Practical Techniques for Purging Deleted Data Using Liveness Information

David Boutcher
Department of Computer Science
University of Minnesota
Minneapolis, MN 55455
boutcher@cs.umn.edu

Abhishek Chandra
Department of Computer Science
University of Minnesota
Minneapolis, MN 55455
chandra@cs.umn.edu

ABSTRACT

The layered design of the Linux operating system hides the *liveness* of file system data from the underlying block layers. This lack of liveness information prevents the storage system from discarding blocks deleted by the file system, often resulting in poor utilization, security problems, inefficient caching, and migration overheads. In this paper, we define a generic "purge" operation that can be used by a file system to pass liveness information to the block layer with minimal changes in the layer interfaces, allowing the storage system to discard deleted data. We present three approaches for implementing such a purge operation: direct call, zero blocks, and flagged writes, each of which differs in their architectural complexity and potential performance overhead. We evaluate the feasibility of these techniques through a reference implementation of a dynamically resizable copy on write (COW) data store in User Mode Linux (UML). Performance results obtained from this reference implementation show that all these techniques can achieve significant storage savings with a reasonable execution time overhead. At the same time, our results indicate that while the direct call approach has the best performance, the zero block approach provides the best compromise in terms of performance overhead and its semantic and architectural simplicity. Overall, 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

The Linux operating system is implemented in a layered architecture, where the storage subsystem is separated from the file system by a narrow interface. This interface typically supports only simple operations such as "read block" and "write block", 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* [16] 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 [11], security [9, 10], backing up unnecessary data [12], inefficient caching [3, 13], and migration overheads [5]. Passing liveness information to the block layer allows for many potential optimizations:

• A storage subsystem with an intelligent cache hierarchy can ensure that only live, and therefore useful, blocks occupy space in the cache, leading to better cache utilization [3, 13].

• A Storage Area Network (SAN) that dynamically allocates data blocks as data is written by clients can free dead data blocks, preserving storage as well as network and I/O bandwidth [11].

• 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, ensuring minimally sized COW files. One potential use of smaller COW files is to substantially reduce the storage requirements for multiple virtual machines sharing identical file systems, and also reduce their migration overheads [5].

• 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 [23].

• From a security perspective, removing deleted blocks can be useful for physically purging the data of deleted files to prevent it from being retrieved later [10].

Some techniques have been proposed for passing liveness information to the block layer [16, 13, 6]. These techniques have typically involved either significant changes to the block layer interface or are specific to the structure of the data stored on the block device. These limitations have made these techniques impractical for widespread adoption into existing 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 evaluate their utility in a real implementation[1].

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. The purge operation is designed with the following objectives in mind:

• Minimize changes to the interface between the file system and block layers to make it practical for instantiation in real systems.

• Be backward compatible, so that the purge operation can be independently introduced into any file system or block layer, providing benefits when both layers cooperate, while supporting the functionality of liveness-agnostic file systems and block layers.

• Have minimal impact on the performance of the system and avoiding any changes to normal I/O operations.

[1]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, such as database systems [13].

We investigate three practical techniques to instantiate this purge operation in a real system: *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. The zero blocks technique 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. While the direct call technique corresponds to the "explicit detection" technique described by Sivathanu et. al.[16], the other two techniques are introduced in this paper.

We demonstrate the feasibility of these techniques through a reference implementation of a dynamically resizable copy on write 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 46% across the different benchmarks. At the same time, while our results indicate that direct call is the most efficient mechanism, the zero block approach provides the best compromise in terms of performance overhead and its semantic and architectural simplicity. Overall, our results demonstrate that passing liveness information across the file system-block layer interface with minimal changes is not only feasible but practical.

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 resizable 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 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 [4] file system runs identically on IDE, SCSI, and loop-back file systems. Similarly, new file systems, with semantics such as journaling [21], or log-based file systems [15] 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. In particular, such a change should be permitted independently in any file system or block layer, with benefits occurring when both layers cooperate. Any layer that does not provide or understand liveness semantics should continue to function as before. Next, we introduce a purge operation that meets these requirements.

