Improving Virtualized Storage Performance with Sky

Leo Arulraj, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

Department of Computer Sciences,
University of Wisconsin, Madison
{arulraj,dusseau,remzi}@cs.wisc.edu

Abstract

We introduce Sky, an extension to the VMM that gathers insights and information by intercepting system calls made by guest applications. We show how Sky gains three specific insights – guest file-size information, metadata-data distinction, and file-content hints – and uses said information to enhance virtualized-storage performance. By caching small files and metadata with higher priority, Sky reduces the runtime by 2.3 to 8.8 times for certain workloads. Sky also achieves 4.5 to 18.7 times reduction in the runtime of an open-source block-layer deduplication system by exploiting hints about file contents. Sky works underneath both Linux and FreeBSD guests, as well as under a range of file systems, thus enabling portable and general VMM-level optimization underneath a wide range of storage stacks.

1 Introduction

Virtual machine monitors (VMMs) are an integral part of the cloud computing infrastructure and offer several important advantages over more traditional approaches, including server consolidation [26, 42], reduction in infrastructure costs [56], simpler failure handling [27], ease of management [34], support for legacy applications [7, 43, 58], improved security [10, 14, 15, 22, 39], and better reliability [6, 9]. Virtualized storage, found within said VMMs, adds the benefits of storage consolidation, shared storage across VMs, out-of-box support across several guest OSes, reduction of costs, improved availability, efficient backups and quick snapshots [33, 38, 45, 53, 55, 57]. Not surprisingly, both server and storage virtualization are prominent and together form a central part of all modern cloud computing infrastructures.

As the lowest level in the software stack, the VMM [4] must manage system resources, including memory, disk, CPU, and network. In doing so, the VMM must optimize their usage for high performance, fairness, and other important system-wide goals.

Managing resources effectively fundamentally requires *information:* which I/O request is latency sensitive, and thus should be scheduled soon? Which block is likely to be accessed again soon, and thus would benefit from placement within a cache? Without this type of information, making the decisions a resource manager must make are at best arduous and often impossible. For example, a VMM cannot typically differentiate whether an I/O request consists of application data or is file-system journaling traffic. Without such basic knowledge, the VMM is inherently limited in its resource-management capacity.

The main hypothesis that underlies this paper is that the VMM can efficiently gain access to a wealth of important and necessary information through judicious usage of *system-call interposition*. In such a configuration, OS-level system call entry and exit are routed through the VMM. At these critical junctures, the VMM can record relevant pieces of information as well as take necessary actions in order to gain access to facts pertinent to its operation.

To explore this hypothesis, we have designed and implemented Sky, a prototype VMM with system-call interception at its core.[1] Sky is implemented for the x86-64 architecture and it supports Linux and FreeBSD OSes. Sky extends KVM with system-call interception to facilitate a range of new information-gathering techniques. Specifically, Sky implements a core interception framework to track specific processes and threads, and then obtains storage-specific insights atop this basic machinery. The insights include information such as the size of currently accessed files, the classification of block I/O into data and metadata, and file content assessment. Some of this information is approximate (i.e., not guaranteed to be correct); however, as we show, it is still useful in building various storage-system optimizations. To aid its information gathering, Sky also (on occasion) injects its own system calls into the OS above; said *insightcalls* are a useful general knowledge-acquisition technique atop the base interception mechanism.

We demonstrate the utility of Sky by implementing three case studies, each showcasing different possibilities within

VEE '17, April 08 - 09, 2017, Xi'an, China

© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-4948-2/17/04…$15.00

DOI: http://dx.doi.org/10.1145/3050748.3050755

[1] The acronym for System Call Interception, SCI, and one possible pronunciation, motivates our name.

the Sky infrastructure. The first is a simple monitoring tool (§6), which can provide generic information such as block lifetimes and the amount of metadata generated by different file systems. With such monitoring in place, a VMM can serve as a single point of detailed knowledge about guest file-system behavior.

The second case study, which we refer to as iCache (§7), implements an aggressive VMM-level caching policy [31], leading to a 2.3 to 8.8 times improvement in run time for both search and database workloads. This approach gives higher priority to small files and file-system metadata and thus can improve run-time significantly. We also show how an application (the MySQL database server) can provide further hints to the caching layer via Sky and improve performance further.

The third case study, known as iDedup (§8), takes advantage of Sky's file-content information to improve performance of a block-layer deduplication system [52]. Sky provides hints to iDedup about block usage patterns (§3.5), and iDedup uses such hints to avoid expensive lookups and thus improves performance. Specifically, this optimization reduces run time by 4.5 to 18.7 times for file-copy and encryption workloads (§8.2).

In each of these cases, Sky implements improvements within a VMM that previously had required full-stack modifications to obtain the information needed to implement said functionality. Sky, in contrast, functions across operating systems (Linux and FreeBSD), and different file systems (Linux Ext4, Btrfs, and XFS, for example). In this manner, Sky consolidates implementation of its optimizations, instead of replicating such effort across different file systems and operating systems.

The rest of this paper is structured as follows. We first provide further motivation (§2). We then describe the design (§3) and implementation (§4) of Sky. Finally, we evaluate Sky (§5 to §9), discuss related work (§10), and conclude (§11).

2 Motivation

In modern virtualized storage systems, better performance, quality of service, and other critical optimizations and features can be achieved through access to information. For example, previous work has shown that classifying I/O requests, and treating each class differently, can greatly improve performance for some workloads [28, 31, 53].

Unfortunately, due to the simple, restrictive interfaces exposed by each of the layers, information cannot be easily passed through the many layers of the storage stack. This reality leads to the so-called "semantic gap" [8] across said layers, thus leading to many missed opportunities in the storage stack [11, 19, 31, 32, 40, 44, 46–49, 59].

Many examples exist in the literature that showcase the benefits of information (and control) throughout the storage stack. For example, Thereska et al. classify I/O requests into flows and associate policies for each of the I/O flows to allow differentiated treatment [53]. Mesnier et al. explicitly classify I/O requests to improve performance, by modifying the application, file system, and low-level storage interfaces [31]. Sonam et al. use I/O classification to improve inline block-layer deduplication by modifying the application, file system, and block I/O interface to generate hints about file-system metadata and file contents [28]. All approaches require changes across many layers of the storage stack.

Compared Research Work	**Number of modified components (Names of the supported components)**		
	OS	**FS**	**Storage Interfaces**
Differentiated Storage Services [31]	2 (Linux, Windows)	2 (Ext3,NTFS)	3 (VFS,Block I/O, iSCSI)
Deduplication with hints [28]	1 (Linux)	4 (Ext2/3/4,Nilfs2)	2 (VFS,Block I/O)
Sky	**0** (Linux, FreeBSD)	**0** (All)†‡	**0** (All)†

Table 1: **Ease of Adoption.** *This table compares the number of components of various types that are supported by Sky and other relevant past research work without any additional implementation effort. † Sky currently supports FSs that do not change user-supplied content (e.g., due to compression or encryption within the FS). Sky uses system-call interception and therefore is not affected by the choice of the storage layers (e.g., FS and device drivers) or their interfaces (e.g., VFS, Block I/O and SCSI) present in-between the system call and the VMM. ‡ We have tested Sky with the FSs UFS, ZFS in FreeBSD guest OS and with Ext3, Ext4, XFS, JFS, Nilfs2, Reiserfs, and Btrfs in Linux guest OS.*

While these systems all provide significant benefits, we believe there are important reasons that they often do not reach deployment. One prominent reason is that any idea that must be realized throughout the storage stack creates a large burden upon developers. Table 1 compares the number of different types of operating systems, file systems, and storage interfaces that must be modified to support various storage optimizations [28, 31]. Being able to run underneath multiple operating systems, file systems, and interfaces has a multiplicative effect on developers, who must modify each of these components to reach wide-scale deployment. In contrast, as the table shows, Sky works across different file systems and storage interfaces, and provides the infrastructure needed to work underneath Linux and FreeBSD; developers of new storage optimizations can thus implement them once within the Sky framework and deploy them underneath a wide range of systems.

Of course, if all vendors agree on a set of information to pass across layers, it is possible that new standards could be developed and adopted. However, as others have discussed, changing interfaces is difficult and time consuming [47]; even small changes to the low-level disk standards, such as the evolution from block-based to object-based storage [16], may take many years to come to fruition, or never reach wide-scale adoption at all.

Tracked Information	Used for
List of monitored processes, their PIDs and their page directory base address.	Interception Framework (§3.1)
Threads of monitored processes, their TIDs, stack base pointers and stack size.	Interception Framework (§3.1)
Parent child relationship between monitored processes.	Interception Framework (§3.1)
System Calls in progress for monitored processes along with their arguments and userspace stack pointer value.	Interception Framework (§3.1)
A per-process pool of 8KB userspace buffers allocated by Sky for issuing insightcalls that need their arguments to be in memory.	Interception Framework (§3.2)
List of monitored file descriptors, their current file offsets and the maximum file offset accessed so far.	Insight I (§3.3)
List of memory-mapped pages for monitored files in monitored processes and their corresponding guest-physical addresses.	Insight II (§3.4)
Checksums of 4096-byte chunks in data payload of I/O-related system calls are stored while the system call is getting processed. Checksums of 4096-byte chunks in data read from or written to the virtual disk are retained for a certain time period.	Insight II (§3.4)
Whether a process has file-copy I/O pattern or is encrypting files. Detecting file-copy I/O pattern requires storing checksums of 4096-byte chunks of data read or written by applications temporarily.	Insight III (§3.5)
The checksums of 4096-byte chunks at various file offsets when block lifetimes need to be calculated.	Block Lifetime (§6.2)

Table 2: **Information tracked by Sky.** *This table lists the information tracked by Sky about guest OS and its processes.*

The best system to support cross-layer optimizations requires modification only at a single spot in the stack, not requiring changes throughout many layers. The optimizations realized in such a framework should then work across a broad range of systems with little or no effort. We now describe one such system that we have built, Sky, which is implemented as part of the VMM and enables interesting information-based storage services to be realized.

