whereas only performing metadata invalidation not only ensures that dirty metadata is not used for redirection, but also keeps the write-response timings low, especially in case of a write-back cache.

Possible write paths that contribute to write response latency for Vanilla, DRIVE and IODEDUP are listed in Table 4.6. As shown, when the cache is in write-through mode, metadata update latency makes little difference to the write response latency since disk write latency is the dominant component in this case. However, when the block-cache is in write-back mode, writes are only written to cache, so metadata update contributes comparable overhead to the write path latency. So, we explore two options of (i) updating metadata, and (ii) not updating metadata, and the corresponding hourly throughput is reported in Table 4.6. Note that IODEDUP has no overhead in the write path when the block-cache is in write-back mode. This is because, as described earlier, IODEDUP is positioned downstream from the block-cache and encounters write requests only when they are flushed from (not when they are written to) cache in case of a write-back cache.

Depending upon whether the block-cache is in write-through or write-back mode, and whether the metadata is updated (MeU) or not-updated (MeNU) upon write request, the read and write response latencies can vary. Moreover, the number of cache-hits in each case is also influenced by the same reasons. Next, we present comparative evaluation of the three systems using various metrics like cache-hit ratios, number of disk reads averted, average read response latency and throughput.

4.7.2 Cost-benefit analysis of metadata space usage in DRIVE versus IODEDUP

Both the DRIVE and IODEDUP systems maintain metadata stores, which basically identify similarity across blocks using content fingerprints. The difference between the two systems, though, is the placement of the metadata store—in IODEDUP, the metadata store is present downstream of the host-cache, whereas DRIVE system has it upstream of the cache. Metadata

space is basically occupying precious space [85] otherwise available for caching, hence, we need to analyze the cost and benefit of using a metadata store.

In the previous section, we demonstrated that DRIVE system is able to achieve better performance than Vanilla system, with the use of the metadata store and simple hint-based I/O redirection. Thus, the benefit of the metadata store is justified in that case. The IODEDUP system also uses a metadata store, and performs I/O deduplication, so here we perform a comparison to see which one of the two systems (DRIVE or IODEDUP) is able to make better use of the precious cache space.

Both the IODEDUP and DRIVE systems build metadata at runtime, i.e., intercept requests and compare content fingerprints to determine whether content is new or duplicate—in both cases, metadata is updated accordingly. If these systems are deployed on production systems that are used long-term, the metadata store will keep growing, unless it is managed as a fixed-size metadata cache. Note that, pruning of metadata in this manner may result in some lost deduplication opportunities, but the rest of the metadata can still yield better performance than the Vanilla system.

For the purpose of this thesis, the aspect of metadata space management is not within thesis scope (it is future work), and metadata space management has not been discussed in IODEDUP paper [68] either. So, for comparision between IODEDUP and DRIVE system using the available traces [81], we assume that metadata storage space is sufficient to store all metadata for these traces. Observe that, this is the best case scenario for both DRIVE and IODEDUP systems, since there will no lost deduplication opportunities here. Moreover, this space usage will be approximately equal (say, x MB) in both DRIVE and IODEDUP systems, so here we compare the cost and benefit of using x MB space as a metadata store in both systems.

Cost of metadata store: As mentioned above, the positioning of the metadata store is different (downstream versus upstream) in IODEDUP and DRIVE systems. Despite this, both systems have to track every "new" content and update metadata accordingly—and hence, space occupied by metadata in both systems will be equal for any given workload/trace. Thus, the "cost" of metadata store is equivalent in both systems. However, the benefit of metadata store is different in both the systems, and we demonstrate it next.

Benefits of metadata store: Metadata can be said to be beneficial when it results in a cache hit (i.e., disk fetch is averted) which would not have been otherwise possible. For example, if blocks A and B are duplicates of each other, and both are in cache, then a cache hit for either of them can not be considered as a "benefit" of metadata. On the other hand, if only block A is present in cache, and a request for block B could be satisfied due to deduplication, that is a benefit of having relevant metadata. We define the former case as a `self-hit` and the latter as a `dedup-hit`, and claim that only the number of `dedup-hits` determines the benefit of gathering similarity-based metatadata.

Table 4.7: Number of metadata hits encountered for read requests

Trace name	Scheme name	Num. of read requests	Num. of metadata hits
homes	IODEDUP	4,052,176	647,012
homes	DRIVE	4,052,176	695,747
mail	IODEDUP	2,375,409	382,119
mail	DRIVE	2,375,409	625,938
webvm	IODEDUP	3,116,456	2,223,457
webvm	DRIVE	3,116,456	2,800,411

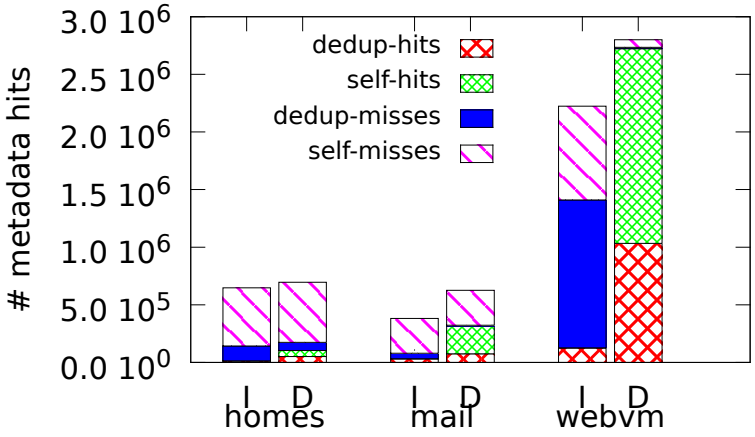


Fig. 4.18: Classification of metadata hits in IODEDUP(I) Vs DRIVE(D) for the *homes*, *mail* and *webvm* traces.

Number of read requests encountering metadata hits: In DRIVE, every block-cache lookup (and disk fetch, if required) happens after `metadatahit` or `metadatamiss`. However, in case of IODEDUP, only requests that encounter `blockmiss` get looked up into metadata (refer the discussion in previous section for details). This is because DRIVE metadata store is upstream of the block-cache, whereas IODEDUP metadata store is downstream. This implies that, the total number of metadata hits in DRIVE will be potentially higher than in IODEDUP, and this is reflected in Table 4.7. It can be seen that for each trace—*webvm*, *homes*, *mail*—the total number of metadata hits in DRIVE system is higher than in IODEDUP.

Classification of metadata hits: As mentioned earlier, a `metadatahit` does not guarantee that a cache-hit will occur—that is, `metadatahit` can result in either a cache hit or a miss. And each of these hits/misses can further be classified into (i) `self-hit`, (ii) `self-miss`, (iii) `dedup-hit`, and (iv) `dedup-miss`. Basically the prefix "self" indicates that the request was for the same block itself, whereas "dedup" indicates that the request was for another block and the metadata store could successfully identify it as a duplicate. In Fig. 4.18, we present a classification of the total number of metadata hits into one of these four classes, for each trace under both IODEDUP and DRIVE systems.

From Fig. 4.18, we can see that the total number of metadata hits is greater in DRIVE system

Table 4.8: Capturing *metadata benefit ratio*

Trace name	Scheme name	Num. of metadata hits	Num. of dedup-hits	Benefit ratio(%)
homes	IODEDUP	647,012	14,041	2.17
homes	DRIVE	695,747	51,090	7.34
mail	IODEDUP	382,119	31,934	8.36
mail	DRIVE	625,938	74,673	11.93
webvm	IODEDUP	2,223,457	127,156	5.72
webvm	DRIVE	2,800,411	1,033,328	36.90

than IODEDUP system, for all traces. Moreover, in case of *webvm* trace, most of metadata hits in IODEDUP system result in `dedup-misses` or `self-misses`, while DRIVE is able to convert most of its metadata hits into `dedup-hits` and `self-hits`. This leads to the huge performance improvement by DRIVE system for *webvm* trace, as compared to IODEDUP system.

Differentiating self-hits and dedup-hits: The above Fig. 4.18 shows that for *webvm* trace, DRIVE has higher number of `dedup-hits` *plus* `self-hits` than IODEDUP. However, as explained earlier, only the number of `dedup-hits` should be considered as a "benefit" of using the metadata store. Hence, if we want to capture metadata benefits using a single number, it would be the ratio of number of `dedup-hits` to number of `metadatahits`. Table 4.8 presents this ratio for each of the traces, and we can see that DRIVE system has a higher metadata benefit ratio than IODEDUP, for all three workloads considered. Especially, for *webvm* trace which was found to be an ideal candidate to benefit from I/O deduplication, the DRIVE system has more than 6× higher metadata benefit ratio than IODEDUP system.

Implications of metadata benefit ratio on metadata store size: In Table 4.8, we see that when the metadata store size is equal, the benefits achieved by DRIVE system is higher than by IODEDUP system. This effectively implies the following:-

1. For IODEDUP to achieve similar benefits as DRIVE, it needs a much larger metadata store and/or content-cache, whereas

2. For DRIVE to attain the same performance as IODEDUP, it may only need a relatively smaller metadata store.

Based on the analysis presented in this section, we have proven that the DRIVE system achieves much better I/O deduplication performance than IODEDUP system, for the same size of metadata store. Specifically, the DRIVE system achieves lower read response latencies, and higher metadata benefit ratio than the IODEDUP system.

4.8 Discussion and future work

4.8.1 Fixed vs variable-sized similarity identification

The work in this chapter is based on content-similarity identification using fixed-size blocks. However, workloads like *homes* and *mail* potentially have content-similarity across variable-sized chunks. For example, multiple collaborators may create multiple revisions of the same document and these revisions may have similar content but not exactly aligned at 4K boundaries. Similarly, mail servers may serve mail for potentially thousands of users with multiple mailing lists and groups. Moreover, even at the datacenter level, most virtual machine image files are instantiated from a small number of golden master images[66], and they "age" over a period of months and years with administrators modifying files and causing byte-shifts at various locations within the image[65]. These observations motivate the need for variable-sized chunk based similarity identification, as future work.

4.8.2 Applicability to other storage systems

The IODEDUP system uses Adaptive Cache Replacement (ARC) [92, 74] policy to boost up its content-cache hit ratio by 3-4X [68]. If the cache replacement policy could be changed to ARC in our system, DRIVE performance would also benefit from a similar performance boost. Although this may not be possible in a standard Linux host, ARC caches are already present in IBM's DS6000/DS8000 storage controllers [93] and adoption of DRIVE may help increase their cache performance.

4.8.3 Interaction with storage deduplication systems

The primary difference between I/O Deduplication and Storage Deduplication[85, 94, 95], is that the former avoids duplicate read requests (hence improving storage read access performance) while the latter avoids duplicate writes to the storage (hence saving disk space). For example, ZFS[95] is a file system kernel module that performs storage deduplication. Additionally, ZFS also maintains an ARC cache, hence resulting in improved read performance (similar to IODEDUP). However, distinct blocks with duplicate content will still populate the host cache. By employing the DRIVE system in conjunction with storage deduplication systems, better read performance can be achieved along-with storage space savings.

4.8.4 Metadata space management

As discussed above, metadata occupies decent amount of space per block entry. Hence, for deployment on a production system or on large System Volume Controllers (SVC), we need a mechanism to ensure that memory space usage to store metadata is as low as possible, otherwise losing precious buffer cache space. One option is to distribute metadata between memory and

disk, such that the more "important" metadata are present in memory for quick access. The work in [70] achieves around 99% metadata cache-hits using this approach. However, a key assumption there is of sequential data access, and hence sequential metadata access pattern. This is true for backup workloads as targeted in [70], however in case of random-access workloads, metadata access prediction is non-trivial.

Random-access workloads exist in inline storage deduplication scenarios [85], where a simple LRU cache is used to accommodate metadata, such that only the most-recently-used metadata stays in cache, while the rest is discarded. This, of course, comes at the cost of lost deduplication opportunities. However, a significant difference from our work of I/O deduplication is that storage deduplication needs to track only write requests whereas tracking read requests is our area of interest. This implies that whereas [85] found no significant performance improvement by using different caching policies, our work could expect to benefit from a frequency-aware caching mechanism like Adaptive Replacement Cache (ARC) [92, 74]. Thus, we propose to use a framework for metadata store which ensures that more-frequently-used and/or more-recently-used metadata stay in cache, while the rest are moved to disk.

4.9 Conclusions

I/O reduction refers to reducing the number of disk read accesses by employing better cache management strategies. Typically, caches are referenced by block number and mitigate the necessity to fetch the block from disk. However, traditional caches can not recognize content similarity across multiple blocks and hence, the system ends up fetching and storing multiple copies of the same content in cache. Elimination of duplicate read I/O requests which fetch the same content repeatedly is referred as I/O Deduplication. Existing work uses a split-cache approach, with a part of the block-based cache reserved as a content-based cache. In this work, we demonstrated that the split-cache approach is sub-optimal, and presented an I/O reduction system called DRIVE which performs I/O redirection to *implicitly* manipulate the whole underlying cache as a content-deduplicated cache. Only the VM's own disk access history is introspected to obtain implicit hints regarding host cache state, to be used for read I/O redirection.

We performed comparative evaluation by implementing prototypes and performed trace-based evaluation in a custom simulator. The evaluations showed that, the DRIVE system manipulates the entire available host cache space effectively like a content-cache, achieving a high content deduplication factor of up to 97%. This is the key reason for better performance, with up to 20% higher cache-hit ratios, and up to 80% higher number of disk reads reduced than the Vanilla system.

Chapter 5

The Case for I/O Deduplication Benchmarks

In the previous chapter, we described an I/O deduplication and reduction mechanism called DRIVE and presented its trace-based evaluation within a custom simulator, called `SimReplay`. The traces we used were the three (*homes*, *mail*, *webvm*) 21 day-long production traces available at [81], from which we concluded that the DRIVE system could perform exceptionally well for the *webvm* trace. However, to further stress the system's functioning as well as to understand its limits, we need to test it with more real-world traces. In this chapter, we build the case for generating synthetic I/O deduplication benchmarks for more comprehensive evaluation of I/O deduplication techniques.

5.1 Motivation

In general, we can intuitively understand that higher the amount of duplicate content in the traces, higher the effectiveness of any I/O deduplication technique as compared to the Vanilla case. However, as our previous evaluation has shown, even though the *webvm* trace had high levels of duplicate content and was an ideal candidate to benefit from I/O deduplication techniques, the deduplication benefits achievable for it was extremely poor within the IODEDUP system while being exceptionally good within the DRIVE system. This seems to indicate that it is not just the degree of duplicate content but possibly other characteristics of the *webvm* trace also that contribute to such varied performance.

In our pursuit to test the DRIVE system further, we were faced with the following options:-

- **Trace collection toolkit**: Build our own tracing toolkit (`preadwritedump`) to capture more traces.
- **Capture production traces**: Deploy above toolkit on production servers to capture real-world traces

- **Capture synthetic benchmarks**: Deploy above toolkit to capture synthetic benchmark traces
- **Dataset survey**: Perform an exhaustive literature survey to determine if there are any other real-world datasets available that can be used for I/O deduplication performance benchmarking
- **DRIVE evaluation using datasets**: If public datasets or traces available, use them for further testing of DRIVE system
- **Dataset characterization**: Extensively characterize the available traces to learn which particular properties of the *webvm* trace contributed to its enhanced performance in the DRIVE system
- **I/O deduplication benchmarks**: If public datasets are not available, build a case for creating synthetic I/O deduplication benchmarks

In the rest of this chapter, we describe the work done in pursuing each of these options, and which finally builds a case for creating synthetic and realistic I/O deduplication benchmarks.

5.1.1 Building custom trace collection toolkit

Of the above alternatives, we started off with building our own tracing toolkit with a view to deploying it on production servers within our department. We built the toolkit called `preadwritedump`, however, we were unable to obtain requisite permission from the systems administrators to deploy the same on the departmental servers. Although we could not fully utilize the developed tool, we believe that it would be helpful for future tracing efforts. Moreover, if we wish to develop sub-block level deduplication techniques, we need to have a tracing tool that can perform block-level tracing and dump of I/O content, because merely dumping hashes of the blocks is not suitable for sub-block duplicate identification. With this in mind, we have presented the design and implementation of our toolkit in the Appendix III of this thesis.

5.1.2 Tracing of synthetic benchmarks

The next alternative was to generate synthetic workloads and capture corresponding traces. However, the challenge here is to develop synthetic traces that are "realistic". For example, many public storage and I/O benchmarks exist, however, their focus is only on the number of I/Os that can be generated per second, and not on the actual content being read or written. Hence, these benchmarks tend to generate "realistic" workload levels (i.e., number of IO operations per second) while paying scant attention to whether the content being generated as part of the read and write operations are also realistic or not. For example, these benchmarks may generate randomized content [96] or heavily duplicate content [58] or even write zeros [97], in some

cases. For our purposes, usage of benchmarks that write randomized content will result in very low duplicates in the workload, whereas those that heavily generate duplicate content or simply write zeros will overestimate the benefit of deduplication. Hence we conclude that, existing public benchmarks, though purported to be "realistic" are not realistic in terms of the content generated for the I/O and hence are not useful for evaluation of deduplication techniques.

With respect to the use of existing benchmarks for deduplication study, it deserves mention that benchmarks like HiBench [98] and RUBiS [58] have been used in earlier works [99] for storage duplicate characterization, wherein high levels of duplicate content were reported (eg. 70-80% duplicate content observed in RUBiS benchmark). However, we take a critical view towards such work because the intended usage of these benchmark applications (eg. RUBiS [58]) is the study of application performance, with zero focus on the application's content characterization. Basically, these benchmarks generate either heavily-duplicate or heavily-random content because the content is irrelevant for the benchmark's intended usage. However this lack of "realistic content characterization" disqualifies their use for evaluating content deduplication techniques. Specifically, RUBiS is an application benchmark mimicking an e-commerce website where most of the content pages are dummy pages, and hence tend to have repetition of content (eg. repetitive "item descriptions" or "comments") just to populate the web pages. This content should not be considered as legitimate duplicate content, since real e-commerce applications would typically not consist of dummy repetitive comments. Alternatively, the duplicate data created in HiBench benchmark is for the express purpose of storage availability in bigdata or MapReduce environments, hence deduplicating such blocks on storage is counter-productive.

5.1.3 Survey to identify relevant public datasets

Due to above unsuccessful attempts at generating synthetic workload traces for evaluation of DRIVE, we once again turned towards datasets in literature and performed an extensive survey to determine whether any relevant useful datasets are available online. To begin with, we classified the dataset surveyed into two sets: (i) *Public datasets*—these traces are publicly available online and can be used by any researcher with an Internet connection, (ii) *Proprietary datasets*—these are datasets mentioned in various papers, however they have not been made publicly available. For example, the research groups from NetApp, IBM, etc use trace datasets that are internally available to them, but are not published online. We describe the surveyed datasets in Section 5.2.

After performing the dataset survey with a very wide net, we conclude that there are no datasets available for testing I/O deduplication techniques apart from the ones that were provided at [81] which we have already used. Further, we conclude that the next best way of preparing datasets for evaluation would be to synthetically generate realistic workload traces by capturing the characteristics of the available trace itself.

(a) Conference-name tag cloud

(b) Year-of-publication tag cloud

Fig. 5.1: Representation of the conferences and the years of publication covered in our survey

5.1.4 Synthetic generation of realistic traces

We performed another detailed literature survey of relevant papers that develop a workload model based on measurements from real-world, and then generate synthetic realistic benchmarks using the developed model. The survey revealed that there is related work that generates realistic benchmarks for I/O access (i.e. only concerned with block numbers accessed, not their content) as well as for storage deduplication (i.e. concerned with content stored on disk, not content accessed). However, there is no related work so far that generates realistic I/O access traces that account for the content as well.

In the rest of this chapter, we build the case for **I/O deduplication benchmarks** using the conducted survey results, and establish the generation of realistic benchmarks as useful future work. First, we present the results of our dataset survey which shows that there are no suitable datasets available publicly. Next, we present a detailed trace characterization of the available *webvm* traces [81] and finally, we make the case that the detailed trace characterization can be used for generating realistic benchmarks for evaluation of I/O deduplication techniques.

5.2 Survey of datasets and benchmarks in literature

We surveyed the datasets used in over 100 publications amounting to a total of over 350 datasets. A similar survey for storage deduplication datasets (covering 120 datasets from 33 research papers) is presented in [100], wherein the requirements specified for such datasets is that they should be realistic, sufficiently large, as well as easily accessible/distributed to other researchers.

The paper [100] does not mention by name which publications were surveyed by the authors, except for stating that the 33 research papers were from top conferences from the years 2010 and 2011. Due to such limited information available, we undertook the survey all over again, while at the same time not restricting ourselves to only a couple of years—instead, we cast a wider net for our survey by including all papers that were related to any of the following topics:-

(i) Storage deduplication, (ii) Memory deduplication, (iii) Storage characterization, and (iv) I/O characterization.

Since the number of papers considered in our survey is huge (more than 100), instead of listing out the conference names and the year of publication, we present the following tag clouds (refer Fig. 5.1(a) and (b)) which are meant to represent the survey coverage. As can be seen from the tagcloud in Fig. 5.1(a), our survey covers the important conferences in the areas of storage and workload characterization, namely **FAST**, **ATC**, **SIGMETRICS** and **IISWC**, to name a few. Moreover, the years of publication (refer Fig. 5.1(b)) for the surveyed papers are mostly within 2009 to 2014, with a few lying even beyond as well. With this wide coverage of conferences and years of publication, we are positive that we have covered the necessary ground for this survey.

As mentioned earlier, we classify the dataset surveyed into two sets (i) *Public datasets* and (ii) *Proprietary datasets*, and for each of these sets, we need to distinguish the dataset further based on whether they are one of the following:

1. I/O traces with content representation,

2. I/O traces without content representation,

3. Storage metadata with content representation, or

4. Storage metadata without content representation

The first item in the above list is the kind of trace we are looking for, i.e. I/O activity traces with content representation. However, we found that almost all of the public datasets available online fall in one of the other three categories listed above.

5.2.1 Public dataset repositories

By the term *public datasets*, we allude to those datasets or traces that have been made available online by the respective owners. A popular repository for storage and I/O traces is the SNIA-IOTTA Repository [101] which is a collection of traces from multiple publications maintained under the Storage Network Industry Association's Input/Output Traces, Tools and Analysis Technical Work Group. Another popular repository is the one maintained by HP Labs [102] which contains many tools and benchmarks as well, in addition to traces. Note that, there are several other small repositories, each hosting the traces or datasets for a single or few publications, we list them separately in the next section under *Public individual datasets*. Below we provide a listing of those online trace repositories, which contain huge number of traces (i.e. multiple trace sets, not just one) and are available to the researchers for development and analysis of network and storage systems.

1. **SNIA IOTTA Repository** [101]: collaborative effort under SNIA IOTTA TWG to provide storage-related I/O traces, tools and analysis to the entire research community free of cost.

2. **HP Labs Repository** [102]: provides several file system and application traces captured using HP Labs' tools like DataSeries and Lintel. However, some of these traces are very old, and may not be representative of current systems.

3. **UMass Trace Repository** [103]: provides network, storage, and other traces, some of which are generated at UMass while some others are donated.

4. **Internet Traffic Archive** [104]: provides access to traces of Internet network traffic.

5. **VMware image repository** [105]: provides many VMware images with various Linux distributions installed like CentOS, Debian, Fedora, FreeBSD and OpenSUSE.

Next, we discuss the suitability of the traces, hosted at each of the above public dataset repositories, for the purpose of I/O deduplication evaluation.

1. SNIA IOTTA Repository traces

This repository contains several trace sets including Block I/O traces, NFS traces and System call traces. The publications that make use of these traces are [106, 68, 107, 108, 109, 110], among which [68] is the only paper which deals with I/O deduplication-related traces—none of the other trace sets at this repository have data content representation, and hence they are not suitable for evaluating I/O deduplication techniques.

2. HP Labs Repository traces

This repository contains several file system and application traces, however some of these traces are very old, and also none of these traces have any content representation either.

3. UMass Trace Repository traces

This repository provides traces for network, storage, memory, etc and is a collection of traces used in various publications like [106, 111, 73]. Apart from the traces provided of memory contents, none of the other traces in this repository has any content representation. Moreover, the memory traces are also published as snapshots of memory contents and not in the form of an I/O trace, indicative of which block is being inserted or evicted from cache at which time. Hence, even these traces are not suitable for evaluation of I/O deduplication techniques.

4. Internet Traffic Archive traces

This repository provides access to traces of Internet network traffic, for potential characterization, benchmark generation, etc and even though a few of these traces (eg. LBL-TCP-3, LBL-PKT) are packet traces, instead of just web requests or HTTP traces, however they still do not represent the packet content or payload. Hence, none of these traces are suitable for I/O deduplication analysis, not even as synthetic workloads.

5. VMware image repository

This is a repository of 78 VMware virtual machine images with various Linux distributions installed in the default configuration. This repository has been used in a few publications [112, 113] to motivate storage deduplication and compression. However, since these are static images and not I/O workloads in themselves, they are not suitable for evaluation of I/O deduplication techniques.

5.2.2 Public individual datasets

1. **I/O deduplication traces** [81]: Traces provided by the I/O deduplication paper [68] which we also used in our evaluation

2. **Plan 9 traces** [114]: File system snapshots of the Plan 9 file system deployed in the Computing Sciences Research center of Bell Labs. Has content representation, but is suitable only for storage deduplication studies, not for I/O deduplication

3. **Animation-bear dataset** [115]: Traces released by HP Labs, collected at a feature animation company—these are NFS traces and have no content representation

4. **Linux kernel & GCC sources** [116, 117]: A few papers [112, 113] use kernel and GCC sources to perform study of storage deduplication systems, by exploiting the fact that multiple successive versions of the same software tend to have a lot of overlapping content.