2.1 Purge Operation

We define a *purge* operation to be provided by the block layer to the file system to explicitly allow for passing of liveness information. We define the semantics of the purge operation as follows:

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

Defining the purge operation in this manner meets 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 the 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 use 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 without being able to pass liveness information. Finally, and most importantly, the purge operation would allow the underlying storage system to discard data which is no longer relevant to the file system.

2.2 Purge Implementation

We examine three techniques to instantiate the purge operation. All these techniques are designed to implement the purge semantics as described above. However, they differ in their architectural complexity and potential performance overheads.

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 intuitive way to implement the purge operation, however it bypasses many layers of the operating system and is incompatible with many I/O scheduling algorithms.

Zero blocks: The file system zeros dead blocks. The block layer detects blocks that consist only of zeroes and treats them as dead. This is the simplest and least intrusive technique in terms of layer changes, though clearly it causes a significant number of additional I/O operations when a file is deleted.

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. This technique is semantically simple, however, it can require changes to multiple OS layers, such as the page cache. Since flags are associated with data written, it requires the same number of additional I/O operations as the zero block approach.

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]By "alive", we mean part of a current file, or of file system metadata.

[3]Other semantics can be defined, e.g., [18] describes cases where it is advantageous to preserve deleted data.

2.3 Direct Call

Perhaps the most intuitive 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[4], all block devices export the following structure:

```
struct block_device_operations {
   int (*open)(...)
   int (*release)(...)
   int (*ioctl)(...)
   long (*unlocked_ioctl)(...)
   ...
```

