Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 EECS Building
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 07-009

Practical Techniques for Eliminating Storage of Deleted Data

David Boutcher and Abhishek Chandra

March 20, 2007

Practical Techniques for Eliminating Storage of Deleted Data

David Boutcher and Abhishek Chandra
Department of Computer Science
University of Minnesota
Minneapolis, MN 55455
{boutcher, chandra}@cs.umn.edu

Abstract

The layered design of modern file systems hides the *liveness* of data from the underlying storage systems. In this paper, we define a generic "purge" operation that can be used by a file system to pass liveness information to the storage system with minimal changes in the layer interfaces. We present three approaches for implementing such a purge operation: direct call, zero pages, and flagged writes. We demonstrate the feasibility of these techniques through a reference implementation in User-mode Linux to dynamically manage a copy-on-write (COW) data store. Performance results obtained from this reference implementation show that these techniques can achieve significant storage savings with a reasonable execution time overhead. Our results demonstrate that passing liveness information across the file system-block layer interface with minimal changes is not only feasible but practical.

1 Introduction

Modern storage systems are implemented in a layered architecture, where the storage subsystem is separated from the file system, with a narrow interface between them. This interface typically supports only simple operations such as "read block" and "write block" at the block layer, and no additional information is provided about the type of data being written. While such a layered approach is desirable for modular design and maintenance of storage systems, the narrow interface between the file system and the block layer also prevents useful performance optimizations [8]. In particular, the block layer has no awareness of the *liveness* [18] of data: whether a given block contains data associated with a file currently existing in the file system. Such lack of liveness information at the block level prevents the storage system from discarding blocks deleted by the file system, often leading to issues with utilization [13], security [10, 12], backing up unnecessary data [14], inefficient caching [2, 15], and migration overheads [4]. To quote [10]:

> The implementations and interfaces of today's operating systems, programming languages, libraries and other software components generally overlook data lifetime issues.

Fundamentally this is another example of the end-to-end system design issue described by Saltzer et. al. [17]

Traditional fixed-sized media, such as hard drives, often do not distinguish between live and dead data blocks. However, more intelligent storage subsystems can make use of this information in several ways:

- A storage subsystem with an intelligent cache hierarchy can ensure that only live, and therefore useful, blocks occupy space in the cache, thus leading to better cache utilization [2, 15].
- A Storage Area Network (SAN) that dynamically allocates data blocks as data is written by clients, can free dead data blocks, thus preserving storage as well as network and I/O bandwidth, which might be wasted on useless data otherwise [13].
- A copy-on-write (COW) storage driver, as described later in Section 3, can ensure that only live data occupies space in the COW copy of the data, thus ensuring small-size COW files. One potential use of smaller COW files can be to substantially reduce the storage requirements for multiple virtual machines sharing identical file systems, and also to reduce their migration overheads [4].
- Even on traditional fixed-sized media, awareness of the liveness of blocks can be used to store rotationally optimal replicas of data in blocks that are otherwise not in use [24].
- From security perspective, removing deleted blocks can be useful for physically purging the data of deleted files to prevent it from being retrieved later [12].

Some techniques have been proposed for passing liveness information to the block layer [18, 15, 5].

These techniques have typically involved either significant changes to the block layer interface used in the industry, or are very specific to the structure of the data, such as the file systems, stored on the block device. These limitations have made these techniques impractical for wide-spread adoption into existing storage system interfaces. The goal of this paper is to investigate practical techniques for passing liveness data that overcome some of the limitations of the existing techniques, and show their utility in a real implementation.

In this paper, we define a generic *purge* operation that can be used by a file system to pass liveness information to the storage system with minimal changes in the layer interface. This operation is designed to be back-compatible, so that file systems and block devices that do not support this operation can continue to function as expected. To instantiate this purge operation in a real system, we investigate three practical techniques: *direct call*, *zero blocks*, and *flagged writes*. Direct call provides an explicit call that a file system can make to the block layer to indicate that a block is dead. Zero blocks allows a dead block to be indicated by writing zeros to it. Finally, flagged writes use a special flag to indicate a block's liveness status.

We demonstrate the feasibility of these techniques through a reference implementation of a dynamically re-sizable copy-on-write (COW) data store in User-mode Linux (UML) [7]. Performance results obtained from this reference implementation show that we can recover virtually all the blocks occupied by deleted data (95% to 99.7% across our benchmarks) as compared to recovery of no blocks in the default UML COW implementation. Further, these techniques showed reasonable execution time overhead ranging from 2% to 26% across the different benchmarks. These results demonstrate that *it is not only feasible but also practical to implement techniques for passing liveness information in a storage system.*

While we focus on using liveness information in support of more efficient storage of file systems, the same techniques could be used by any interface to the block device. Many database systems, for example, interact directly with storage devices at the block layer [15]. These same techniques can be applied to such database systems, to indicate to the storage layer when a record or blob has been deleted.

The rest of the paper is organized as follows. We present the purge operation and the various techniques for its instantiation in Section 2. The reference implementation of a dynamically re-sizable COW storage system is described in Section 3, followed by its experimental evaluation in Section 4. We present related work in Section 5, and conclude in Section 6.

2 Passing Liveness Information

The narrow, and fairly standard, interface between the file system layer and the block layer simplifies development of both new file systems and new storage devices. For example, the ext2 [3] file system runs identically on IDE, SCSI, and loop-back file systems. Similarly, new file systems, with semantics such as journaling [23], or log-based file systems [16] have been developed without requiring changes to underlying block devices.

Any change to pass liveness data between the file system and the block layer should be made in a way that minimizes changes to the interface. In addition, to support backward compatibility, such a change should not *require* that all storage devices or file systems be modified. Thus, to be practical, any mechanism designed to pass liveness information between the file system and the block layer must satisfy the following key requirements:

1. The changes to the file system-block layer interface must be minimal.
2. Normal I/O operations such as reads and writes must remain unchanged. For instance, writing a block of data and subsequently reading it must give the same data.
3. Block devices must continue to support any file systems that do not provide liveness information, and file systems must function correctly on block devices that do not support the purge operation.

2.1 Purge Operation

To meet the above requirements, we define a *"purge"* operation to be provided by the file system. We define the semantics of the purge operation as follows:

1. The purge operation identifies blocks whose data is no longer alive.
2. Purging a block of data results in subsequent reads from that block returning zeros.
3. Purging a block of data does not affect subsequent writes to the block.
4. Purging a block does not impact any read/write operations on other blocks in the storage system.

By defining the purge operation in this manner, we meet the practicality requirements as follows. First of all, the purge operation is only a single operation added to the file system-storage system interface. As we will describe below, this purge operation can be instantiated in different ways to minimize the breakdown in the transparency of the file system and storage system layers. Thus, it

requires minimal changes to the interface. Second, the purge operation maintains the expected semantics of all reads and writes. In particular, any reads and writes to the purged block are in essence same as reading or writing to an unused block, while reads and writes to all other blocks remain unaffected. Third, the purge operation is provided mainly as a hint to the storage system, which does not require it for the management of data blocks. If a file system does not provide the purge operation, it will continue to function as before, with the storage system working without the liveness information. Similarly, if the block layer does not support the purge operation, the file system will function correctly.