3 Design

This section describes the basic techniques used by Sky (implemented as part of the VMM) to intercept system calls (§3.1 and §3.2) and then details the insights gained by intercepting I/O-related system calls (§3.3 to §3.5). Our design was influenced by the following desirables:

- *Simple and Universal:* Favor simple techniques that are widely applicable across OSes.
- *Timely:* Generate reliable hints as early as possible.

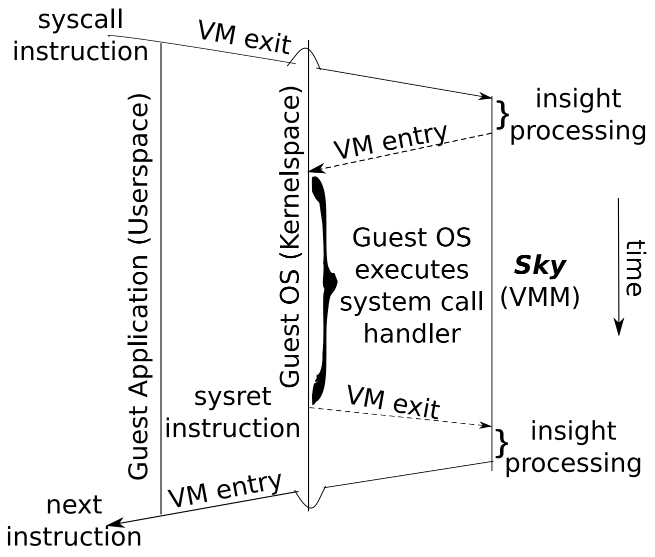

Figure 1: **System-Call Interception.** *This figure shows the control flow between the guest application, guest OS and the VMM during a system-call interception in a monitored process. Sky turns off the hardware interception techniques when unmonitored processes are scheduled on a processor.*

- *Robust and Lightweight:* Keep Sky robust and its overheads low.

Table 2 is a summary of all the information tracked by Sky and where the information is used. Sky tracks information about processes, threads, process parent-child relationship and in-progress system calls in order to provide the basic interception framework upon which meaningful insight gathering techniques can be implemented. Depending on the insight generated, additional information is tracked by Sky as listed in Table 2.

3.1 System-Call Interception

Sky intercepts the entry and exit of a subset of I/O and process-management related system calls executed by the guest application in order to gain insights that can be used as hints to improve virtualized-storage performance. Sky configures the processor before a VM entry so that execution of a system call entry or exit instruction causes a VM exit and transfers control back to Sky (part of VMM). Figure 1 shows the control flow during system-call interception. With this ability to intercept system call entries and exits, Sky can monitor the arguments and return codes of system calls to gather insights about the guest application. Being part of the VMM, Sky can access all of the guest VM's memory enabling further optimizations and new features.

3.1.1 Selective System-Call Interception

Sky avoids the overheads due to intercepting all userspace applications in the guest VM by monitoring only a targeted set of I/O-bound processes. Sky automatically monitors and unmonitors the children of the monitored processes by intercepting process-management related system calls like `fork`, `clone` and `kill`. Sky monitors all threads in a monitored process by default. Whenever a new guest process is scheduled on a virtual processor, Sky checks if the new process is monitored or not and turns system-call interception on or off respectively.

In our prototype, monitoring of applications is bootstrapped by a helper application calling a library function with its own PID and then launching the benchmark application. The library function sends the PID to Sky (which is part of the VMM in the host machine) through the network. Sky automatically monitors the launched benchmark application and any other processes it subsequently creates. More sophisticated policies could be built on top of this scheme when appropriate in the future: e.g. tracking and identifying certain applications that are known beforehand to benefit from Sky or periodically monitoring the latency of I/O-related system calls made by guest applications and dynamically turning system-call interception on or off based on these latencies. Our prototype version does not do this.

Identifying Processes and Threads: Sky keeps track of the guest-OS assigned process identifiers (PIDs) and thread identifiers (TIDs) for the monitored processes and threads respectively. When intercepting system call entries and exits, Sky uses only the virtual-CPU state to identify the currently executing process or thread. Specifically, processes are identified using the page directory base register (PDBR) that contains the guest-physical address of the currently executing process's page directory base. Sky maps a PID to the page directory base address by issuing a `getpid` insight-call (described in §3.2). Sky captures the userspace stack base address, stack size and the thread identifier while intercepting thread-creation system calls. Sky differentiates between threads within a process using the stack pointer (SP) register that contains the guest-physical address of the top of the userspace stack. Given the values of the PDBR and SP register, Sky identifies the guest-OS issued process and thread identifiers respectively. Sky maintains a set of all monitored processes, their PIDs, PDBRs, threads, thread TIDs, thread SP values and thread stack sizes.

Tracking Guest-OS Scheduling: Sky intercepts all guest-OS process scheduling by intercepting overwrites to the PDBR of the virtual processor through hardware mechanisms. The PDBR has to be compulsorily overwritten with the page directory base address of the new process during process context switch. Since thread rescheduling does not involve a PDBR overwrite, Sky uses the following technique: during a system-call interception, if Sky detects a different currently running TID from the one that last executed on the same virtual processor during the last system-call interception, it knows a thread switch has occurred. Such delayed detection of a thread switch only when a system call is intercepted is sufficient for matching a system call exit correctly with its entry.

3.2 Insight-Calls: Sky-Introduced System Calls

In certain scenarios, Sky needs access to the state maintained by the guest OS in order to gather more insights efficiently, easily and in a manner that eases portability across different guest OSes. Sky (which is part of the VMM) issues system calls to the guest OS in the context of an intercepted guest application in order to read state from the guest OS. We call such Sky-issued system calls as Insight-Calls. Sky currently issues insight-calls only when it is intercepting an actual system call made by a guest application and to only read state from the guest OS. Insight-calls never change the state of the guest OS because that would be outside the knowledge of the guest application and could lead to erroneous application behavior.

Insight-Call: To issue an insight-call, Sky first saves the intercepted system call entry's system call number and arguments (call it *'syscallinformation'*) into a private data structure and then replaces them with those corresponding to the insight-call that it wishes to issue to the guest OS. When Sky subsequently intercepts the system call exit, it restores back *'syscallinformation'* into the appropriate registers and additionally decrements the current instruction pointer (IP) appropriately to point back to the system call entry instruction. This way, the original guest-issued system call is now executed by the guest OS. Sky can also issue a series of such insight-calls when more complex information needs to be gathered from the guest OS. Sky uses insight-calls in several scenarios like: getting the PID of the process currently executing (§3.1.1), getting the current size of the file backing an open file descriptor (§3.3) and handling 'misaligned or small I/O requests' (§3.4.4).

We note that, customers who do not completely trust their service providers (e.g., in a IaaS cloud computing model) with the usage of insight-calls could be given an option to opt-out of Sky during their sign-up process. Also, to improve performance in some cases, it is possible to avoid insight-calls and instead directly access the guest-OS internal state to get the required information [12]. Sky does not use such optimizations because the effort does not seem justified given the relatively small number of insight-calls. Exploring such optimizations is left as future work.

3.3 Insight I: Guest FS File Size Information

A storage system cache can achieve improved cache hit rates by knowing whether an I/O request is issued on a small file or a large file [31]. This is because large files are usually laid out sequentially on a magnetic disk and therefore cache misses on reads to small files are costlier than misses on a large file. Moreover, more small files can be cached in the same cache space occupied by a large file. Sky implicitly classifies I/O requests based on the size of the corresponding file by keeping track of the current file sizes of all files opened by a monitored process. An example of such file size based classification is shown later in Table 6 of the *iCache* case study (§7).

Sky achieves such file-size based classification by intercepting the I/O-related system calls like `open`, `read`, `write`, `lseek` and `close` in order to capture information like: current open file descriptors in a process, the current file offsets for those file descriptors, the files behind those file descriptors and their current file sizes.

Selectively monitoring only certain files: Sky can be instructed to selectively intercept I/O to only specific files using a control command sent through the network. The current prototype version of Sky allows specifying such files using their path prefix that denotes their location in the guest FS hierarchy. Sky issues a `getcwd` insight-call in order to get the current working directory of the process before prefix matching files opened using relative file paths. A variety of other policies for specifying which files to selectively monitor are possible.

Tracking File Sizes: Sky tracks the current file offset and the highest file offset accessed so far for all the monitored file descriptors whenever an I/O is performed by a guest application using read, write and other similar system calls. Sky also tracks the current file size for all monitored file descriptors using an `lstat` or `lseek` insight-call.

Sky translates the file size information into an *I/O class* and hints the VMM-level storage cache to adjust the priority based on the *I/O class*. Sky always associates gathered insights with that particular I/O request rather than with the virtual-disk sectors to which the I/O request is destined in order to avoid insights becoming stale when the corresponding sectors are reallocated. A case study on such a smart storage cache called *iCache* with its performance compared against a normal storage cache is detailed in §7.

3.4 Insight II: Guest FS Metadata vs. Data Classification

Cache hits can be improved by distinguishing FS metadata from application data and giving higher priority to FS metadata [31]. The FS inside the guest-OS kernel organizes information on the virtual disk by writing metadata information (e.g. block allocation bitmap, file offset to disk block translation) in addition to the data from the guest application. However, the distinction between guest-FS written metadata and guest-application written data is not available at the virtual disk in the VMM. Sky provides useful hints to the virtual disk to distinguish metadata I/O requests from data I/O requests. The basic idea behind this insight is the observation that all data I/O requests originate from the guest application while all metadata I/O requests originate from the in-kernel FS within the guest OS. Sky tracks the set of all data I/O requests that originate from the guest application using system-call interception and identifies metadata I/O requests by exclusion from this set. §3.4.1 to §3.4.4 detail how to handle different types of I/O system calls using this basic technique.

3.4.1 Handling Synchronous I/O

Synchronous I/O is performed primarily using the `read` and `write` system calls. Both take three arguments: the open file descriptor on which I/O is requested, the address of a userspace buffer for I/O contents and the number of bytes in the I/O. These system calls return the number of bytes successfully accessed upon success and a negative error code upon failure. Other system calls for performing synchronous I/O like `pread`, `pwrite`, `preadv` and `pwritev` are handled similarly.