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,
                     unsigned long *length)
```

that can be used by the file system. The function takes a pointer to the structure describing block device, 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)
    fops->purge(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. Linux file systems do 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. For instance, 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. By introducing a direct call from the file system to the block layer, an architecturally difficult layer violation is introduced that completely bypasses I/O scheduling and ordering. The situation is made even worse in the case of journaling file systems, since such file systems have very precise I/O ordering requirements to support, for example, the ability to roll back operations during crash recovery. In order to ensure the ability to roll back, the data must not be purged from the disk before any metadata associated with the delete is written.

A synchronous direct call is also undesirable due to the potential I/O overheads that may result due to the purge operation. For example, 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 rearranged in the COW file. Carrying out these purge operations using a synchronous direct call could bring the file system to a stop while a long-running synchronous operation is performed. To avoid this situation, any I/O resulting from the purge operation should 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.)

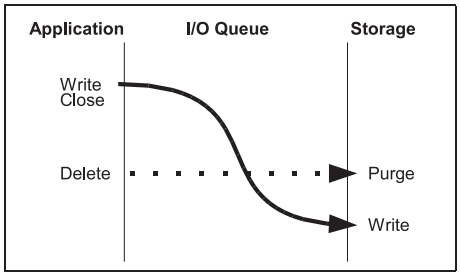


Figure 1: Direct Call ordering example

Finally, a direct call creates severe I/O ordering issues leading to potential data corruption. The less severe reordering issue occurs if a write is followed by a purge. 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 reordering does not introduce correctness issues, but will lead to dead data being written to the media, which 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. 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.

2.3.2 Implementation Issues

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 will never occur.

2.4 Zero Blocks

A simple and architecturally non-intrusive 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[5]. The block layer then examines blocks of data written, and if they are zero, treats them as "dead" data. As described in Section 2.1, an important semantic that must be maintained by the block layer is that a block detected as "dead",

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

[5]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 therefore purged from the system, must return zeros on subsequent read operations.

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 can impose a potential 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[6].

A more significant penalty is the requirement to issue I/O operations to every block of a file when it is deleted. This overhead is non-trivial, and its impact must be traded off against the benefits of subsequently recovering deleted blocks. Having a file system that zeros deleted blocks and a block layer that is *not* zero-block aware maintains correct file system semantics. This situation is functionally correct, but bears the burden of significantly increased I/O without the benefits of improved storage utilization.

2.5 Flagged Writes

Finally, it is possible to define a new "flag" that accompanies I/O requests flowing between the file system and block layers 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. A new flag to indicate a purge operations can be set by the file system, and recognized by the block layer. 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. This approach results in a similar number of additional I/O operations as the zero block approach, but without requiring that the pages first be written with zeros, or requiring that the storage layer examine each page to determine if it contains only zeros.

2.5.1 Advantages and Disadvantages

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.

[6]A more efficient alternative to the zero block approach could be to map a purged page in the page cache to the *zero page*—a special page maintained by Linux for mapping zero-initialized blocks of memory.

The main disadvantage of a new I/O flag is that it is a more invasive change than those previously described. For instance, 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. In the case of Linux implementation, the flag must be defined in three different kernel structures: (i) the buffer_head struct, (ii) the bio struct, and (iii) 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. However, purge and non-purge I/O operations cannot be merged together lest the purge bit be applied to an incorrect I/O or dropped from a purge I/O, thus preventing the performance benefit of merging. Finally, since the only way to inform the block layer of a change to the status of the page is to schedule it for I/O, a purged page will generate the same additional I/O overhead as zeroing all blocks of a deleted file.