2.2 Techniques to Implement Purge Operation

We examine three techniques to instantiate the purge operation as defined above. All these techniques are designed to implement the purge semantics as described above.

Direct Call The block device driver provides a direct call interface that the file system layer can use to identify dead blocks. This is the most straightforward way to implement the purge operation, however it bypasses many layers of the operating system.

Zero blocks The file system zeros dead blocks. The block layer detects blocks that consist only of zeroes and treats them as dead. A special case of this technique is to use *zero pages* for zeroing dead blocks. Many operating system implementations have one or more pages explicitly set to zero. By issuing write operations through the block layer referencing the zero page, the block device can identify dead pages without requiring that every byte of the block be set to zero.

Flagged Writes Flags are typically provided on I/O operations through the block layer. Additional flags, indicating dead data, can be defined, so that the file system can set these flags for a dead block to indicate it should be purged.

We now describe each of these techniques in more detail, presenting the implementation issues, as well as the pros and cons associated with each of them.

2.3 Direct Call

Perhaps the simplest technique for passing additional information between the layers is the addition of a new function call to the list of calls exported by the block layer. In the Linux kernel[1] all block devices export the following structure:

```
struct block_device_operations {
   int (*open)(...)
   int (*release)(...)
   int (*ioctl)(...)
   long (*unlocked_ioctl)(...)
   long (*compat_ioctl)(...)
   int (*direct_access)(...)
   int (*media_changed)(...)
   int (*revalidate_disk)(...)
   struct module *owner;
};
```

This structure provides a set of primitive functions (open, release, ioctl, etc.) supported by the block level. The simplest method of declaring a block "dead" is to implement an additional function, such as

```
int (*purge_sectors)(struct block_device *,
                     sector_t sector,
                     unsignedlong *length)
```

that can be used by the file system. The function takes a pointer to a the structure describing block devices, a starting sector, and the number of sectors to mark as "dead". To support compatibility with block devices that may not implement the new function, the file system can check whether the function pointer is valid before calling it. The behavior for block devices not implementing the function will be the same as their default behavior.

```
if (fops->purge_sectors)
    fops->purge_sectors(bdev,
                        sector,
                        num_sector);
```

2.3.1 Advantages and Disadvantages

The primary advantage of the direct call approach is its simplicity. By adding a single, architected call to the block device operations structure, the file system can inform the block layer of deleted blocks. The disadvantages of this approach are the layering and ordering violations it introduces.

In the current Linux system, for example, the file system does not deal directly with the block layer. Rather there is a page cache in between. The file system deals with pages, and the page cache takes care of asynchronously scheduling I/O to the physical devices. The file system marks a page as "dirty", and relies on the page cache to queue and schedule the write operations to move the page to disk. Even in the case of synchronous I/O, the file system relies on the page cache to manage I/O operations. By introducing a direct call from the file system

[1] All references to the Linux kernel in this paper will be made to the 2.6.17 kernel.

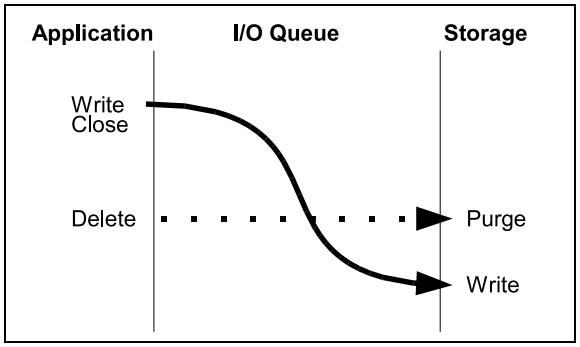


Figure 1: Direct Call ordering example

to the block layer, an architecturally difficult layer violation is introduced. Additionally, since I/O is queued up between the file system and the block layer, the file system does not know, at the time the block is purged, what other I/O operations may or may not have completed.

The situation is made worse by the fact that the purge operation, which may result in block device I/O operations, should not be a synchronous operation. In the case of cache management with an intelligent adapter, the purge operation may require an I/O operation to the adapter. In the COW device implementation described in section 3, a purge operation may result in multiple data I/O operations as data is moved around in the COW file. The file system should not stop dead while a long-running synchronous operation is performed. This requires that any I/O resulting from the purge operation be enqueued, presumably in a queue internal to the block device. The block device must then ensure that the integrity of the file system data is maintained while processing two asynchronous queues of operations (i.e. not incorrectly purging data.)

If a write is followed by a purge, there is no data corruption issue if the operations are re-ordered, and the only consequence is that dead data will be written to the media. Figure 1 shows such a situation, where the curved solid line shows I/O queued typically, and the dashed line shows the direct call bypassing any I/O queueing. This does not introduce correctness issues, but may be problematic if, for example, the goal of liveness detection is purging data for security reasons.

There is a much greater problem if the purge is incorrectly scheduled *after* the write, since valuable data may be lost or if a purge operation is incorrectly scheduled ahead of a read operation, since the read will not correctly return data. It is unclear what file system semantics would result in a such operations being concurrently scheduled. Further, some I/O operations are treated as "barrier" operations to enforce certain I/O ordering. A direct call from the file system to the block layer ignores scheduling constraints placed by barrier operations in flight.

In our reference implementation of direct call (described in Section 4), the call to the block layer results in purge operations being inserted at the tail of the queue of scheduled I/O operations (i.e. operations already scheduled at the block layer by the page cache.) The purge operation will occur prior to any subsequent write operations. This ensures that while the purge may skip ahead of a write operation (write:purge gets re-scheduled as purge:write), the inverse well never occur.

Our implementation currently ignores the issue of barrier operations. Barrier operations in a file system are typically required to ensure that the data on the storage device is always consistent, especially in the case of a system failure during a sequence of operations [9][6]. Ignoring barrier operations allowed for a simpler implementation that enabled the benchmarking described in section 4, however it could result in inconsistent on-disk data if a system failure occurs. One such scenario involves the deletion of a file, where the file system first updates its meta-data to indicated that the file is deleted and the blocks are freed, then issues the direct-call purge operation to remove the blocks. If the purge operation occurs *before* the meta-data gets written to disk, a window exists where the data blocks have been purged from the storage device before the file system meta-data is updated to indicate that the file is deleted. In the reference implementation this would result in a file that contains all zero blocks. This scenario highlights the risks involved in bypassing operating system layers to implement a new function.

2.3.2 Using IOCTL for the Direct Call

An alternative to adding the function pointer to the block device operation structure would be to implement the call as an `ioctl` call. There are some further advantages and disadvantages to this approach.