Writes: When a guest application issues a `write` system call to write data to a file, Sky intercepts the system call entry and calculates the checksums of every 4096-byte sized chunk in the userspace buffer supplied by the application. Sky, being part of the VMM, easily translates the guest-virtual address of the userspace buffer to host-virtual addresses while accessing the userspace buffer. Sky stores these checksums in a hashtable. The *I/O class* based on the file size insight described earlier in §3.3 can also be stored in this hashtable. When this system call is then serviced by the guest OS, it eventually causes a write I/O request to the virtual disk. Sky also interposes on this subsequent write request to the virtual disk to calculate the checksums of every 4096-byte chunk and looks up the checksums in the hashtable. If the checksum is found, it indicates that the content originated from the guest application and hence is a data I/O request. Checksums for metadata I/O requests will never be found in the hashtable. Sky removes the checksums from the hashtable after the lookup to avoid any future misclassification.

Reads: Reads have to be handled slightly differently by Sky. When a guest application issues a `read` system call, the data is available in the userspace buffer only after the system call completes because the data has to be read from the virtual disk or the buffer cache as part of the `read` system call servicing by the guest OS. Hence, Sky interposes all read requests to the virtual disk and calculates the checksums of every 4096-byte sized chunk being read and stores them in a hashtable along with the corresponding sector number in the virtual disk and a timestamp. Subsequently, Sky also interposes the exit of the `read` system call that caused the read request to the virtual disk, calculates the checksums of every 4096-byte chunk in the userspace buffer and looks up the checksums from the hashtable. If the lookup succeeds, Sky classifies the request as data I/O and removes the checksums from the hashtable. All entries remaining in the hashtable after a configurable sufficiently long delay (currently set at 4 seconds) are classified as reads due to metadata I/O requests. In the experiments presented in this paper, we never had to re-configure this delay value.

3.4.2 Handling Asynchronous I/O:

When a guest application issues an asynchronous I/O system call, the I/O is not complete when the system call returns. Rather, the I/O completes at a later point in time and the guest application learns about the completion later using a separate system call. Hence, for asynchronous read I/O system calls, Sky performs the checksum calculation and lookups after the I/O request is completed by the guest OS. For asynchronous write I/O requests, the checksum calculation occurs during the I/O-submission system call entry.

3.4.3 Handling Memory Mapped I/O:

Memory mapped I/O is performed by mapping a region of the file address space to a region of the process's virtual-memory address space. I/O requests to the virtual disk are

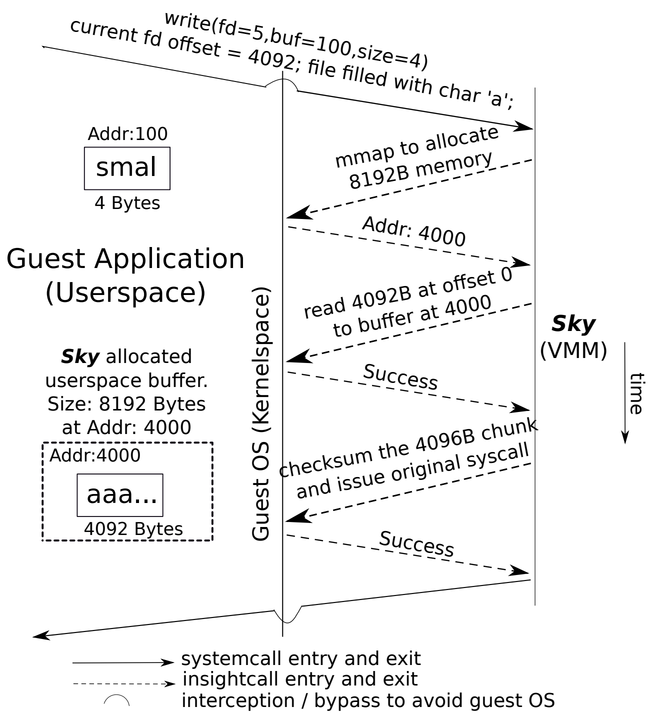
write(fd=5,buf=100,size=4)
current fd offset = 4092; file filled with char 'a';
Addr:100
smal
4 Bytes
Guest Application (Userspace)
Sky allocated userspace buffer. Size: 8192 Bytes at Addr: 4000
Addr:4000
aaa...
4092 Bytes
Guest OS (Kernelspace)
mmap to allocate 8192B memory
Addr: 4000
read 4092B at offset 0 to buffer at 4000
Success
checksum the 4096B chunk and issue original syscall
Success
Sky (VMM)
time
systemcall entry and exit
insightcall entry and exit
interception / bypass to avoid guest OS

Figure 2: **Insight-Calls for handling a Small Write.** *This figure shows how Sky handles an example small write using a series of insight-calls.*

automatically issued by the guest OS when the memory mapped address space is accessed by the guest application. Because there are no system calls to intercept for insights when I/O happens, memory-mapped I/O is specially handled by Sky. Sky intercepts the `mmap` system call that initially performs the memory mapping with parameters that specify the starting virtual-memory address and the length of the memory-mapped region. At this time, Sky write protects the host-OS memory pages that contain the guest page-table entries behind the memory-mapped virtual address space. This write protection ensures traps to Sky whenever the guest OS changes the guest-physical pages backing the memory-mapped virtual address space. Thus, Sky continuously keeps track of the most recent guest-physical pages (and their host-physical pages) backing the memory-mapped virtual address space in a global hashtable.

A memory-mapped I/O request automatically issued by the guest OS to the virtual disk always contains the guest-physical address backing the memory-mapped virtual page that was accessed. This is because memory-mapped I/O skips the buffer cache in the guest OS. When Sky intercepts the I/O requests to the virtual disk, it also looks up the host-physical addresses of the pages behind every I/O request in the global hashtable described above. A successful lookup indicates a data I/O request while a failure means metadata. Sky removes the old addresses and adds new ones to the hashtable when the guest OS unmaps the old guest-physical pages and maps new ones for the memory-mapped virtual address space.

3.4.4 Handling Misaligned and Small I/O:

Sky always calculates checksums of 4096-byte chunks that are aligned with 4096-byte file offset boundaries so that the checksums remain valid when interposing the I/O requests to the virtual disk despite the guest OS splitting or merging I/O requests. 4096 bytes or a sector is the smallest addressable unit for modern disk drives and is smaller than or equal to the FS block size. However, guest applications can issue I/O system calls that result in chunks smaller than 4096 bytes either due to small I/O requests or due to I/O requests misaligned with the 4096-byte file offset boundaries. Sky handles such I/O requests by reading the necessary contents from the file using Insight-Calls to form 4096-byte chunks as outlined in Figure 2 and described below. It should be noted that because our prototype uses the insights as hints for performance improvements, it could skip handling misaligned and small I/O requests.

1. Sky allocates a 8192- byte private anonymous userspace buffer in the guest-application's virtual address space using a `mmap` insight-call. Sky also adds this 8192- byte userspace buffer into a free memory pool that it maintains for every guest process so that future Insight-Calls for this process can reuse the same buffer.

2. Sky then reads any necessary additional content located before and after the small or misaligned I/O request's file offset as needed using `pread` insight-call into the first and second 4096 bytes of the userspace buffer allocated in the previous step.

3. Sky then calculates checksums of resulting aligned 4096-byte chunks before finally reissuing the original guest application's system call. For misaligned reads alone, Sky issues the original guest-application's system call before issuing the `pread` insight-calls so as to avoid multiple disk requests for large multiblock reads. This way the total number of disk I/O requests generated remains the same while handling misaligned or small read requests.

As an optimization, Sky also caches the contents of the most recent I/O along with the file descriptor, file offset and data size information for monitored processes. Using this information, misaligned I/O resulting from a strided access pattern that starts at an unaligned offset can be handled without using the process described above that uses additional insight-calls.

Note: Certain FSs store both metadata and data in the same virtual-disk block: e.g. Ext4's inline data feature stores tiny files inside the inode structure. Sky classifies such blocks as metadata. There is a very small window of chance when a monitored guest application could generate data blocks that match the metadata blocks generated by the guest FS within the short duration of the virtual-disk access times for those data blocks. Since, our prototype uses insights as hints, such small chances of misclassification are tolerable.

3.5 Insight III: Application I/O semantics and patterns

Tracking the I/O semantics and patterns of applications can be helpful in improving their performance. Sky can detect I/O patterns without any modifications to the application or the guest OS. Sky sends the I/O-pattern insights as hints to the storage system. Example I/O-pattern insights are 'knowing when an application is encrypting data' or 'knowing when an application is copying data from one file to another'. We show how these I/O-pattern insights can be used to improve the performance of a deduplication system in §8.

Detecting File Encryption: Sky uses the names of the executables and the file name extensions of the destination files to derive hints about encryption. Sky issues a `sysctl` insight-call or a `readlink` insight-call to get the name of the executable depending on whether the guest OS is Linux or FreeBSD respectively. The file name extensions are available as an argument while intercepting `open` system calls. Sophisticated executable identification and file type detection by examining the contents of the executable and destination file is left as future work.

Detecting File Copy: Sky detects file-copy I/O patterns by first targeting certain guest applications by using the executable names as a hint: e.g. Unix tools like *cp* and *dd*. Sky then stores the checksums of 4096-byte chunks being read by such targeted application in a hashtable. Finally, Sky looks up the checksums of 4096-byte data chunks being written by these applications in the hashtable to confirm the file-copy I/O pattern. Repeated matches indicate a file-copy I/O pattern in progress.

3.6 Application Supplied Insights

Certain applications already perform or can be easily modified to perform better I/O classification because they have the most information about the I/O requests that they issue. The exact policy used for I/O classification depends on the guest application. For example: a cloud file server can associate its premium customers with a higher priority I/O classification ensuring a better QoS, a database server can associate I/O requests to certain data structures like the 'secondary index' with lower priority to ensure better overall throughput (§7.2.2).