These are individual datasets that were used for the characterization and evaluation in a couple of papers and have been made available online for use by other researchers. Among these, the first dataset (i.e. the I/O deduplication traces at [81]) is the one we have used for DRIVE evaluation in the previous chapter. This traces are also present at the SNIA IOTTA repository mentioned above, and is the only I/O access trace which has content representation included. The remaining traces are all either storage metadata traces or I/O traces without content representation, hence unfit for evaluation of I/O deduplication techniques.

5.2.3 Proprietary datasets

So far we have seen the publicly-available storage and I/O datasets. However, this is a small fraction of all the datasets that have been used and cited in all the papers that we surveyed.

Table 5.1: Summary of *public* datasets uncovered in the survey

No.	Dataset source	Cited by	Related to I/O, storage or memory	Comments regarding suitability for I/O dedup evaluation
1	IODEDUP traces	[68]	I/O	Suitable for I/O dedup evaluation
2	SNIA IOTTA Repo	[106, 108, 109, 110]	I/O	No content representation
3	HP Labs Repo	[118, 102]	Storage, I/O	No content representation
4	UMass Trace Repo	[106, 111, 73]	Storage, I/O	No content representation
5	Internet Traffic Archive	[119, 120, 104]	Internet	No content representation
6	VMware Image Repo	[112, 113]	Storage	Storage metadata, not I/O
7	Plan 9 traces	[87]	Storage	Storage metadata, not I/O
8	Animation-bear dataset	[115]	I/O	No content representation
9	Kernel & GCC source	[112, 113]	Storage	Storage metadata, not I/O

In other words, most of the datasets that are used for evaluation in literature are rarely hosted publicly [100] or made available to other researchers for further perusal and analysis. In what follows, we present a brief categorization of such datasets and cite the works which have used each category of dataset.

1. *Homes:* This basically consists of the home directories of several employees or students or researchers hosted on a common or shared storage. The access workload is such that the files are read and written by a single user each, although the user may end up making several copies of the same file for purposes of editing, translations, conversions, versioning, etc. Such datasets are used in [121, 122, 123, 124, 125].

2. *WebServer:* Consists of traces (storage or I/O) from webserver workloads, and includes Web "search" workloads also in some cases. Such workloads are traced and characterized in several research efforts like [126, 127, 128, 22, 129, 130, 131, 132].

3. *CollaborationShares:* This consists of shared storage among a group of researchers or collaborators such that the files are created by one user and accessed by many others for read and/or updates. This dataset may also have duplicates due to multiple copies for editing and versioning [128, 85, 124, 125]

4. *SoftwareDeployment:* This consists of the data from a server which contains VM images, softwares and binaries to be used for deployment by users, and the workload is such that the files are created by one or more system administrators and used or accessed by a bigger user population. This dataset may have duplicates due to similarities across the softwares and/or VM images [65]. Such datasets are used for the evaluation of storage deduplication in [85, 125].

5. *VM-Dataset:* A set of a few hundred VM images with installations of different versions of operating systems, applications and libraries is considered under this dataset. The simi-

Table 5.2: Summary of *proprietary* datasets uncovered in the survey

No.	Dataset category	Dataset description	Cited by
1	*Homes*	Home directories on shared storage	[121, 122, 123, 124, 125]
2	*WebServer*	Traces from webserver workloads	[127, 128, 130, 131, 132]
3	*CollaborationShares*	Files created by one, read by many	[85, 124, 125, 128]
4	*SoftwareDeployment*	Files created by one, used by many	[85, 121, 125]
5	*VM-Dataset*	Few hundred VM images	[65, 66, 121, 133, 134]
6	*VM-Backup*	Full or partial backup of VMs	[113, 135, 136, 137]
7	*DatabaseBackup*	Backup of production databases	[113, 122, 138, 139]

larities across VM images are due to multiple VM images being instantiated from a single golden master or template image [65, 133, 121, 66, 134].

6. *VM-Backup:* This consists of a setup where one or more VMs are fully or partially backed up regularly, such that the backups would have lot of redundancies amongst them [113, 135, 136, 137].

7. *DatabaseBackup:* Similar to the VM-Backup dataset mentioned above, several works also use database backup workloads for evaluation of storage deduplication techniques [113, 122, 138, 139].

We summarize the results of our above dataset survey in Table 5.1 for *publicly-available* datasets and in Table 5.2 for *proprietary* datasets. For each dataset, we identify it by a name, and list a few representative reference papers in which the dataset has been utilized. For each dataset, we also mention whether it is related to I/O, storage or memory traces and present comments regarding the dataset's suitability for use in evaluation of I/O deduplication techniques.

5.2.4 Benchmarks

Some benchmarking tools and application benchmarks have also been used for workload generation in literature. Some of the most popular benchmarks for Storage I/O workload generation are as under:-

1. **Filebench** [140]: It is a file system and storage benchmark that can generate both micro and macro workloads. It allows detailed workload specification using Workload Model Language (WML) as well. It includes several popular macro-workloads like web server, mail server and database server.

2. **dbench** [141]: This tool can be used to stress test a storage system to figure out its saturation performance level. It allows complex load specification, however the point is not to just generate specific load levels, rather the point is to push the storage system to its limits for the purpose of benchmarking its performance.

3. **HiBench** [98]: This is a benchmark suite for Hadoop mapreduce application benchmarking, comprising of both micro-benchmarks as well as application workloads like web search, machine learning and analytic querying.

4. **IOzone** [142]: This is a file system benchmark tool that generates and measures the performance of a wide variety of file system operations like `read`, `write`, `re-read`, `re-write`, `random read`, `mmap` and so on.

5. **IOMeter** [143]: This is a load generation and characterization tool for both storage and network, and can generate loads on single or multiple systems.

6. **PostMark** [96]: This is a benchmark to emulate and measure the functionality of email servers, web servers and news servers.

7. **FIO** [144]: Flexible IO tester (FIO) is a tool that allows the user to write a job file according to the load that needs to be simulated, and the tool spawns multiple processes to generate the requested load.

8. **CloudSuite** [145]: CloudSuite is a benchmark suite currently consisting of 8 application benchmarks that are based on real-world setups in today's datacenters. The benchmarks supported include data serving, data caching, web serving benchmarks and so on.

9. **TPC Benchmarks** [146]: The Transaction Processing Performance Council (TPC) defines benchmarks related to transaction processing and databases, which can be used to benchmark the performance of various applications, like OLTP, decision support and many more.

10. **SPEC Benchmarks** [147]: The Standard Performance Evaluation Corporation (SPEC) establishes, maintains and endorses a standardized set of benchmarks for high-performance computers, like compute-intensive benchmarks, mail server benchmarks (now retired), virtualization performance benchmarks, etc.

11. **NAS Parallel Benchmarks** [148]: This is a suite of programs designed to evaluate parallel supercomputers, at the NASA Advanced Supercomputing Division.

12. **HPC Challenge Benchmark** [149]: This is a benchmark suite comprising of 7 tests, measuring various metrics like rate of memory operations sustainable, total communication capacity of the network, latency and bandwidth of simultaneous communications, etc.

13. **PARSEC** [150]: The Princeton Application Repository for Shared-Memory Computers (PARSEC) is a benchmark suite containing many multi-threaded programs simulating diverse workloads, like image processing, financial analytics, and video encoding.

14. **Kernel** [116] **and other sources** [151, 152] **compile benchmark**

Most of the above benchmarks are either too simplistic or have so many control knobs that it is a daunting task to choose the correct settings [100]. For example, the work in [153] shows that to exhaustively try all settings in the benchmarks like FIO, IOzone and Postmark is too time consuming—instead, it recommends that only the minimum and maximum value for every setting need be tested. It makes an arguable assumption that the intermediate settings will result in output that lies between the outputs for the minimum and maximum settings.

Given any benchmarking tool which has several knobs that can be tweaked, we might still try tweaking the knobs exhaustively to determine those settings which produce a compatible workload for I/O deduplication evaluation. However, the onus of proving that the resulting tweaked workload is a realistic workload would still loom large. Particularly, most of these benchmarks do not have realistic content representation [154], and generate either highly duplicate (eg. Bonnie++ [155]) or highly random content (eg. PostMark [96], Fstress [156]). We thus make the claim that, after having established the initial worth of the DRIVE system using the available traces, there is nothing more to be gained by evaluation using synthetic benchmarks. Further evaluation of the DRIVE system is valuable only if performed using real-world workloads or using "realistic" benchmarks.

A study of various benchmarks and analysis by "benchmarking" of the benchmarks is done in [157], which shows that most of the literature consists of researchers using their own customized or *ad-hoc* synthetic benchmarks, which results in incomparable results across papers. This is an undesirable situation, and the work in [157] proposes that there should first be a consensus regarding the dimensions to be benchmarked, and then agreement on the methodology to benchmark each dimension and comparison of results across systems. It proposes several dimensions for file system benchmarking, like *I/O, on-disk, metadata, caching* and *scaling*. Due to lack of consensus regarding dimensions, and the unsuitability of most existing benchmarks for IO deduplication evaluation, we turn to characterizing the available traces instead.

5.2.5 Summary of survey findings

The purpose of this survey is to find publicly-available datasets of I/O traces with content representation. Our survey revealed that among the publicly-available datasets, only the ones available via IODEDUP paper [68] have content representation, and we have already used them for our evaluation in the previous chapter. All other publicly-available datasets lie in one of the other three categories.

Based on our findings that (i) existing public datasets are not relevant for evaluation of I/O deduplication, (ii) many relevant datasets are proprietary and not publicly-available, as well as, (iii) existing benchmarks are not "realistic" enough for our purposes, we turn to the next avenue of using real-world traces to build synthetic traces ourselves. We performed a detailed literature survey of this area (i.e., generating realistic traces) and find that all such efforts rely on trace characterization of some available real-world traces, to generate realistic synthetic traces.

Table 5.3: Summary statistics of one week I/O workload traces *webvm* and *homes* [68]

Workload type	Filesystem size (GB)	Memory size (GB)	Filesystem accessed (%)	Filesystem size (no. of 4KB blocks)
webvm	70	2	2.8	18 million
homes	470	8	1.44	123 million

Therefore, next we perform trace characterization of the available *homes* and *webvm* traces, which may be helpful to build realistic synthetic traces in future.

5.3 Trace characterization of available dataset

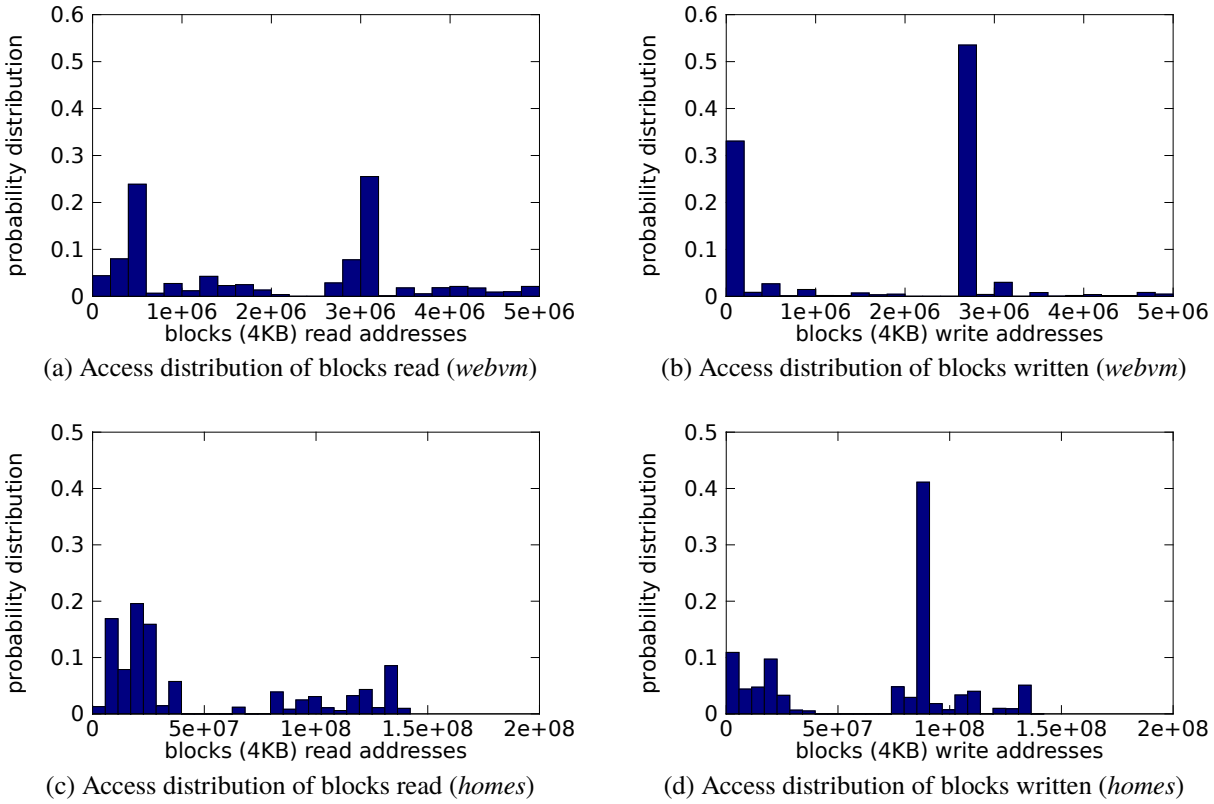
In Section 4.3.3, we presented a brief study of content similarity in the IODEDUP traces available online at [81], by presenting metrics like the "sharing" factor and the content "occurrence" factor in the traces. In this section, we perform a more detailed study to capture more characteristics of the workload which can be used for synthetic generation of realistic benchmarks, in future.

We characterize the *webvm* and *homes* traces from [81] using the following four major metrics for both read and write requests separately. Note that, I/O deduplication techniques attempt to optimize only the disk read performance, so it would be preferable to capture the disk read and write characteristics of the trace separately, rather than together.

1. Blocks accessed distribution
2. Run length distribution
3. Reuse distribution
4. Jump length distribution

For each of the above metrics, we first present a basic definition accompanied by an example and then present the characterization of the traces based on that metric. Along-with the block-based interpretation of the metric, we also present the "content-based interpretation", the point being that I/O deduplication is to be performed for traces that have duplicate content so the content-defined trace characterization is especially relevant.

Before presenting the distributions, we present summary statistics regarding the traces considered in Table 5.3—these are statistics reproduced from the original paper [68]. As can be seen in the table, *webvm* trace is from a filesystem of size only 70 GB whereas *homes* trace has a filesystem of size 470 GB backing it. This implies that the range of block addresses accessed in the latter filesystem is expected to be greater than the former. The primary memory allocated in both systems is also different—2 GB and 8 GB, respectively. Thus, the *homes* trace has been



(a) Access distribution of blocks read (*webvm*)

(b) Access distribution of blocks written (*webvm*)

(c) Access distribution of blocks read (*homes*)

(d) Access distribution of blocks written (*homes*)

Fig. 5.2: Block access popularity distribution for reads and writes in *webvm* and *homes* traces

captured downstream of a much bigger cache than the *webvm* trace—this may be partly the reason why the *homes* trace does not seem to have much duplicate content to be exploited by the I/O deduplication techniques (refer Fig. 4.3 in Section 4.3.3).

In Table 5.3, the percentage (%) of filesystem accessed indicates that in both concerned systems, the total number of block addresses accessed in the trace constitutes only a small fraction of the entire file system size. This implies that if the metadata is constructed dynamically (i.e., while the application is executing), it needs to represent only that portion of the filesystem that is accessed in the workload, and not the whole filesystem. Note that, this requirement is different from that of a storage deduplication system which needs to persistently retain metadata regarding the entire data within the system.

For trace characterization, we consider the entire three-week *webvm* and *homes* trace, and report various distributions (instead of averages) in accordance with the insight in [158] that "distribution matters, not the average".

5.3.1 Blocks accessed distribution

Fig. 5.2(a) presents a probability distribution of the 4KB block addresses read in the *webvm* trace, while Fig. 5.2(c) presents the same for the *homes* trace. The x-axis plots the addresses of the blocks read or written, respectively, such that each address is for a 4 KB block or 8 sectors or 4096 bytes. It can be seen that a major portion of the read accesses in both traces are limited to

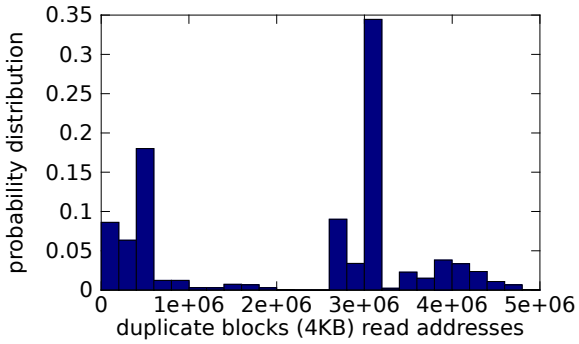


Fig. 5.3: Distribution of accesses of duplicate content blocks

only some portions of the address space, while the rest of the address space gets much fewer read accesses. Similar behaviour is observed in the write accesses as well, as shown in Fig. 5.2(b) and Fig. 5.2(d), respectively. Note that, the total range of blocks accessed is bigger in case of the *homes* trace as compared to the *webvm* trace—this is because the file system size in case of the former is around an order larger than the latter, as shown in Table 5.3.

The above block accessed distributions prove that there is *spatial locality* in the blocks being accessed in the traces, i.e., the accesses tend to be clustered more in particular regions instead of being uniformly distributed throughout the address range. This behaviour is typical of most storage I/O workloads and several models have been proposed in literature to capture this characteristic into realistic benchmarks [159, 126, 160, 161].

Fig. 5.2 refers to all (i.e., unique as well as duplicate) blocks being accessed in the trace, however since we are interested in I/O deduplication, it would be interesting to know whether such spatial locality of access exists for duplicate data as well. The work in [85] certainly demonstrates this to be true of real-world storage content, however the question that faces us is whether this is true of I/O workloads as well.

To answer the question of spatial locality in duplicate content, we consider only those records from the trace which read "duplicate" data, i.e., each trace record considered in this scenario is associated with content that has already been read or written in a previous trace record. For every block of content that we encounter in the trace, we assign it an index called the `dedupID` and if a content is repeated, it is assigned the same `dedupID` as the previous instance of that content in the trace. Thus, `dedupID` is a unique identifier for block content, and in Fig. 5.3 we plot the access distribution for only those content which were found to be duplicate in the *webvm* trace. As expected, we can see in Fig. 5.3 that there is spatial locality within the accesses of duplicate data as well. Thus, our observations echo those present in [85].

5.3.2 Run length distribution

In general, run length refers to the length of a sequential run. In the context of I/O traces, *run length* refers to the length of a sequence of blocks being requested. For example, if the block accesses occur as shown in Fig. 5.4, the first sequence of block accesses results in a run length

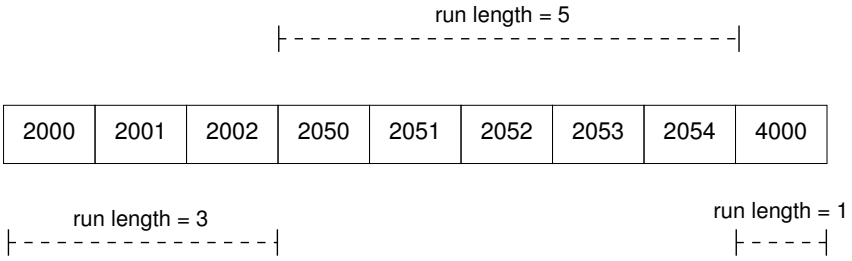
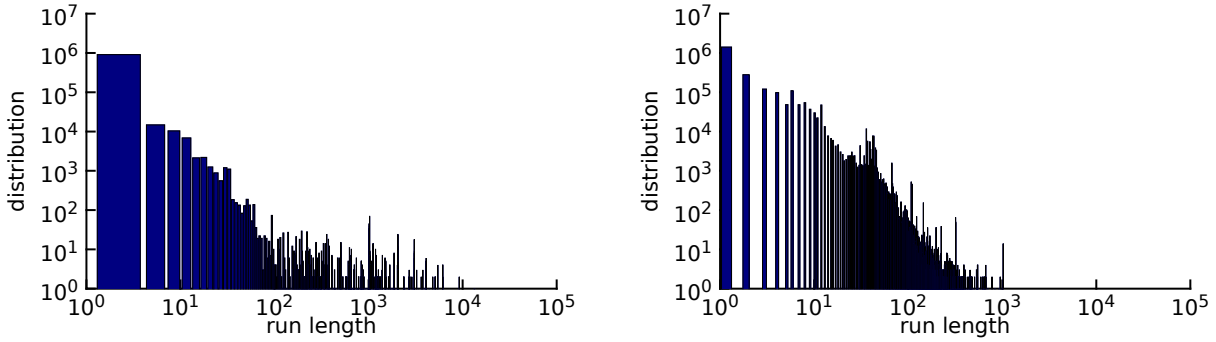
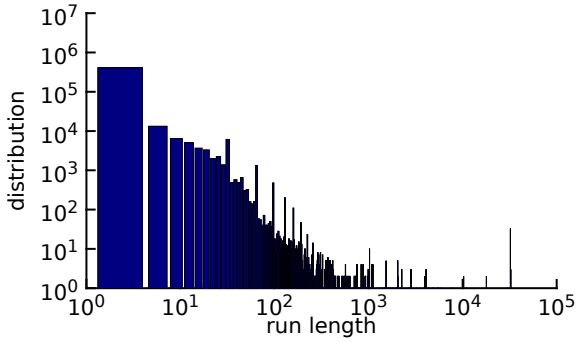


Fig. 5.4: Example to explain the definition of *run length*

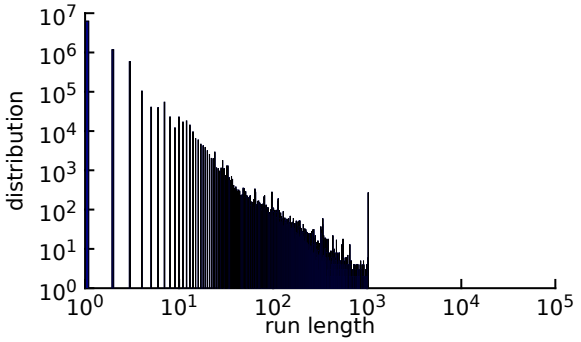


(a) Block run length for reads (*webvm*)

(b) Block run length for writes (*webvm*)



(c) Block run length for reads (*homes*)



(d) Block run length for writes (*homes*)

Fig. 5.5: *Block run length* distribution for reads and writes in *webvm* and *homes* traces

of 2, the second sequence has a run length of 5, and the third sequence has a run length of 1.

For *webvm* trace characterization, we present its run length distribution, i.e., a distribution of the different length sequences in the trace. This metric will indicate whether the requests in the trace have the sequentiality property, or whether they are essentially random access. It may be noted that if there are many huge files being accessed or read from the file system, it will cause bigger run lengths to be recorded in the trace. On the other hand, if the files being read are small (i.e. less than 4KB in size), then the read workload will seem to be random instead of sequential, since entire files each will get read in a single I/O [107].

The run length distributions for read and write requests have been plotted in Fig. 5.5, where it can be seen that as the run length increases, the number of occurrences of that run length reduces. More specifically, read request run length distribution for *webvm* and *homes* reveals the

following important points:-

1. Read run length of less than 10 blocks occur close to 10^6 times in both *webvm* and *homes* traces

2. Read run length of between 10 to 100 blocks occur between 10 to 10^4 times in both traces

3. Read run length of greater than 100 blocks occur around 10 times or less

The above observations enable us to conclude that very long run lengths in read I/O are rare whereas run length of less than 10 blocks is mostly the norm. Our observation of 82% run lengths being of 1 block also corroborates the finding in [126] that most I/O in web service applications is found to be random access and not sequential.