3. IMPLEMENTING A LIVENESS-AWARE COPY ON WRITE STORAGE SYSTEM

To explore the above described purge techniques in a practical environment, we implemented a dynamically resizable copy on write 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

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.

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 [19], 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 of the 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".

COW storage is becoming an increasingly important technique for managing storage overcommitment in virtual machine environments. VMWare [19], Xen [1], KVM [14], and other virtual machine monitors make use of COW storage to allow multiple virtual machines to share common blocks. One fundamental attribute of existing COW storage in these environments, however, 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. Allowing COW storage to be dynamically resizable provides a significant benefit for the storage overcommit scenarios frequent in large virtualized environments.

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 Linux 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. However, data blocks are not allocated on the media for all the intermediate data. A read to any region of the file not previously written to, will return zero[7].

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[8]. 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, while regions not written to remain sparse.

In addition, a bitmap is maintained indicating which blocks of the COW store have been written to, with one bit for each region of the backing data. Initially, the bitmap is all zero. 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 of 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. Note that the Linux-specific `madvise(MADV_REMOVE)` call does include the semantics for removing a section of a file, however it has only been implemented in the tmpfs and shmfs file systems, and thus is not usable for disks backed by physical media.

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.

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

[8]In practice, the COW file will likely consume a minimum of one data block, as well as any associated file system metadata.

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 evaluating the purge techniques described in the previous section.

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 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. 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.

When a block is purged from the COW file, the COW file should shrink. In order to implement this 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". This current implementation adds overhead every time a block is purged, however allows for immediate evaluation of COW file size. A more optimized solution would use lazy garbage collection techniques to amortize and avoid the cost of purging blocks.

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

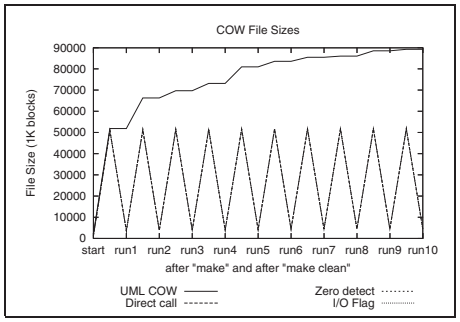


Figure 2: Linux build COW file sizes.

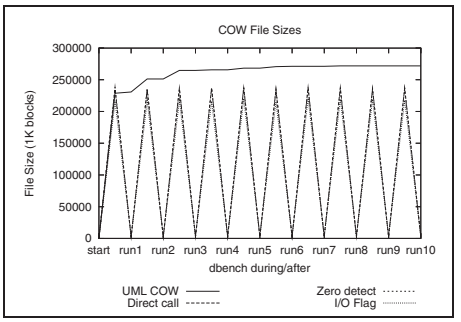


Figure 3: dbench COW file sizes.

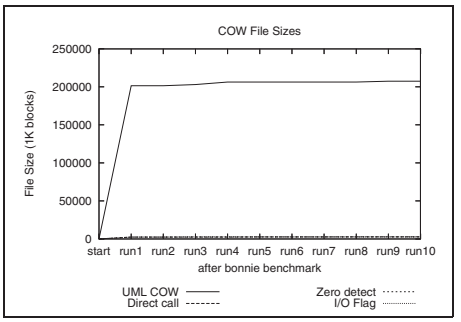


Figure 4: bonnie COW file sizes.

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. First, 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. In our experiments, a 500MB ext2 [4] file system was created on UML. 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 benchmarks were run on 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. While the ext2 file system is more commonly replaced by journaling file systems on modern systems, it has the advantage of being more tolerant of the I/O ordering violations introduced by the direct call implementation.

We used the following benchmarks to analyze the performance of our dynamic COW implementation: (i) *Linux build:* This benchmark performs 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. (ii) *dbench:* dbench [20] 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. (iii) *bonnie:* bonnie [2] 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, each using one 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.