An advantage is that the ioctl can be used from user programs, allowing increased management of the block device from administrative programs. One example of this is formatting a file system using a user program such as the Linux "mke2fs" program. Such programs, when initializing a block device, could inform the block layer of all the dead blocks (in the simple case, performing a purge starting at the beginning of the disk and extending to the end before initializing file system meta-data structures.) A disadvantage to the ioctl approach is the lack of architecture to the interface. The ioctl function is difficult to validate at compile time, and the data passed through in an invalidated form. There are very few ioctl() calls made from one part of the Linux kernel to another for

these reasons.

2.4 Zero Blocks

A brute force method for indicating dead data is to always write blocks of zero to any area of the file system that does not contain live data. Existing file system implementations already use this technique, primarily to ensure that deleted data is not recoverable.[2] The block layer then examines blocks of data written, and if they are zero, treats them as "dead" data.

It is critically important that any read to a block that has previously been written with zeros subsequently return an identical block. This can be problematic if a block of zeros is used to indicate that the storage layer can purge, and not store, the block of data. Subsequent read operations must return zeros even if the contents of the block are not physically stored.

2.4.1 Advantages and Disadvantages

The advantage of the zero block approach is that it requires no changes to the file system-block layer interface, and block devices that do not treat zero blocks specially will be semantically identical to devices that treat zero blocks as purged. A further advantage of the zero block approach is that it can be used in any component of the I/O infrastructure. For example, if data is being stored on a storage area network (SAN), the device drivers on the host system need not perform the zero block detection or be aware of the purgability of data. The data can be passed all the way across the SAN fabric to the SAN controller, which could perform the zero detection.

The main disadvantage with this technique is that the file system must physically write zero to each byte of a dead block, and the storage layer must physically examine each byte in the block to see if it is zero. This is a serious performance penalty, particularly if the data must all be loaded into the processor's cache. In the case of a write containing non-zero data, it is likely such a test will detect non-zero data early in the block. In the case of zero data, however, the block layer must examine the full block before concluding that it is purgable.

2.4.2 Using Zero Page Mapping

A more elegant alternative to the zero block approach is to take advantage of the fact that many operating systems maintain one or more pages whose contents are always zero. These pages are used in any case that a zero-initialized block of memory is required (such as the BSS segment of a program.) Such pages are often COW memory pages, that will be mapped to a unique writable page on the first write operation.

When the file system wishes to delete a block in a file, it can re-map the page in the page cache to the zero page and issue the appropriate write. The block layer then detects write operations whose data is coming from the address range of the zero page and recognizes those as zero data as in the zero block case. Even in the case that the zero block detection occurs on some remote device this approach may have some performance advantages, since the system need not write zeros to every byte of the blocks to be purged.

2.5 Flagged Writes

Perhaps the architecturally cleanest approach to passing liveness data between the file system and the block layer is to introduce a new "flag" that accompanies the I/O request indicating that the I/O is purging data. There are a number of flags already in the Linux kernel, such as REQ_FASTFAIL, that indicates an I/O should not be retried, and REQ_HARDBARRIER that indicates other I/O operations should not be re-scheduled around this one. Similarly, to indicate a purge operation, a new flag, REQ_PURGE can be defined and set by the file system, and recognized by the block layer. It is important to note that since the file system deals with page through the page cache, the purge flag is set on all *pages* to be purged. Those pages then migrate out to disk through normal I/O mechanisms and are purged from, rather than written to, the media when they reach the block layer with the flag set.

2.5.1 Advantages and Disadvantages

The main disadvantage of a new I/O flag is that it is a more invasive change than those previously described. The flag (in the case of Linux) must be defined in three different kernel structures:

- The buffer_head struct
- The bio struct
- The request struct

The buffer_head structure is the primary link to the page cache updated by the file system. The page cache then creates *bio*s, which in turn are grouped into requests. A flag indicating purgable data must be set by the file system in the buffer head, and that information in turn passed along through the other layers. An additional concern is the way I/O operations are merged. Multiple bio operations are merged into a single request. Purge

[2] As an example, see the zero-free patch to the Linux kernel at http://intgat.tigress.co.uk/rmy/uml/linux-2.6.16-zerofree.patch

and non-purge I/O operations should not be merged together lest the purge bit be applied to an incorrect I/O or dropped from a purge I/O. Finally, since the pages are flagged as purgable, they must be scheduled to be written out by the block layer, resulting in additional I/O operations.

The main advantage of the flagged write approach is that it is architecturally clean. The purged I/O operations are scheduled along with other I/O operations, and respect ordering with operations such as barriers. If the purging operations are performed in storage components such as a SAN, the flag must be added to the interface to the SAN controller (e.g. to the fiber protocol) in order to be effective.

2.6 Comparison of the Techniques

There is one significant difference in the approach of the three techniques. The direct call technique provides a very efficient interface to the block layer, at the expense of violating architected layering. The interface provides a starting sector and range to be purged, and that call is the only new invocation of the block layer in support of purging data. The zero block and flagged I/O techniques, on the other hand, require that all pages associated with the purged data be marked in the page cache. The zero block technique marks them by filling the pages with zeros whereas the flagged I/O technique marks them by setting a flag on the page. Both the zero block and flagged I/O techniques cause significantly more I/O operations to occur between the page cache and the block layer, since all those marked pages must then be scheduled to be written out by the block layer.

3 Implementing a Liveness-Aware Copy-On-Write Storage Subsystem

To explore the above described purge techniques in a practical environment, we implemented a dynamically resizable Copy-On-Write (COW) storage system that uses each of these techniques to pass liveness information to the block layer. We first give a background of COW storage, followed by a description of our implementation.

3.1 Background

3.1.1 Copy-On-Write

Copy-On-Write (COW) is a lazy optimization mechanism for data storage that maximizes shared copies of data for as long as possible. Multiple users of a piece of data all reference the same copy of the data, with a private copy being created only when a user attempts to modify the data. This mechanism is typically implemented transparently to users of the data.

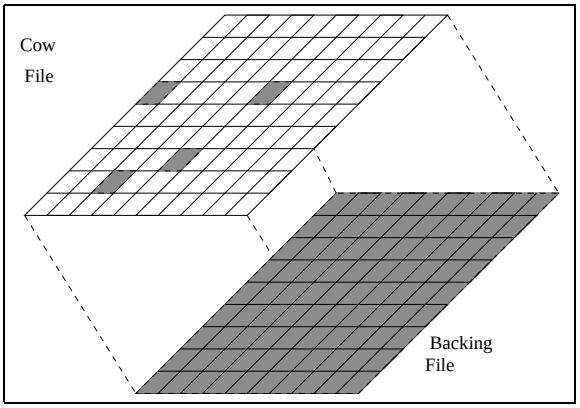


Figure 2: COW storage

Copy-on-write is used for disk storage to reduce the storage required when multiple similar copies of a file system are created. Two common examples of this are multiple similar virtual machines executing within a single physical machine [21], and "snapshot" copies of file system state. In both of these cases, the original disk storage remains unaltered. Any updates by the file system are made to separate, smaller, disk storage that contains only the altered blocks. In the rest if this paper, we refer to the original, immutable, storage as the "backing data", and the storage containing only the changed regions of the data as the "COW data". Figure 2 illustrates a COW storage system. The figure shows the allocated regions of COW file in gray. The COW file only allocates storage for the blocks of the the backing file that have been written to, while the unmodified blocks are read from the backing file.