Guest applications can pass the I/O classification information on a per-system-call basis by calling an alternate library function that is similar to its counterpart in standard libraries like libc. This alternate function takes an additional last argument for the I/O class. For example, a C application calls `iwrite(file descriptor, buffer, size, ioclass)` library function to perform writes instead of the usual `write(file descriptor, buffer, size)` libc function. The `iwrite` library function issues the `write` system call with two additional last arguments: an additional magic number argument and the I/O classification number. During system-call interception, if Sky sees that a system call has the same magic number as the second-to-last argument, then it indicates that the guest application is supplying explicit I/O classification information in the last argument. Therefore, Sky uses the guest application supplied I/O classification and turns off its own implicit I/O classification based on 'file size', 'FS metadata vs. data' and 'application I/O semantics' for that I/O request. This approach of passing I/O classification along with every system call allows fine granular control on every I/O request rather than over an entire file. Also, it allows passing down the I/O classification information from the guest application to Sky very efficiently in a timely manner. Guest applications that directly access a virtual disk without a FS can also pass I/O classification using this approach.

4 Implementation

4.1 Interception Techniques

Intercepting System Call entries and exits: Sky uses previously known techniques [10, 39] to intercept all x86-64 system call instructions except the IRET instruction for which we describe a new technique below. All the experiments in this paper use 64-bit guest OSes and they run on an Intel processor; therefore, they all used the Syscall, Sysret and IRET instructions for performing system calls. We tested out some of the other previously known interception techniques listed below but did not use them with Sky. A comparison of Sky with related work in the field of Virtual Machine Introspection is in §10.

- ***INT 80:*** The Interrupt Descriptor Table (IDT) size is restricted to cause faults during software interrupts.
- ***Syscall and Sysret:*** The SCE flag is unset so that the syscall/sysret instruction causes a VM exit.
- ***Sysenter and Sysexit:*** The SYSENTER_CS_MSR machine status register is set to NULL so that the sysenter/sysexit instruction causes a VM exit.
- ***IRET:*** Both Linux and FreeBSD kernels use the IRET instruction for returning from a system call during slow return scenarios that include situations where userspace signal handlers are invoked before returning from the system call. We could not easily intercept the IRET instruction directly using architectural support with the Intel virtualization extensions unlike the AMD virtualization technology [3, 20].[2] Sky intercepts the system call exits that use IRET instruction using the following technique. Whenever Sky intercepts a system call entry instruction, Sky subtracts the size of the syscall instruction from the userspace IP register and keeps track of any such subtraction made. This subtraction guarantees an interception during the system call exit because the syscall instruction will be re-executed again artificially upon system call exit. There are two possibilities during the subsequent system call exit: the guest OS either uses the Sysret/Sysexit instruction or uses the IRET instruction. In the former case, Sky intercepts the Sysret/Sysexit instruction using hardware mechanisms, gathers insight and undoes the subtraction because it is no longer needed. In the latter case, though the IRET instruction cannot be intercepted, the artificial re-execution of the syscall instruction will be intercepted by Sky, at which point Sky completes the insight

[2] The Intel manual [20] mentions that a VM exit occurs upon executing an IRET instruction if the 'NMI-window exiting', 'NMI Blocking', 'Virtual NMIs' and 'NMI Exiting' control bits are set. Using this technique, a VMM can queue a virtual NMI to a guest and subsequently inject a virtual NMI when the guest is ready after execution of an IRET instruction. However, we have not verified that this technique can be used for intercepting IRET instructions for the purposes of Sky.

processing and skips executing the artificial syscall instruction.

Intercepting Guest-OS Scheduling: Both Intel and AMD hardware virtualization extensions allow intercepting writes to the PDBR by setting a specific bit in the VM execution control register.

4.2 Handling Process Rescheduling

Sky gathers insights by analyzing system call arguments and the returned values. However, there are some tricky scenarios that arise when the guest OS reschedules monitored processes across different virtual processors. In these scenarios, associating the value returned by a system call to its earlier invocation and the corresponding arguments needs additional effort as detailed in the rest of this section.

Matching system call exits with entries: Sky matches system call exits with entries based on the fact that system calls are synchronous. There is at most one outstanding system call for any given process (or thread) at any point in time. Sky copies over the system call number, arguments and the PID of the currently executing process while intercepting a system call entry on any of the virtual processors. Sky matches the subsequent system call exit that occurs on a virtual processor with the currently outstanding system call entry on the same virtual processor.

Split system call entries and exits: I/O-related system calls that usually involve a disk access often get rescheduled to a different virtual processor after issuing the system call but before the guest OS returns after processing the system call. Sky needs to correctly match a system call exit that occurs on the new virtual processor with its system call entry that occurred earlier on a different virtual processor. To this end, whenever a new process is scheduled on a virtual processor, Sky checks if there is an outstanding system call entry in that virtual processor. If so, Sky stores that system call information into a global hashtable with the PID or TID as the key. In order to match a system call exit to its system call entry, Sky first looks up in this global hashtable with the PID or TID of the currently executing process or thread for any matching outstanding system call. If a match is found, it is removed and the system call is processed for insights. A match won't be found if the previous virtual processor is still idle and no new process has been scheduled on it yet by the guest OS. In this case, Sky looks up each of the other virtual processors for an outstanding system call entry that matches the PID or TID for the system call exit being matched. This look up always succeeds because the outstanding system call entry either has to be in the global hashtable or with one of the other virtual processors.

Handling Signal Handlers: When there are pending signals for a process, the corresponding userspace signal handlers (if any) are invoked by the guest OS before a system call exit occurs. The signal handlers could possibly issue new system calls too even before the previous system call is finished. Since the signal handler is invoked using a signal stack, the new system call will have a different userspace SP value during system-call interception. Sky detects such signal-handler invocations by noticing this difference in userspace SP and stores the outstanding system-call information into the global hashtable. Subsequently, when the outstanding system call's exit occurs after the signal-handler invocation is complete, Sky looks up the global hashtable to find the system call information and process it for gathering insights.

4.3 Linux vs. FreeBSD System Call Interface

Guest OS identification: Our prototype takes the guest OS type as a configuration parameter but it is possible to infer this automatically. Known techniques that use VM memory analysis [1] can be used to distinguish Linux from Windows. Linux and FreeBSD differ in the system call number for `exit` which is the last system call executed by a process and it does not return anything. VM memory analysis coupled with observation of system call numbers, their arguments, return values and frequencies could be used as a general approach to detect the guest-OS type automatically in a future version of Sky.

Sky handles the following differences between Linux and FreeBSD system call interfaces:

- Thread-related system calls: FreeBSD guest OS uses system calls like `thr_new`, `thr_kill` and `thr_exit` for threads while Linux guest OS uses `clone`, `kill` and `exit`.
- FreeBSD `nosys` system call: System Call numbered 0 in x86-64 FreeBSD is an indirect system call that takes another system call number as its first argument and invokes its system call handler. Sky intercepts such `nosys` system calls and gathers insights corresponding to the actual system call that gets executed.
- For failed system calls, the FreeBSD guest OS sets the Carry Flag in the virtual processor and returns a positive error code integer while the Linux guest OS just returns the negated value of the error code integer.

4.4 Prototype

We implemented a prototype of Sky using the KVM/Qemu VMM for the Linux OS on an x86-64 machine. Figure 3 shows the organization of a typical setup of running VMs using KVM/Qemu [5, 24]. The Host machine runs a Linux OS that has a KVM kernel module. Each of the guest VMs is by itself a userspace process running the Qemu emulation program. Our prototype supports both Linux and FreeBSD guest OSes. The KVM kernel module exposes the hardware virtualization features of the processor to accelerate running the userspace Qemu-emulated guest VM.

Almost all of the Sky logic is implemented within the KVM kernel module. We hope that Sky will become part of the mainstream KVM with options to turn it off if users want to. Sky is 7.8 KLOC of new code added to 43.6 KLOC of unmodified KVM source code. This is a modest increase in the hypervisor codebase. Our prototype uses a pseudo device driver in the host OS (3.8 KLOC of source code) to intercept the I/O requests to the virtual disk rather than intercepting them in the Qemu userspace emulator. This avoids the

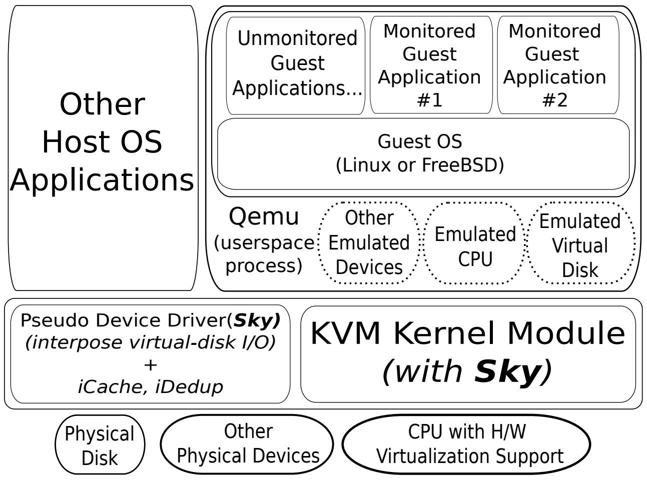
Other Host OS Applications
Unmonitored Guest Applications...
Monitored Guest Application #1
Monitored Guest Application #2
Guest OS (Linux or FreeBSD)
Qemu (userspace process)
Other Emulated Devices
Emulated CPU
Emulated Virtual Disk
Pseudo Device Driver(Sky) (interpose virtual-disk I/O) + iCache, iDedup
KVM Kernel Module (with Sky)
Physical Disk
Other Physical Devices
CPU with H/W Virtualization Support

Figure 3: **Sky Prototype Organization.** *This figure shows how Sky prototype is implemented for Qemu/KVM as part of the KVM kernel module along with the pseudo device driver within the Linux Host-OS Kernel. The guest VMs are userspace processes emulated by Qemu.*

Aspect	**Specification**
Host Processor	Intel i5,3.3Ghz,VT-x/EPT
Host OS	Linux (Kernel v3.11.5)
Guest OS	Linux or FreeBSD
Qemu Version	Qemu v2.5.0
KVM Version	KVM v3.10.1
Host, Guest Memory	16 GB, 6 GB
Virtual Disk	16 GB Paravirt (RAW Disk format)
Backing Disk	80 GB, 7200 RPM Magnetic Disk
Cache Device	2 GB In-Memory Disk
Bcache Version	Comes with Linux Kernel
Host FS	Ext3

Table 3: **Experimental setup.** *This table shows the experimental setup used to evaluate the Sky prototype.*

overheads associated with the communication between host-userspace and host-kernelspace while keeping the number of modified components minimal. Because the I/O requests to the virtual disk are intercepted within the host-OS kernel, Sky will also intercept disk I/O requests from the VMM and the host FS that are necessary for laying out the virtual disk on the backing physical disk. Sky correctly classifies such I/O requests as metadata because they won't be found in the set of data I/O requests tracked by Sky. Sky uses 64 bit checksums calculated using the 64 bit FNV-1a hash algorithm [13]. Bloom filters are used for quick lookups when appropriate: to check whether a process is monitored or unmonitored and to check whether a system call is I/O or process-management related or not. All the results reported in the following sections (§5 to §9) are the average of three trials.