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 2 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 corresponds to the default UML COW, while the other curves correspond to the three resizable COW implementations (these curves are so close that they are indistinguishable). It is clear from 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). Note that when using the default UML COW the file continues to grow after each run since the same disk blocks are not necessarily re-used in each make/clean cycle.

Figure 3 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. We modified the dbench benchmark slightly to store one non-zero byte in each 512 byte block to produce the results shown in Figure 3.

Finally, the COW file size for the bonnie runs is shown in Figure 4. 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 5(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 5(b). As shown in the figure, the time to delete the binaries (make clean) was increased by 46% from 6.1 seconds to 9.0 seconds in the worst case (flagged purge), with direct call taking 7.5 seconds and zero detect taking 8.3 seconds. The average 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 6(a) shows the average throughput achieved over 10 runs of dbench. As can be seen from the figure, the throughput 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 6(b) explains the performance of the zero-detect approach. The figure shows the impact of writing zeros to all deleted blocks. The second bar, "UML COW with zero" shows the performance of merely writing zeros to all deleted blocks, as compared to the "Zero detect" bar, which additionally examines all written blocks in the storage layer to determine if they are zero, and removes them from the COW file if they are. The zero detection adds only a 3% degradation over writing zeros to all deleted blocks. This result indicates that the main overhead in the zero detect approach is due to the write I/O's generated, and not due to detecting zero blocks.

The above result also helps in understanding the similarity in the performance of the flagged I/O approach to that of the zero page approach in Figure 6(a). Since every page that is flagged must be scheduled for I/O by the file system, the flagged I/O approach will perform a comparable number of additional I/O operations to those required for writing zeros to the deleted blocks of a file. Moreover, these flagged I/O operations cannot be coalesced with other I/O operations, as explained in Section 2.5.

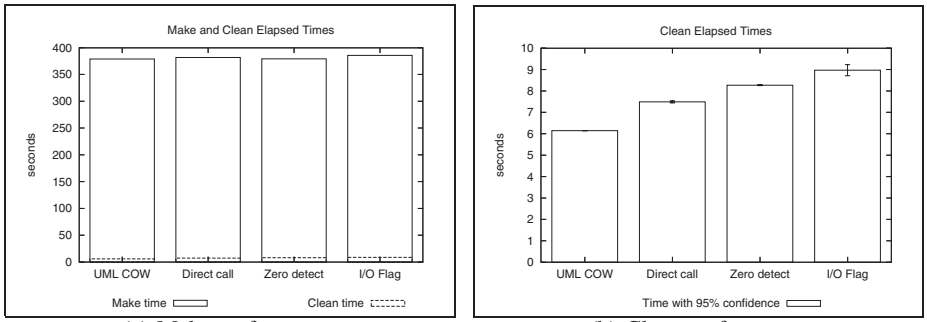
Figure 7(a) shows the elapsed time to complete the bonnie benchmark, which is a multi-stage I/O-intensive benchmark measuring a number of I/O functions such as sequential and random reading/writing, creating/deleting files, seeking, etc. Note that while on the dbench throughput graph, higher is better, in the bonnie elapsed time graph, lower is better. As seen from Figure 7(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 increase from an average of 69 seconds to 86 seconds and 74 seconds, respectively. Figure 7(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 the overhead.

The differences between the three approaches are highlighted by Figures 8(a) and 8(b), which show the number of creates and deletes that bonnie performed per second. Using the direct call purge approach has very good performance on creates, but much worse performance on deletes. This is because, on a file deletion, a direct and immediate call is made to the block layer to purge the file blocks. This adds overhead of resizing the COW file at the time of the delete operation itself. The zero block and the flagged I/O approaches, on the other hand, show much better performance on deletes, but poor performance on creates. This is because when a file is deleted, the relevant pages are simply written with zeros or flagged for purging, but the purge operation itself does not occur until the overwritten or marked pages naturally migrate to disk. The poor performance on creates can be attributed to the additional I/O operations already in the queue generated from previous purges. In other words, the overhead of purging blocks from the COW file with these approaches is deferred until subsequent I/O operations, and is thus amortized over a longer time period.

4.4 Discussion

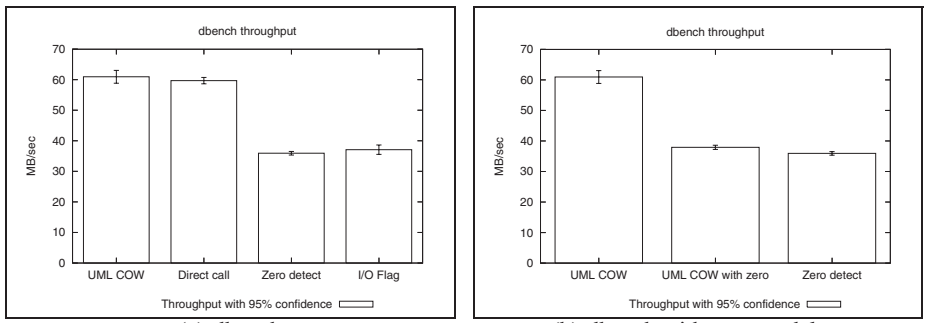