One fundamental attribute of existing COW storage is that it is *grow only*. In other words, the COW data increases in size as blocks are written to the storage, but it never decreases in size. The prototypical example would be copying a 1GB file to a file system backed by COW storage, and then deleting the file. 1GB of blocks would be written to the COW file, but would not be freed when the file is deleted.

3.1.2 Sparse File COW Implementations

Sparse files are used to reduce storage requirements by formalizing the assumption that files are initially filled with zeros, and therefore data blocks need only be allocated when new, non-zero data is written to regions of the file.

In a Posix file system, for example, a sparse file is created by opening the file, seeking to some, potentially

large, offset in the file, and writing data at that offset. The size of the file, as recorded by the file system, will be the byte last written. Data blocks are not allocated on the media, however, for all the intermediate data. A read to any region of the file not previously written to, will return zero.[3]

The use of sparse files makes for a very elegant implementation of COW data store. A sparse COW data file is created, with the same size as the original backing data. Assuming that the file is created by writing single zero byte at an offset equal to the size of the backing file, the COW file nominally occupies a single byte of storage. [4]

In addition, a bitmap is maintained indicating which blocks of the COW store have been written to. Initially the bitmap is all zero. There is one bit for each "region" of the backing data. In many implementations, a region is considered to be a standard 512 byte sector, though other region sizes can be used. There is, therefore, overhead of 1 bit for each 512 bytes of the backing data store. This works out to an overhead of 0.024%, which is very acceptable for most applications.

As writes are made to the COW data store, the writes are made to the COW sparse file, at the same offset as the original file. Writes to the sparse file cause those regions to be allocated, however regions not written to remain sparse. The bitmap is updated whenever a new region of the file is written to. When read operations are performed, the COW data store examines the bitmap to determine if the read should be made from the COW sparse file, or the original backing file.

Since the bitmap of used blocks is an integral part of the data store, it is often stored at the beginning of the COW file, and the offsets of I/O operations done to the COW file are adjusted to account for the size of the bitmap at the beginning of the file.

Limitations of Sparse File COW Storage

The most significant limitation to sparse file COW implementations is that sparse files do not have an interface for making a region of the file sparse again once it has been written to. In other words, once a region has been written to, and storage allocated for that region, even writing zeros back to that region will not deallocate the space. The allocated size of sparse files can only grow. [5]

This limitation on deallocating storage is significant for COW implementations that allow for data to be marked "deleted", and therefore no longer requiring space in the COW store. Techniques do exist for shrinking sparse COW files. The existing techniques, however, are limited by the inability to add holes back into the sparse file. They rely, therefore, on first writing zero blocks to any non-allocated blocks in the file system, and then copying the COW file to a new file, skipping over any zero regions. Moreover, this resizing operation can only be performed when the COW file is not in use, preventing the use of these techniques in an online manner.

3.2 Dynamically Resizable COW Store

A dynamically resizable COW store supports growing as data is written to the store as well as shrinking as data is deleted from the store. The dynamic COW file adjusts its size while the file is in use. Implementing a dynamic COW file requires interfaces from the file system to determine the liveness of data, and is thus a very suitable implementation for testing the techniques described in the previous section.

The management of storage in a resizable COW file is a significant issue by itself. As data is purged from the COW file, generally from the middle of the file, the remaining data needs to be re-arranged to reduce the size of the file. Techniques for this kind of storage management exist in other computer science disciplines [11], and are very similar to the management of a database file where records are added and deleted arbitrarily from the file. As the focus of our implementation was to explore the liveness issue, rather than data management, we employed a simple technique to manage storage in the file. As a data block is purged from the COW file, the data at the end of the file is immediately moved to the purged area, reducing the size of the COW file. This technique has the advantage of simplicity, but it can lead to inefficiencies. If a large amount of data is being deleted from a COW file, for example, it is likely that the moved data will be deleted very shortly. More efficient mechanisms, involving lazy deletion, for example, would likely improve the performance of the system.

3.2.1 Resizable COW Data Structures

Our implementation of resizable COW file uses an "indirect map" that maps any changed blocks in the backing data file to blocks at some arbitrary location in the COW file. This indirect map is used in conjunction with a bitmap of allocated COW blocks that keeps track of which blocks have been changed from the backing file. The indirect map allows changed blocks in the backing data to be stored at any location in the COW data. Additionally, blocks in the COW data can be moved as long

[3] Interestingly, the Windows NTFS file system allows the default value returned for sparse regions to be set to a value other than zero.

[4] In practice, the COW file will likely consume one block in the file system, depending on how the file system deals with writes of less than a block size. That is file system dependent.

[5] Peter Braam (braam@clusterfilesystem.com) created a prototype syscall in Linux, named "sys_punch" to punch holes in a file. In other words, return regions of a file to a sparse state. This syscall was never accepted into Linux.

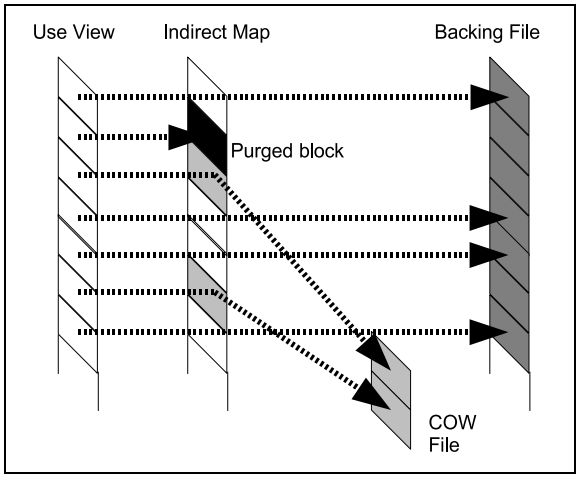


Figure 3: Indirect Mapping Data-structure

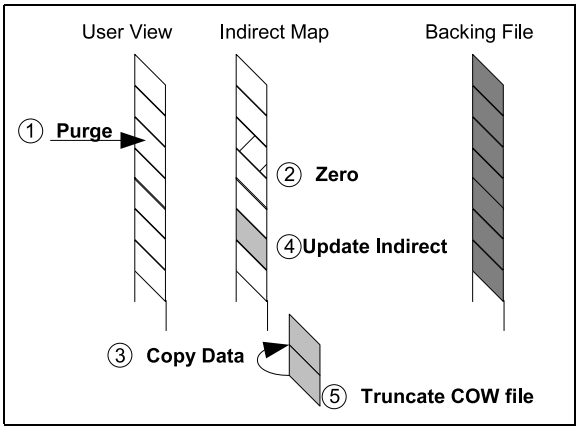


Figure 4: Purging a block

as the indirect map is updated. This feature is required to shrink the COW file as blocks are deleted, since shrinking the COW file may require rearranging the remaining blocks. Figure 3 shows the indirect mapping of storage regions in the COW file. The indirect map is only looked up if the bitmap indicates that a COW block exists. This in turn allows for the recording of purged blocks, whose contents should be returned as zero. Any block with a value of one in the bitmap, but a value of zero in the indirect map is considered a purged block and will return zeros on subsequent read operations.