5 Overhead Evaluation

Our experimental setup is outlined in Table 3. The virtual disk is loaded in KVM/Qemu with cache parameter as 'none' so that the Qemu-provided cache is disabled for evaluating the effects of the enhanced caching with insights. For all experiments in this paper, the measurements reported as when running without Sky are taken by disabling the Sky relevant code in our modified KVM module as opposed to using an untouched vanilla KVM version. We saw no measured difference in runtime or memory consumed when running a vanilla KVM version versus our modified KVM with Sky related code disabled.

During each VM exit caused by a system-call interception, KVM code gets executed in order to figure out the exit reason, to handle the exit and to emulate the instruction that caused the VM exit. In addition to this, Sky performs some computational work and hash table lookups to do the following: check whether the current process is monitored or not, check whether there has been a thread switch since the last system call interception for the same process, check if this is a split system call or if there has been a signal-handler invocation and perform statistics update for timing measurements to aid experimentation. This leads to CPU cache pollution.

We used a set of micro and macro benchmarks to evaluate the overheads due to Sky. The benchmarks were run both with and without Sky. *iDedup* and *iCache* were both disabled for these experiments. We ran these measurements on two different guest OSes: Linux and FreeBSD. When run without Sky, there is no system-call interception happening. The difference in runtime is used to calculate the overheads introduced by Sky as shown in Table 4. The percentage overhead that Sky introduces for real applications and macro benchmarks is minimal (under 5%) as seen from the last five rows of the table. The overall overhead is also split up to show how much of it is due to VM exits, basic Sky system-call interception and insight generation.

Micro-benchmarks: We use three types of micro benchmarks to measure the overhead imposed by Sky and their results are presented in Table 4. The repeated reads and writes benchmark accesses the same offset in a file leading to no actual virtual-disk I/O. When there is high-latency disk I/O (e.g. Random Writes workload), the overhead introduced by Sky for every I/O request is negligible when compared to the disk latency. However, the overhead of Sky is relatively high compared to the request latency when there is no disk I/O (e.g. Repeated Write to the same offset of a file). Sky is designed for I/O applications that issue I/O requests that involve accessing the disk. The repeated reads and writes micro benchmark is a worst case workload for Sky and hence system-call interception should be turned off for such applications.

Macro-benchmark and applications: We measured the overhead introduced by Sky for file encryption using the *gpg* command, file search using the *find* command, file copy using the *cp* command, Filebench varmail benchmark and TPC-H query on a MySQL database server. As shown in §7.2, the overheads are minimal (under 5%).

Workload	With Linux Guest OS and Ext3 FS			With FreeBSD Guest OS and UFS FS		
	With Sky (secs)	Without Sky (secs)	% Total Overhead (Splitup: VM Exits /Sky Interception /Insights)	With Sky (secs)	Without Sky (secs)	% Total Overhead (Splitup: VM Exits /Sky Interception /Insights)
Random Reads	99.4	98.5	1 (1/ 0/ 0)	185.2	183.3	1 (1/ 0/ 0)
Random Writes	54.9	54.6	1 (1/ 0/ 0)	272.5	269.8	1 (1/ 0/ 0)
Sequential Reads	14.6	14.9	-2 (-/ -/ -)	30.1	25.5	18 (13/ 2/ 3)
Sequential Writes	25.5	23.4	9 (4/ 2/ 3)	50.2	39.5	27 (24/ 1/ 2)
Repeated Reads	20.3	5.7	256 (157/ 41/ 58)	31.7	15.4	106 (68/ 16/ 22)
Repeated Writes	23	6.6	248 (153/ 36/ 59)	33	15.8	109 (68/ 16/ 25)
Encryption	33.8	32.3	5 (0/ 0/ 0)	25.3	24.8	2 (0/ 1/ 1)
File Search	78.3	76.4	4 (2/ 2/ 0)	54.6	53.4	2 (1/ 1/ 0)
File Copy	24.2	24.4	-1 (-/ -/ -)	34.9	34.7	1 (0/ 1/ 0)
Mail Server	385.1	381.1	1 (0/ 1/ 0)	163.5	160.8	2 (0/ 1/ 1)
TPC-H (MySQL)	36.1	35	3 (3/ 0/ 0)	21	21.7	-3 (-/ -/ -)

Table 4: **System-Call Interception introduced overheads.** *This table compares the time taken for various workloads when run with and without Sky on both Linux and FreeBSD guest OSs. System-Call Interception was turned on when running with Sky and was turned off when running without Sky.* iCache *and* iDedup *were both disabled. The total percentage overhead is shown and also splitup into sub components of percentage overhead due to VM exits, Sky interception and Sky insight computation.*

Guest FSs	Guest OS	Misclassification Error
Ext3,Ext4,XFS, JFS,Nilfs2,Reiserfs	Linux	0%
Btrfs	Linux	3.9%
UFS	BSD	0%
ZFS	BSD	0.7%

Table 5: **Accuracy of Sky.** *This table shows the data writes misclassification error percentage for Filebench varmail on different FSs.*

Memory overhead: Across all the experiments we ran, Sky used a peak memory usage of 33 MB of memory for its data structures including the various hashtables (not shown in Table 4). The amount of state maintained by Sky is on the order of 10s of bytes for every 4096 bytes of in-progress I/O (system call has been issued but virtual-disk I/O is not yet complete). Therefore, even for write-intensive workloads with 100s of 4K IOPS and a write delay of 10 seconds due to guest-OS page cache, the memory overhead will be on the order of 10s of MB.

6 Case Study #1: Information Gathering

In this case study, we show two examples of information gathering using Sky that are either useful by itself or can be used to improve storage performance.

6.1 Accuracy of data classification for different FS

We ran the varmail workload from the Filebench [29] benchmark suite after configuring it to finish after running a total of 100000 operations like file delete, file create, file append, file sync and whole file read. We also modified the Filebench benchmark to report the total number of 4096-byte chunks of data written to files. Sky classifies all the written data with a zero error percentage for all but two copy-on-write FSs as shown in Table 5. We saw small inaccuracies for the copy-on-write FSs Btrfs and ZFS (3.9% and 0.7%) which is a limitation of our current prototype. All writes that are not data are classified as metadata. This information is useful to evaluate the accuracy of Sky as well as to take a closer look into performance of different guest FSs as part of a virtualized-storage stack.

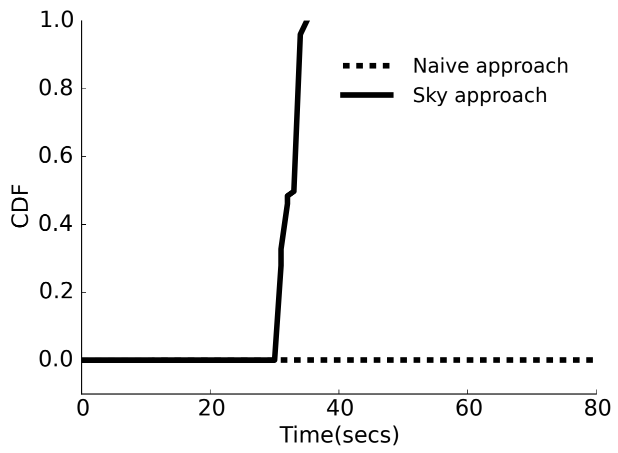

Figure 4: **CDF of block lifetimes for a synthetic workload.** *This figure shows the CDF of block lifetimes calculated using two approaches for a synthetic workload that writes 80 MB of data and deletes the files after a 30 secs delay.*

6.2 Block lifetimes

A VMM could use information about block lifetimes in order to schedule write back caching using a persistent cache device, to perform data reorganization on the backing physical disk, to intelligently prefetch content that skips dead blocks or to optimize recovery by skipping dead blocks [41, 47]. Sky allows a VMM to get the block liveness informa-

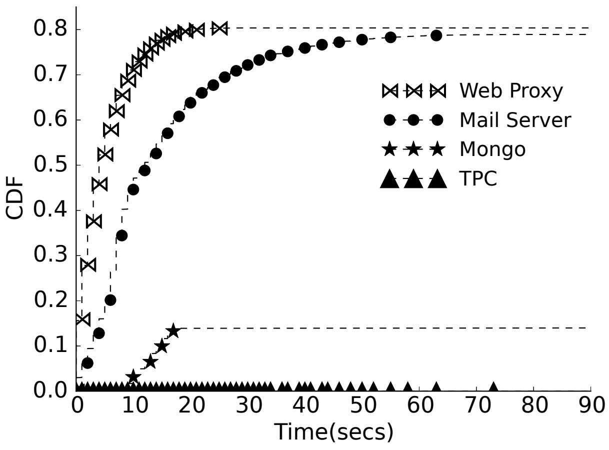

Figure 5: **CDF of block lifetimes for Filebench workloads.** *The CDF does not reach 1.0 because there are some blocks still alive at the end of these Filebench workloads.*

tion in a virtual machine setting without modifying the guest OS, FS or the applications. When the virtual disk and the FS support TRIM or UNMAP commands, Sky could track block liveness with less effort by tracking only those blocks that get deleted and are quickly reallocated before a TRIM or UNMAP command is issued to the virtual disk. Our current prototype of Sky targets applications that can tolerate a small level of inaccuracy due to checksum collisions. A detailed comparison with past related work on block liveness is in §10.