The write request run length distribution plotted in Fig. 5.5(b) shows that a similar characteristic (as observed with reads) is present here as well. The following points may be noted:-

1. In both cases, there is a huge number of instances of run length = 1

2. The maximum write run length is only around 3×10^3 whereas the maximum read run length is over a magnitude larger at around 3×10^4

3. The maximum run length for reads in *homes* trace is 32,510 blocks compared to 1024 blocks for write run length

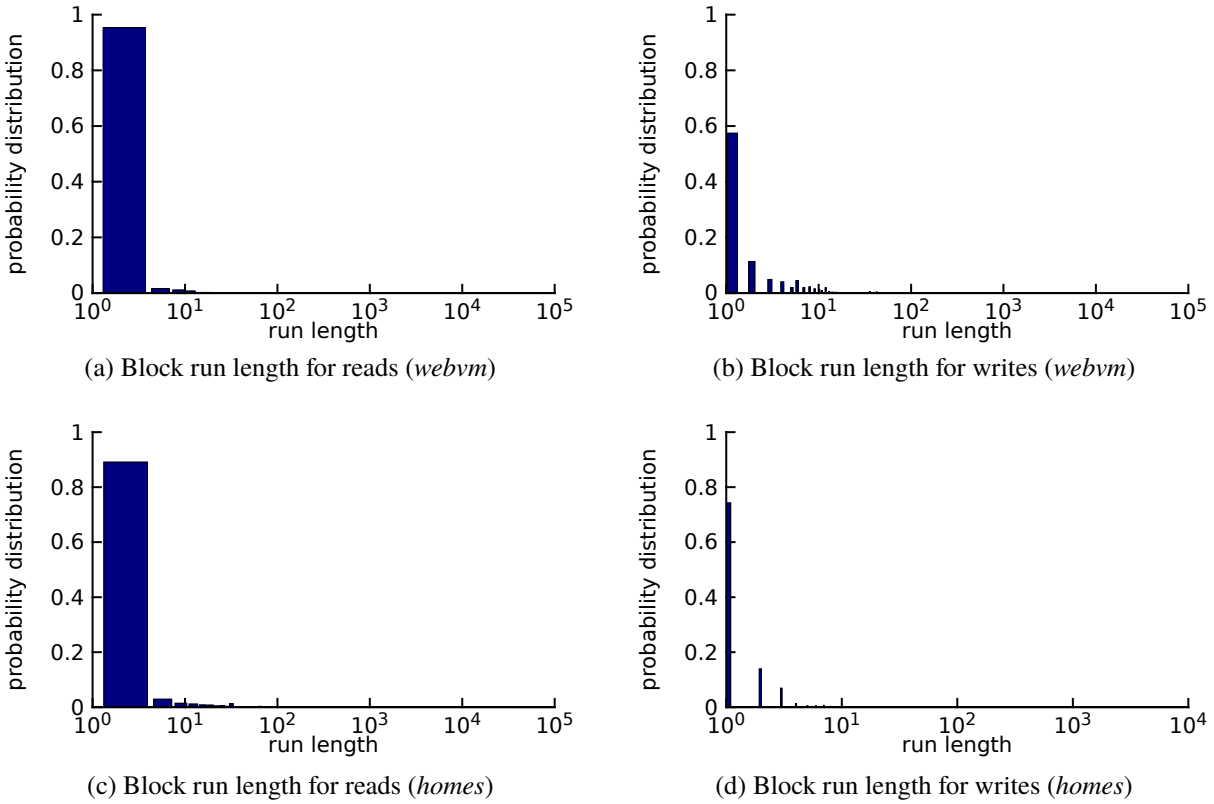
To achieve a better comparison across these different plots, we present the probability distributions below in Fig. 5.6, such that the total number of requests within the trace doesn't skew the number of instances in each bucket. In fact, the probability distribution captures which is the most expected and least expected run length in each of the considered cases. From Fig. 5.6, we can see that

1. The number of instances of run length = 1 in case of read requests in *webvm* trace is proportionally much higher than the corresponding write requests, specifically 86% in reads as opposed to 57% in writes

2. 75% of the values in the both the read and write run length distribution of *homes* trace are for run length = 1 block

3. Additionally, the read requests of both the *webvm* and *homes* trace have more instances of run length less than 10, as compared to the write requests

From the above, we can conclude that, in both the *webvm* and *homes* traces, read run lengths are generally shorter than write lengths, which is indicated by the read run length distributions having taller bars for shorter run lengths, and shorter bars for longer run lengths, in comparison to the write run length distributions.



(a) Block run length for reads (*webvm*)

(b) Block run length for writes (*webvm*)

(c) Block run length for reads (*homes*)

(d) Block run length for writes (*homes*)

Fig. 5.6: *Block run length* probability distribution for reads and writes in *webvm* and *homes* traces

Since our study is geared towards understanding the duplicate content characteristics of the trace, we further present a content-defined version of the *run length* metric, which we refer to as the *duplicate content run length*. Basically, this metric tries to capture whether the duplicate content blocks are those that have a run length of one, or those with longer run lengths.

By investigating the *duplicate content run length* metric, we observed that **all** the 158852 read requests that were found to have duplicate content in *webvm* trace were of run length = 1 block. This is understandable, since most of the read requests in the trace were of run length = 1 block as well (86% as reported above).

5.3.3 Reuse distance distribution

The term *reuse distance* refers to the number of blocks (or content) between two successive accesses to the same block (or content) [68]. An example of block reuse distance is presented in Fig. 5.7 which shows a series of blocks—2000, 2001, 4000, 4001, 2000, 2001, 2002, 4000, 4001—being accessed. In this sequence, the blocks 2000, 2001, 4000 and 4001 have been "reused", i.e., accessed more than once. The reuse distance in each of these cases is as marked in the figure, i.e. reuse distance = 3, 3, 4, 4, respectively. Similarly, *content reuse distance* is defined per access of same content, and is irrespective of its block address. Thus, if blocks 2000 and 4000 have duplicate content as shown in Fig. 5.8, then content reuse distances for that

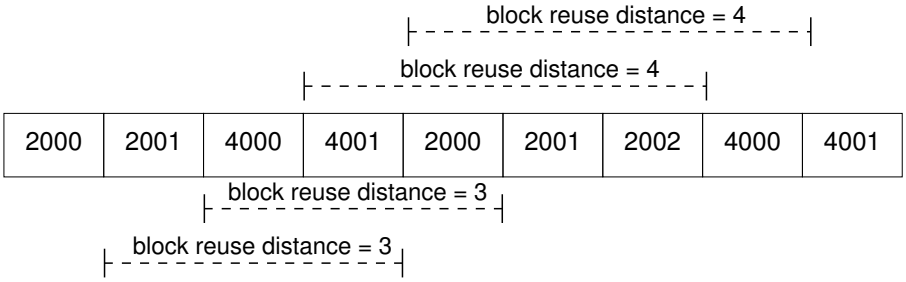


Fig. 5.7: Example to explain the definition of *block reuse distance*

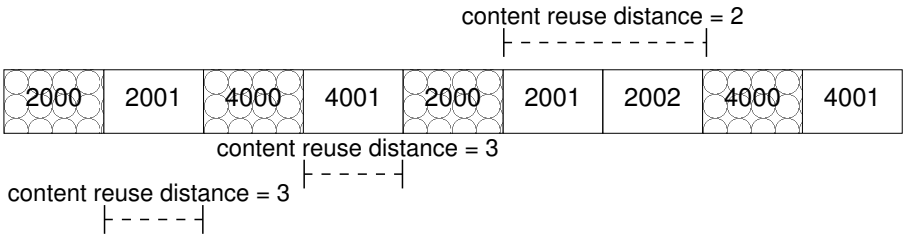


Fig. 5.8: Example to explain the definition of *content reuse distance*

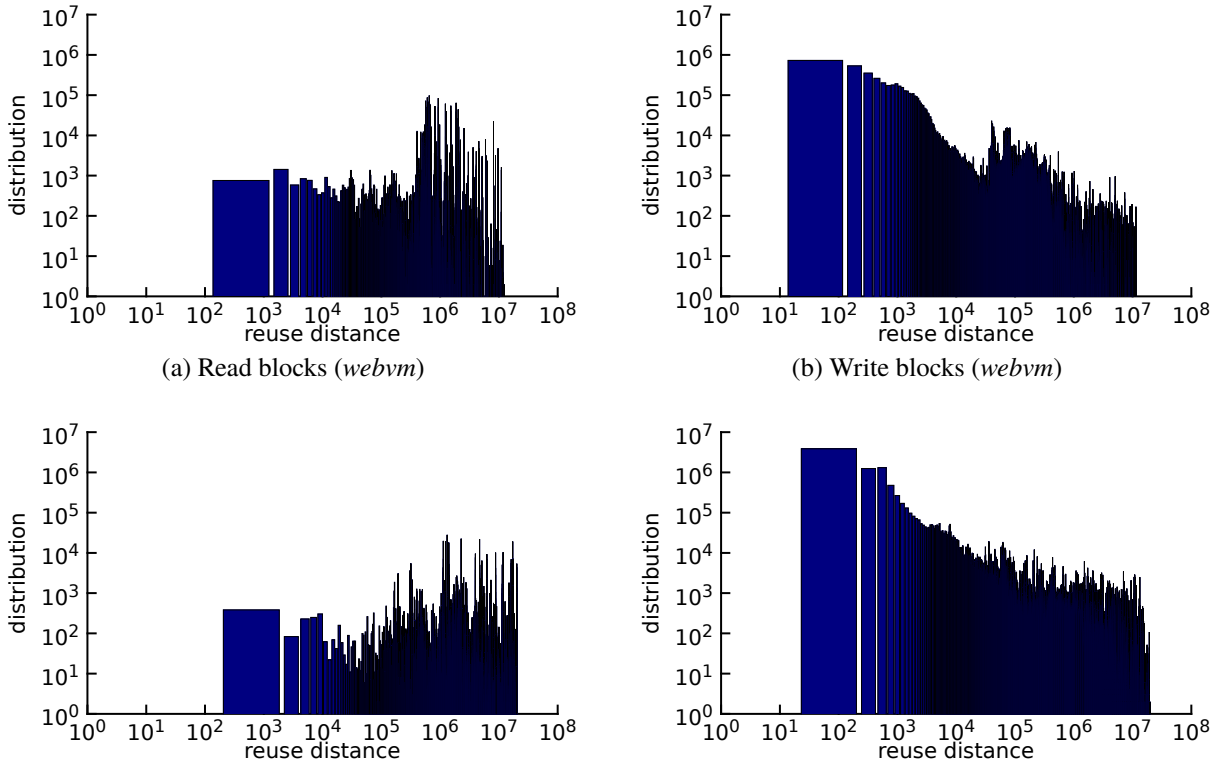
content would be 1, 1 and 2 in the given example.

A study of *block reuse distance* versus *content reuse distance* for the *webvm* trace is present in [68], however only "average" distances are reported whereas we present the "distribution" here. Similar to the observation in [158], we also believe that distributions paint a better picture of the overall scenario than mere averages.

The block reuse distance distributions for the read and write requests for both the *webvm* and the *homes* traces are plotted in Fig. 5.9. In both read and write distributions, *webvm* trace has greater occurrence of smaller reuse distances that the *homes* trace. Also, both traces have smaller reuse distances for write requests as compared to read requests. This may be because when files are over-written using an editor, its corresponding blocks that have been fetched into page cache, tend to get freed up and new blocks allocated for writing the new content [162]. The blocks that get freed up thereby, might get re-allocated soon as and when new write requests come in—this may result in a higher churn of write blocks, in write-intensive workloads.

As per our observations in Section 4.3.3 regarding the similarity study within the three traces, we had concluded that the number of occurrences of each piece of content was greater than the number of distinct block numbers that had the same piece of content. In other words, occurrence factor was found to be greater than sharing factor for each content. This would imply that the distance between accesses to the "same" content should be lower than the distance between accesses to the same block. To verify this, we present a distribution of the content reuse distance in Fig. 5.10.

By comparing each of the graphs Fig. 5.10 (a), (b), (c) and (d) with Fig. 5.8 (a), (b), (c) and (d), respectively, we can see that in each case, the *content reuse distances* are smaller than the *block reuse distances*, as expected. Also, the number of instances of the smallest content reuse



(a) Read blocks (*webvm*)

(b) Write blocks (*webvm*)

(c) Read blocks (*homes*)

(d) Write blocks (*homes*)

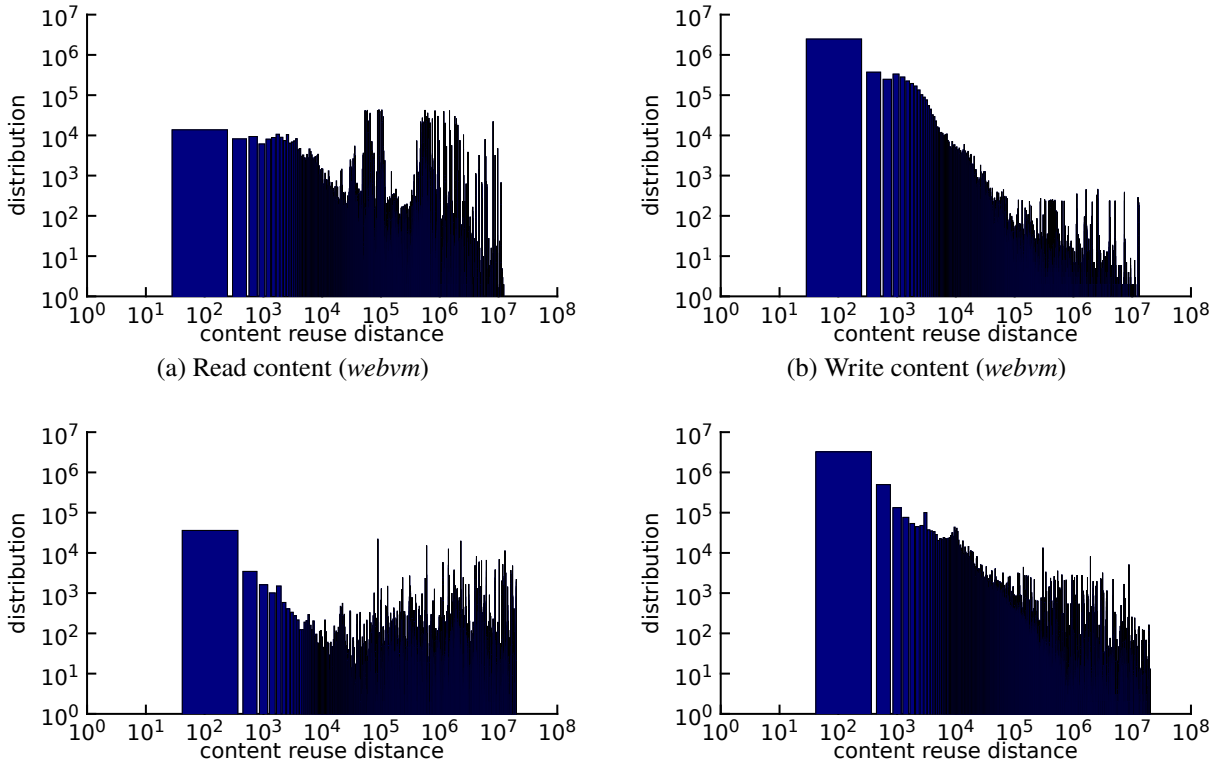
Fig. 5.9: *Block reuse* distance distribution for reads and writes in *webvm* and *homes* traces

Table 5.4: Reuse distance statistics for reads in *webvm* and *homes* traces

Workload type	**Total reads (#)**	**Reads with block reuse (#)**	**Reads with block reuse (% of total)**	**Reads with content reuse (#)**	**Reads with content reuse (% of total)**
webvm	3,116,456	2,800,411	90	2,873,998	92
homes	4,052,176	695,747	17	732,265	18

distance in each case, is at least an order of magnitude larger than the number of instances of the smallest block reuse distance, respectively. Note that, shorter content reuse distance indicates that if the cache is operated in a content-based manner rather than block-based manner, it would provide greater benefits [68].

At a high-level view, it may seem that both the *webvm* and *homes* traces have similar block and content reuse-distance distributions. Note however that, the axes in these graphs are log-log, so even a "slight" difference can be considered as a significant difference in normal scale. To demonstrate that the difference between them is significant enough to result in differential performance for the two traces, we present the statistics in Table 5.4 regarding the total number of read requests in each trace, and the number of read requests resulting in block reuse and content reuse in each of them.



(a) Read content (*webvm*)

(b) Write content (*webvm*)

(c) Read content (*homes*)

(d) Write content (*homes*)

Fig. 5.10: *Content reuse* distance distribution for reads and writes in *webvm* trace

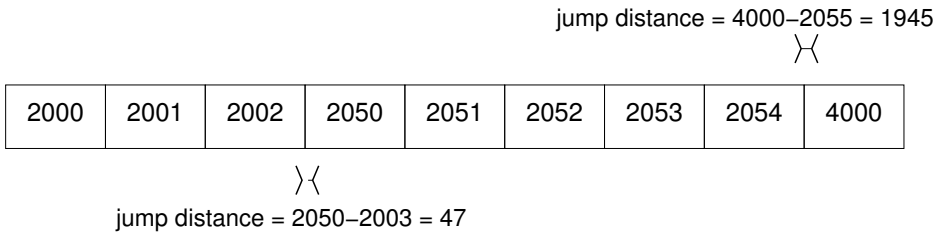
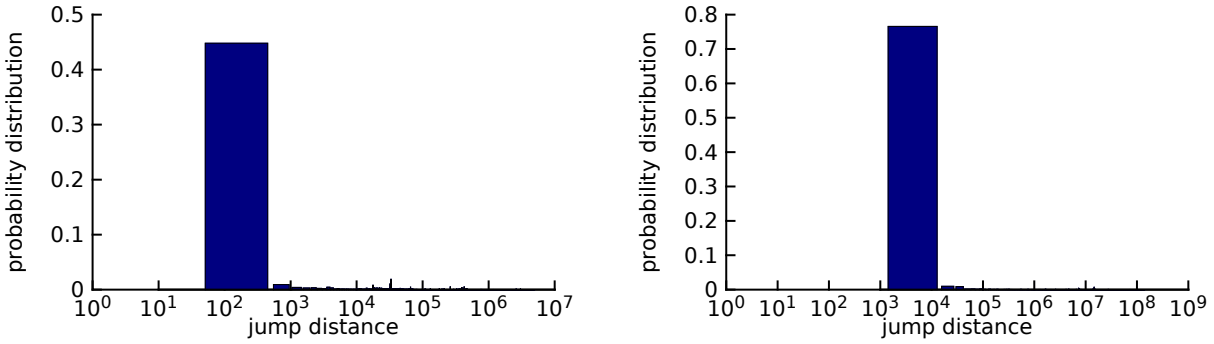


Fig. 5.11: Example to explain the definition of *jump distance*

5.3.4 Jump distance distribution

In the context of I/O traces, *jump distance* refers to the distance between successive I/O requests. Specifically, if consecutive requests are continuous in the logical block address (LBA or LBN) space, then the jump distance between them is zero. However, when there are two consecutive requests (read or write combined) such that they are not continuous, there is said to be a positive jump distance between them, equal to the difference between the two block addresses. Considering the example in Fig. 5.11, we can see that the given request trace consists of three sequences of accesses—2000 to 2002, 2050 to 2054, and 4000. Thus, two non-zero "jump distances" can be noted in the example, one after access to block 2002 and the other after block number 2054.

Although for all of the above metrics, we had presented separate distribution for reads and



(a) Jump distance distribution for *webvm* trace

(b) Jump distance distribution for *homes* trace

Fig. 5.12: Jump distance probability distribution for read/write trace of *webvm* and *homes*

writes, here we present a single distribution for all reads and write requests combined. This is similar to the approach adopted in [163] and makes sense because jump distance is a characteristic of the access trace which indicates how randomly or not, the blocks are accessed (read or written) in the workload.

Fig. 5.12 presents the probabilistic jump distance distributions for both the *webvm* and the *homes* traces. We can see that the majority of jump distances in the *webvm* trace lie between 100 to 1000 whereas they are between 1000 to 10,000 for the *homes*. Moreover, the largest jump distance in the *homes* trace is an order of magnitude larger than the largest jump distance in the *webvm* trace. This is possibly because the filesystem of the *homes* system is similarly larger (470 GB) than the *webvm* system (70 GB), as mentioned earlier in Table 5.3.

In this section, we described various characteristics of the I/O traces and quantified their content-defined versions for the available *homes* and *webvm* traces. In the next section, we make the case that we need to use such characterization to generate synthetic traces for evaluating new I/O deduplication techniques comprehensively.

5.4 The need for I/O deduplication benchmarks

Recently, there has been significant research momentum in the direction of surveying existing benchmarks and datasets to determine whether enough realistic datasets or workloads are available for faithful comparison and evaluation of competing storage optimization techniques [100]. In particular, the fields of I/O performance benchmarking as well as storage deduplication characterization have been found wanting with regards to availability of realistic benchmarks and datasets. In this section, we make the case that there is significant literature in the areas of realistic benchmark generation for network [164] and storage I/O performance [126, 160, 161, 106, 165, 163, 159, 166] as well as storage deduplication evaluation [100], but none in the area of I/O deduplication.

Basically, the I/O trace characterization and benchmark generation efforts focus on the request inter-arrival times as well as the spatial and temporal locality of the access traces whereas

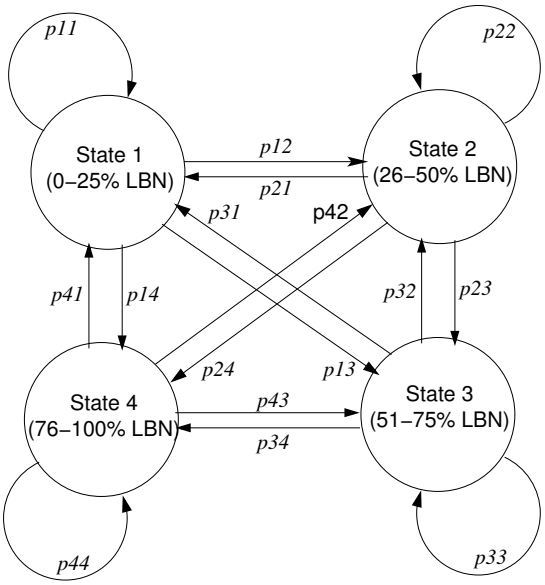


Fig. 5.13: Markov model from [126]: Each arrow *pij* represents probability of transition from one LBN range (state *i*) to itself or another (state *j*)

the storage deduplication benchmark generation effort focusses on modeling the content duplication across temporal snapshots of the same dataset. On the other hand, the requirement for I/O deduplication benchmarks is a merging of the above two types of work, such that both the spatial/temporal locality aspects as well as the duplicate content aspect be characterized and captured in the benchmark realistically. In the rest of this section, we describe the existing work in realistic benchmark generation under each of the categories of (i) Storage I/O, (ii) Network I/O activity, (iii) File system metadata, and (iv) Storage data deduplication.

5.4.1 Generating realistic storage I/O performance benchmarks

The generation of realistic I/O performance benchmarks has received considerable attention in recent literature [126, 160, 161, 106, 165, 159, 166]. The work in [126] builds a Markov model to capture *spatial locality*, with each state representing one-quarter of the block address range (called logical block number range or LBN range) and each transition representing the probability of transitioning from one LBN range to another. The idea is that due to the spatial nature of the workload, most of the transitions would stay within the same state and the probabilities of transitioning out of one state to another would be quite low. Thus, a request stream is perceived as a state machine which transitions between LBN ranges with certain probabilities.

Fig. 5.13 is a reproduction of the Markov model built in [126], and as can be seen, the model has 4 states and 16 transitions between the states. The probability of each transition in this model is determined by characterizing real-world traces, and could be workload-specific (i.e., dependent on the type of workload that needs to be captured by the model). The traces used were for web services like a message store for an email service, image tile storage for a large-scale geo-mapping service, and a blob storage service which hosted huge amounts of user-generated

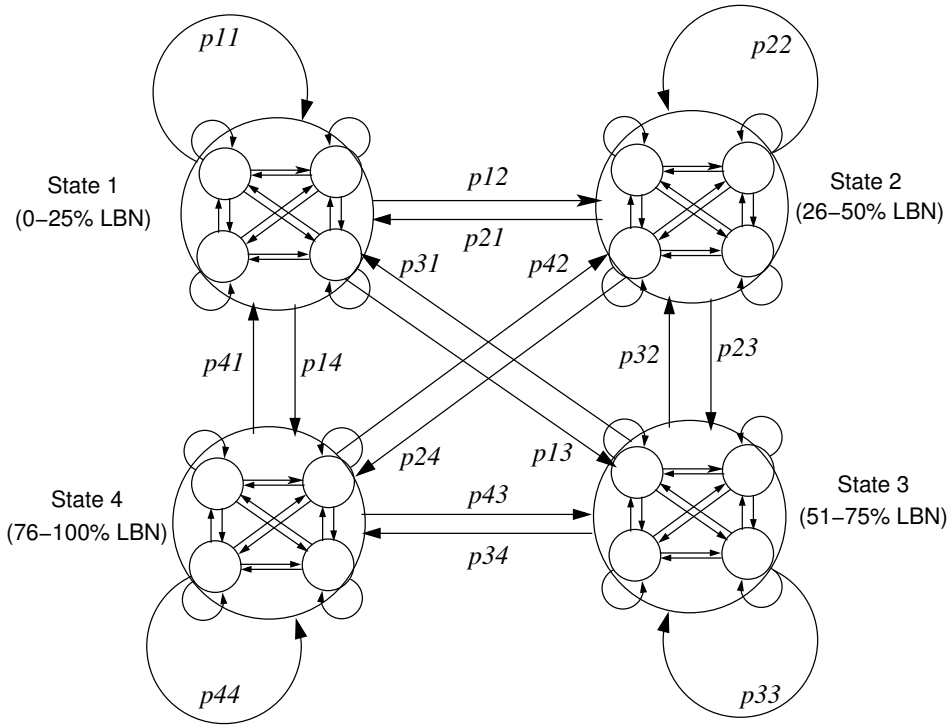


Fig. 5.14: Markov model from [165]: Each arrow *pij* represents probability of transition from one LBN range (state *i*) to itself or another (state *j*). Also, each state *i* is further divided into 4 sub-states each, having internal transition probabilities as well.

content [126].

The work in [161, 160, 165] builds on top of the above Markov model idea by building a hierarchical Markov model such that each of the above states is further divided into 4 finer granularity states. For example, the LBN range 0-25% from Fig. 5.13 is broken into 4 sub-ranges like 0-6.25%, 6.25-12.5%, 12.5-18.75% and 18.75-25%. Pictorially, the hierarchical Markov looks as depicted in Fig. 5.14, although the inner states have not been marked with all the LBN sub-ranges therein.

Representing the storage I/O model as a hierarchical Markov model enables to represent information of a finer granularity than [126]. Thus, the two level diagram has a total of 16 states and 76 transitions, which are once again parameterized using a thorough characterization of available real-world storage I/O traces.

The work in [159] claims that to characterize and recreate a realistic storage I/O workload, it is necessary to not only capture the block accessed distribution, but also the jump distance distribution. In view of this, the approach proposed in [159] is to transform the synthetic trace generation problem into the Hamiltonian Path problem, and then to apply a brute-force, depth-first search to find a Hamiltonian path (this path is expected to have the same jump distance characteristics as the original trace). However, if a complete Hamiltonian Path is not found, approximation techniques are used to construct the access pattern. The evaluation therein shows that if the trace to be generated needs to have less than 150 I/O requests, only then this brute-

force approach is able to find complete Hamiltonian paths—in most other cases, the algorithm is forced to rely on the latter approximation techniques.