When an I/O operation is performed, the block number of the I/O is first used as an index into the bitmap, and then used as an index into the indirect mapping entry array. If the indirect map entry is non-zero, the indirect map indicates the block number in the COW file that contains the current copy of the block of data. Write operations to a block that does not currently have a copy in the COW data require the allocation of new space in the COW file. New allocations are always made at the end of the COW file, and the block where the data was written is stored in the indirect map. Read operations either are directed to the backing file, if the bitmap entry is zero, or to the COW file, if both the bitmap and the indirect map entries are non-zero. If the bitmap entry is non-zero but the indirect map entry is zero, it indicates a purged block and the read returns zeros.

The memory overhead of this COW store is significantly larger than the bitmap used in the sparse file implementation described in section 3.1.2. Assuming that the indirect map stores the offset in the COW file in "blocks", or sectors, the overhead is dependent on the size of the block. The size of the index directly affects the maximum file size supported by the COW implementation. For a sector size of 512 bytes, a 32 bit index will support file sizes up to 2TB. The overhead of that implementation is 4 bytes for each 512 bytes of data, or 0.8%.

Using this implementation, the memory overhead is constant regardless of how much data is written to the COW store. More complex and dynamic allocation schemes, such as a tree, a hash table, or a multi-level index are also possible. These schemes could have the advantage of growing the overhead associated with the COW store proportionally to the amount of data written to it. The current implementation of dynamic COW files always keeps a copy of the indirect map in memory. A more scalable approach would cache portions of the indirect map in memory without requiring that the whole map be resident.

3.2.2 Purging Blocks from the COW File

When a block is purged from the COW file, the COW file should shrink. In order to implement that semantic, the last block in the COW file is moved to the purged block, and the appropriate indirect maps updated. The indirect map for the purged block is set to zero, and the indirect map for the moved block is changed from the last block in the file to the deleted block. This ensures that the COW file is always "packed". The ordering of these operations is very important to preserve ensure the correctness of the data in the event of a failure during the operation. Figure 4 shows the mapping of storage regions in gray. The ordering must be:

1. file system calls purge
2. write 0 to purged indirect map entry
3. Copy data block from end of file to the purged block
4. Update the moved block's indirect entry

5. Truncate the file

There are advantages and disadvantages to keeping the COW file packed. One advantage is that no "utilization" bits need be kept and managed. All blocks in the COW file are always used. A disadvantage, though, is that purge operations are expensive and synchronous. When a block is purged, a block must be read and written, and the indirect map must be updated.

4 Experimental Evaluation

4.1 Implementation and Experimental Setup

We have implemented a reference implementation of our dynamically resizable COW storage system in User Mode Linux (UML) [7], UML is a virtual machine implementation of Linux that allows a version of Linux to run as an application on top of an underlying Linux system. We chose UML for our implementation for two reasons. Firstly, it has an existing grow-only COW implementation against which the resizable COW implementation can be compared. Secondly, it provides a simple environment for developing and testing new Linux kernel code. A 500MB file system was created and an ext2 [3] file system created on it. The Linux system running the benchmarks had the "zerofree" patch applied, allowing the ext2 file system to write zeros to deleted blocks for the zero detection algorithms.

The system on which the benchmarks were run was an AMD Athlon(tm) XP 2600+ with 1GB of memory. A 2.6.17 Linux kernel was used for both the host system as well as the UML kernel.

We used the following benchmarks to analyze the performance of our dynamic COW implementation:

- *Linux build:* This benchmark performed a make and clean of Linux source code. This benchmark was chosen since building the kernel involves the creation of a large number of files with a significant amount of data, and performing the "make clean" operation causes all the files to be deleted again.

- *dbench:* dbench [22] is an I/O benchmark that simulates the I/O load produced by the netbench benchmark, without the network component. It creates, populates, and deletes files.

- *bonnie:* bonnie [1] is a Linux I/O benchmark that performs a number of I/O tests, including reading/writing data and creating/deleting files.

We implemented three versions of the dynamically resizable COW system, using each of the three purge techniques described in Section 2 (namely, direct call, zero page, and flagged writes) to pass liveness information from the file system to the COW layer. We ran our benchmarks on these three versions and compared the results to those obtained on the default UML COW implementation.

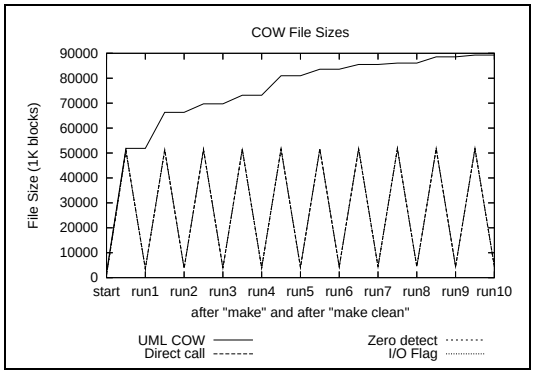


Figure 5: Linux build COW file sizes.

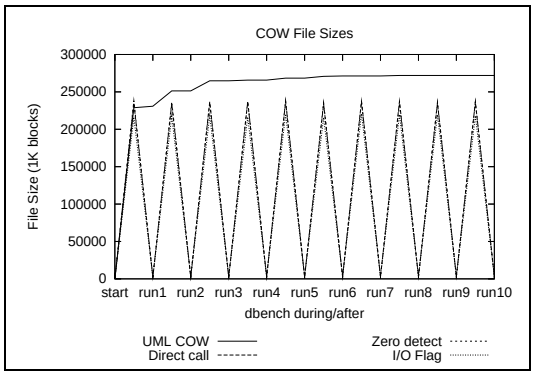


Figure 6: dbench COW file sizes.

4.2 Storage Savings

The main reason for implementing a resizable COW store is to minimize the amount of storage used. Here, we present results in the storage savings achieved by the various purge techniques in the resizable COW implementation.