A comparison of the different liveness inference techniques must include both quantitative as well as qualitative analysis. As seen from our results, the direct call technique is the most efficient among the three. However, the efficiency of the direct call approach comes at the expense of heavy violations of layering between the file system and the block layers, causing several inconsistencies in I/O ordering and semantics, such as barrier operations, as described in Section 2.3.1. These architectural and ordering violations make direct call a potentially dangerous and difficult technique to implement. The zero block and the flagged I/O techniques are much simpler and honor all existing I/O ordering and scheduling assumptions in the storage system. Moreover, the largely synchronous nature of the direct call approach also results in higher overhead at the time of the purge operation itself, as indicated by Figure 8(b). The zero block and flagged I/O approaches, on the other hand, are able to defer these costs until the I/O operations naturally migrate to the storage device.

In terms of performance, both the zero block and the flagged I/O techniques cause significantly more I/O operations to occur between the file system and the block layer, compared to the direct call approach. This is because both the zero block and the flagged I/O techniques require that all pages associated with the purged data must either be written to or be marked in the page cache. This overhead is a necessary cost for communicating the liveness information to the block layer in order to avoid the layering and ordering violations of the direct call. Interestingly, our results show that both the zero-detect and I/O flag approaches have similar performance characteristics across all benchmarks. This result indicates that the cost of both zeroing pages and subsequently detecting blocks of zeros does not appear to be a significant factor in the overhead associated with purging blocks. This result was contrary to our



(a) Make performance (b) Clean performance

Figure 5: Performance of Linux build benchmark.



(a) dbench (b) dbench with zero on delete

Figure 6: dbench throughput.

expectations, as we had originally expected that the overhead of writing and detecting zeros would be significantly higher than that of flagging I/O operations. This indicates that the additional I/O generated from the purge operations is the dominating factor in the performance of both these techniques.

Qualitatively, the flagged I/O approach requires much more significant architectural changes to the operating system, as described in Section 2.5.1. Further, if such flags need to be passed to additional storage layers such as in a SAN, they would require even more invasive changes. The zero block approach, on the other hand, does not require such changes, as it relies on an implicit agreement as to what it means to "purge" a block, and is the most non-invasive technique among the three.

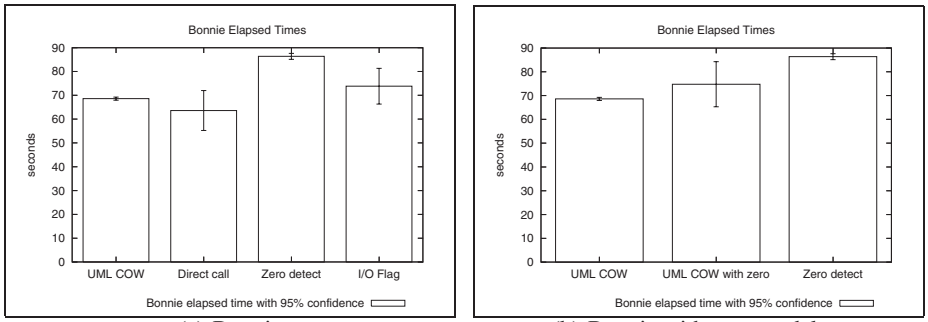
Based on both qualitative and quantitative considerations, our results suggest that *while direct call is the most efficient technique, the zero page approach provides the best compromise in terms of performance overhead and architectural simplicity of the three approaches we evaluated.*

5. RELATED WORK

The narrow interface between the file system and the block layer is addressed in [17]. This work advocates creating "semantically-smart disk systems" that can understand the file systems above them to make more intelligent storage decisions. 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.

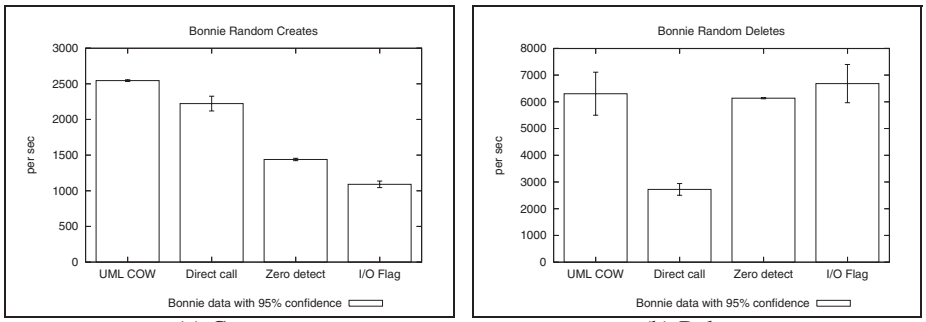
Techniques have been developed [16] to determine the liveness of data blocks, classified as explicit and implicit. Their explicit detection approach is similar to our direct call interface, without necessarily addressing the scheduling difficulties associated with calling the block layer from the file system. Our analysis of the direct call approach is designed to quantitatively evaluate this technique for passing liveness data in comparison to the other techniques described herein. 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 intrusive, there are a number of limitations to it. The first is that to be portable, the storage layer must be modified to understand the semantics of every file system. Further, if any 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.

A completely new interface between file systems and storage devices has been proposed [6] 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



(a) Bonnie (b) Bonnie with zero on delete

Figure 7: Bonnie elapsed time



(a) Creates (b) Deletes

Figure 8: Comparison of the purge techniques for file creations and deletions.

paper.

Another approach provides write hints [13] to the storage devices servicing an on-line transaction processing (OLTP) workload. Since this 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.

The issue of data stored in layered caches is addressed by Yagar, Factor, and Schuster using a system named KARMA [22]. While they address removing data from a cache as it migrates up the cache hierarchy, they do not address the issue of purging data from the cache if it is deleted. While this paper has primarily addressed data deleted from traditional file systems, applying deletion hints to a database cache system such as KARMA would provide additional knowledge in cache management.