The work in [166] hypothesizes that although synthetic workloads are flexible and easy to obtain, the challenge is that synthetic workloads are accurate only if they share certain key properties with the original production workload(s). The unfortunate aspect here is that we do not know which properties are "key" for any given workload or storage system. Thus, regarding the selection of key properties for workload characterization and generation, the work in [166] presents a tool called `Distiller` that can automatically identify them using an iterative trial-and-error method. As input, the `Distiller` tool needs a library of workload attributes, from which it picks one additional attribute at each iteration, and checks whether the chosen set of attributes is representative enough of the given workload. If so, the modeling is done and if not, it chooses one more attribute in the next iteration and so on until a realistic representation is achieved.

5.4.2 Realistic network activity modeling and benchmark generation

The work in [164] presents a modeling scheme called ECHO, which captures the temporal and spatial behaviour of network traffic in large-scale datacenter applications. Two models are built, wherein the first is a distribution-fitting model (called the *single-server temporal model*) that generates per-server network traffic and the second is a Markov chain model (called the *system-wide spatial model*) that captures server-to-server interactions.

The single-server temporal model of [164] consists of using a network trace as an input and identifying known distributions—Gaussian, Poisson, Zipf, etc—in the network activity pattern. The output mathematical expression from this model would be a superposition of multiple known distributions such that sampling of this composite distribution allows to generate network activity patterns that have similar temporal patterns as the ones present in the original trace. However, this method captures the network activity only for individual servers and can not address server-to-server activity patterns or network traffic across sets of machines.

The second model developed in [164] is a hierarchical Markov chain model which captures the network activity patterns for individual servers, across servers within a rack, as well as across racks within a group of racks. Fig. 5.15 is a lifted reproduction from [164] of the schematic of the hierarchical spatial Markov chain model. As can be seen, this model is hierarchical, and the level of detail in the model can be adjusted according to the requirements of each application.

5.4.3 Generating realistic file system benchmarks

Similar to the quest in [166] regarding the key properties of a given workload, the work in [167] is motivated by the key properties needed to realistically represent a file system image. The claim therein, is that, depending on the workload to be emulated or evaluated, the file system image to be used may need different levels of detail. For example, a data mirroring scheme like

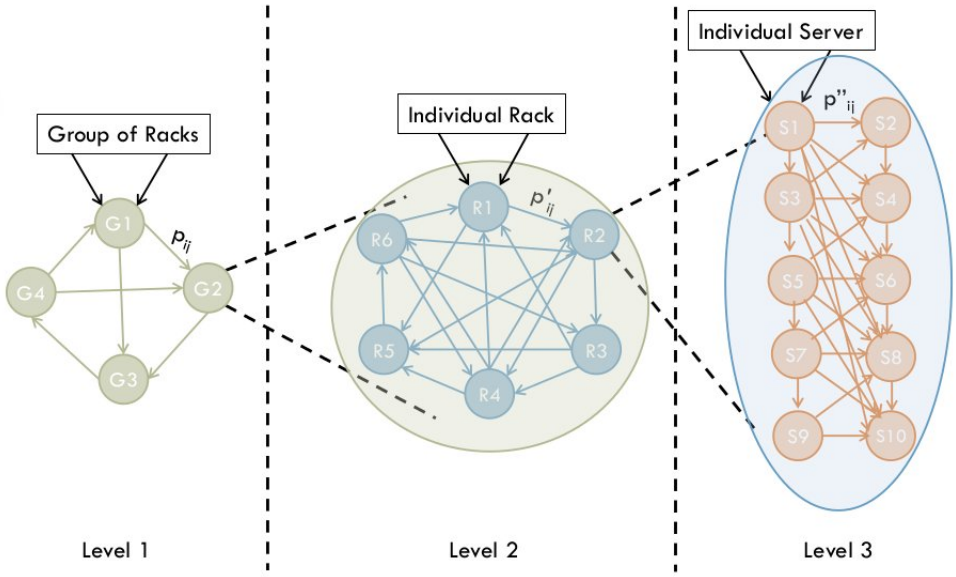


Fig. 5.15: Schematic of the hierarchical spatial Markov chain model [164]

RAID or a system that takes full backups would be independent of both metadata and content of the filesystem. On the other extreme, a desktop search engine's performance would depend on both the metadata and the exact content of the file system on which it operates.

Although the work in [167] does not address the evaluation or analysis of deduplication techniques, the framework (called *Impressions*) presented therein would be useful to capture the content profile of any real file system such that deduplication techniques can be evaluated over it [100].

5.4.4 Generating realistic storage deduplication benchmarks

To justify the creation of benchmarks for storage deduplication, the work in [100] claims that the benchmark or datasets should be such that they should be :-

1. Sufficiently large
2. Having controllable characteristics
3. Easy to distribute to other researchers
4. Easily accessible or reproducible by other researchers
5. Realistic

The requirement of "realistic" datasets is straight-forward—unless the dataset is realistic, there is no way to judge whether any proposed technique is useful in the real world. However, if the dataset is "realistic" because it contains proprietary or private information, then the owner of the dataset may be reluctant to make the dataset publicly available. Further, the requirement that the dataset is sufficiently large, is also at crossroads with the requirement that the dataset be

easily accessible to other researchers, because larger the dataset, tougher the task of hosting and maintaining it online.

Due to above conflicting requirements, the work in [100] suggests a framework which involves capturing the important characteristics of a real-world workload into a model which can be represented with a much smaller storage footprint than the actual workload, and hence can be easily distributed across research groups. The work in [100] generates benchmarks which capture the changes in file system across multiple snapshots (referred to as *mutation*) along-with its content representation, using a combination of a Markov model and a multi-dimensional distribution model.

The work in [154] presents DEDISbench, a realistic storage deduplication benchmark. The I/O generation framework in DEDISbench consists of two components: (i) Access pattern generator, and (ii) Content generator. The access pattern generator can generate sequential, uniform-random or random-with-hotspots accesses depending on user choice. The random-with-hotspots access pattern uses TPC-C NURand function to simulate access hotspots in write requests—access hotspot implies that a few blocks will be accessed multiple times while other blocks may only be accessed once each. If the request is a write request, the content generator generates the random content to be written to the specified block, based on an input distribution of duplicates. A tool called DEDISgen is also presented [154], which can be used to compute the cumulative distributions of block accesses for a real filesystem such that this distribution can be used as an input to DEDISbench for generating the realistic benchmarks for storage deduplication.

Since our requirement is to evaluate read I/O deduplication techniques, and not storage deduplication, hence to use DEDISbench, we would need to tweak it such that access hotspots are created for read accesses as well, and not just for write requests. Moreover, the realistic nature of not only the content in the filesystem (i.e., on storage) but also the content in the read I/O (i.e., in I/O traces) needs to be captured. Specifically, DEDISbench uses the duplicate distribution to guide the content generation for write requests. We need the read access hotspots to also be similarly guided by a duplicate distribution of content present in real I/O traces—basically to capture the dual metrics of *sharing factor* and *occurrence factor* of every content in the trace.

Based on the above surveyed literature, we conclude that there are no realistic benchmarking tools or trace generation frameworks that can be used for evaluation of I/O deduplication techniques. Thus, for the evaluation of such techniques, researchers need to either collect multiple production workload traces for their own research, or create frameworks that can synthesize realistic I/O traces along-with content representation.

Chapter 6

Open Directions and Future Work

This chapter acknowledges other gaps yet to be filled in the area of resource management and performance optimization for virtualized services. Some of the following problems are offshoots of the research contained in this thesis, and some others are crucial open problems that are orthogonal to the work in this thesis.

6.1 Affinity-aware CPU Usage Estimation for P2V-Transitioning Services

The first and foremost issue that needs to be addressed to provision applications in virtual execution environments, is the mapping of resource requirements from physical to virtual environments. Maximum service level agreements and resource guarantees need to be met by the mapping that is determined, while simultaneously maximizing the resource multiplexing potential so that the total required capacity be minimized. This problem is the P2V transition problem.

Timothy et. al. [4] have demonstrated a profiling and modeling approach to developing a generic model for predicting a VM's virtualized CPU requirements, given various resource usage profiles in the physical domain. Thus, this work considers each VM a single independent entity. However, in a typical virtualized shared hosting platform, there may exist several applications with mutually communicating components/tiers placed in separate VMs. Thus, these VMs are no longer independent entities and the communication that flows between them could potentially affect their individual CPU resource requirements. Taking cognizance of the changed CPU requirements is essential to guarantee both minimal performance degradation and maximal multiplexing of physical resources.

6.2 Tracking workload upon migration

Our models in the first component of this thesis are constructed and validated with the assumption that the same workload in dispersed-VMs case is to be serviced in the colocated-VMs case.

However, in a real server consolidation algorithm, it is worth determining whether this assumption holds across a VM migration. That is, when a VM is migrated out of a colocated case due to overload, so that it can be allocated more resources on another PM, what is the impact of this on its observed workload?

Alternatively, in a dynamic server consolidation algorithm, VMs may be migrated due to increasing load level. Thus, the load levels before and after the VM migration need not necessarily be the same. To handle this, we need a workload tracking component within the server consolidation algorithm [7], which can then give inputs to our CPU estimation models for accurate estimation of CPU usage on the target PM.

6.3 Handling diversity in network topology

The affinity-aware CPU usage estimation models have been developed for VM pairs that are connected to the same layer 2 switch, when dispersed. As discussed before, the absolute CPU usage of dispersed VMs seem to be significantly different even when sustaining the same network traffic rate over a 100Mbps switch versus a 1Gbps switch. In a large data-center network, there exist several such switches and any pair of VMs might be communicating with one another over an array of interconnecting switches. To begin with, we need to validate that the CPU usage prediction models that have been developed by benchmarking over a single switch are still applicable across PMs that are connected by multiple switches. The next point to consider is whether the network consists of only one type of switch (either 100Mbps or 1Gbps or 10Gbps, etc) or whether it has a mixture of all types. If all switches in the network topology are homogeneous, then the same model can be used for all predictions. However, if there is heterogeneity in switch capacities, then we might need separate models to be developed per switch type. Thus, when a VM migration is to be initiated, the CPU usage prediction model to be used should be the one corresponding to the switch to which the target PM is connected. We would also need to consider the effective rate of data transfer when two mutually communicating VMs are connected to their respective switches via links of different capacities.

6.4 Metadata space management for I/O reduction

For deployment of I/O deduplication systems on large System Volume Controllers, metadata space requirement can be huge. An option is to distribute metadata between memory and disk, such that only "important" metadata are present in memory for quick access. The work in [70] achieves around 99% metadata cache-hits using this approach, under a key assumption that backup workloads access metadata sequentially. However in case of random-access workloads, metadata access prediction is non-trivial.

Random-access workloads exist in inline storage deduplication scenarios [85], where a simple LRU cache is used to accommodate metadata. However, a significant difference from our

work of I/O deduplication is that storage deduplication needs to track only write requests whereas we require to track read requests. This implies that though [85] found no significant performance improvement by using different caching policies, our work could expect to benefit from a frequency-aware caching mechanism like Adaptive Replacement Cache (ARC). Thus, we propose to use a framework for metadata store which ensures that more-frequently-used and/or more-recently-used metadata stay in cache, while the rest are moved to disk.

6.5 I/O reduction using variable-length blocks

This problem deals with the feasibility of performing disk read I/O reduction by identifying duplicate content in terms of variable-sized sub-blocks, instead of fixed-size blocks. We refer to this optimization as vDRIVE. Similar to the DRIVE system, upon the receipt of every block read request at the virtual disk front-end driver, the metadata can looked-up and the I/O request redirected so as to manipulate the underlying sector-based cache effectively like a content-based cache. However, the difference between DRIVE and vDRIVE would be that, DRIVE redirects read requests by substituting one block address by another, whereas vDRIVE metadata can potentially map one block address to multiple sub-blocks/chunk, which in turn, map to multiple blocks. Thus, every block read request in vDRIVE may potentially result in extra block read requests. However, if the duplication ratio is significant, the improved cache efficiency will outweigh the impact of the extra block reads. Due to lack of access to real-world traces at the moment, we do not include the design and implementation of vDRIVE in current thesis scope. However, as future work, we can implement a prototype of vDRIVE and evaluate using real-world traces (if available) or realistic benchmarks to demonstrate capabilities of the system.

6.6 Generating realistic I/O deduplication benchmarks

As per our discussion in Chapter 5, there is a dire need of research in the direction of generating realistic I/O traces along-with content representation. Most research regarding I/O trace generation so far has focussed only on metrics like working set sizes [168], I/O sizes [106], request inter-arrival times [161], block accessed distribution and jump distance distribution [159]. However, without any form of content representation, it is not possible to evaluate I/O deduplication techniques using such traces. A future work of this thesis could be to develop such trace characterization and generation methods using real-world workloads, such that it can present some content representation within the trace without sacrificing anonymity or privacy of the system and its users.

6.7 Empirical approach to performance-aware resource requirement estimation

In this component, the problem is to perform empirical measurements for a given virtualized service and its resource configuration to determine the peak workload supported for a given performance requirement. Further, if the supported workload falls short of the requirement, then we should be able to identify the bottleneck resources and allocate more until a configuration is reached where the target workload is met [7]. Since this is an offline task, and not a real-time one, it might be acceptable to do empirical measurements to determine the resource requirement. However, it would still be desirable to compute this "optimal" resource configuration as quickly as possible [169] since it might not be feasible to do empirical measurements at all resource configurations exhaustively to find the optimal allocation.

Chapter 7

Summary and Conclusions

To address the problem of server consolidation or migration to a target machine (example, for alleviating resource usage hotspots [7]), the foremost requirement is to estimate the expected resource requirements of the virtual machine on the target machine. This knowledge in turn, can enable us to make the decision of whether such a migration or server consolidation is advisable. As part of this work, we concern ourselves with the Xen virtualization environment and are interested in predicting the CPU resource requirements of VMs. In Xen virtualization environment, since the privileged domain Dom0 needs CPU scheduling to perform IO operations (network operations and disk operations for NFS-mounted VM images) on behalf of the VMs, we are also interested in predicting the corresponding Dom0 CPU resource requirement. Though we work with Xen, our approach is generic enough to be applicable to other virtualization solutions.

In this thesis, first we considered the problem of server consolidation with an affinity-aware approach. Two VMs are said to have *network-affinity* if they communicate with each other, and we show that colocation of communicating VMs on a single PM helps reduce the total CPU usage. With detailed measurements, we justified the need for affinity-aware placement and presented our approach to estimate the colocated CPU usage for dispersed VMs and vice versa. We further presented evaluation of the constructed estimation models with synthetic workloads and a real application. We observe that the pair-wise models to predict different CPU usage perform well, achieving maximum prediction error within 2% absolute CPU utilization. We also applied the pair-wise models to multi-VM scenarios using a multi-phase prediction methodology, and evaluation showed maximum error to be within 2%.

Next, we considered the problem of improving disk access performance by using the disk cache more efficiently. I/O reduction refers to reducing the number of disk read accesses by employing better cache management strategies. Typically, caches are referenced by block number and mitigate the necessity to fetch the block from disk. However, traditional caches can not recognize content similarity across multiple blocks and hence, the system ends up fetching and storing multiple copies of the same content in cache. Elimination of duplicate read I/O requests

which fetch the same content repeatedly is referred as I/O Deduplication. Existing work uses a split-cache approach, with a part of the block-based cache reserved as a content-based cache. In this work, we demonstrated that the split-cache approach is sub-optimal, and presented an I/O reduction system called DRIVE which performs I/O redirection to *implicitly* manipulate the whole underlying cache as a content-deduplicated cache. Only the VM's own disk access history is introspected to obtain implicit hints regarding host cache state, to be used for read I/O redirection.

We performed comparative evaluation by implementing prototypes and performed trace-based evaluation in a custom simulator. The evaluations showed that, the DRIVE system manipulates the entire available host cache space effectively like a content-cache, achieving a high content deduplication factor of up to 97%. This is the key reason for better performance, with up to 20% higher cache-hit ratios, and up to 80% higher number of disk reads reduced than the Vanilla system.

Appendices

Appendix I

LoadGen: A custom micro-benchmarking toolkit

This chapter describes the process of workload generation in detail. In order to build the model(s) proposed in Chapter 3, we adopt an empirical approach based on micro-benchmark profiling on the virtual machines. For this, we intended to generate various types of loads (CPU, disk, intra-PM and inter-PM network loads) at different levels and measure the resultant CPU usage. There are quite a few tools available off-the-shelf for load generation. However, in many cases, these tools were insufficient for our purposes. In this chapter, we describe the existing tools and their insufficiency, and also present the design of our custom micro-benchmarks.

I.1 Toolkit requirements

The prime requirements of our workload generation step were as follows,

1. We should be able to generate load on various resources (CPU, network and disk) at variable rates, i.e., we need to exercise progressively increasing (or decreasing) levels of activity within a range.

2. To create more realistic scenarios, we also need to generate workloads in combination, i.e. load on multiple resources simultaneously

3. Any load level chosen should generate a fixed load, unless resource constrained by another parallelly executing workload, i.e., the interference should only happen if the resource capacity is saturated, and not for any other reason.

The load generation, logging and log transfer are to be automated via a script on a central *controller* machine in our setup. The script should ssh to specified VMs and PMs, and start off load, start off logging, and later stop logging, stop load, and collect logs into a specified central repository. To enable automatic ssh without prompting for manual password input each time, need to perform secure ssh key exchange between *controller* machine and all others.

I.2 Unsuitability of existing tools for our micro-benchmarking experiments

Our first recourse was to search for existing load generators. We were able to find a huge set of such tools per load type, say CPU, network I/O and disk I/O. We also found a few tools that could generate more than one of these different load types. However, we found all of these tools unsuitable for our purpose due to various reasons, as cited below.

I.2.1 Existing tools for network traffic

For generating rate-specific network loads, we initially experimented with creating UDP workloads. For this, we explored tools like bwUDP and iperf [170]. However, we observed that the tools themselves were causing high CPU utilization in the DomU. We viewed such extra CPU usage as unacceptable "noise". Besides, UDP workloads were not representative of most applications—which would likely use TCP connections for communication. We also tried httperf [171] to generate file uploads and downloads. However, we observed that the network traffic generation tools had high CPU overhead for their usage, which was against our requirements.

I.2.2 Existing tools for disk load generation

For disk load generation, we explored several tools including IOzone [142], Iogen [172], Rugg [173], LMBench [174] and Vdbench [175]. However, we found that tools like Iogen, IOzone, Rugg and LMBench are designed to "stress test" the disk and measure the resulting saturated I/O performance. In other words, these tools offer no rate control. Moreover, the Vdbench tool is written in Java, and resulted in around 60% CPU utilization, which was not acceptable since we required to measure the CPU utilization resulting purely due to the disk load being generated and not due to the workload generator overhead.

There are other tools which we were unable to use, like HP's Disk Bench (DB), which is available only as part of HP Developer & Solution Partner Program (DSPP), and fio [144] whose documentation was not very adequate at the time. Intel IOMeter [143] was a possible option for disk load generation, however it is a platform specific tool and had no AMD support (at that time, our working setup had AMD machines).

I.2.3 Existing tools for CPU load generation

For generating CPU load, we checked out a few tools like Lookbusy [176], Load [177], and Stress [178]. Lookbusy attempts to keep the total CPU utilization at a particular level, by manipulating its own activity level according to any other CPU usage that maybe already being incurred on the machine. For example, if we request Lookbusy to generate 60% CPU load,

and if there is some other workload already executing which uses 20% CPU, then Lookbusy generates only an additional 40% CPU usage to bring the total to 60%. This is contrary to our needs, since we need CPU load of a particular level to be generated irrespective of any other CPU utilization that may be present.

The Stress tool simply "stress-tests" the system on various resource axis, and was not of much use for rate-controlled micro-benchmarking. The Load tool generated CPU using two threads—a controller thread and a worker thread. The controller thread starts off the worker and goes to sleep for a certain interval x, and stops the worker thread after waking up. After waiting for an interval of 100–x time units, the controller thread starts off the execution of the worker thread. Hence the expected CPU utilization caused by the worker thread = x/100 = x%. However, this is true only if the Load program is the only process executing on the machine, and the presence of any other program would be unnoticed by the controller thread, hence potentially resulting in lower than x% utilization. Such kind of inherent interference during load generation is unacceptable for our purposes, at least during combinational load generation.

Due to the above problems, we resolved to build a custom load generation tool, using simple system commands & programs, like `dd` for disk loads, `wget` and/or TCP socket programming for network loads, and Fibonacci series computation programs for CPU load. Our language of choice was C. The program can take inputs regarding different types of loads and can spawn multiple threads, one for each workload type. Next, we describe in more detail the steps adopted to create various types of workloads.

I.3 Custom micro-benchmarks

Due to the inadequacy of existing workload generation tools for micro-benchmarking, we designed and implemented a customized workload generation tool. Implemented in C language, it spawns separate threads for each workload type and can be used to generate CPU load (C), disk read load (R), disk write load (W), and network load (N). This tool can be executed on any machine (physical or virtualized) on which load generation is required, with appropriate input parameters regarding the load level for each requested workload type. Combinational loads (i.e., combination of any or all among cpu, disk-read, disk-write, intra-PM and inter-PM network traffic) can be generated by specifying inputs for each workload type and multiple load generating threads are spawned accordingly.

As mentioned previously, our intention is to generate various levels of loads for each workload type, and then generate profiles for them. However, accuracy of load generation is not mandatory, in the sense that, if the load requested is 90% CPU utilization, the load generation tool should generate utilization approximately close to 90%, although it need not be exact. Similarly, when asked to generate 50M bps network load, the tool may generate some network load in the range of 45 to 55M bps. However, the load level thus generated (and observed) should be

deterministic for the requested load (i.e., it should generate the same requested level each time), and this is sufficient for our cause.

I.3.1 CPU micro-benchmark

We generate CPU workload by first computing off-line, the first $2,000,000$ Fibonacci numbers and noting the time T taken, say $T = 10ms$. Multiple such Fibonacci computation operations consecutively form a string of Fibonacci operations. Different levels of CPU utilization are attained by varying the length of such Fibonacci operation strings (*active time*) and the length of sleep time between consecutive strings (*sleep time*). To generate a CPU load of $X\%$, X consecutive strings of $2,000,000$ Fibonacci numbers are computed and then the computation thread sleeps for $(100 - X) \times T$.

The difference between our approach, and the approach in Load [177] tool is that we use a single thread to compute and sleep in such a manner that the required load is generated, whereas the Load tool uses two threads—worker and controller, such that worker does computation and controller causes it to sleep or awaken. In the worker/controller setup, if there are other CPU-intensive loads present, the worker may not have gotten enough computation time but the controller will still suspend it because it is unaware of the external interference. However, in our case, a single thread is responsible for the CPU load generation, and can better keep track of its own activity periods.

I.3.2 Disk read & write micro-benchmark

We generate disk read (or write) workload by reading (or writing) files of a specific size, continuously one after the other, for a period of time T (say, one minute). In order to vary the disk read access rates (in read load), we need to to overflow the RAM in such a manner that every read fetches data directly from the disk instead of finding it in cache. To ensure this, for every file size input (example, x KB) for disk read, we pre-create a base file set containing files of size x KB, such that the total size of the dataset is equal to twice the RAM size. Since the files are read in the same order each time, it is ensured that after a file is read from disk once, the next read request to that same file will happen only after that file's previously fetched contents have been flushed out of the RAM and cache, thus ensuring that disk access occurs.

I.3.3 Network micro-benchmark

In the first version of the tool, network receive load at a VM was generated by having the load generation tool invoke `wget` on a URL to fetch a 128 KB file. The receive rate (or bandwidth) was controlled by varying the sleep between consecutive file fetches. Similarly, to create network transmit load at the VM, another VM invokes `wget` targeted at a URL on this VM. In other words, to create network transmission load at VM1, `wget` requests are generated at VM2 directed at

VM1, which causes VM1 to respond by transmitting the requested file.

In the second version of the tool, we changed the network load generation functionality such that the segment size of network transmission also could be varied. Specifically, instead of using 128 KB fixed size files only, we generated various levels of network traffic between a pair of VMs by using a TCP-based custom application that sends strings of bytes on a TCP socket with different periodicity—the length of the byte string (referred to as segment size) can also be varied to generate different levels of network traffic. We assume the maximum available capacity to be $100Mbps$ and vary the load on each VM by steps of $10Mbps$, from 10 to $90Mbps$. To generate various levels of network traffic, we varied the transmission interval as well as the segment size of transmission.

I.3.4 Mixed workloads

In the above benchmarks, we have generated workloads that utilize one resource axis at a time. However, most applications would use more than one resource for their execution. Our load generation tool has a multi-threaded implementation, such that a thread can be spawned for each type of load (CPU, network or disk) to be generated. Thus, combination workloads can be generated by spawning multiple threads, one for each load type and level.

I.4 Micro-benchmark workload intensities

For our profiling step, we use a generic client-server setup (described earlier in Section 3.4.1), wherein a client (the controller machine) remotely connects to the servers (each PM or Dom0, and VM or DomU) where workload is to be generated. The workload generation tool resides on each such logical "server" and complies with the load generation requests received from the logical "client" machine.

For CPU micro-benchmarking, we split the CPU load into nine different intensities ranging from 10% to 90%. At each of the two VMs, we vary the CPU load from 10% to 90%, thus resulting in 9 $\times$ 9 = 81 CPU workload combinations, as input for the modeling process. For network loads, we assume the maximum available capacity to be 100 Mbps and vary the load on each VM by steps of 10 Mbps, from 10 to 90 Mbps. For disk read and write micro-benchmarking, we vary the file size from 4KB to 8MB, in powers of 2.