Figure 5 shows the size of the COW file (in KB) for the Linux build benchmark. The figure shows the size of the COW file at two points in each run: the size of the COW file after the compile, and the size of the COW file after a "make clean" command to delete the resulting binaries. The top line in the graph correspond to the default UML COW, while the other curves corresponds to the three resizable COW implementations (these curves are so close that they are indistinguishable). It is clear from

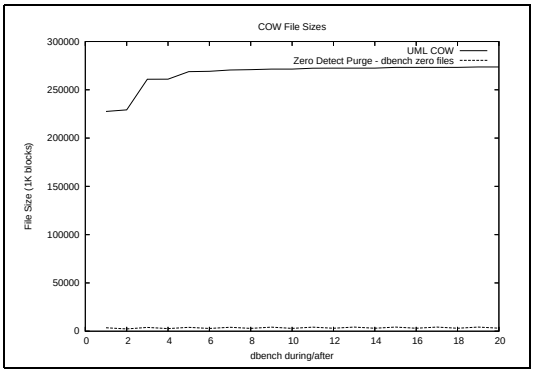


Figure 7: Behavior of zero page purge implementation with the default dbench benchmark.

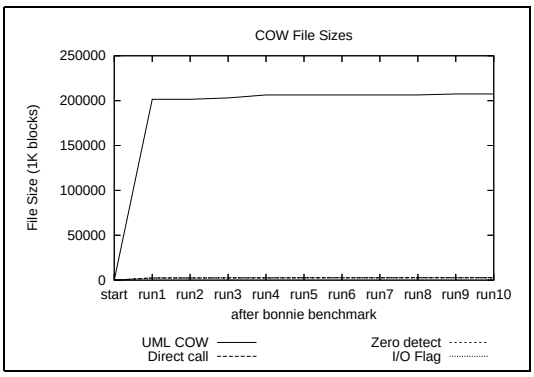


Figure 8: bonnie COW file sizes.

this graph that the resizable COW implementation provides significant storage benefits. In the existing COW implementation, the storage occupied by the COW file continues to grow, even after the "make clean" operation deletes the binaries produced by the kernel compile. The resizable COW implementation, by comparison, grows and shrinks on demand. After 10 cycles of compile and clean, the resizable file is only 5% the size of the UML COW implementation (4MB compared to 87MB).

Figure 6 shows a similar graph for the dbench benchmark, with snapshots taken 20 seconds into each benchmark run, and at the end of each run. Once again, we can see that resizable COW implementations are able to reclaim the storage for all the deleted data, and by the end of each run, the COW file is at its minimal size.

We noticed an interesting aspect of dbench during our experiments. By default, the dbench benchmark writes only zeros to all its data files. Using this default behavior with the "zero page detection" approach results in the COW file never growing at all as illustrated in Figure 7: the barely visible line at the bottom of the graph is the size of the COW file using zero page detection. We in fact modified the dbench benchmark slightly to store one non-zero byte in each 512 byte block to produce the results shown in Figure 6.

Finally, the COW file size for the bonnie runs is shown in Figure 8. The figure shows only the size of the COW file *at the end* of each benchmark run. We did not measure the file size during the runs, because bonnie executes a number of phases (a block I/O phase, a file seek phase, a file creation phase, etc.), making it difficult to determine a fixed point during the benchmark where files are created/deleted. However, as can be seen from the figure, the size of the COW file remains minimal (it is the barely visible curve very close to the x-axis), with a saving of 99.7% over the default UML COW after 10 runs of the benchmark.

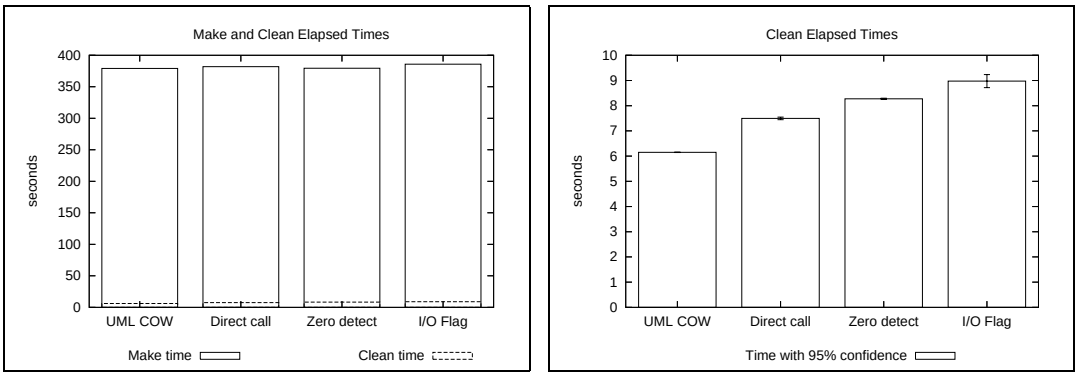
These results demonstrate that *purging deleted data based on liveness information passed from the file system has significant savings in terms of storage.*

4.3 Performance

Next we look at the performance of the various purge techniques, and the overheads involved in resizing the COW file using these techniques. Figure 9(a) shows timing comparisons for building the kernel and subsequently deleting the binaries. As can be seen from the figure, the overall compilation and clean time of the resizable COW implementations was comparable to the UML COW implementation. The wall-clock time to build the kernel was increased by only 1% in the worst case (the "flagged purge" approach) and was statistically identical for the other approaches. The overall time for the make/clean cycle is dominated by the compilation, which is primarily a CPU-intensive operation. Since the file deletion is much more I/O-intensive, we show the performance for cleaning the binaries separately in Figure 9(b). As shown in the figure, the time to delete the binaries (make clean) was increased by 46% in the worst case (flagged purge), while the aggregate of both make and clean operations is increased by only 2% (6 minutes, 26 seconds compared to 6 minutes 19 seconds.) This level of overhead seems reasonable given the storage savings achieved.

As mentioned above, the kernel compilation has a significant CPU component to its performance in addition to I/O. The dbench benchmark was used to analyze the overhead of the resizable COW implementation in a much more I/O intensive environment. Figure 10(a) shows the average throughput achieved over 10 runs of dbench. As can be seen from the figure, the performance of direct call purge is statistically identical to that of UML COW, while those of zero-detect purge and flagged purge are 41% and 39% less respectively.

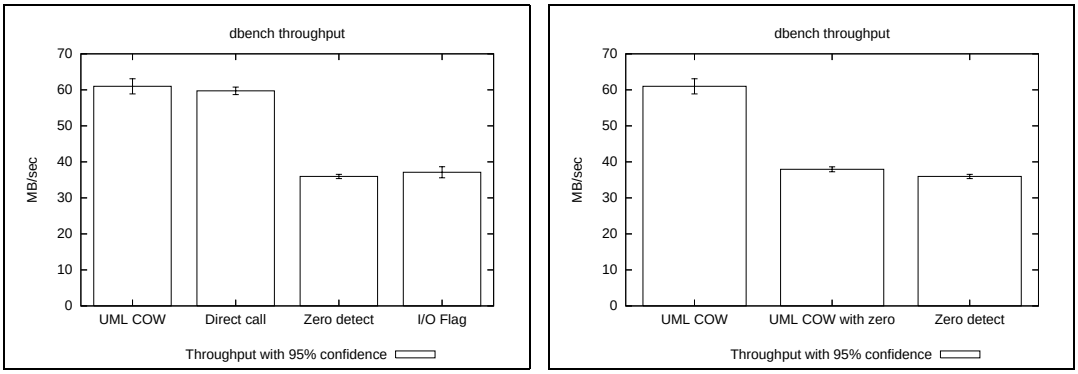
Figure 10(b) explains why the zero-detect purge may



(a) Make performance

(b) Clean performance

Figure 9: Performance of Linux build benchmark.