Approach: Sky uniquely identifies files by the guest disk device number and inode number. Sky maintains checksums of 4096-byte chunks at various file offsets by intercepting `write` and other related system calls. It detects block lifetimes by detecting overwrites to content already present at various file offsets. Sky also intercepts `unlink`, `truncate` and related system calls to accurately take into account file deletes and truncates. Figure 4 compares the block lifetimes calculated using a naive approach that just uses block overwrites with that calculated using Sky for a synthetic workload that writes 80 MB worth of file contents, sleeps for 30 seconds and finally deletes the files. This synthetic workload helps illustrate that Sky correctly handles file deletes. Since the naive approach does not know about the file deletes or truncates, it thinks all the blocks are still alive, while Sky correctly calculates the block lifetimes as approximately 30 seconds for all the blocks.

Figure 5 shows the cumulative distribution function (CDF) of block lifetimes for four Filebench workloads. Sky could use such block lifetime information about the running workloads to adjust the delay while scheduling write-back from a faster persistent cache device to the slower disk. The CDF does not reach 1.0 because there are some blocks still alive at the end of these Filebench workloads.

File Size (MB)	**<14**	**<10**	**<5**	**<2**	**<1**	**Meta Data**	**Skip Dedup.**
I/O class	0	1	2	3	4	5	32

Table 6: **Policy to assign I/O class to disk I/O requests.** *The default* I/O class *is 0. Priority increases with increasing* I/O class *values (0-lowest,5-highest). Sky uses* I/O class *32 to hint that the payload is unique and deduplication can be skipped.*

7 Case Study #2: *iCache*

In this case study, we show how the effectiveness of a storage cache can be improved by using the policy of caching small files and FS metadata with higher priority. This policy is complementary to the traditional cache-management algorithms like LRU, LFU, MQ [60] and ARC [30]. Such traditional cache-management algorithms differentiate disk blocks only based on their access patterns and do not associate any semantic meaning to them. Our work adds this missing semantic meaning to the traditional cache-management algorithms. Sky helps VMs make better use of their fair share of cache allocated by the hypervisor and is complementary to algorithms for fair cache partitioning between VMs. We also show how a MySQL server can be easily modified to pass insights directly to Sky bypassing the guest OS.

7.1 Implementation

Bcache External Disk Caching Module: We integrated the Bcache block device caching layer from the Linux kernel with Sky's pseudo device driver. I/O to the slower magnetic disk that contains the virtual disk is cached using an in-memory disk. We made the following modifications to the stock Bcache code (adding 10% new code):

- Added statistics collection to track and report sector-level hits and misses rather than the default request-level reporting.
- Added code to search for the slot holding a particular sector and change its priority.
- Added code to not cache guest FS journal writes
- Allowed 'clearing the cache' and 'reading the list of cached sectors' from the userspace for experimental convenience.

Sky sets the priority of Bcache slots that contain FS metadata and small files to higher values than the default.

7.2 Evaluation

We now show how the performance of a variety of real applications and macro benchmarks can be improved by using this *iCache*. The policy used by the *iCache* is to give the highest priority to metadata followed by lower priorities for data depending on the size of the file as shown in Table 6. The emphasis is on how classification of I/O requests and their differential treatment can bring benefits rather than the particular choice of this policy. Applications that fit a different policy profile can use their own policies as described in §7.2.2.

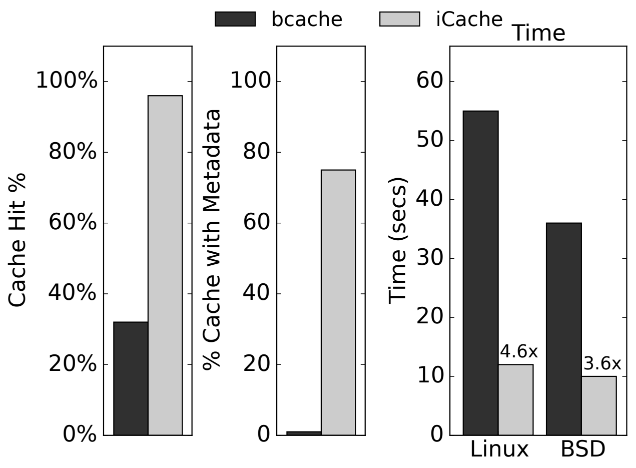

Figure 6: **File Name Search (*find*) Results.** *This figure compares the cache hit percentage, cache contents and the runtimes for the file name search workload on bcache and* iCache*. The numbers above the runtime bars for* iCache *are the speedups achieved over bcache.*

We run the experiments both with a normal cache as well as with the *iCache* and compare. The amount of physical resources available is the same for the normal cache and *iCache*. Also, when using the normal cache, system calls are not intercepted, thereby avoiding interception overheads. The differences in results are due to the differential caching done by *iCache*.

7.2.1 File Name Search (*find*)

A Linux kernel source code archive of size 115 MB is unzipped and untarred into a newly created FS 10 times creating 450K files with total disk usage of 6 GB. The *find* command is used to search for a non-existent file. The *iCache* retains the FS metadata when the Linux kernel source files are written due to its policy of giving higher priority to metadata than for the file contents. Because searching for a file using the *find* command only reads the FS metadata, the *iCache* outperforms the normal cache for this workload by 3.6 to 4.6 times as shown in Figure 6.

7.2.2 TPC-H on MySQL Database Server

We now show how a more sophisticated real world application can be modified in a way that it can explicitly classify I/O requests for differential caching. We changed the MySQL database server in order to differentiate I/O requests to the Clustered Index (which also contains the table data) from those to the Secondary Index by storing a tag in a thread local store at the various I/O-generating functions. Overheads due to our modifications to MySQL [36] were negligible. The mechanism described earlier in §3.6 is used to pass the I/O class along with the system calls. Since the secondary index can be huge in size and is also sequential on disk, the current policy we use is to give the secondary index data a lower priority than all other I/O requests. This is similar to the policy used by Mesnier et al [31].

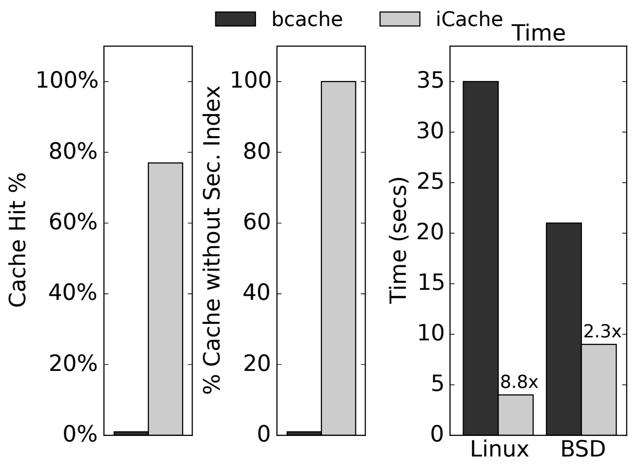

Figure 7: **TPCH on MySQL Server Results.** *This figure compares the cache hit percentage, cache contents and the runtimes for a TPC-H query workload on bcache and* iCache*. The numbers above the runtime bars for* iCache *are the speedups achieved over bcache.*

We load the TPC-H tables relevant to query number 16 with scale factor 1.7, create a few secondary indices on some columns on the tables and finally execute the query [54]. The *iCache* is able to retain the table data while the secondary index is created due to the policy of lower priority to secondary data. Hence, as shown in Figure 7, query number 16 which performs a join on two tables without using any secondary index executes faster on the *iCache* by 2.3 to 8.8 times.

8 Case Study #3: *iDedup*

In this case study, we demonstrate how the performance of a deduplication system can be improved by using hints about the semantics of the I/O workload. First, we show how an application that copies one file to another could greatly benefit by avoiding expensive disk-backed hashtable lookups. Second, we show how such hashtable lookups and additions can be avoided for encryption workloads that very rarely get deduped [50]. The time taken for hash lookups and additions can be substantial because hashes are randomly distributed and it is impractical to keep all the hashes in memory; therefore, a disk-backed hashtable that is persisted by frequent flushes leads to slow random I/Os during lookups and additions [28, 61].

8.1 Implementation

Dmdedup *Block Layer Deduplication:* We made the following modifications to the stock *dmdedup* [52] code (adding 14.5% new code):

• Added code to specially handle writes that are known beforehand to contain unique payload by skipping the initial search and the subsequent addition to the hashtable mapping the block checksums to the corresponding physical block numbers.

• Added code to maintain an in-memory cache of block checksum to physical-block number mappings. This in-

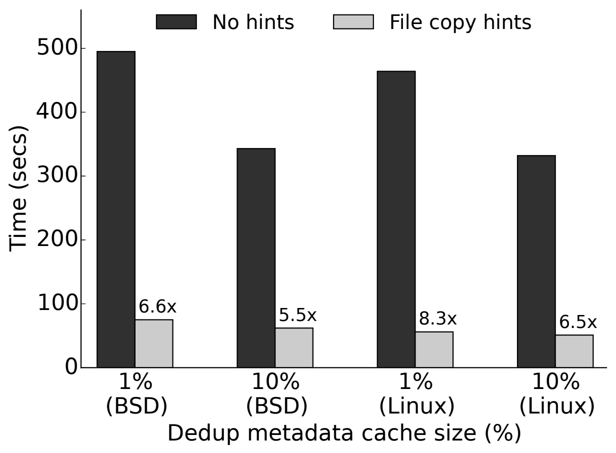

Figure 8: **File Copy Results.** *The numbers on top of the light colored bars show the speedup achieved for the file copy workload when run with file-copy hints on* iDedup.