In order to train the model on some realistic data, we also needed combinational workloads, i.e., multiple workload types executing in parallel. However, to conduct exhaustive experiments that cover the entire combination input set is not possible, since there is an exponential number of cases in the input space of combinational load. Thus, we adopted a small workaround of choosing random input numbers for each workload type. This is intended to keep the sample points uniformly distributed throughout the available sample space. Thus, a combinational load input to a single load generator instance would be a 5-tuple of the form $< c\%$, a Mbps, n Mbps,

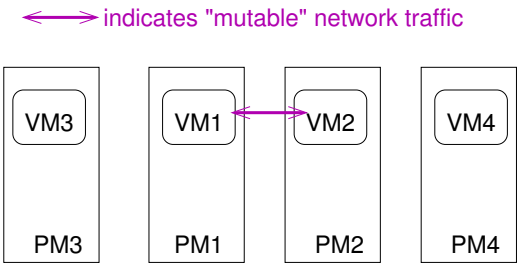


Fig. I.1: Setup for *mutable* network traffic generation: *VM1 and VM2 communicate with each other, and get dispersed or colocated*

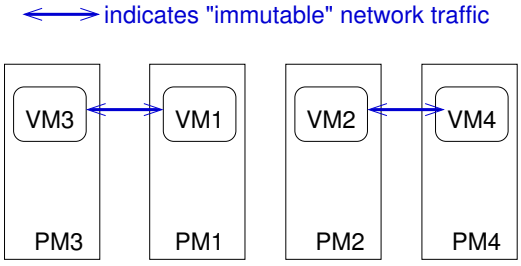


Fig. I.2: Setup for *immutable* network traffic generation: *VM1 and VM2 have communication with other VMs (i.e., not with each other), and get dispersed or colocated*

r, w $>$ where c is a random number in [1-100] for generating c% CPU load, a is a random number in [1-90] for generating a Mbps intra-PM network receive traffic, n is similarly random for generating inter-PM network receive traffic, r and w are randomly chosen file sizes between 4 KB and 128 MB.

I.5 Illustration of difference between mutable & immutable network traffic

The workload generation program itself does not distinguish between the way mutable and immutable network traffic is generated. This terminology is of semantic importance for the modeling only, since the network load generation tool simply proceeds with whichever input IP address is specified. Semantically, to create intra-PM network traffic between a pair of VMs, the VMs should be colocated on a single PM and the network transmission is done from one of the VMs under consideration to the other.

Our experimental setup consisted of 4 VMs hosted on 4 PMs. Fig. I.1 shows the setup used to generate mutable network traffic, and in contrast Fig. I.2 shows the setup for immutable network traffic experiment. Here, the resource usage profiling is being performed on VM1 and VM2 by placing them on dispersed and colocated configurations, while the other two VMs are stationary. Hence, the difference between these two setups is that mutable network traffic is generated between the pair VM1 and VM2, whereas immutable network traffic is between two pairs, (i) VM1 and VM3, (ii) VM2 and VM4. Note that, in case of mutable traffic setup, colocation of VM1 and VM2 (on PM1 or PM2) changes the nature of network traffic from *inter-PM* to *intra-PM*. On the other hand, in case of immutable traffic setup, colocation or dispersion (on PM1 and PM2) of VM1 and VM2 causes no change in the nature of their network communication with VM3 and VM4, respectively—it stays *inter-PM* in both cases.

Note that, above we have assumed that colocation can happen only on PM1 or PM2, and not on PM3 or PM4. This is because of two main reasons. Firstly, the above description is with respect to the VM pair of VM1 and VM2. Secondly, if a single VM migration can cause it to

be colocated with multiple VMs or to be dispersed from one VM and colocated with another, these scenarios are considered multi-VM scenarios, and are handled separately as discussed in Section 3.5.3.

I.6 Tool setup

Every experiment is automated—simply specifying the IP addresses of the VMs and PMs, specifying whether they need to be colocated or dispersed, and specifying load to be generated, is sufficient. The scripts on *controller* machine take these inputs and setup the experiment scenario accordingly—VMs are instantiated on corresponding PMs (if they are not already running) and the systems prepped for load generation. For example, disk read workload requires the pre-creation of large dataset of size twice that of RAM. Additionally, `iptables` rules are setup to capture the mutable and/or immutable network traffic being generated in the given setup.

For each benchmark load, we start load generation on the two VMs and then start off the logging on all concerned systems (the VMs and corresponding PMs). The metrics logging is done every second, and continues for a minute. After a minute, logging is shut down and then the load generation is stopped. The generated logs are then transferred to the log repository, for further analysis and model generation.

For resource usage measurements, tools like `Xentop`, `top`, `sar` and `iptables` are used. We need the measurements along-with timestamp information, so that different measurements can be correlated with each other. Tools like `sar` and `top` report timestamps themselves, however, `Xentop` does not provide timestamp information. Hence, we use a perl-based wrapper for `Xentop`—the wrapper invokes `Xentop` in batch mode every second, and notes the timestamp along-with Xentop output.

I.7 Tool usage

The main load generation tool in our `LoadGen` toolkit, is a C program `generate_loads`. Its usage snippet is presented in Listing I.1.

```
Usage: ./generate_loads <daemonize> <expt-duration> [[<thread-type> <rate>
    <number> <name>]]

<expt-duration> should be integer
<thread-type> can be R, W, N, C
<rate> in Kbps for net with 1Mbps=1000Kbps and max=100Mbps & <rate> in perc
     for cpu
<number> can be 'number of 4k blocks' for R/W, 'number of 4kB' for N and
    xxxx for C
<name> can be file-prefix for R/W, web-server IP for N and xxxx for C
```

Listing I.1: Listing of generate_loads usage

The 4-tuple [[<thread-type> <rate> <number> <name>]] seen in Listing I.1 indicates the input per workload-type to be generated. Multiple such tuples can be input on the command-line to indicate a mixed workload, as well. The different types of workloads are indicated using the <thread-type> option—'R' for disk read, 'W' for disk write, 'N' for network and 'C' for cpu workload.

Appendix II

SimReplay: A custom simulator to study cache management effectiveness

In the evaluation section for DRIVE system, we briefly mentioned the custom simulator that we built to perform the quantitative analysis of IODEDUP system's efficiency, as well as to evaluate the performance of the DRIVE system in comparison with the Vanilla and IODEDUP systems. In this Appendix, we describe the design and implementation of the simulator, called `SimReplay`, in more detail.

The main requirement from the simulator is that it capture the functionality of the Vanilla, IODEDUP and DRIVE systems and demonstrate the effect of each of these I/O access mechanisms on the cache efficiency and read access performance. We have built the simulator to be modular so that it could be used to develop and test new I/O deduplicate & redirection strategies as well.

II.1 High-level functioning and requirements

The simulator basically accepts virtual disk requests from multiple VMs hosted on a single physical machine, and simulates its I/O execution on the physical machine. The I/O execution in the standard case is referred to as Vanilla, while the augmented execution scenario presented in [68] is referred to as IODEDUP. A third execution scenario in this simulator is our system called DRIVE. Since the simulation is of virtual disk requests sent by VMs, an input mapping needs to specify which virtual disk block of a VM maps to which corresponding physical disk block on the physical host's storage. Two main components of the simulator are the simulation of two caches on the physical host: (i) a block-based (LRU) cache and (ii) a content-based (ARC) cache, both of configurable sizes and using write-through policy. Additionally, the physical host's disk or storage also needs to be simulated.

Fig. II.1 depicts the functioning of the custom simulator. As shown, a virtual disk request is mapped from the virtual address space to the host physical address space, and then looked-up

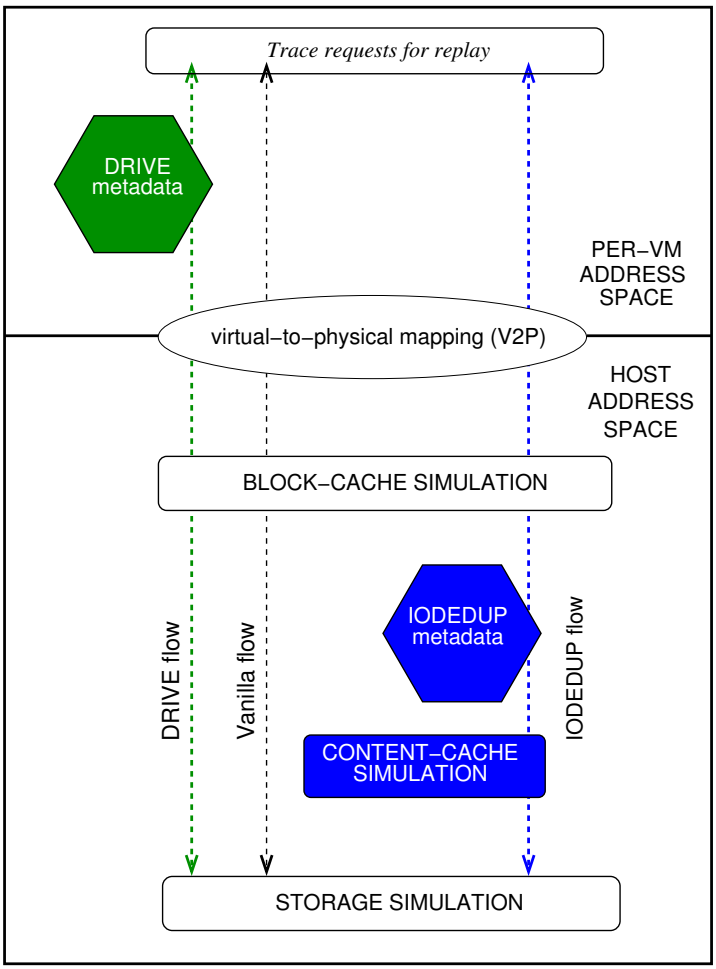


Fig. II.1: High-level functioning and requirements: *The simulator should perform simulation of I/O replay by simulation of a per-VM address space and a host address space. It should accept an input virtual-to-physical (V2P) mapping for these two address spaces. It should also simulate a block-cache as well as storage for Vanilla, IODEDUP and DRIVE invocations, and a content-cache especially for IODEDUP invocation.*

in host's block-cache. In case of Vanilla flow, the requests flow from the block-cache straight to the host storage. On the other hand, in case of IODEDUP flow, metadata is looked up for every request that is not satisfied by the block-cache, and content is searched in the content-cache, before hitting the host storage. In case of DRIVE flow, the requests are redirected within the virtual machine address space itself, so that the flow path within the physical machine address space remains unchanged (i.e., same as Vanilla path on physical machine).

The input to the simulator is a single trace file which contains multiple disk read/write requests. Each request in the trace should contain information regarding (i) whether it is a read or write request, (ii) block number to be read or written, (iii) data to be read or written, and its size. The output from the simulator involves the collection of metrics like number of cache-hits and cache-misses incurred over the execution of all requests in the trace file. Next, we present the high-level and low-level design descriptions for the custom simulator. Finally, we present details regarding the implemented input options and demonstrate the usage of the simulator.

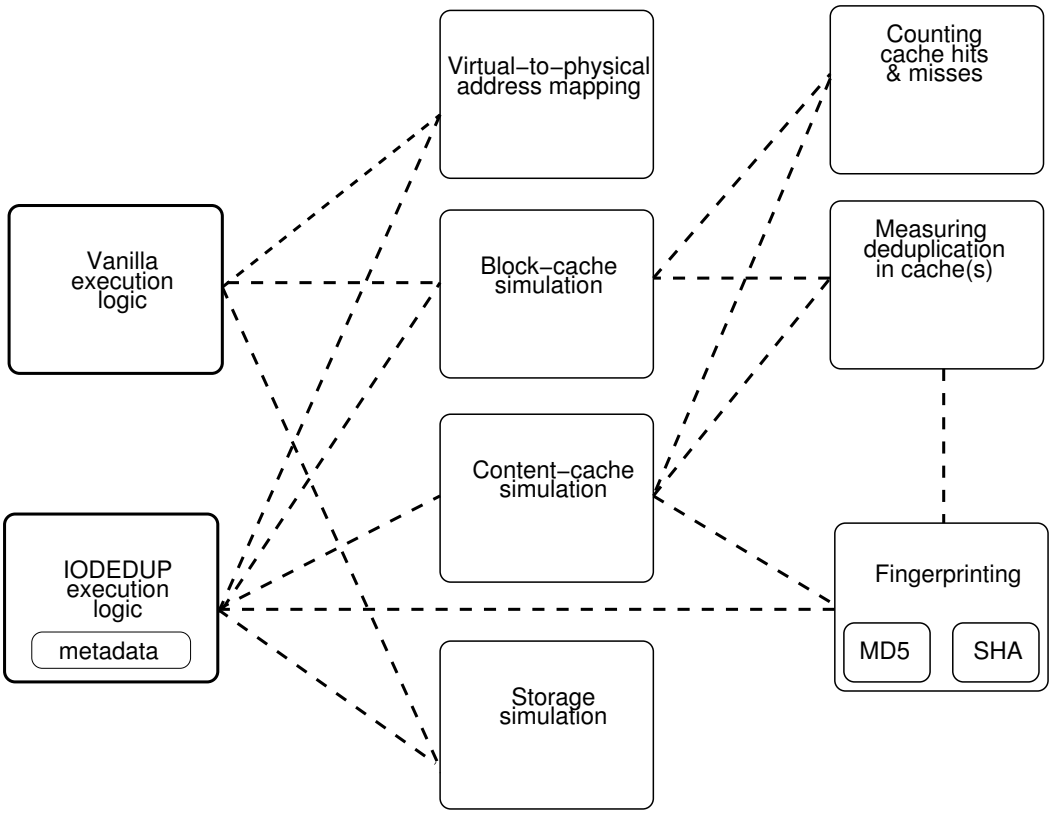


Fig. II.2: Modules of the simulator: *The IODEDUP execution logic needs to maintain deduplication metadata, and hence requires a content finger-printing module as well (eg. MD5, SHA). Additionally, modules to measure number of cache-hits & misses, as well as to measure content deduplication ratio achieved in total cache space, are shown.*

II.2 High-level design

Fig. II.2 depicts the different modules of the custom simulator, and the interaction among them. For each of the above modules, a brief description is given next.

II.2.1 Virtual-to-physical address mapping

The input Virtual-to-physical (V2P) address mapping indicates the range of physical blocks (i.e. blocks on host storage) that map to the address space of each VM (i.e. blocks of virtual disk) for simulation. This input applies to all the three invocations of the simulator—Vanilla, IODEDUP and DRIVE.

Assumptions. The assumptions made regarding the V2P map are the following: (i) All blocks belonging to a single virtual disk are contiguous blocks on the host storage, and (ii) Block requests are aligned at 4KB (i.e. 8 sectors) boundaries. The first assumption is a simplifying assumption for ease of implementation of the simulator, and discarding this assumption should not have any significant impact on the results presented in this thesis. The second assumption holds true if the starting sector number of the host storage partition is a multiple of eight (8),

which should be the case in efficiently configured storage systems. However, if this is not the case, the performance of the system/applications would be affected in general [179], and an administrative fix during configuration is recommended to avoid such inefficiency [180, 181].

```
root@PM5:~# fdisk -l

Disk /dev/sda: 500.1 GB, 500107862016 bytes
255 heads, 63 sectors/track, 60801 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x0003741c

   Device Boot      Start         End      Blocks   Id  System
/dev/sda1   *           1       30395   244140032   83  Linux
/dev/sda2           60055       60802     5995521    5  Extended
/dev/sda3           30395       60055   238247936   83  Linux
/dev/sda5           60055       60802     5995520   82  Linux swap

Partition table entries are not in disk order

Disk /dev/sdb: 1000.2 GB, 1000204886016 bytes
255 heads, 63 sectors/track, 121601 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Sector size (logical/physical): 512 bytes / 4096 bytes
I/O size (minimum/optimal): 4096 bytes / 4096 bytes
Disk identifier: 0x000aff6c

   Device Boot      Start         End      Blocks   Id  System
/dev/sdb1               1      121601   976760001   83  Linux
Partition 1 does not start on physical sector boundary.
```

Listing II.1: Output of command `fdisk -l` on test machine.

Example Partition Layout. The list of partitions and the starting sector of all partitions on a linux host can be viewed using the commands `df -h` and `fdisk -l`, respectively. The sample output from one of our test machines is shown below in Listing II.1. The listing II.1 shows the "Start" sector number, "End" sector number and the total number of "Blocks" for each partition, for each hard-disk present. It can be seen that none of the "Start" sector numbers are aligned at 4K boundaries. This is the case because the system was installed for test usage only, and not for performance-optimized usage. However, in production servers and cloud usage, it is expected that the disk would be formatted and partitioned correctly for optimized performance.

Effect of non-aligned block requests. The block-cache operates at 4KB page granularity, and if a block write request is received for a *whole* 4KB block, the block is directly written to cache without needing to first fetch it from disk. However, if the block write request is only for a *partial* block, the block needs to be first fetched from disk, and then the partial write is performed to it so that when the resulting buffer is flushed to disk, the unwritten portion of the block remains

unchanged. Given the above, if a block being written to the block-cache is not aligned (i.e. one block write request results in two partial block writes), two blocks would need to be first fetched from disk into cache, and then both would need to be partially over-written. Similarly, a single block read request would also necessitate the read of two blocks from disk. In our simulation, we have done away with such complexities by assuming that all block requests are block-aligned. As specified earlier, this can be achieved in a straight-forward manner by correct configuration of the physical storage.

Example V2P map input. Since all blocks of a single VM are assumed to be in a contiguous chunk on host storage, the representation of the resulting V2P map is simplified. Suppose there are three VMs, each having a virtual disk of 1000 blocks each. Thus, on the host storage, these three VMs can be assumed to be laid out such that the first VM's (VM1) blocks map to physical blocks 0-999, the second VM's (VM2) blocks map to physical blocks 1000-1999, and the third VM's (VM3) blocks map to physical blocks 2000-2999. In our simulator, the above information is input in terms of a 3-tuple per VM like <*vmname*, *capacity*, *base-address*>. In the 3-tuple representation, *capacity* refers to the number of blocks in the VM (i.e., 1000 in above example) and *base-address* refers to the starting address in the host storage (0, 1000, 2000 for each VM respectively, in above example). Note that the *base-address* can be any value, provided that the range of addresses for none of the VMs overlap with each other.

II.2.2 Block-cache & Content-cache simulation

The block-cache is simulated with the Least Recently Used (LRU) policy for cache eviction as well as the write-through policy for handling block writes. The size of the block-cache is configured by default to be 1GB, and can be tuned according to requirement using input options. The block-cache size specification applies to Vanilla, IODEDUP and DRIVE invocations of the simulator.

To prototype the IODEDUP system within the simulator, we implement a content-based cache with Adaptive Replacement Cache (ARC) policy. The size of the content-cache can be configured, and this configuration option is valid only if IODEDUP replay is requested. Also, since the content-cache basically occupies part of the space that is available for the block-cache, so the block-cache size is reduced accordingly. Thus, if IODEDUP replay is requested, the block-cache size specified earlier is assumed to include the content-cache's size as well. For example, with a block-cache size specification of 1 GB (i.e., 1024 MB) and a content-cache size specification of 100 MB, only 924 MB (i.e., 1024 *minus* 100) is used for simulating the block-cache while the remaining 100 MB forms the content-cache, as requested.

II.2.3 Storage simulation

When a block read request execution is to be simulated, the data of the block needs to be read from the disk. In our simulator, an actual disk read should not have been necessary since the traces already contain the read block content for every request. However, during our experience with the online traces at [81], we found that there appeared to be some missing trace records from the files, resulting in some inconsistencies. For example, a write request to block 1 with data A, was subsequently followed by a read request to the same block 1 (and hence should ideally contain same data A) with another data B instead of A. To address such inconsistency issues, we chose to simulate the storage as well, wherein we could explicitly track the content of each block that has been read or requested so far and hence catch these inconsistencies during replay. More details regarding inconsistencies and how the approach of simulated storage helped to address them, are presented in Section II.3.2.

II.2.4 Vanilla execution logic

As mentioned earlier, Vanilla invocation of the simulator implies that the I/O execution replay for the requests is performed as would happen in a standard linux host. Thus, for every write request, the content of the block is written into the cache, and flushed to (simulated) storage. Moreover, for every read request, the block address is looked up in the block-cache and returned, if cache hit. In case of a cache miss for a read request, the block is read from storage.

II.2.5 IODEDUP execution logic

The IODEDUP execution begins in a similar way as Vanilla, i.e. performs virtual-to-physical address mapping, and then block-cache lookup. However, the IODEDUP module intercepts the path between the block-cache and the storage, where it performs content-deduplication based on fingerprint comparison. Thus, implementation of the IODEDUP prototype requires a content-based cache and a fingerprinting mechanism as well. Additionally, IODEDUP maintains a meta-data store, containing information regarding the duplicates identified, which can be used to intercept block read requests and serve content directly from the content-based cache, if available.

II.2.6 DRIVE execution logic

The DRIVE execution logic in the physical address space (i.e. for redirected block) is the same as the Vanilla execution, although DRIVE performs block-level redirection within the virtual address space. Similar to IODEDUP, the DRIVE system also maintains a metadata store and requires a fingerprinting mechanism. The metadata store is held within the virtual disk driver in the virtual address space and metadata lookups are done to perform disk I/O redirection.

II.2.7 Measuring cache-hits & cache-deduplication

For all read & write requests, whether the block is found in cache or not, contributes to incrementing the cache-hit and cache-miss counters, respectively. Also, for every new block inserted in the cache, it is compared with existing blocks to measure the amount of deduplicated content in the cache. For Vanilla and DRIVE invocations, these measurements apply to block-cache only, whereas for IODEDUP invocation, these measurements are performed across both the block-cache and the content-cache.

Note that the "measurement of deduplication factor" is not to be confused with "performing deduplication"—the measurement of deduplication merely produces a number that determines the amount of unique content present in cache(s)—this may or may not be due to deduplication being performed. Thus, the measurement of deduplication can be done for Vanilla system also, though it does not actively perform any deduplication. For example, if a cache of capacity 10 blocks has 8 unique blocks and 2 blocks that are duplicates of the others, the cache can be said to have 8 *deduplicated* blocks and hence, its content deduplication factor evaluates to 80%.

II.3 Low-level design & implementation

Fig. II.3 depicts the lower-level design for each module specified in the earlier design diagram of Fig. II.2. In this section, we describe the details of implementation of each of these modules in our simulator.

II.3.1 Trace input & parsing

The trace files present at [81] has the following format for every request:-

```
[ts in ns] [pid] [process] [lba] [size in 512 Bytes blocks] [Write or Read] [major device number] [minor device number] [MD5 per 4096 Bytes]
```

The last column of the above format is in terms of the hexadecimal representation of the MD5 of the content. Hence, all above fields have content in deterministic range, and parsing of the trace file can be performed in a straight-forward line-by-line fashion.

The above trace files represent a single workload each—*webvm*, *homes* or *mail*, as mentioned earlier. However, our simulator is meant to simulate the replay of block requests of multiple VMs on a single host. Hence, we accommodate an additional attribute, a VM identifier, within each record in the trace. Accordingly, the data-structure representing a single request within the simulator is as shown in Listing II.2.

II.3.2 Addressing trace file inconsistencies by storage simulation

The trace files present at [81] have the content (or rather, MD5 hash representation of content) along-with every read and write request. This implies that a replay performed at a later date can

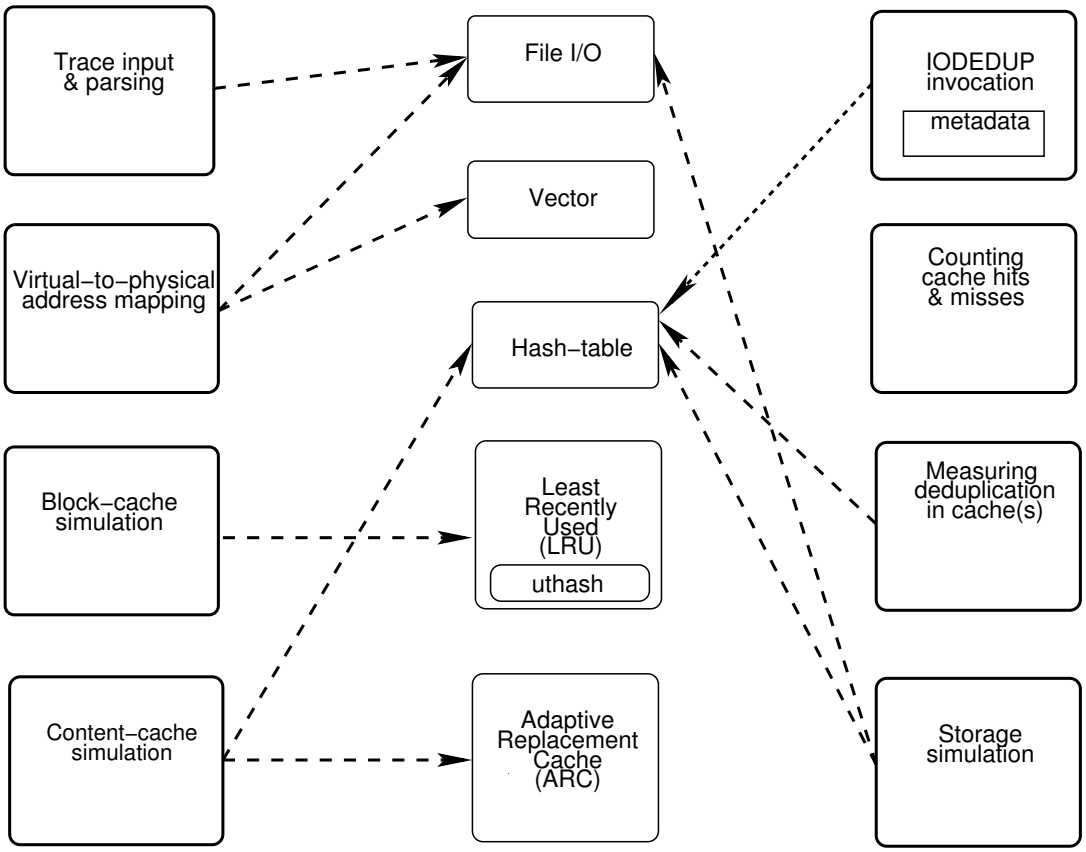


Fig. II.3: Low-level design of simulator: *The left-most and the right-most columns are the high-level components presented in Fig. II.2 and the center column contains the lower-level design components which underlie the higher-level design (as indicated by the unidirectional arrows).*

rely on these content for simulating storage reads and writes, without having to actually perform the I/O on storage. However, our experience with usage of these traces revealed that there were inconsistencies in the content reported in the traces.