(a) dbench

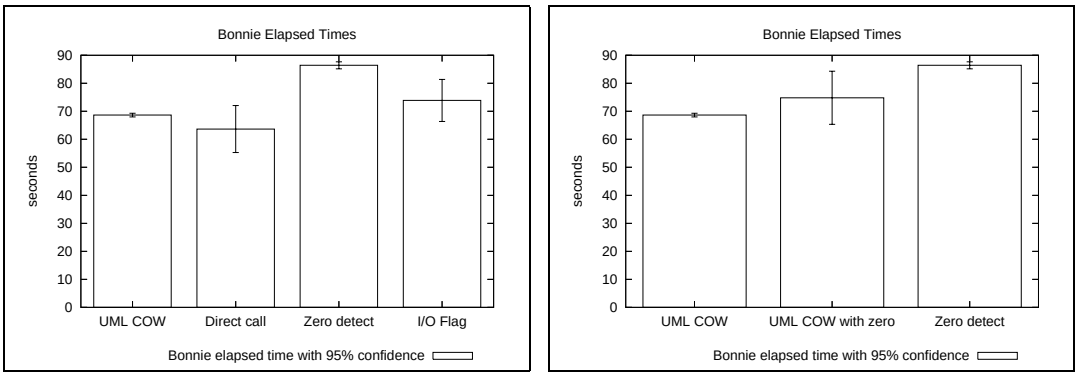
(b) dbench with zero on delete

Figure 10: dbench throughput.

be doing so poorly. The figure shows the impact of writing zeros to all deleted blocks. It compares the dbench throughput of UML COW with a UML COW version implementing zero writes to deleted blocks (without purging and resizing), and the zero-detect purge implementation. As can be seen from the figure, the performance of the UML COW with zero writes is comparable to that of the zero-detect purge, indicating that most of the overhead is occurring due to the writing of zeros to the deleted blocks.

The degradation in the performance of flagged purges is because of the extra I/O operations it generates. We hypothesize that there is also a degradation due to the fact that it prevents coalescing of non-flagged operations with flagged operations, as explained in Section 2.5. Thus several extra operations are performed which would not have been done if they could have been coalesced.

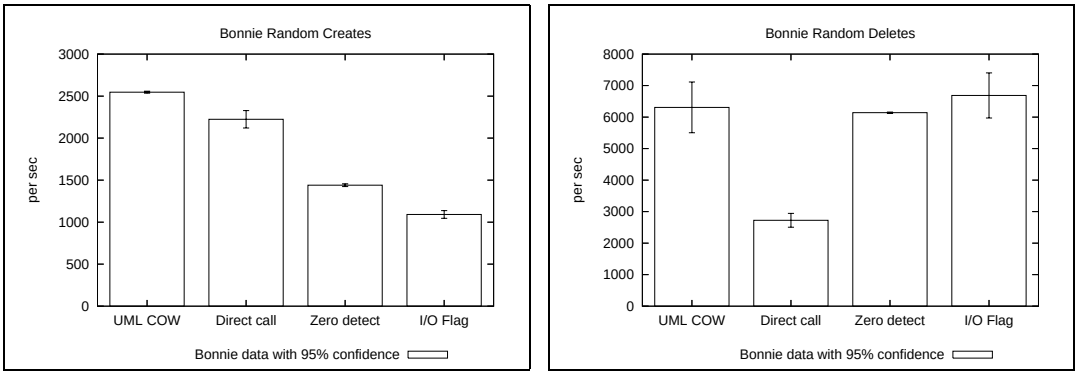
Figure 11(a) shows the elapsed time to complete the bonnie benchmark, which is a multi-stage I/O-intensive benchmark measuring a number of I/O functions such as sequential and random reading/writing, creating/deleting files, seeking, etc. Note that while on the dbench throughput graph, higher is better, in the bonnie elapsed time graph, lower is better. As seen from Figure 11(a), the elapsed time for direct call purge is again comparable to that for UML COW, while those of zero-detect purge and flagged purge are 26% and 8% higher respectively. Figure 11(b) shows the impact of writing zero blocks on deletion, and, we can see that, as with dbench, the addition of writing zeros to all deleted blocks adds most of



(a) Bonnie

(b) Bonnie with zero on delete

Figure 11: Bonnie elapsed time



(a) Creates

(b) Deletes

Figure 12: Comparison of the purge techniques for file creations and deletions.

the overhead.

The differences between the three approaches are highlighted by Figures12(a) and 12(b), which show the number of creates and deletes respectively that bonnie performed per second. Note that using the direct call purge approach has very good performance on creates, but much worse performance on deletes, as might be expected since on a file deletion a direct and immediate call is made to the block layer. This adds overhead of resizing the COW file at the time of the delete operation itself. The flagged purge approach, on the other hand, shows much better performance on deletes, but poor performance on creates. This is because when a file is deleted, the relevant pages are just flagged for purging, but the purge operation itself does not occur until the marked pages naturally migrate to disk. The poor performance on creates can be attributed to the additional I/O operations already in the queue.

The two measurements focused most closely on the cost of the delete operation itself are the “make clean” times, shown in Figure 9(b) and the bonnie delete measurements, shown in Figure 12(b). Those two measurements agree that the direct call approach has the worst performance, and the flagged purge approach has the best performance. On more mixed workloads, however, such as overall dbench throughput or the bonnie overall elapsed time, the ordering is reversed with direct call providing the best performance. We hypothesize that this is because the flagged I/O approach affects the overall I/O scheduling and merging of other I/O operations, whereas

the direct call approach restricts its impact to the delete operations themselves. Finally, it is difficult to separate the overhead of writing zeros to deleted buffers from the other overhead of resizing COW files. However, Figures 10(b) and 11(b) indicate that writing zeros to blocks has a significant impact on the performance of the zero block detection technique. This overhead can perhaps be reduced by using mechanisms such as zero page mapping, as described in Section 2.4.2.

Overall, we see that *the purge operations have acceptable performance* particularly given the amount of storage savings. Moreover, each of these purge operations may be suited for a different environment based on what I/O operations are expected to dominate the workload in that environment.