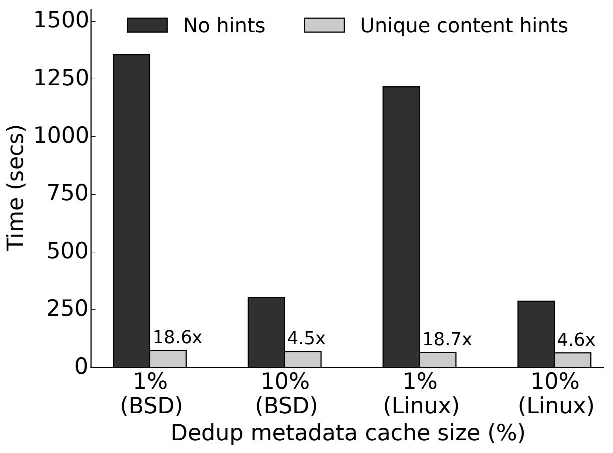

Figure 9: **File Encryption Results.** *The numbers on top of the light colored bars show the speedup achieved for the file encryption workload when run with unique content hints on* iDedup.

memory cache is populated during a read issued by a process flagged by Sky as exhibiting the file-copy I/O pattern. *iDedup* checks this in-memory cache for every write before issuing an expensive lookup to the disk backed hashtable.

- Added statistics collection to track and report the count of unique and file-copy hints, the hits in the in-memory cache of checksums to physical-block numbers, and the total time spent by all write requests in *dmdedup*.

8.2 Evaluation

We compare the performance of two different applications on *iDedup* against *dmdedup*. *Dmdedup* uses Dmbufio to buffer the I/O accesses to its disk backend. We ran the following experiments with a Dmbufio size of 1% and 10% of the peak metadata storage needs for the corresponding workload. 1% to 10% metadata cache sizes are typical in real deduplication systems [28]. The system calls are not intercepted for gathering insights when using *dmdedup* avoiding the overheads of system-call interception.

8.2.1 File Copy (cp)

The deduplication system is first warmed up by copying a 500 MB file full of random data on a newly created FS. Next the experiment is run which copies the just copied 500 MB file to another new file using the Unix *cp* command. Sky detects the file-copy I/O pattern and sends down hints to *iDedup* for every read issued by application. For such hinted reads, *iDedup* caches the mapping between the block checksum and the physical-disk block in memory. Upon a subsequent write of the same payload, *iDedup* looks up the in-memory cache and avoids the expensive lookups in the disk-backed hashtable. The stock *dmdedup* does not get such hints and hence *iDedup* is faster by 5.5 to 8.3 times as shown in Figure 8.

8.2.2 File Encryption (gpg)

A 500 MB file full of random data is encrypted using the GNU Privacy Guard (*gpg*) program. Sky infers the encryption by using the executable name of the *gpg* program and passes down hints to *iDedup* about unique data content. *iDedup* uses the hint to avoid looking up and subsequently adding a new entry to the disk-backed hashtable that maps block checksums to their physical-block numbers. Because most FS metadata is unique [28], Sky sends unique hints for all guest FS metadata writes also. *iDedup* is able to improve the runtime by 4.5 to 18.7 times over *dmdedup* as shown in Figure 9.

Case Study	**Workload**	**W Sky** (secs)	**W/O Sky** (secs)	**SSD Speedup/** (Overhead)	**HDD Speedup**
iCache (§7)	File Name Search	3.8	4.6	1.2x	4.6x
	TPC-H on MySQL	3.9	3.6	(0.9x)	8.8x
iDedup (§8) (with 1% dedup metadata cache size)	File Copy	32.3	34.1	1.1x	8.3x
	File Encryption	31.5	468.6	14.9x	18.7x
iDedup (§8) (with 10% dedup metadata cache size)	File Copy	18.1	24.8	1.4x	6.5x
	File Encryption	33.5	48.2	1.4x	4.6x

Table 7: **Sky with SSD Backing Disk.** *Sky provides good improvements when used with* iDedup *and provides nominal improvements when used with* iCache *on a SSD backing disk. Sky poses about 8% overhead for the TPC-H query on MySQL Server workload alone. The last column shows the speedup achieved on a magnetic disk for comparision.*

9 Fast Storage Devices

Sky's system-call interception imposes an overhead that is independent of whether a fast or slow storage device is used; therefore, the interception overhead is relatively higher when used with low-latency storage devices. Because of this, the benefits of *iCache* (§7) on a SSD storage device are not as high as those on a magnetic disk. Table 7 lists the speedups achieved with *iCache* and *iDedup* for various workloads on a Linux guest OS. *iDedup* (§8) has significant benefits even when used on a SSD storage device (though not as high as when used on a magnetic disk). Decreasing the system-call interception overhead is a good future research direction in order to make Sky more beneficial when used with fast storage devices.

10 Related Work

Mesnier et al. implement I/O classification by modifying the OS and application to pass down classification information that can be used by the storage system for better caching [31]. IOFlow, a software defined storage architecture, classified I/O requests at the VM granularity and enforced policies at various points in the I/O path [53]. Sonam et al. improve the performance of the *dmdedup* deduplication system by modifying the guest applications and FSs to generate hints [28]. In contrast, Sky obtains I/O-classification hints on a per system call basis without modifying the guest OS or the FSs and reaps similar benefits. Sky also provides an equally expressive interface to modified I/O applications when compared to the above previous works as discussed in §3.6. However, Sky targets virtualized-storage in the context of VMM while the previous works are more broadly applicable.

Several caching algorithms have been proposed in the past such as LRU-K [35], ARC [30], 2Q [21], MQ [60], LRFU [25] etc. Our work on associating priorities with insights is complementary to these caching algorithms because these algorithms differentiate between disk blocks only based on their access patterns while Sky associates semantic meaning to the blocks. For example, Sky allows hits on data from a high-paying customer to be better than hits on data from a low-paying customer.

Several past research works have shown that insights can be gained by the storage systems using the knowledge of the on-disk layout of FSs [2, 47–49, 51]. Having more complex logic in the storage systems can make them less robust and more expensive. Sky generates insights with considerably lesser complexity in the storage.

Virtual Machine Introspection [18] in general and system-call interception specifically [10, 39] have been applied for malware analysis and other security applications in the past. LibVMI [1, 37] is a library to access the guest VM details that primary supports memory accesses and events based on memory accesses. LibVMI has examples to show how to intercept system call entries but not system call exits. Sky uses system-call interception on both system call entry and exit for generating hints to improve storage performance. Sky introduces a new technique to intercept system call exits that use the IRET instruction in the Intel processors when compared to the past work on system-call interception using hardware extensions [10, 39]. Virtuoso [12] automatically generates programs for accessing guest OS information using training programs, trace collection and dynamic slicing. Techniques in Virtuoso could be used to fasten up certain parts of Sky like getting the PID of a process.

Roselli et al. use the auditing infrastructure in UNIX and the filter driver in Windows NT in order to collect and analyze traces to understand different FS workloads (e.g. block lifetimes) [41]. Sky obtains similar information about the block lifetimes through system-call interception. FADED [47] provides secure file deletes by providing block liveness to the storage device. It detects file deletes and truncates implicitly by tracking FS on-disk data structures and also making small modification to FSs when necessary. Sky directly intercepts `unlink`, `truncate` and related system calls to know about file deletes and truncates. Because Sky uses checksums, there is a loss of accuracy in rare scenarios when FS metadata and data content generate the same checksums. A more sophisticated future version of Sky could avoid this inaccuracy by using FS knowledge.

VirtFS is a paravirtualized FS that avoids the overheads associated with a generic networked FS by leveraging the 9p distributed FS protocol directly on top of a paravirtualized transport [23]. Sky could be used with VirtFS in order to allow guest applications to pass hints to VMM without modifying the 9p protocol and the VirtIO transport.

Gu et al. bridge the semantic gap between a VM and the VMM by running a process from the host on the guest VM under the cover of an existing running process in the guest [17]. Such an approach will be costly for Sky because intercepting system calls at userspace level is expensive due to the kernel boundary crossings and the context switches between the monitored and monitoring processes.

11 Conclusion

We proposed system-call interception as a core technique so that a VMM can gather insights and information without requiring modifications to the guest OS or the guest application. We built Sky, a prototype VMM that uses system-call interception to gather insights. We showed through experiments that system-call interception is an efficient way to obtain useful insights about I/O-bound guest applications with minimal overheads (under 5%). We also used Sky to implement a better storage cache called *iCache* and a better deduplication system called *iDedup*; these new features can improve the runtime of a variety of real workloads by 2.3 to 18.7 times.

12 Acknowledgments

We thank the anonymous reviewers and the members of the ADSL research group at UW-Madison for their insightful comments and feedback. This material was supported by funding from NSF grants CNS-1419199, CNS-1421033 and CNS-1319405, DOE grant DE-SC0014935, as well as donations from EMC, Facebook, Google, Huawei, Microsoft, NetApp, Samsung, Seagate, Veritas, and VMware. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of NSF, DOE, or other institutions.

References

[1] Libvmi: Virtual machine introspection library, 2016. `http://libvmi.com/` `https://github.com/libvmi/libvmi`.

[2] Agrawal, N., Arulraj, L., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Emulating Goliath Storage Systems with David. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies* (2011).

[3] AMD Technology. AMD64 Architecture Programmers Manual Volume 2: System Programming. `http://developer.amd.com/wordpress/media/2012/10/24593_APM_v21.pdf`.

[4] Arpaci-Dusseau, R. H., and Arpaci-Dusseau, A. C. *Operating Systems: Three Easy Pieces*, 0.91 ed. Arpaci-Dusseau Books, May 2015.

[5] Bellard, F. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference, FREENIX Track* (2005), pp. 41–46.

[6] Bressoud, T. C., and Schneider, F. B. Hypervisor-based Fault Tolerance. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1995), SOSP '95, ACM, pp. 1–11.

[7] Bugnion, E., Devine, S., and Rosenblum, M. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. In *SOSP '97* (Saint-Malo, France, October 1997), pp. 143–156.

[8] Chen, P., and Noble, B. When virtual is better than real [operating system relocation to virtual machines]. In *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on* (2001), pp. 133–138.

[9] Cully, B., Lefebvre, G., Meyer, D., Feeley, M., Hutchinson, N., and Warfield, A. Remus: High Availability via Asynchronous Virtual Machine Replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2008), NSDI'08, USENIX Association, pp. 161–174.

[10] Dinaburg, A., Royal, P., Sharif, M., and Lee, W. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security* (New York, NY, USA, 2008), CCS '08, ACM, pp. 51–62.