We define trace *inconsistency* as the case where a piece of content was reportedly read from or written to a block, but the next read request to the same block reports another piece of content, although there were no writes to that block in the interim. This could be caused due to some missing records in the trace. However, to evaluate the correctness of any replay module, the traces themselves would have to be consistent. Hence, in our simulator, we maintain a simulated storage wherein the block content is written to a file, and can be read back when required. In this model, the first access to every block (whether read or write) is considered as the ground truth and written to the file (i.e. simulated storage). Naturally, subsequent write requests do not face any issues of consistency, since they are just new data to be written to storage. However, for every subsequent read request, the content present on simulated storage is used for replay (since it is the consistent version), instead of the content present in the trace (since it was found to be inconsistent at times).

The implementation of storage simulation in the custom simulator involves usage of hash-tables and file I/O, as indicated in Fig. II.3. The file I/O module is required because a file (re-

```
/* @vmname:  VM identifier
 * @time:    Time stamp when trace was emitted
 * @block:   VM I/O block identifier
 * @bytes:   Number of bytes transferred
 * @content: Content to be transferred
 * @rw:      Read (1) or write (0)
 */
struct vmreq_spec {
    char vmname[HOSTNAME_LEN];
    __u64 time;
    __u32 block;
    __u32 bytes;
    __u8 *content;
    unsigned char rw:1;
};
```

Listing II.2: Data-structure representing a single request.

ferred as *simdisk*) is used to store the content (or MD5 hash of content). Additionally, to enable "storage-lookup" using the block address, we use a hash-table with the block address (string formatted) as the key. Fig. II.4 presents a pictorial view of how storage simulation is implemented in our simulator. Within the hash-table entry that is identified for a given block, we maintain a file offset value which indicates the offset within the *simdisk* file, at which the corresponding content is stored.

II.3.3 Mapping from virtual disk space to physical disk space

As mentioned in Section II.2.1, the input V2P mapping indicates the range of physical blocks (i.e. blocks on host storage) that map to the address space of each VM (i.e. blocks of virtual disk) for simulation. For a trace replay file that has a single VM's requests, this input file is not mandatory, since the simulator can infer this map dynamically from the trace file itself. In fact, at the end of the simulator invocation, the inferred V2P map is output to the same file. Thus, with a single-VM trace file, it is possible to determine the range of blocks (i.e. maximum block number accessed) for that VM. However, in case of a trace input file that has aggregated requests from multiple VMs, this input file is mandatory and it is validated that the ranges of blocks specified for each VM does not overlap with the range of any other VM. In order to build this multi-VM input V2P file, the output V2P file(s) from the single-VM trace executions can be used additively, as indicated in Section II.2.1.

The input of V2P mapping requires the file I/O module (refer Fig. II.3) since the mapping has to be read from and written to a file. Also, after receiving the input mapping, the simulator uses a vector data-structure for storing the 3-tuples for each VM involved in replay. The *vector* data-structure is basically the same as an array, except that it can grow as required, unlike an array which has static size allocation.

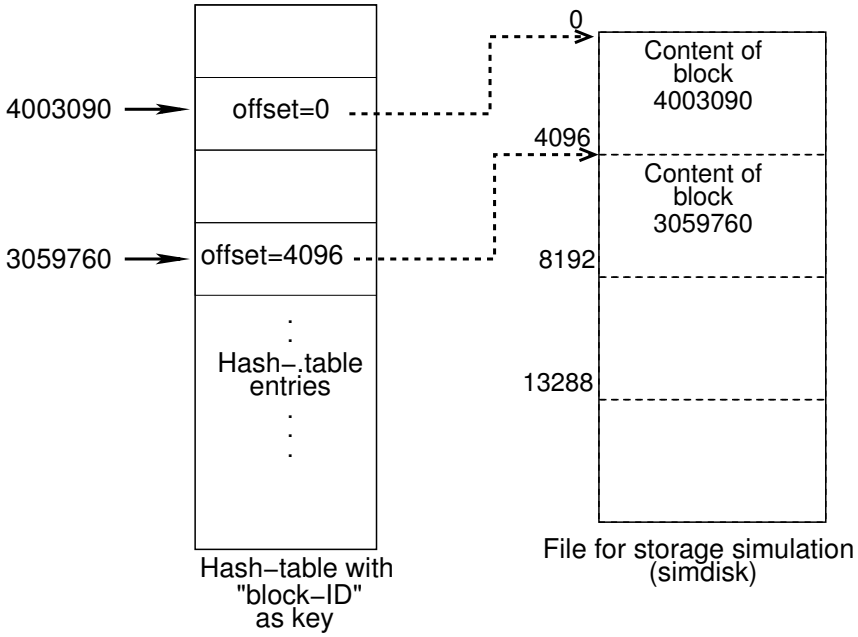


Fig. II.4: Implementation of storage simulation in custom simulator. Each hash-table entry contains an offset into the *simdisk* file.

II.3.4 Block-cache simulation

Since the block-cache needs to be looked-up by block ID, a hash-table implementation is required. Also, since the LRU policy has to be applied to the elements in the block-cache, all such elements should also form part of a linked list. Instead of implementing this cache from scratch, we used an existing hash-table implementation called `uthash`[182], which is basically a hash-table for storing C data-structures. It requires that a *UT_hash_handle* element be added to the data-structure that is to be stored into the hash-table and another field (say blockID) of the same data-structure can be used as the key to lookup the hash-table.

The `uthash` module also maintains a linked list of all the elements in the hash-table in "insertion" order. In order to simulate LRU policy for the block-cache, we have to tweak the ordering slightly, as follows. If a block is already present in the hash-table, inserting it again for the same key is an error in the `uthash` module. Modifying it in-place is not an error though, but it will not cause the LRU ordering that we require. So, when we find that the block being requested is in cache, we first delete it from the hash-table, and then re-insert it, to simulate LRU ordering.

II.3.5 IODEDUP metadata store

In our simulator, we have designed and implemented the IODEDUP metadata store in a similar fashion as the DRIVE metadata store except for two major differences: (i) The DRIVE metadata store is present within the virtual address space whereas the IODEDUP metadata store holds mapping within the physical storage address space, (ii) The DRIVE metadata store maintains

implicit caching hints to aid in request redirection whereas IODEDUP metadata store is used to retrieve the content hash which is further used for content-cache lookup.

II.3.6 Content-cache simulation

The content-cache is implemented using a hash-table implementation, and has the Adaptive Replacement Cache (ARC) as the cache replacement policy. The ARC policy basically maintains four lists—(i) a list of in-cache elements in the LRU order, (ii) a list of in-cache elements in the LFU order, (iii) a list of recently-evicted (ghost) elements in the LRU order, and (iv) a list of ghost elements in the LFU order. The ARC algorithm uses these four lists and maintains their relative sizes in order to most effectively keep the most recently used and the most frequently used elements in cache.

In our implementation, we use an online implementation of the ARC algorithm for the content-cache simulation. The content-cache can be looked up based on the hash of the content, and the insertion/deletion in cache is performed according to ARC policy mentioned above. For a read request, the content-cache needs to be looked up only upon a metadata hit in the IODEDUP system. In case of a metadata miss, the read request is forwarded to the storage instead. For a write request, the content-cache is written into and the metadata is updated accordingly. The content-cache is maintained with write-through semantics, as specified in [68].

II.3.7 Quantifying content-deduplication in cache(s)

The aim of this simulator is to study cache management effectiveness, and we quantify the content deduplication effected within the cache(s) (including the content-cache as well, for IODEDUP) as a measure of its effectiveness. In our simulator, we simulate traps at every block insertion and eviction from cache(s). A separate hash-table is maintained which keeps track of the duplicate content among the blocks currently present in the simulated cache(s), and this hash-table is modified at every such interrupt/trap occurrence.

In case of a block being inserted into the cache, in the trap that ensues, we fingerprint it and lookup the hash-table to see if there are any other blocks already in cache(s) with the same fingerprint. If so, its reference counter is incremented. When a block is evicted from cache, the trap processing again involves fingerprinting it and searching the hash-table. If the content is present in the hash-table, its reference counter is decremented, and the entry is removed from the hash-table if the reference counter reaches value of zero. At any instance during replay, the *content deduplication factor* can be defined as the ratio of number of unique blocks in cache(s) to number of total blocks in cache(s). Higher the content deduplication factor, higher the efficiency of the cache space utilization, and hence higher the storage access performance.

II.3.8 Verifying the correctness of the simulator

To ensure correctness of the simulator, each individual module was first unit-tested manually with small input sizes, and then the entire module as a whole was integration-tested with whole trace files. For every block that is present in cache (whether block-cache or content-cache), it is verified that the content present on the simulated storage is actually the same as the content returned from cache. Note that this is only a simulation-correctness check and does not count towards number of disk reads for replay. Additionally, for each invocation of the simulator, the following are verified in code (using assert statements).

- The total number of *reads* replayed equals the sum of *cache hits* and *cache misses*.
- The total number of *cache hits* equals the sum of *read cache hits* and *write cache hits*.
- The number of *read cache misses* equals the number of *disk reads*.
- The number of *read cache misses* equals the sum of *compulsory* and *capacity misses*.
- The total number of *writes* replayed equals the total number of blocks written to disk.

II.4 Input options and usage

Listing II.3 shows the basic usage options for the custom simulator, *simreplay*, in alphabetic order. Each option is explained below.

```
static char usage_str[] =                                                    \
        "\n"                                                                 \
        "\t[ -C          : --iodedup-logformat          ] Default: 0\n"      \
        "\t[ -d <dir>    : --input-directory=<dir>      ] Default: .\n"      \
        "\t[ -e          : --read-enable                ] Default: Off\n"    \
        "\t[ -E          : --write-enable               ] Default: Off\n"    \
        "\t[ -f <file>   : --input-file=<file>          ] Default: None\n"   \
        "\t[ -h          : --help                       ] Default: Off\n"    \
        "\t[ -m <file>   : --mapv2p=<file>              ] Default: None\n"   \
        "\t[ -O <in-MB>: --overall-RAM-size=<in-MB>  ] Default: 1024\n"     \
        "\t[ -o <in-MB>: --contentcache-size=<in-MB>] Default: 100\n"       \
        "\t[ -Q          : --Vanilla-replay             ] Default: On\n"     \
        "\t[ -R          : --DRIVE-replay               ] Default: Off\n"    \
        "\t[ -T          : --IODEDUP-replay             ] Default: Off\n"    \
        "\t[ -V          : --version                    ] Default: Off\n"    \
        "\n";
```

Listing II.3: Listing of simreplay usage

The option `iodedup-logformat` indicates that the content information present in each trace request is in terms of the MD5 hash of the content, as is the case with the traces *webvm*, *homes* and *mail* available online at [81]. This flag is especially useful for parsing of the input trace

file. When the flag is specified, the content field is expected to be 32 characters (i.e hexadecimal representation of 16-byte MD5 hash). On the other hand, if this flag is not specified, the original content is expected to be present in the trace. For example, if `iodedup-logformat` is not indicated, the size of the content is equal to the value in the `nbytes` field in the trace request data-structure.

The options `input-directory` and `input-file` are used to specify the input trace file to be used for simulated I/O replay. The file contains multiple read/write requests, each in the format described earlier. Parsing of the file is performed by the "Trace input & parsing module" as discussed in Section II.3.1.

The options `read-enable` and `write-enable` indicate whether read requests or write requests should be replayed, respectively. If `read-enable` is not specified, read requests replay will not be simulated and if `write-enable` is not specified, write requests will not be simulated. Note that, at least one of these two options has to be specified otherwise there is no simulation to be done. By default, both options are disabled, however a typical invocation of simulator would enable both options so that all read and write requests present in the trace file would be replayed/simulated.

The option `mapv2p` can be used to specify the input V2P mapping indicating the range of physical blocks that map to the address space of each VM, as described earlier. If the trace file has only a single VM's requests, the input V2P mapping is optional, but is mandatory otherwise.

The option `overall-RAM-size` indicates the space in number of megabytes (MB) to be simulated as the block-cache in the Vanilla system. By default, the value is 1024 MB, i.e. 1GB block-cache. This option is also applicable to the IODEDUP system, but in conjunction with the value specified using the option `contentcache-size`, as follows. The option `contentcache-size` specifies what portion of the `overall-RAM-size` should be simulated as a content-cache, and defaults to 100 MB. This option is valid only for IODEDUP replay, and re-adjusts the configured size of the block-cache to `overall-RAM-size` minus `contentcache-size`. It follows that, the value of `contentcache-size` must be less than or equal to `overall-RAM-size`, for successful invocation of the simulator.

```
$ cat stats-O1024-webvm-iodedup-sreplay_rw.txt
#READ=3116456, #WRITE=11177702
RAM-size = 1024 (MB)
buffer cache: hits=12303451, misses=1990707, readhits=1471112, writehits
    =10832339
disk hits=12823046
disk hits read=1645344, writes=11177702
```

Listing II.4: Sample output of `SimReplay` for Vanilla invocation.

The options `Vanilla-replay` and `IODEDUP-replay` indicate that the simulator is to be invoked to simulate Vanilla execution and IODEDUP execution, respectively. Listings II.4 and II.5 show sample outputs of the simulator when used for Vanilla and IODEDUP invocations,

```
$ cat stats-O1024-o100-webvm-iodedup-ioreplay_rw.txt
io-redirections: self-is-leader=813572, self-is-not-leader=1409885
read-responses: compulsory-misses=316045, cache-hits=704111, capacity-
    misses=2096300
metadata-hit-conversions: deduphits=127156, selfhits=1, dedupmisses
    =1282729, selfmisses=813571
#READ=3116456, #WRITE=11177702
RAM-size = 1024 (MB)
CCACHE-size = 100 (MB)
buffer cache: hits=11185632, misses=3108526, readhits=576954, writehits
    =10608678
content metadata: hits=2223457, misses=316045 dirties=0
content cache: hits=127157, misses=8035550
content cache: dedup hits=127156, nondedup hits=1
disk hits=13590047
disk hits read=2412345, writes=11177702
```

Listing II.5: Sample output of `SimReplay` for IODEDUP invocation.

```
cat stats-O1024-webvm-iodedup-freplay_rw.txt
io-redirections: self-is-leader=1754745, self-is-not-leader=1045666
read-responses: compulsory-misses=316045, cache-hits=2722411, capacity-
    misses=78000
metadata-hit-conversions: deduphits=1033328, selfhits=1689083, dedupmisses
    =12338, selfmisses=65662
#READ=3116456, #WRITE=11177702
RAM-size = 1024 (MB)
confided metadata: hits=2800411, misses=316045, mapmisscachehits=0, dirties
    =0, mapdirtycachehits=0
fcollisions=0, fcollisionstp=3141002, fzeros=0
buffer cache: hits=13635026, misses=659132, readhits=2722411, writehits
    =10912615
disk hits=11571747
disk hits read=394045, writes=11177702
```

Listing II.6: Sample output of `SimReplay` for DRIVE invocation.

respectively. Similarly, the option `DRIVE-replay` indicates that the simulator is to be invoked to simulate DRIVE execution. A sample output of DRIVE invocation of the simulator is shown in Listing II.6.

The sample output of Vanilla invocation in Listing II.4 has information regarding the number of read and write requests in the trace, the total cache (RAM) size used, the number of hits and misses in the buffer (block) cache, and a classification of the hits into reads hits or write hits as well. Finally, it also reports the number of requests that went to disk—in case of reads, this is the number of disk reads that happened, and in case of writes, this is the total number of writes since we assume that all writes are flushed to disk eventually.

The sample output of IODEDUP and DRIVE invocation also have similar fields as above, however they have some additional fields too, to track the efficiency achieved by each system. For example, Listing II.5 presents information regarding the configured size of content-cache, the number of metadata hits incurred, and how many of those were for reads and writes, each. It also presents information regarding how many of the metadata hits eventually resulting in content-cache hits and misses. Additionally, for all of the content-cache hits, it lists how many were hits for duplicate content and how many were just regular hits.

The `help` option shows the usage listing for the `SimReplay` tool, and the `version` option can be used to identify different versions of the custom simulator, for future extension.

Appendix III

Trace logging & collection toolkit : `preadwritedump`

In this section, we describe the block request tracing tools that we have developed as part of our effort to capture traces for evaluation. Initially, we contacted the authors of [68] regarding the block request tracing tools developed by them. They were kind enough to allow us access to the source code of the kernel module, called `collect.ko`, which could generate traces in a *similar* format as described at [81] (though not exactly the same format as at [81]). However, the output of `collect.ko` was in terms of the MD5 hash of the content, whereas we wished to dump the entire content as-is, so that it could be used with other fingerprinting mechanisms as well, example Rabin hashing [89]. Moreover, the `collect.ko` kernel module used the mode of *relay channels*[183] to write trace records into the `debugfs`[184] filesystem, but the userspace tools to siphon the traces from the `debugfs` filesystem into persistent storage were missing.

The above issues motivated us to build our own toolkit which included kernel modules to dump content into `debugfs` filesystem, as well as a generic userspace tool that runs parallelly and collects the `debugfs` filesystem traces into output files. We also incorporated the functionality of communication across the kernel module and userspace tool such that removal of the kernel module would signal the userspace tool and wait for a handshake before exiting gracefully. Moreover, since the content to be dumped in the traces was binary in nature and could consist of any arbitrary characters, parsing of the output traces became a significantly non-trivial task compared to the parsing of finite-sized records, as done for traces at [81]. To enable correct parsing of every record, we devised our own trace record format as well. In the rest of this section, we first provide a brief background of the basic features used in the toolkit implementation, and then describe details of the various tools implemented.

III.1 Background for implementing tracing toolkit

In this sub-section, we present a brief description of basic features like `relayfs` channels, `debugfs` file system, `kprobes` and *kernel signaling*, which were used for developing the block tracing toolkit.

III.1.1 Relayfs channels

The Relayfs[183] filesystem is designed to enable large amounts or sustained data delivery from kernel modules to user space. Basically, it is a "channel" that routes data from the kernel module into a set of per-CPU buffers that are operated essentially like a file. The Relayfs API consists of two sets of API—the first set for in-kernel clients, and the second set for userspace tools to dump out the data into persistent storage. We used both sets of APIs to build our toolkit—the first set to build our content-dumping kernel module(s), and the second set to build a generic userspace tool that can be used with any kernel module that uses the in-kernel relayfs APIs. Note that, the relay channels work in tandem with the `debugfs` filesystem, as explained next.

III.1.2 Debugfs filesystem

Although relay channels route the data from in-kernel to outwards, the userspace tools can access the data only if the buffers are exposed to userspace like a filesystem. This is where the `debugfs`[184] filesystem comes into play. When the in-kernel Relayfs API called `relay_open()` is invoked, it results in abstract files being created within the Debugfs filesystem, per CPU. These files can be accessed via `poll()`, `open()` and `mmap()` from userspace tools.

III.1.3 Kprobes

Kprobes[185] form the basic essence of the block request tracing kernel modules. Kprobes provide a minimally-invasive way to dynamically trap into any kernel routine, and collect debugging information non-disruptively. Thus, we can trap any kernel address, and specify a handler to be executed before the original execution begins. There are three types of kernel probes: (i) `kprobes`—can be inserted on *any* instruction in the kernel, (ii) `jprobes`—can be inserted at kernel function entry, and provides convenient access to function's arguments, (iii) `kretprobes`—inserted to be fired at function exit. In our kernel modules, we use `jprobes` at the entry of the kernel function `generic_make_request` to gain access to its function arguments, of type `struct bio`. More details regarding usage of `jprobes` in tracing tool implementation, are presented in the next section.

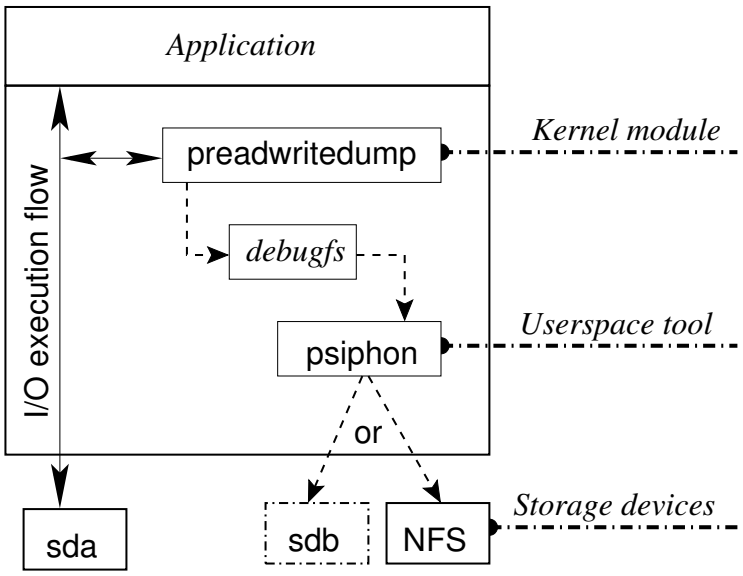


Fig. III.1: Toolkit usage: *Figure shows the data flow for the trace collection, from the kernel module to* `debugfs` *and then to either NFS or another local storage device.*

III.1.4 Signaling between kernel and user space

An example instantiation of how to "Send signals from kernel to user space" is presented at [186]. We borrow the source code and tweak it to fit it into our kernel-module and the userspace log-siphoning program. The basic idea is that the kernel can signal a userspace process only if its PID is known. Hence, the userspace process is required to let its own PID be known/accessible to the kernel, by writing it into a pre-determined location, such as a file in the `debugfs` filesystem described above. Note that, from within a kernel module, it is unsafe/disallowed to open() or close() regular files. However, creating, opening and closing files in the `debugfs` filesystem is allowed, as mentioned above. Additionally, the kernel can register handlers to specify the functionality to be performed upon read() or write() of the file created in `debugfs` filesystem. Thus, by using the `debugsfs`, any userspace process's PID can be conveyed to the kernel-module, so that signals may be sent to that PID from kernel space.

III.2 Tools developed for tracing: `pdatadump`, `preadwritedump`, `psiphon`

There are primarily three tools in our trace generation toolkit, as listed below. The first two are both kernel modules, with a slight difference in the `jprobe` handlers used in each, whereas the third one is a userspace tool.

1. `pdatadump`: For every block write request, this kernel module dumps the written content as well as other identifying information like sector address, major number, minor number,

etc. For every read request, only the identifying information is output, and the data content itself is not dumped.

2. `preadwritedump`: Similar to `pdatadump`, with the only difference that data content is dumped for read requests as well, in addition to write requests.

3. `psiphon`: This is the userspace tool that can siphon logs written in the `debugfs` files, into regular files. High-speed transfer is achieved using `mmap` of the regular files. This is a generic tool, in the sense that it can be used to transfer logs from `debugfs` filesystem into the regular filesystem for any files, provided the filenames are provided as input. Thus, it can be used in tandem with any kernel module that dumps output into the `debugfs` filesystem.

Use-ability of the toolkit. The `pdatadump`, `preadwritedump` and the `psiphon` tools can be used within any linux machine (both physical or virtual), and can be used to trace the block read and write requests for any specified device (namely, `/dev/sda`, `/dev/sdb`, etc) or even for a specific hard-disk partition (namely, `/dev/sda1`, `/dev/sda2` and so on).

Measures to avoid the problem of *recursive tracing*. When the `psiphon` tool transfers the trace data from `debugfs` filesystem into regular files, if the destination file is on the same disk/-partition as the one being traced, it can result in "recursive tracing"—traces that are being output themselves getting traced. To avoid such a scenario, we prefer the usage of a separate network-attached (i.e. remote storage, eg. NFS) storage as the destination file system for `psiphon` output files. This option is built into the installation script for the tracing toolkit. Although it is possible to use local storage also for log collection, it should be ensured that the specified partitions/devices for tracing and log collection are distinct, to avoid the "recursive tracing" issue.

Fig. III.1 shows ways in which the tracing toolkit can be used in both physical and virtual machines. Physical machines may potentially have multiple hard-disks, so choosing one for tracing and another for log collection is a possibility. However, typical virtual machines are created with a single partition, by default. In this case, creating an NFS partition to be used for log collection is a straight-forward choice.

III.3 Implementation details

In this sub-section, we present details of implementation of the kernel modules (`pdatadump`, and `preadwritedump`) as well as the userspace tool (`psiphon`).

III.3.1 Trace file format to enable correct parsing

The output trace files generated by this toolkit are to be eventually used for replay in the *simreplay* simulator, described earlier. Thus, all the elements of the previous trace format were retained in

our format as well, albeit in a different ordering. Especially, the entire binary dump of the content per request was done, instead of dumping only the MD5 hash of the content. However, the most important difference introduced is that because of the arbitrary nature of binary characters, it was no longer feasible to simply use any ASCII character as a delimiting character across the elements of each trace.

Usually, a delimiting character is picked such that it is itself not part of any of the fields within the record. Since the content dump could be arbitrary (i.e., we can not control it), we could no longer rely on a delimiting character to help parse/differentiate distinct elements of a single trace file. The workaround identified is to indicate the beginning of each record using a special struct, namely `struct trace_event_element`. The definition of this struct is presented below in Listing III.1.

```
struct trace_event_element
{
    __u32 magic;
    __u32 elt_len;  /* length of data in next trace element */
};
```

Listing III.1: Data-structure indicating the beginning of a record in trace files.