Interestingly, Strunk et. al. [18] argue for information survival: the ability to recognize and *preserve* deleted data in some cases. 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 Linux 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 making minimal changes to the interface between the file system and the block layer.

Our analysis indicates that of the three approaches we evaluated, the best approach in terms of achieving minimal invasiveness is a "keep it simple" approach that involves just writing zeros to deleted blocks. It is architecturally clean and consistent with other semantics, such as I/O ordering, passed across the block layer interface. Surprisingly, the overhead of filling pages with zeros, and scanning blocks for zeros is not a significant component of the additional overhead. The main disadvantage and overhead of this approach is the requirement that pages be passed to the block layer for every block of a deleted file.

The I/O flag does not appear to provide significant performance advantages over the zero-block approach, while introducing more invasive changes to the file system and block layers. The direct call approach, while performing the best, appears to be the least attractive of the three approaches, due to its violation of the scheduling

and ordering interfaces in the kernel.

The objective of this paper was to explore techniques that make *minimal* changes to the block layer interface while passing liveness information. The additional I/O operations introduced by our preferred approach suggests that an ultimate solution to passing liveness information will require a more complex implementation, particularly for journaling file systems.

There are a number of areas of interest for future research. Firstly, since the direct call approach offers measurably superior performance to the other two approaches, a provably correct implementation of this "explicit" interface would likely offer the best performance for passing liveness data. Secondly, we plan to investigate other environments, such as Xen virtual machines, where these techniques could provide value, for instance, for live VM migration. Finally, scalable data management within resizable COW files is also an area that deserves more attention.

7. REFERENCES

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, October 2003.

[2] T. Bray. bonnie. `http://www.textuality.com/bonnie/`.

[3] 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.

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

[5] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 273–286, May 2005.

[6] 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.

[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, Dec. 2001.

[9] T. Garfinkel, B. Pfaff, J. Chow, and M. Rosenblum. Data lifetime is a systems problem. In *Proceedings of the 11th workshop on ACM SIGOPS European workshop: beyond the PC*, Leuven, Belgium, 2004.

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

[11] 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.

[12] N. C. Hutchinson, S. Manley, M. Federwisch, G. Harris, D. Hitz, S. Kleiman, and S. Malley. Logical vs. Physical File System Backup. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, New Orleans, Louisiana, February 1999.

[13] 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.

[14] Qumranet. Kvm: Kernel-based virtualization driver. `http://www.qumranet.com/wp/kvm_wp.pdf`, 2006. Technical Report.

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

[16] M. Sivathanu, L. N. Bairavasundaram, A. C. Arpaci-Dusseau, and R. 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.

[17] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-Smart Disk Systems. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST 2003)*, March 2003.

[18] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. 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.

[19] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the 2001 USENIX Annual Technical Conference*, June 2001.

[20] A. Tridgell. dbench. `http://samba.org/ftp/tridge/dbench/`.

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

[22] G. Yadgar, M. Factor, and A. Schuster. Karma: know-it-all replacement for a multilevel cache. In *FAST '07: Proceedings of the 5th USENIX conference on File and Storage Technologies*, pages 25–25, Berkeley, CA, USA, 2007. USENIX Association.

[23] X. Yu, B. Gum, Y. Chen, R. Y. Wang, K. Li, A. Krishnamurthy, and T. 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.