5 Related Work

[19] also addresses the narrow interface between the file system and the block layer and advocates creating "semantically-smart disk systems" that can understand the file systems above them to make more intelligent storage decisions. They state that changing the interface to storage "is fraught with peril, requiring broad industry consensus, and massive upheaval in existing infrastructure." While the approach described in our paper addresses only the liveness of data blocks, we believe we have demonstrated that it is practical to make minimal interface changes in support of new and more intelligent storage behaviors.

[18] presents techniques to determine the liveness of data blocks, classified as explicit and implicit. Their explicit detection approach is similar to our direct call interface, without necessarily addressing the scheduling difficulties associated with calling the block layer from the file system. The implicit detection technique relies on the block layer making inferences about the liveness of data without any direct interaction with the file system. While the implicit detection approach is the least non-intrusive, there are a number of limitations to it. The first is that the storage layer must be modified to understand the semantics of each file system. Further, if file system enhancements or fixes are made that subtly alter the semantics, a corresponding change must be made to the storage layer. Another disadvantage is that not all file systems provide sufficient semantics to fully understand the liveness of data. Moreover, since file system structures may be cached in memory, it is not always possible to define provably correct semantics for understanding block liveness. For these reasons, we did not explore the implicit detection approach in this paper, as our focus was on a practical and general approach that can be applied across different file systems and block layers.

[5] proposed a completely new interface between file systems and storage devices that provides a much richer set of functions, including a delete operation. While being more versatile, this approach is also much more invasive than the one described in our paper.

[15] describes an interesting approach for providing "write hints" to the storage devices servicing an on-line transaction processing (OLTP) workload. Since their approach focuses on database transactions, their hints indicate whether a given write is "synchronous", "asynchronous replacement", or "asynchronous recoverability". There appears to be some synergy with the approach described in this paper, since in dealing with second level cache management, the deletion (purge) of an OLTP record would appear to be of interest to a cache management system.

Interestingly, Strunk et. al. [20] argue that in some cases it is desirable to be able to recognize and *preserve* deleted data. They refer to this as "information survival." Both the flagged I/O and direct call techniques described in this paper lend themselves to this requirement. The zero block approach is less applicable, though a block layer that detects when data is being overwritten by a block of zeros could also trigger the preservation of the previous contents of the block.

6 Concluding Remarks

We have demonstrated the practicality of making minimal changes to the file system-block level interface to inform storage devices of the liveness of data blocks. We defined a purge operation to pass this liveness information, and presented three techniques to instantiate this operation: direct call, zero blocks and flagged writes, with each technique having some advantages and disadvantages. We evaluated these techniques in a reference implementation of a dynamically resizable copy-on-write storage system. Our results demonstrate that these techniques provide significant storage savings over existing COW implementations, while incurring acceptable performance overhead.

In the future, we plan to investigate other environments, such as Xen virtual machines, where these techniques could provide value, for instance, for live VM migration. We will also continue to explore more efficient liveness passing mechanisms, such as the zero page approach describe earlier. Scalable data management within resizable COW files is also an area that deserves more attention.

References

[1] Tim Bray. bonnie. `http://www.textuality.com/bonnie/`.

[2] N. Burnett, J. Bent, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Exploiting Gray-Box Knowledge of Buffer-Cache Management.

In *Proceedings of the USENIX Annual Technical Conference*, June 2002.

[3] Remy Card, Theodore T'so, and Stephen Tweedie. Design and Implementation of the Second Extended Filesystem. In *Proceedings of the First Dutch International Symposium on Linux*, State University of Groningen, 1995.

[4] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live Migration of Virtual Machines. In *Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 273–286, Boston, MA, May 2005.

[5] W. de Jonge, F. Kaashoek, and W. C. Hsieh. Logical Disk: A simple new approach to improving file system performance. Technical Report MIT/LCS/TR-566, Massachusetts Institute of Technology, 1993.

[6] P. J. Denning. Effects of scheduling on file memory operations. In *Proceedings on AFIPS Spring Joint Computer Conference*, Reston, Va., April 1967.

[7] J. Dike. A user-mode port of the Linux kernel. In *Proceedings of 4th Annual Linux Showcase and Conference*, pages 63–72, 2000.

[8] G. Ganger. Blurring the Line Between Oses and Storage Devices. Technical Report CMU-CS-01-166, Carnegie Mellon University, December 2001.

[9] Gregory R. Ganger, Marshall Kirk McKusick, Craig A. N. Soules, and Yale N. Patt. Soft updates: a solution to the metadata update problem in file systems. *ACM Transactions on Computer Systems*, 18(2):127–153, 2000.

[10] Tal Garfinkel, Ben Pfaff, Jim Chow, and Mendel Rosenblum. Data lifetime is a systems problem. In *Proceedings of the 11th workshop on ACM SIGOPS European workshop: beyond the PC*, Leuven, Belgium, 2004.

[11] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[12] Peter Gutmann. Secure Deletion of Data from Magnetic and Solid-State Memory. In *Proceedings of the Sixth USENIX Security Symposium*, pages 77–89, July 1996.

[13] B. Hong, D. Plantenberg, D. D. E. Long, and M. Sivan-Zimet. Duplicate data elimination in a SAN file system. In *Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST 2004)*, page 301, 2004.

[14] Norman C. Hutchinson, Stephen Manley, Mike Federwisch, Guy Harris, Dave Hitz, Steven Kleiman, and Sean Malley. Logical vs. Physical File System Backup. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, New Orleans, Louisiana, February 1999.

[15] X. Li, A. Aboulnaga, K. Salem, A. Sachedina, , and S. Gao. Second-Tier Cache Management Using Write Hints. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST '05)*, pages 115–128, December 2005.

[16] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[17] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-To-End Arguments in System Design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[18] Muthian Sivathanu, Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Life or death at block-level. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, pages 379–394, 2004.

[19] Muthian Sivathanu, Vijayan Prabhakaran, Florentina I. Popovici, Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Semantically-Smart Disk Systems. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST 2003)*, March 2003.

[20] John D. Strunk, Garth R. Goodson, Michael L. Scheinholtz, Craig A. N. Soules, and Gregory R. Ganger. Self-Securing Storage: Protecting Data in Compromised Systems. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, pages 165–180, San Diego, CA, October 2000.

[21] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the 2001 USENIX Annual Technical Conference*, Boston, Massachusetts, USA, June 2001.

[22] Andrew Tridgell. dbench. `http://samba.org/ftp/tridge/dbench/`.

[23] Stephen Tweedie. Journaling the Linux ext2fs Filesystem. In *LinuxExpo '98*, 1998.

[24] Xiang Yu, Benjamin Gum, Yuqun Chen, Randolph Y. Wang, Kai Li, Arvind Krishnamurthy, and Thomas E. Anderson. Trading Capacity for Performance in a Disk Array. In *Proceedings of the 2000 Symposium on Operating Systems Design and Implementation*, pages 243–258, San Diego, 2000. USENIX Association.