As is shown, struct `trace_event_element` consists of two entities: (i) *magic*, and (ii) *elt_len*. The *magic* value is a fixed value defined within the module, such that if the defined value does not match the value in the trace file, a parsing error can be declared. The second entity is basically a value that reports the number of bytes following this structure, to form the rest of the current record. Thus, based on the value of `elt_len`, those many bytes can be read from the trace file, and considered as a single record. Within every single record, the binary content field is always the last field and hence, the delimiter for all of the previous fields is still retained as a blank space. Consequently, after the second-to-last field has been parsed, it can be safely assumed that remaining buffer content is itself the binary content dumped as part of the current trace record.

III.3.2 Using relay channels with debugfs to dump output logs

As mentioned in the previous section, we use relay channels and debugfs to dump output logs from the kernel module, and use another userspace module to siphon the logs from debugfs into persistent storage repository. Using the `relaychan_init()` call, we create relay channels per CPU with name *pioevents*, such that a file named *pioevents.i* is created within `debugfs` for each CPU core numbered *i*. For example, if the VM has two vCPUs, two files named *pioevents.0* and *pioevents.1* would be created. This is done during kernel module initialization, so that immediately after initialization, the relay channels would be ready for use.

Once the kernel module is in use, it will keep intercepting disk read and write requests and dumping relevant output into the relay channels. Note that, depending on which CPU is pro-

cessing the disk request, the corresponding relay channel file will be written into. For example, events processed by CPU-0 will cause output to be written into *pioevents.0* file whereas those processed by CPU-1 will have output written into *pioevents.1* file, and so on. Thus, our entire trace would be split up into multiple trace files when dumped via the relaychannel subsystem. Because we eventually want to obtain a single trace file at a later time, we also print a timestamp for each event that is logged. After the tracing is complete and the logging has been terminated, we use the timestamps to merge all the output files and get an integrated timestamp-ordered output trace file.

III.3.3 Difference between `pdatadump` and `preadwritedump`

As mentioned earlier, the `pdatadump` module dumps data content only for write requests, and not for read requests. This is because, it also has another kernel thread running parallelly that performs a "scan" of the entire disk and outputs per-block content into another relay channel. Thus, for all read requests, the corresponding content is expected to be gather separately by the "scanning thread" of the `pdatadump` module. Consequently, the `pdatadump` module operates two relay channels, while the `preadwritedump` module operates just one.

In `pdatadump`, since content needs to be dumped only for write requests, and not read requests, only a single-point probe/intervention is sufficient to gather the requisite information, i.e. the `jprobe` at `generic_make_request`. This is because at this probe point, the I/O requests are just being queued for execution, and at this time, the read buffers are empty while the write buffers contain data. The read buffers are populated only after execution, in the return path, at which time, the write buffers have already been consumed (and potentially even reused elsewhere). Thus, using a single probe, either the read content or the write content can be dumped, not both. Hence, the `pdatadump` module makes the choice of dumping content only for the write requests, while simultaneously "scanning" all the blocks to gather remaining content.

The advantage of the approach in `pdatadump` module is that, it simplifies the design of the kernel module itself. However, the disadvantage of this approach is that, usage of the collected traces for replay becomes slightly unwieldy than if the read request content had also been present along-with. More specifically, the absence of data content along-with the read request implies that simulated disk "creation" had to be performed using the "scanning" traces, in advance of performing each simulated replay invocation. Moreover, the "scanning" traces could be huge due to large size storage capacity, even if the actual content being read may only be a fraction of the total size [68]. With this in mind, we soon graduated to the newer implementation of the kernel module, i.e. the `preadwritedump` module.

As mentioned above, dumping the data content only for write requests enabled the implementation of the kernel module to be relatively simple. However, since our new requirement is to dump the data content for read requests as well, we adopted a similar approach as was done in the `collect` kernel module (this was the incomplete source code received from authors of [68],

mentioned earlier). The idea is that for a read request that has been queued up to be associated with the corresponding completed read request, we need to do some stacking and un-stacking of function pointers. This requires in-kernel manipulation of I/O buffers of type `struct bio`, as explained next.

III.3.4 Using jprobes to capture disk read and write requests

In both `pdatadump` and `preadwritedump`, read and write requests are intercepted by using a `jprobe` of the following form, shown in Listing III.2.

```
static struct jprobe jp_make_request = {
    .kp.addr = (kprobe_opcode_t *) generic_make_request,
    .entry = (kprobe_opcode_t *) my_make_request
};
```

Listing III.2: The jprobe defined within the kernel module, to intercept disk read and write requests

```
struct old_bio_data_prwd {
  void          *bi_private;     //to stow away the original struct bio
  bio_end_io_t  *bi_end_io;      //pointer to original bi_end_io()
  sector_t      bi_sector;       //starting sector of data being requested
  unsigned      bi_size;         //size of the data being requested
  dev_t         bd_dev;        //device of data being requested
  unsigned      major;         //major number of device
  unsigned      minor;         //minor number of device
  unsigned      pid;           //process id of the current process
  char          processname[PNAME_LEN]; //process name of above process
};
```

Listing III.3: The structure for `oldbio`, similar to original `struct bio` of the Linux kernel

During kernel module initialization, the above jprobe is registered using `register_jprobe()` so that for every disk read and write request, the call to `my_make_request` would be invoked, hence giving us access to its `struct bio` arguments. Thus, `my_make_request()` is our jprobe handler, and for a write request, it dumps the write content as well as identifying information into a relay channel. However, the behaviour of the `jprobe` handler when encountering a read request, is slightly modified. Instead of immediately dumping to a relay channel, the handler simply notes the identifying information in a newly-defined `struct oldbio` which is quite similar to the original `struct bio` (refer Listing III.3). Additionally, the actual dumping of content and identifying information for a read request is delegated to a separate function call, namely `bio_end_prwd()`.

Originally, the I/O completion routine is the `bi_end_io` call w which is invoked for each `bio` that is ready for consumption. However, to associate the "queue" and "completion" events for the various read requests, the original function pointer for `bi_end_io` is stowed away into

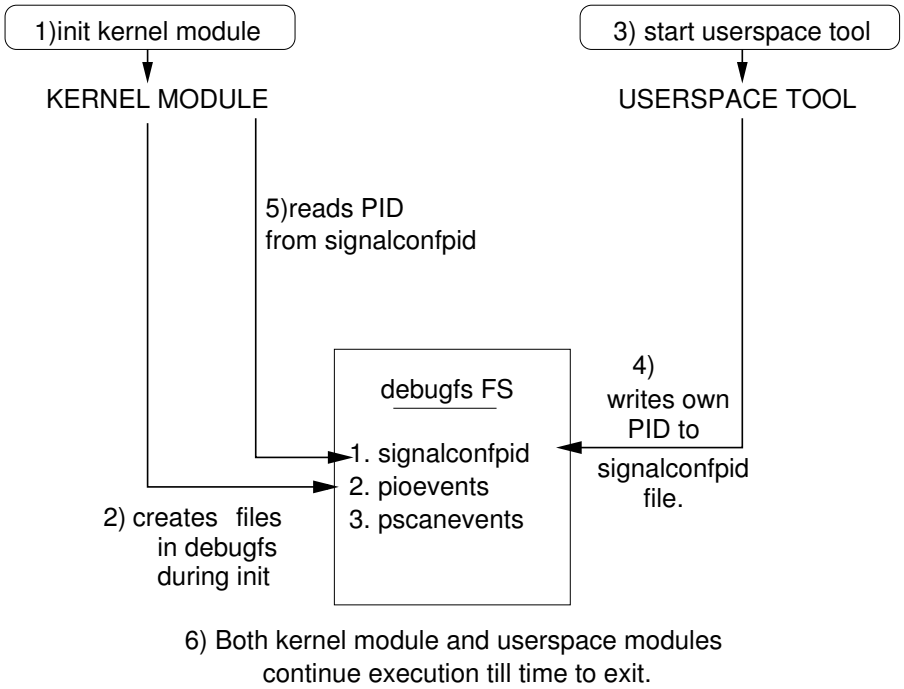


Fig. III.2: Exit handshake setup: *Setting up for the exit handshake between the kernel module and the userspace process involves writing of self PID into a* `debugfs` *file by the userspace process. The other two files listed here are the* `debugfs` *output files of the kernel module.*

`struct oldbio`, whereas the function pointer for `bio_end_prwd()` is "stacked" into the original `struct bio` in its place. This causes our new routine `bio_end_prwd()` to be invoked first during read I/O completion, at which time, the dumping of read content happens. At the end of the `bio_end_prwd()` routine, we invoke the original `bi_end_io` so that read I/O execution can complete as usual.

III.3.5 Exit handshake between kernel module and userspace process

Fig. III.2 shows the setup for performing the signaling from the kernel to the userspace process. As mentioned earlier, this requires that the kernel module is aware of the PID of the userspace process. As shown in the figure, when the kernel module is init-ed, it creates a file named `signalconfpid` in the `debugfs` filesystem, in addition to other files for relay channels. After the kernel module has been installed, the userspace tool is instantiated and it determines its own process ID (PID) to write to the `signalconfpid` file. When the kernel module detects that a PID has been written, it notes the value in local memory for future use. After this initial exchange of PID, both the kernel module and the userspace tool continue their trace dumping & collection process, until the kernel module requires to be un-installed. The actual handshake to achieve graceful exit is performed when the kernel module is un-installed, as explained next.

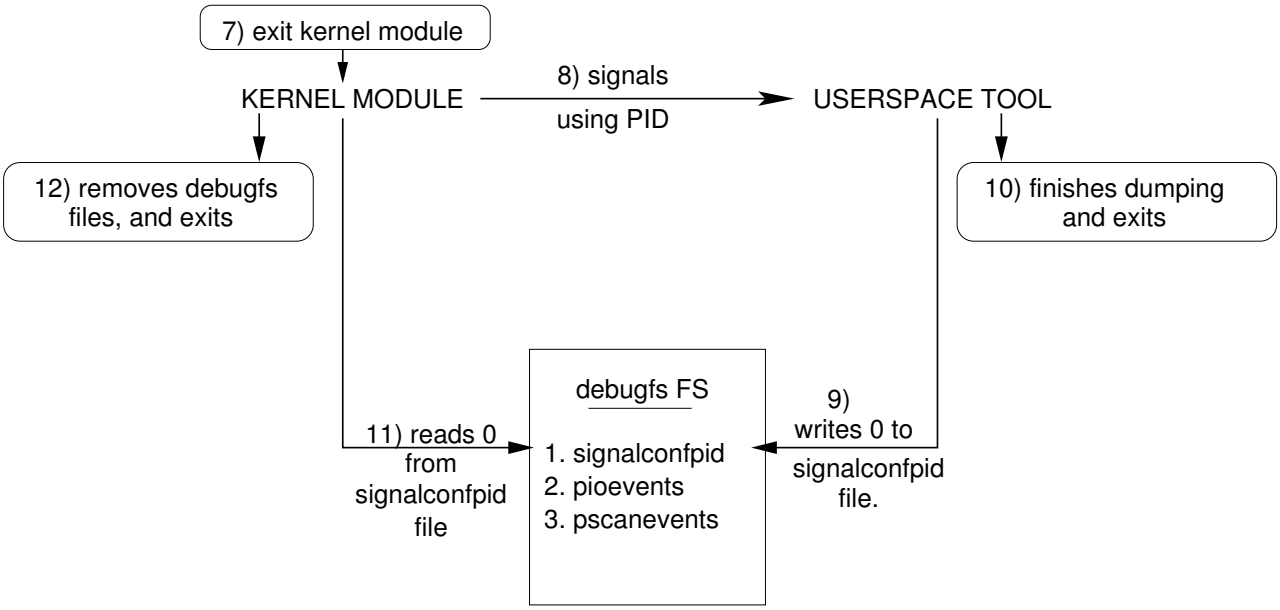


Fig. III.3: *Execution of the exit handshake between the kernel module and the userspace process.*

III.3.6 Executing the exit handshake

Signaling is used so that after the kernel stops dumping output to the relay channels, it can notify the userspace process. An exit handshake is accomplished which ensures that the data being logged has been captured in its entirety. Fig. III.3 presents the handshake performed between the two modules, before the kernel module exit. Upon receiving the signal from the kernel module, the userspace process finishes transferring any remaining data from the `debugfs` file system to the regular files, and exits. However, before exiting, the userspace program indicates its exit to the kernel module by over-writing the previous PID value with zero. Upon receiving a value of zero (0), the kernel module can safely exit.

Bibliography

[1] The AWS Community. Amazon Elastic Compute Cloud (Amazon EC2). Website. http://aws.amazon.com/ec2/. Accessed Apr 24, 2015.

[2] Ubuntu Community. Bootstack—Your Managed Cloud. Website. http://www.ubuntu.com/cloud/managed-cloud. Accessed May 11, 2015.

[3] Ubuntu Community. Cloud tools. Website. http://www.ubuntu.com/cloud/tools. Accessed May 11, 2015.

[4] Timothy Wood, Ludmila Cherkasova, Kivanc Ozonat, and Prashant Shenoy. Profiling and Modeling Resource Usage of Virtualized Applications. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, pages 366–387, New York, NY, USA, 2008. Springer-Verlag New York, Inc.

[5] Chris Hyser, Bret Mckee, Rob Gardner, and Brian J. Watson. Autonomic Virtual Machine Placement in the Data Center. Technical report, HPL, 2007.

[6] Hien Nguyen Van and Frederic Dang Tran and Jean-Marc Menaud. Autonomic Virtual Resource Management for Service Hosting Platforms. *ICSE Workshop on Software Engineering Challenges of Cloud Computing*, 0:1–8, 2009.

[7] Timothy Wood, Prashant Shenoy, and Arun. Black-box and Gray-box Strategies for Virtual Machine Migration. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design & Implementation*, pages 229–242, 2007.

[8] Peter Radkov, Li Yin, Pawan Goyal, Prasenjit Sarkar, and Prashant Shenoy. A Performance Comparison of NFS and iSCSI for IP-Networked Storage. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 101–114. USENIX Association, 2004.

[9] N. Bobroff, A. Kochut, and K. Beaty. Dynamic Placement of Virtual Machines for Managing SLA Violations. In *Proceedings of the 10th IFIP/IEEE International Symposium on Integrated Network Management*, pages 119–128, 2007.

[10] Daniel Gmach, Jerry Rolia, Ludmila Cherkasova, and Alfons Kemper. Capacity Management and Demand Prediction for Next Generation Data Centers. In *Proceedings of the International Conference on Web Services*, pages 43–50, 2007.

[11] Fabien Hermenier, Xavier Lorca, Jean-Marc Menaud, Gilles Muller, and Julia 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.

[12] Deshi Ye, Hua Chen, and Qinming He. Load Balancing in Server Consolidation. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing with Applications*, pages 170–174, Aug 2009.

[13] S. Dutta, S. Gera, Akshat Verma, and B. Viswanathan. SmartScale: Automatic Application Scaling in Enterprise Clouds. In *Proceedings of the 5th IEEE International Conference on Cloud Computing (CLOUD)*, pages 221–228, June 2012.

[14] Martin Bichler, Thomas Setzer, and Benjamin Speitkamp. Capacity Planning for Virtualized Servers. *Workshop on Information Technologies and Systems (WITS), Milwaukee, Wisconsin, USA, 2006*, 2007.

[15] Victor Pankratius. *Emerging Research Directions in Computer Science : Contributions from the Young Informatics Faculty in Karlsruhe*. KIT Scientific Publishing, 2010.

[16] Shubham Agrawal, Sumit Kumar Bose, and Srikanth Sundarrajan. Grouping Genetic Algorithm for Solving the Server Consolidation Problem with Conflicts. In *Proceedings of the first ACM/SIGEVO Summit on Genetic and Evolutionary Computation*, pages 1–8. ACM, New York, USA, 2009.

[17] Rohit Gupta, Sumit Kumar Bose, Srikanth Sundarrajan, Manogna Chebiyam, and Anirban Chakrabarti. A Two Stage Heuristic Algorithm for Solving the Server Consolidation Problem with Item-Item and Bin-Item Incompatibility Constraints. *International Conference on Services Computing*, 2:39–46, 2008.

[18] Padala Padala, Xiaoyun Zhu, Zhikui Wang, Sharad Singhal, and Kang G. Shin. Performance Evaluation of Virtualization Technologies for Server Consolidation. Technical report, HPL, 2007.

[19] Varsha Apte, Purushottam Kulkarni, Sujesha Sudevalayam, and Piyush Masrani. Balancing Response Time and CPU allocation in Virtualized Data Centers using Optimal Controllers. Technical report, IIT Bombay, 2010. http://www.cse.iitb.ac.in/internal/techreports/reports/TR-CSE-2010-29.pdf.

[20] Miche Baker-Harvey. Google Compute Engine uses Live Migration technology to service infrastructure without application downtime. Online. http://googlecloudplatform.blogspot.com/2015/03/Google-Compute-Engine-uses-Live-Migration-technology-to-service-infrastructur html. Accessed May 1, 2015.

[21] Ludmila Cherkasova and Rob Gardner. Measuring CPU Overhead for I/O Processing in the Xen Virtual Machine Monitor. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATC)*, pages 24–24, Berkeley, CA, USA, 2005. USENIX Association.

[22] Theodore M. Wong and John Wilkes. My Cache or Yours? Making Storage More Exclusive. In *Proceedings of the General Track of the USENIX Annual Technical Conference (ATC)*, pages 161–175, 2002.

[23] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM symposium on Operating Systems Principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[24] Ian Pratt. Xen and the Art of Virtualization Revisited. Online. https://www.usenix.org/legacy/events/nsdi08/tech/pratt.pdf. Accessed May 1, 2015.

[25] Microsoft. Microsoft Azure. Online. http://azure.microsoft.com. Accessed May 1, 2015.

[26] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A View of Cloud Computing. *Communications of the ACM*, 53(4):50–58, April 2010.

[27] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal. Dynamic Provisioning of Multi-tier Internet Applications. In *Proceedings of the Second International Conference on Autonomic Computing (ICAC)*, pages 217–228, June 2005.

[28] Diwaker Gupta, Ludmila Cherkasova, Rob Gardner, and Amin Vahdat. Enforcing Performance Isolation Across Virtual Machines in Xen. In *Proceedings of the ACM/IFIP/USENIX International Conference on Middleware*, Middleware '06, pages 342–362, 2006.

[29] Xiaoqiao Meng, Canturk Isci, Jeffrey Kephart, Li Zhang, Eric Bouillet, and Dimitrios Pendarakis. Efficient resource provisioning in compute clouds via vm multiplexing. In *Proceedings of the 7th International Conference on Autonomic Computing*, ICAC '10, pages 11–20, New York, NY, USA, 2010. ACM.

[30] Understanding Full Virtualization, Paravirtualization and Hardware Assist. VMware Whitepaper, November 2007. http://www.vmware.com/files/pdf/VMware_paravirtualization.pdf. Accessed Apr 24, 2015.

[31] The Xen Community. Xen. Website. http://www.xen.org/. Accessed Apr 24, 2015.

[32] Jeanna Neefe Matthews, Wenjin Hu, Madhujith Hapuarachchi, Todd Deshane, Demetrios Dimatos, Gary Hamilton, and Michael McCabe. Quantifying the Performance Isolation Properties of Virtualization Systems. In *Proceedings of the Workshop on Experimental computer science (ECS)*, pages 5–5, Berkeley, CA, USA, 2007. USENIX Association.

[33] Kernel Based Virtual machine (KVM). http://www.linux-kvm.org/page/Main_Page. Accessed Apr 24, 2015.

[34] Wikipedia. Full Virtualization. Online, 2007. http://en.wikipedia.org/wiki/Full_virtualization. Accessed Jan 5, 2015.

[35] Parallels. OpenVZ. Online. http://wiki.openvz.org/Main_Page. Accessed Jan 5, 2015.

[36] Qumranet Inc. KVM: Kernel-based Virtualization Driver. Technical report, Qumranet, 2006. www.linuxinsight.com/files/kvm_whitepaper.pdf. Accessed Apr 24, 2011.

[37] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the USENIX Annual Technical Conference*, pages 41–41, 2005.

[38] B. Anwer, N. Feamster, A. Nayak, and L. Liu. Network I/O Fairness in Virtual Machines. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Virtualized Infrastructure Systems and Architectures*, pages 73–80, 2010.

[39] Rusty Russell. Virtio: Towards a De-facto Standard for Virtual I/O Devices. *SIGOPS Operating Systems Review*, 42(5):95–103, 2008.

[40] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner. *Internet Small Computer Systems Interface (iSCSI), RFC 3720*, 2004.

[41] A. J. Lewis. LVM HOWTO. Online. http://www.tldp.org/HOWTO/LVM-HOWTO/. Accessed May 8, 2015.

[42] Bhuvan Urgaonkar, Prashant Shenoy, and Timothy Roscoe. Resource Overbooking and Application Profiling in Shared Hosting Platforms. *SIGOPS Operating Systems Review*, 36(SI):239–254, 2002.

[43] J.C. Faugre, B. Folliot, and B.C. Saab. Execution Platform for High Consuming Parallel Applications: A Case Study for Grbner Basis. International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications, 1999.

[44] Gueyoung Jung, Kaustubh R. Joshi, Matti A. Hiltunen, Richard D. Schlichting, and Calton Pu. A Cost-sensitive Adaptation Engine for Server Consolidation of Multitier Applications. In *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '09, pages 9:1–9:20, New York, NY, USA, 2009. Springer-Verlag New York, Inc.

[45] Jason Sonnek and Abhishek Chandra. Virtual Putty: Reshaping the Physical Footprint of Virtual Machines. In *Proceedings of the Workshop on Hot Topics in Cloud Computing*, June 2009.

[46] J. Sonnek, J. Greensky, R. Reutiman, and A. Chandra. Starling: Minimizing Communication Overhead in Virtualized Computing Platforms Using Decentralized Affinity-Aware Migration. In *Proceedings of the 39th International Conference on Parallel Processing (ICPP)*, pages 228–237, Sept 2010.

[47] Aravind Menon, Jose Renato Santos, Yoshio Turner, G. (John) Janakiraman, and Willy Zwaenepoel. Diagnosing Performance Overheads in the Xen Virtual Machine Environment. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments (VEE)*, pages 13–23, 2005.

[48] David Chisnall. *The Definitive Guide to the Xen Hypervisor*. Prentice Hall, 2008.

[49] Olivier Withoff. *Understanding XenServer Networking - The Linux Perspective*. Citrix, 2008.

[50] Christian Benvenuti. *Understanding Linux Network Internals*. O'Reilly, 2005.

[51] Sebastien Godard. Sysstat. Website. http://freshmeat.net/projects/sysstat/. Accessed Jan 5, 2015.

[52] Xen Developer Community. Xentop. Tool in Xen. http://linux.die.net/man/1/xentop. Accessed Apr 24, 2015.

[53] Linux Developer Community. Linux / Unix Command: top. Online. http://linux.about.com/od/commands/l/blcmdl1_top.htm. Accessed Apr 24, 2015.

[54] Rusty Russell. iptables—Linux man page. Online. http://linux.die.net/man/8/iptables. Accessed Mar 3, 2015.

[55] N. R. Draper and H. Smith. *Applied Regression Analysis*. J. Wiley & Sons, 1998.

[56] M Maechler. robustbase: Basic Robust Statistics, [Online]. http://cran.r-project.org/web/packages/robustbase/index.html. Accessed Nov 9, 2009.

[57] Sujesha Sudevalayam and Purushottam Kulkarni. Affinity-aware Modeling of CPU Usage for Provisioning Virtualized Applications. Technical report, Indian Institute of Technology Bombay, 2011.

[58] C. Amza, E. Cecchet, A. Chanda, A. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel. Specification and Implementation of Dynamic Web Site Benchmarks. Proceedings of the 5th Workshop on Workload Characterization, 2001.

[59] Muffin proxy 0.9.3a, Muffin world wide web filtering system., [Online]. http://muffin.doit.org/.

[60] L. Hu, K. Schwan, A. Gulati, J. Zhang, and C. Wang. Net-cohort: Detecting and Managing VM Ensembles in Virtualized Data Centers. In *Proceedings of the 9th International Conference on Autonomic Computing*, pages 3–12, 2012.

[61] Deepal Jayasinghe, Simon Malkowski, Qingyang Wang, and Jack Li et al. Variations in Performance and Scalability when Migrating n-Tier Applications to Different Clouds. In *Proceedings of the Fourth International Conference on Cloud Computing*, 2011.

[62] Muhammad Aufeef and Muhammad Ali Babar. Migrating Service Oriented System to Cloud Computing: An Experience Report. In *Proceedings of the 4th IEEE International Conference on Cloud Computing (CLOUD)*, 2011.

[63] Xin Meng, Jingwei Shi, Xiaowei Liu, Huifeng Liu, and Lian Wang. Legacy Application Migration to Cloud. In *Proceedings of the Fourth International Conference on Cloud Computing*, 2011.

[64] Keren Jin and Ethan L. Miller. The Effectiveness of Deduplication on Virtual Machine Disk Images. In *Proceedings of the Israeli Experimental Systems Conference SYSTOR*, pages 7:1–7:12, 2009.

[65] K. R. Jayaram, Chunyi Peng, Zhe Zhang, Minkyong Kim, Han Chen, and Hui Lei. An Empirical Analysis of Similarity in Virtual Machine Images. In *Proceedings of the Middleware Industry Track Workshop*, pages 6:1–6:6, 2011.

[66] Chunyi Peng, Minkyong Kim, Zhe Zhang, and Hui Lei. VDN: Virtual Machine Image Distribution Network for Cloud Data Centers. In *Proceedings of INFOCOM*, pages 181–189, 2012.

[67] Sean Barker, Timothy Wood, Prashant Shenoy, and Ramesh Sitaraman. An Empirical Study of Memory Sharing in Virtual Machines. In *Proceedings of the USENIX Conference on Annual Technical Conference (ATC)*, pages 25–25, 2012.

[68] Ricardo Koller and Raju Rangaswami. I/O Deduplication: Utilizing Content Similarity to Improve I/O Performance. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 211–224, 2010.