[11] Ding, X., Jiang, S., Chen, F., Davis, K., and Zhang, X. DiskSeen: Exploiting Disk Layout and Access History to Enhance I/O Prefetch. In *Proc. of USENIX07* (2007).

[12] Dolan-Gavitt, B., Leek, T., Zhivich, M., Giffin, J., and Lee, W. Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection. In *Security and Privacy (SP), 2011 IEEE Symposium on* (2011), pp. 297–312.

[13] Fowler, G., Noll, L. C., and Vo, P. Fowler / Noll / Vo (FNV) Hash, 1991.

[14] Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., and Boneh, D. Terra: A Virtual Machine-based Platform for Trusted Computing. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2003), SOSP '03, ACM, pp. 193–206.

[15] Garfinkel, T., Rosenblum, M., et al. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *NDSS* (2003), vol. 3, pp. 191–206.

[16] Gibson, G. A., Rochberg, D., Zelenka, J., Nagle, D. F., Amiri, K., Chang, F. W., Feinberg, E. M., Gobioff, H., Lee, C., Ozceri, B., and Riedel, E. File server scaling with network-attached secure disks. In *SIGMETRICS '97* (Seattle, WA, June 1997), pp. 272–284.

[17] Gu, Z., Deng, Z., Xu, D., and Jiang, X. Process Implanting: A New Active Introspection Framework for Virtualization. In *Proceedings of the 2011 IEEE 30th International Symposium on Reliable Distributed Systems* (Washington, DC, USA, 2011), SRDS '11, IEEE Computer Society, pp. 147–156.

[18] Hebbal, Y., Laniepce, S., and Menaud, J. M. Virtual machine introspection: Techniques and applications. In *2015 10th International Conference on Availability, Reliability and Security* (Aug 2015), pp. 676–685.

[19] Huang, H., Hung, A., and Shin, K. G. FS2: dynamic data replication in free disk space for improving disk performance and energy consumption. In *Proceedings of 20th ACM Symposium on Operating System Principles* (2005), ACM Press, pp. 263–276.

[20] Intel. Intel 64 and IA-32 Architectures Software Developers Manual Combined Volumes: 1, 2ABC, 3ABCD. `https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf`.

[21] Johnson, T., and Shasha, D. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1994), VLDB '94, Morgan Kaufmann Publishers Inc., pp. 439–450.

[22] Jones, S. T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. VMM-based Hidden Process Detection and Identification Using Lycosid. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (New York, NY, USA, 2008), VEE '08, ACM, pp. 91–100.

[23] Jujjuri, V., Van Hensbergen, E., Liguori, A., and Pulavarty, B. VirtFSA virtualization aware File System pass-through. In *Proceedings of the Ottawa Linux Symposium* (2010).

[24] Kivity, A., Kamay, Y., Laor, D., Lublin, U., and Liguori, A. KVM: the Linux virtual machine monitor. In *Proceedings of the Linux Symposium* (2007), vol. 1, pp. 225–230.

[25] Lee, D., Choi, J., Kim, J. H., Noh, S. H., Min, S. L., Cho, Y., and Kim, C. S. LRFU: A Spectrum of Policies That Subsumes the Least Recently Used and Least Frequently Used Policies. *IEEE Trans. Comput. 50*, 12 (Dec. 2001), 1352–1361.

[26] Lee, S., Panigrahy, R., Prabhakaran, V., Ramasubramanian, V., Talwar, K., Uyeda, L., and Wieder, U. Validating Heuristics for Virtual Machines Consolidation. Tech. Rep. MSR-TR-2011-9, Microsoft Research, January 2011.

[27] LeVasseur, J., Uhlig, V., Stoess, J., and Götz, S. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation* (Berkeley, CA, USA, 2004), OSDI'04, USENIX Association.

[28] Mandal, S., Kuenning, G., Ok, D., Shastry, V., Shilane, P., Zhen, S., Tarasov, V., and Zadok, E. Using Hints to Improve Inline Block-layer Deduplication. In *14th USENIX Conference on File and Storage Technologies (FAST 16)* (Santa Clara, CA, Feb. 2016), USENIX Association, pp. 315–322.

[29] McDougall, R., and Mauro, J. Filebench, 2005. `http://filebench.sourceforge.net/`.

[30] Megiddo, N., and Modha, D. S. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2003), FAST '03, USENIX Association, pp. 115–130.

[31] Mesnier, M., Chen, F., Luo, T., and Akers, J. B. Differentiated storage services. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 57–70.

[32] Mesnier, M., Ganger, G. R., and Riedel, E. Object-based storage. *IEEE Communications Magazine 41*, 8 (Aug 2003), 84–90.

[33] Meyer, D. T., Aggarwal, G., Cully, B., Lefebvre, G., Feeley, M. J., Hutchinson, N. C., and Warfield, A. Parallax: Virtual disks for virtual machines. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008* (New York, NY, USA, 2008), Eurosys '08, ACM, pp. 41–54.

[34] Nelson, M., Lim, B.-H., and Hutchins, G. Fast Transparent Migration for Virtual Machines. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2005), ATEC '05, USENIX Association.

[35] O'Neil, E. J., O'Neil, P. E., and Weikum, G. The LRU-K Page Replacement Algorithm for Database Disk Buffering. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 1993), SIGMOD '93, ACM, pp. 297–306.

[36] Oracle Corporation. MySQL White Papers. `https://www.mysql.com/why-mysql/white-papers/`.

[37] Payne, B. D., Carbone, M. D. P. D. A., and Lee, W. Secure and flexible monitoring of virtual machines. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)* (Dec 2007), pp. 385–397.

[38] Pfaff, B., Garfinkel, T., and Rosenblum, M. Virtualization aware file systems: Getting beyond the limitations of virtual disks. In *Proceedings of the 3rd Conference on Networked Systems Design and Implementation - Volume 3* (Berkeley, CA, USA, 2006), NSDI'06, USENIX Association.

[39] Pfoh, J., Schneider, C., and Eckert, C. *Nitro: Hardware-Based System Call Tracing for Virtual Machines*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 96–112.

[40] Rodeh, O., and Teperman, A. zFS: A Scalable Distributed File System Using Object Disks. In *Proceedings of the 20 th IEEE/11 th NASA Goddard Conference on Mass Storage Systems and Technologies (MSS'03)* (Washington, DC, USA, 2003), MSS '03, IEEE Computer Society.

[41] Roselli, D., Lorch, J. R., and Anderson, T. E. A comparison of file system workloads. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2000), ATEC '00, USENIX Association.

[42] Rosenblum, M. The Reincarnation of Virtual Machines. *Queue 2*, 5 (2004).

[43] Rosenblum, M., and Garfinkel, T. Virtual Machine Monitors: Current Technology and Future Trends. *Computer* (2005), 39–47.

[44] Schindler, J., Griffin, J. L., Lumb, C. R., and Ganger, G. R. Track-Aligned Extents: Matching Access Patterns to Disk Drive Characteristics. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2002), FAST '02, USENIX Association.

[45] Shamma, M., Meyer, D. T., Wires, J., Ivanova, M., Hutchinson, N. C., and Warfield, A. Capo: Recapitulating storage for virtual desktops. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2011), FAST'11, USENIX Association.

[46] Sivathanu, G., Sundararaman, S., and Zadok, E. Type-safe Disks. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2006), OSDI '06, USENIX Association, pp. 15–28.

[47] Sivathanu, M., Bairavasundaram, L. N., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Life or Death at Block-level. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation - Volume 6* (Berkeley, CA, USA, 2004), OSDI'04, USENIX Association.

[48] Sivathanu, M., Prabhakaran, V., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Improving Storage System Availability with D-GRAID. *Trans. Storage 1*, 2 (May 2005), 133–170.

[49] Sivathanu, M., Prabhakaran, V., Popovici, F. I., Denehy, T. E., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Semantically-Smart Disk Systems. In *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2003), FAST '03, USENIX Association, pp. 73–88.

[50] Storer, M. W., Greenan, K., Long, D. D., and Miller, E. L. Secure data deduplication. In *Proceedings of the 4th ACM International Workshop on Storage Security and Survivability* (New York, NY, USA, 2008), StorageSS '08, ACM, pp. 1–10.

[51] Tarasov, V., Jain, D., Hildebrand, D., Tewari, R., Kuenning, G., and Zadok, E. Improving I/O Performance Using Virtual Disk Introspection. In *Presented as part of the 5th USENIX Workshop on Hot Topics in Storage and File Systems* (Berkeley, CA, 2013), USENIX.

[52] Tarasov, V., Mandal, S., Shilane, P., Jain, D., Kuenning, G., Palanisami, K., Trehan, S., and Zadok, E. Dmdedup: Device Mapper Target for Data Deduplication.

[53] Thereska, E., Ballani, H., O'Shea, G., Karagiannis, T., Rowstron, A., Talpey, T., Black, R., and Zhu, T. IOFlow: A Software-defined Storage Architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 182–196.

[54] Transaction Processing Council. TPC Benchmark H Standard Specification, Revision 2.17.1, 2014.

`http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.17.1.pdf`.

[55] VMWARE INC. VMware VMFS product datasheet. `http://www.vmware.com/pdf/vmfs_datasheet.pdf`.

[56] VOGELS, W. Beyond Server Consolidation. *Queue 6*, 1 (2008), 20–26.

[57] WALDSPURGER, C., AND ROSENBLUM, M. I/O Virtualization. *Commun. ACM 55*, 1 (Jan. 2012), 66–73.

[58] WHITAKER, A., SHAW, M., AND GRIBBLE, S. D. Scale and Performance in the Denali Isolation Kernel. In *OSDI '02* (Boston, MA, December 2002).

[59] YALAMANCHILI, C., VIJAYASANKAR, K., ZADOK, E., AND SIVATHANU, G. DHIS: Discriminating Hierarchical Storage. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference* (New York, NY, USA, 2009), SYSTOR '09, ACM, pp. 9:1–9:12.

[60] ZHOU, Y., PHILBIN, J., AND LI, K. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference* (Berkeley, CA, USA, 2001), USENIX Association, pp. 91–104.

[61] ZHU, B., LI, K., AND PATTERSON, H. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2008), FAST'08, USENIX Association, pp. 18:1–18:14.