[69] Sergey Brin and Lawrence Page. The Anatomy of a Large-scale Hypertextual Web Search Engine. In *Proceedings of the 7th International Conference on World Wide Web (WWW)*, pages 107–117, 1998.

[70] Benjamin Zhu, Kai Li, and Hugo Patterson. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, pages 18:1–18:14, 2008.

[71] Robert Love. Linux Kernel Development Second Edition. http://www.makelinux.net/books/lkd2/ch15. Accessed Jan 5, 2015.

[72] Grzegorz Miłós, Derek G. Murray, Steven Hand, and Michael A. Fetterman. Satori: Enlightened Page Sharing. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, 2009.

[73] Timothy Wood, Gabriel Tarasuk-Levin, Prashant Shenoy, Peter Desnoyers, Emmanuel Cecchet, and Mark D. Corner. Memory Buddies: Exploiting Page Sharing for Smart Colocation in Virtualized Data Centers. In *Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, pages 31–40, 2009.

[74] Nimrod Megiddo and Dharmendra S. Modha. Outperforming LRU with an Adaptive Replacement Cache Algorithm. *Computer*, 37(4):58–65, 2004.

[75] Zhiming Shen, Zhe Zhang, Andrzej Kochut, Alexei Karve, Han Chen, Minkyong Kim, Hui Lei, and Nicholas Fuller. VMAR: Optimizing I/O Performance and Resource Utilization in the Cloud. In *Proceedings of the ACM/IFIP/USENIX International Middleware Conference*, pages 183–203, 2013.

[76] Min 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 *Proceedings of the 20th International Symposium on Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 209–218, 2012.

[77] Mohammad Shamma, Dutch T. Meyer, Jake Wires, Maria Ivanova, Norman C. Hutchinson, and Andrew Warfield. Capo: Recapitulating Storage for Virtual Desktops. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST)*, pages 3–3, 2011.

[78] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. *SIGOPS Operating Systems Review*, 36(SI):181–194, 2002.

[79] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat. Difference Engine: Harnessing Memory Redundancy in Virtual Machines. In *Proceedings of the 8th USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 309–322, December 2008.

[80] Prateek Sharma and Purushottam Kulkarni. Singleton: System-wide Page Deduplication in Virtual Environments. In *Proceedings of the 21st International symposium on High-Performance Parallel and Distributed Computing (HPDC)*, pages 15–26, 2012.

[81] Ricardo Koller and Raju Rangaswamy. Trace: I/O Deduplication: Utilizing Content Similarity to Improve I/O Performance. Website. http://sylab-srv.cs.fiu.edu/doku.php?id=projects:iodedup:start. Accessed Apr 24, 2011.

[82] R. Rivest. *The MD5 Message-Digest Algorithm*, 1992.

[83] D. Eastlake, 3rd and P. Jones. *US Secure Hash Algorithm 1 (SHA1)*, 2001.

[84] Shay Gueron, Simon Johnson, and Jesse Walker. SHA-512/256. In *Proceedings of the 8th International Conference on Information Technology: New Generations*, pages 354–358, 2011.

[85] Kiran Srinivasan, Tim Bisson, Garth Goodson, and Kaladhar Voruganti. iDedup: Latency-aware, Inline Data Deduplication for Primary Storage. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*, pages 299–312, 2012.

[86] Athicha Muthitacharoen, Benjie Chen, and David Mazières. LBFS: A Low-bandwidth Network File System. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 174–187, 2001.

[87] Sean Quinlan and Sean Dorward. Venti: A New Approach to Archival Data Storage. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST)*, 2002.

[88] Val Henson. An Analysis of Compare-by-hash. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, pages 13–18, 2003.

[89] Michael O. Rabin. Fingerprinting by Random Polynomials. Technical report, Center for Research in Computing Technology, Harvard University, 1981.

[90] Gustavo Duarte. What your computer does while you wait. Online. http://duartes.org/gustavo/blog/post/what-your-computer-does-while-you-wait. Accessed Apr 24, 2015.

[91] Jim Gray and Preshant Shenoy. Rules of Thumb in Data Engineering. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 3–. IEEE Computer Society, 2000.

[92] Nimrod Megiddo and Dharmendra S. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST)*, pages 115–130, 2003.

[93] wikipedia. Adaptive replacement cache. Online. http://en.wikipedia.org/wiki/Adaptive_replacement_cache. Accessed Apr 24, 2015.

[94] Austin T. Clements, Irfan Ahmad, Murali Vilayannur, and Jinyuan Li. Decentralized Deduplication in SAN Cluster File Systems. In *Proceedings of the Conference on USENIX Annual Technical Conference (ATC)*, pages 101–114, 2009.

[95] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. The Zettabyte File System. Technical report, University of California, Santa Cruz, 2003.

[96] Charles P. Wright and Erez Zadok. Postmark. Website. http://www.filesystems.org/docs/auto-pilot/Postmark.html. Accessed Apr 24, 2015.

[97] Dimitris. Beware of benchmarking storage that does inline compression. Website. http://recoverymonkey.org/2013/02/25/beware-of-benchmarking-storage-that-does-inline-compression/. Accessed May 11, 2015.

[98] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *Proceedings of the 26th IEEE International Conference on Data Engineering Workshops (ICDEW)*, pages 41–51, March 2010.

[99] Ruijin Zhou and Ming Liu and Tao Li. Characterizing the Efficiency of Data Deduplication for Big Data Storage Management. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pages 98–108, 2013.

[100] Vasily Tarasov, Amar Mudrankit, Will Buik, Philip Shilane, Geoff Kuenning, and Erez Zadok. Generating Realistic Datasets for Deduplication Analysis. In *Proceedings of the USENIX Conference on Annual Technical Conference*, pages 24–24, 2012.

[101] SNIA IOTTA TWG. Storage Networking Industry Association's Input/Output Traces, Tools, and Analysis repository. Online. http://iotta.snia.org/traces. Accessed Mar 3, 2015.

[102] HP Labs. Open Source software at tesla.hpl.hp.com. Online. {http://tesla.hpl.hp.com/opensource/}.AccessedJan5,2015.

[103] UMass Laboratory for Advanced Software Systems. UMass Trace Repository. Online. http://traces.cs.umass.edu/. Accessed Jan 5, 2015.

[104] Peter Danzig, Jeff Mogul, Vern Paxson, and Mike Schwartz. Internet Traffic Archive. Online. http://www.sigcomm.org/ITA/. Accessed Jan 5, 2015.

[105] Pink Sheep Industries. Some VMware Images. Online. http://www.thoughtpolice.co.uk/vmware/. Accessed Jan 5, 2015.

[106] V. Tarasov, S. Kumar, J. Ma, D. Hildebrand, A. Povzner, G. Kuenning, and E. Zadok. Extracting Flexible, Replayable Models from Large Block Traces. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*, FAST'12, pages 22–22. USENIX Association, 2012.

[107] Eric Anderson. Capture, Conversion, and Analysis of an Intense NFS Workload. In *Proceedings of the 7th Conference on File and Storage Technologies (FAST)*, FAST '09, pages 139–152. USENIX Association, 2009.

[108] S. Kavalanekar, B. Worthington, Qi Zhang, and V. Sharda. Characterization of Storage Workload Traces from Production Windows Servers. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pages 119–128. IEEE, 2008.

[109] Nitin Agrawal, William J. Bolosky, John R. Douceur, and Jacob R. Lorch. A Five-year Study of File-system Metadata. *ACM Transactions on Storage (TOS)*, 3(3), 2007.

[110] Akshat Aranya, Charles P. Wright, and Erez Zadok. Tracefs: A File System to Trace Them All. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, FAST '04, pages 129–145. USENIX Association, 2004.

[111] S. Gurumurthi, S. Sankar, and M.R. Stan. Using Intradisk Parallelism to Build Energy-Efficient Storage Systems. *IEEE Micro*, 29(1):50–61, 2009.

[112] Wen Xia, Hong Jiang, Dan Feng, Lei Tian, Min Fu, and Zhongtao Wang. P-Dedupe: Exploiting Parallelism in Data Deduplication System. In *Proceedings of the 7th IEEE International Conference on Networking, Architecture and Storage (NAS)*, pages 338–347. IEEE, June 2012.

[113] Wen Xia, Hong Jiang, Dan Feng, Lei Tian, Min Fu, and Yukun Zhou. Ddelta: A Deduplication-inspired Fast Delta Compression Approach. *Performance Evaluation*, 79(0):258 – 272, 2014. Special Issue: Performance 2014.

[114] Sean Quinlan. Plan 9 File system traces. Online. http://www.cs.bell-labs.com/who/seanq/p9trace.html. Accessed Jan 5, 2015.

[115] HP Labs. animation-bear dataset. Online. http://apotheca.hpl.hp.com/pub/datasets/animation-bear/. Accessed Jan 5, 2015.

[116] Inc. Linux Kernel Organization. The Linux Kernel Archives. Website. https://www.kernel.org/. Accessed Mar 3, 2015.

[117] GCC team. GCC Releases. Website. https://www.gnu.org/software/gcc/releases.html. Accessed Apr 23, 2015.

[118] Erik Riedel, Mahesh Kallahalla, and Ram Swaminathan. A Framework for Evaluating Storage System Security. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, Berkeley, CA, USA, 2002. USENIX Association.

[119] Vern Paxson and Sally Floyd. Wide Area Traffic: The Failure of Poisson Modeling. *IEEE/ACM Transactions Networks*, 3(3):226–244, June 1995.

[120] Martin F. Arlitt and Carey L. Williamson. Web Server Workload Characterization: The Search for Invariants. In *Proceedings of the 1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '96, pages 126–137, New York, NY, USA, 1996. ACM.

[121] Ahmed El-Shimi, Ran Kalach, Ankit Kumar, Adi Oltean, Jin Li, and Sudipta Sengupta. Primary Data Deduplication-large Scale Study and System Design. In *Proceedings of the USENIX Conference on Annual Technical Conference (ATC)*, pages 26–26. USENIX Association, 2012.

[122] Grant Wallace, Fred Douglis, Hangwei Qian, Philip Shilane, Stephen Smaldone, Mark Chamness, and Windsor Hsu. Characteristics of Backup Workloads in Production Systems. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, FAST'12, pages 4–4. USENIX Association, 2012.

[123] Benjamin Zhu, Kai Li, and Hugo Patterson. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 18:1–18:14. USENIX Association, 2008.

[124] Andrew W. Leung, Shankar Pasupathy, Garth Goodson, and Ethan L. Miller. Measurement and Analysis of Large-scale Network File System Workloads. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 213–226. USENIX Association, 2008.

[125] Calicrates Policroniades and Ian Pratt. Alternatives for Detecting Redundancy in Storage Systems Data. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 6–6. USENIX Association, 2004.

[126] Sriram Sankar and Kushagra Vaid. Storage Characterization for Unstructured Data in Online Services Applications. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 148–157. IEEE Computer Society, 2009.

[127] Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson. A Comparison of File System Workloads. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 4–4. USENIX Association, 2000.

[128] Fei Xie, Michael Condict, and Sandip Shete. Estimating Duplication by Content-based Sampling. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 181–186. USENIX Association, 2013.

[129] A.B. Downey. The structural cause of file size distributions. In *Proceedings of the IEEE/ACM Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 361–370, 2001.

[130] Martin Arlitt, Rich Friedrich, and Tai Jin. Workload Characterization of a Web Proxy in a Cable Modem Environment. *SIGMETRICS Performance Evaluation Review*, 27(2):25–36, 1999.

[131] A. Feldmann, A. C. Gilbert, W. Willinger, and T. G. Kurtz. The Changing Nature of Network Traffic: Scaling Phenomena. *SIGCOMM Computer Communications Review*, 28(2):5–29, 1998.

[132] Paul Barford, Azer Bestavros, Adam Bradley, and Mark Crovella. Changes in Web Client Access Patterns: Characteristics and Caching Implications. Technical report, Boston University, 1998.

[133] Keren Jin and Ethan L. Miller. The Effectiveness of Deduplication on Virtual Machine Disk Images. In *Proceedings of the The Israeli Experimental Systems Conference*, SYSTOR '09, pages 7:1–7:12. ACM, 2009.

[134] Fanglu Guo and Petros Efstathopoulos. Building a High-performance Deduplication System. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'11, pages 25–25, Berkeley, CA, USA, 2011. USENIX Association.

[135] Stephen Smaldone, Grant Wallace, and Windsor Hsu. Efficiently Storing Virtual Machine Backups. In *Proceedings of the 5th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'13, pages 10–10, Berkeley, CA, USA, 2013. USENIX Association.

[136] Bing Zhou and Jiangtao Wen. Hysteresis Re-chunking Based Metadata Harnessing Deduplication of Disk Images. In *Proceedings of the 2013 42nd International Conference on Parallel Processing*, ICPP '13, pages 389–398, Washington, DC, USA, 2013. IEEE Computer Society.

[137] Wei Zhang, Tao Yang, Gautham Narayanasamy, and Hong Tang. Low-cost Data Deduplication for Virtual Machine Backup in Cloud Storage. In *Proceedings of the 5th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'13, pages 12–12, Berkeley, CA, USA, 2013. USENIX Association.

[138] Giridhar Appaji Nag Yasa and P. C. Nagesh. Space Savings and Design Considerations in Variable Length Deduplication. *SIGOPS Operating Systems Review*, 46(3):57–64, 2012.

[139] Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Virtualization Aware File Systems: Getting Beyond the Limitations of Virtual Disks. In *Proceedings of the Conference on Networked Systems Design & Implementation (NSDI)*, pages 26–26. USENIX Association, 2006.

[140] Jim Mauro and Spencer Shepler and Vasily Tarasov. Filebench File System Benchmark. Online. http://sourceforge.net/projects/filebench/. Accessed Jan 5, 2015.

[141] Andrew Tridgell and Ronnie Sahlberg. DBENCH I/O storm. Website. https://dbench.samba.org/. Accessed Apr 24, 2015.

[142] William D. Norcott and Don Capps. Iozone Filesystem Benchmark. Website. http://www.iozone.org/. Accessed Apr 24, 2015.

[143] Intel Corporation and Open Source Development Lab. Iometer. Website. http://www.iometer.org/. Accessed Apr 24, 2015.

[144] axboe@kernel.dk. fio. Website. http://freshmeat.net/projects/fio/. Accessed Apr 24, 2015.

[145] EPFL PARSA. CloudSuite. Online. http://parsa.epfl.ch/cloudsuite/cloudsuite.html. Accessed Apr 23, 2015.

[146] Transaction Processing Performance Council. TPC Benchmarks. Online. http://www.tpc.org/information/benchmarks.asp. Accessed Apr 23, 2015.

[147] SPEC. Standard Performance Evaluation Corporation (SPEC). Online. {http://www.spec.org/index.html}.AccessedApr23,2015.

[148] NASA Advanced Supercomputing Division. NAS Parallel Benchmarks. Online. {http://www.nas.nasa.gov/publications/npb.html}.AccessedApr23,2015.

[149] DARPA HPCS. HPC Challenge Benchmark. Online. {http://icl.cs.utk.edu/hpcc/}.AccessedApr23,2015.

[150] Princeton University. PARSEC Benchmark. Online. {http://parsec.cs.princeton.edu/}.AccessedApr23,2015.

[151] Erez Zadok. The Berkeley Automounter Suite of Utilities. Website. http://www.am-utils.org/. Accessed Apr 23, 2015.

[152] Inc. Free Software Foundation. GNU Emacs. Website. https://www.gnu.org/software/emacs/. Accessed Apr 23, 2015.

[153] N. Park, W. Xiao, K. Choi, and D. Lilja. A Statistical Evaluation of the Impact of Parameter Selection on Storage System Benchmarks. In *Proceedings of the 7th IEEE International Workshop on Storage Network Architecture and Parallel I/O (SNAPI)*, 2011.

[154] J. Paulo, P. Reis, J. Pereira, and A. Sousa. DEDISbench: A Benchmark for Deduplicated Storage Systems. In *On the Move to Meaningful Internet Systems: OTM 2012*, volume 7566 of *Lecture Notes in Computer Science*, pages 584–601. Springer Berlin Heidelberg, 2012.

[155] Russell Coker. Bonnie++. Website. http://www.coker.com.au/bonnie++/. Accessed May 11, 2015.

[156] Jeff Chase and Darrell Anderson. Fstress. Website. http://www.cs.duke.edu/ari/fstress/. Accessed May 11, 2015.

[157] Vasily Tarasov, Saumitra Bhanage, Erez Zadok, and Margo Seltzer. Benchmarking File System Benchmarking: It *IS* Rocket Science. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 9–9. USENIX Association, 2011.

[158] Kimberly Keeton, Alistair Veitch, Doug Obal, and John Wilkes. I/O Characterization of Commercial Workloads. In *Proceedings of the 3rd Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW)*, 2009.

[159] Zachary Kurmas, Jeremy Zito, Lucas Trevino, and Ryan Lush. Generating a Jump Distance Based Synthetic Disk Access Pattern, 2006.

[160] Christina Delimitrou, Sriram Sankar, Kushagra Vaid, and Christos Kozyrakis. Accurate Modeling and Generation of Storage I/O for Datacenter Workloads, 2011.

[161] C. Delimitrou, S. Sankar, K. Vaid, and C. Kozyrakis. Storage I/O Generation and Replay for Datacenter Applications. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 123–124, April 2011.

[162] Tim Gibson, Attn J, Camp Smith Hi, Ethan L. Miller, Darrell D. E. Long, and Jack Baskin School. Long-term File Activity and Inter-Reference Patterns, 1998.

[163] Vasily Tarasov, Dean Hildebrand, Geoff Kuenning, and Erez Zadok. Virtual Machine Workloads: The Case for New Benchmarks for NAS. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, FAST'13, pages 307–320. USENIX Association, 2013.

[164] Christina Delimitrou, Sriram Sankar, Aman Kansal, and Christos Kozyrakis. ECHO: Recreating Network Traffic Maps for Datacenters with Tens of Thousands of Servers. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '12, pages 14–24. IEEE Computer Society, 2012.

[165] Christina Delimitrou, Sriram Sankar, Kushagra Vaid, and Christos Kozyrakis. Decoupling Datacenter Studies from Access to Large-scale Applications: A Modeling Approach for Storage Workloads. In *Proceedings of the 2011 IEEE International Symposium on Workload Characterization*, IISWC '11, pages 51–60. IEEE Computer Society, 2011.

[166] Z. Kurmas, K. Keeton, and K. Mackenzie. Synthesizing Representative I/O Workloads Using Iterative Distillation. In *Proceedings of the 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems (MASCOTS)*, pages 6–15, Oct 2003.

[167] Nitin Agrawal, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Generating Realistic Impressions for File-system Benchmarking. In *Proceedings of the 7th Conference on File and Storage Technologies*, FAST '09, pages 125–138. USENIX Association, 2009.

[168] Peter J. Denning. The Working Set Model for Program Behavior. *Communications of the ACM*, 11(5):323–333, May 1968.

[169] Piyush Shivam, Varun Marupadi, Jeff Chase, Thileepan Subramaniam, and Shivnath Babu. Cutting Corners: Workbench Automation for Server Benchmarking. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2008.

[170] NLANR/DAST. Iperf. Website. http://sourceforge.net/projects/iperf/. Accessed Apr 24, 2015.

[171] David Mosberger and Tai Jin. Httperf— A Tool for Measuring Web Server Performance. *SIGMETRICS Performance Evaluation Review*, 26(3):31–37, 1998.

[172] Marco Peereboom. Iogen. Website. http://linux.softpedia.com/get/Programming/Quality-Assurance-and-Testing/iogen-7894.shtml. Accessed Apr 24, 2015.

[173] Rugg Developer Group. Rugg: Flexible file system and hard drive crash testing. Website. http://rugg.sourceforge.net/. Accessed Apr 24, 2015.

[174] Larry McVoy and Carl Staelin. LMbench - Tools for Performance Analysis. Website. http://www.bitmover.com/lmbench/. Accessed Apr 24, 2015.

[175] Henk Vandenbergh. vdbench. Website. http://sourceforge.net/projects/vdbench/. Accessed Apr 24, 2015.

[176] Devin Carraway. Lookbusy. Website. http://www.devin.com/lookbusy/. Accessed Apr 24, 2015.

[177] Masanori Itoh. Load. Website. http://sourceforge.net/projects/kusanagi/files/load/. Accessed Apr 24, 2015.

[178] Amos Waterland. Stress. Website. http://weather.ou.edu/~apw/projects/stress/. Accessed Apr 24, 2015.

[179] Richard W. M. Jones. Ubuntu manuals virt-alignment-scan. Website. http://manpages.ubuntu.com/manpages/precise/man1/virt-alignment-scan.1.html. Accessed Apr 23, 2015.

[180] Abhinav Joshi, Eric Forgette, Peter Learmonth, and Jon Benedict. Best Practices for File System Alignment in Virtual Environments. Technical report, NetApp, January 2011. http://www.netapp.com/us/media/tr-3747.pdf. Accessed Apr 23, 2015.

[181] Peter Brouwer. Aligning Partitions to Maximize Storage Performance. Technical report, Oracle, November 2012. http://www.oracle.com/technetwork/server-storage/sun-unified-storage/documentation/partitionalign-111512-1875560.pdf. Accessed Apr 23, 2015.

[182] Troy D. Hanson. uthash: A Hash-table for C Structures. Online. https://troydhanson.github.io/uthash/. Accessed Jan 5, 2015.

[183] Michel Dagenais and Richard Moore and Bob Wisniewski and Karim Yaghmour and Tom Zanussi. Relayfs—A High-speed Data Relay Filesystem. Online. http://relayfs.sourceforge.net/relayfs.txt. Accessed Jan 5, 2015.

[184] Jonathan Corbet. Debugfs filesystem. Online. https://www.kernel.org/doc/Documentation/filesystems/debugfs.txt. Accessed Apr 23, 2015.

[185] Jim Keniston and Prasanna S Panchamukhi and Masami Hiramatsu. Kernel Probes (Kprobes). Online. https://www.kernel.org/doc/Documentation/kprobes.txt. Accessed Mar 3, 2015.

[186] Ariane Keller. Sending Signals from the Kernel to the User Space. Online. http://people.ee.ethz.ch/~arkeller/linux/kernel_user_space_howto.html#s6. Accessed Jan 5, 2015.

Publications List

Conference Proceedings

1. *DRIVE: Using Implicit Caching Hints to achieve Disk I/O Reduction in Virtualized Environments*. Proceedings of the 21st International Conference on High Performance Computing (HiPC), 2014. Sujesha Sudevalayam, Purushottam Kulkarni, Rahul Balani and Akshat Verma.

2. *Affinity-aware Modeling of CPU Usage for Provisioning Virtualized Applications*. Proceedings of the 4th International Conference on Cloud Computing (CLOUD), 2011. Sujesha Sudevalayam and Purushottam Kulkarni.

Journal Publications

1. *Affinity-aware Modeling of CPU Usage with Communicating Virtual Machines*. Journal of Systems and Software (JSS), 2013. Sujesha Sudevalayam, Purushottam Kulkarni.

2. *Energy Harvesting Sensor Nodes: Survey & Implications*. IEEE Communications Surveys and Tutorials 2011. Sujesha Sudevalayam and Purushottam Kulkarni.

Technical Reports

1. *CONFIDE: Content-based Fixed-sized I/O Deduplication*. Technical Report, IIT Bombay. Sujesha Sudevalayam, Purushottam Kulkarni, Rahul Balani and Akshat Verma. IITB/CSE/2014/April/60, TR-CSE-2014-60.

2. *Affinity-aware Modeling of CPU Usage for Provisioning Virtualized Applications*. Technical Report, IIT Bombay. Sujesha Sudevalayam and Purushottam Kulkarni. IITB/CSE/2011/February/34, TR-CSE-2011-34.

3. *Colocation-aware Modeling of CPU Usage for P2V Transitioning Applications*. Technical Report, IIT Bombay. Sujesha Sudevalayam. IITB/CSE/2014/March/59, TR-CSE-2014-59.

4. *Balancing Response Time and CPU allocation in Virtualized Data Centers using Optimal Controllers*. Technical Report, IIT Bombay. Varsha Apte, Purushottam Kulkarni, Sujesha Sudevalayam and Piyush Masrani. IITB/CSE/2010/April/29, TR-CSE-2010-29.

5. *Energy Harvesting Sensor Nodes: Survey and Implications*. Technical Report, IIT Bombay. Sujesha Sudevalayam and Purushottam Kulkarni. IITB/CSE/2008/December/19, TR-CSE-2008-19.

Acknowledgements

The journey of my Ph.D. has been an adventurous one, and it is the support of my advisor, Prof. Purushottam Kulkarni that has made this materialize. I am grateful to my father, (late) Dr. K. K. Sudevan for inspiring and encouraging me to take up a challenging Ph.D program at the esteemed Indian Institute of Technology, Bombay.

During the journey of my Ph.D., I have made several friends and acquaintances who have contributed in ways, known and unknown, to my outlook on life. I thank my friends and lab mates (Rijurekha Sen, Vijay Gabale, Uma Sawant, Senthil Nathan, Dhaval Bonde, Viven Rajendra, Mayank Mishra, Swetha Srinivasan, Prashima Sharma, Prajakta Patil, Mukulika Maity, to name a few), for the innumerable discussions, both technical and personal. I thank my friend and roommate, Deepti Shenai, for her continued encouragement and support.

I also thank fellow members on the hostel student council, during my term as General Secretary, especially Bandana Singha, Rohini Karandikar, Rijurekha Sen, Urbashi Sarkar and Neelam Rathore. I also acknowledge my friends and colleagues, Abhisekh Sankaran and Karthik Ramachandra, for engaging with my musical side.

Last but not the least, I am grateful to my mother, Mrs. Usha Sudevan, my husband, Amey Gavand, and my parents-in-law, Dr. Narayan Balaram Gavand and Mrs. Neelam Gavand, for their undying support and encouragement. Without their support, we would not be here today. Thank you.

Index