Opportunistic Storage Maintenance

George Amvrosiadis
Dept. of Computer Science, University of Toronto
gamvrosi@cs.toronto.edu

Angela Demke Brown
Dept. of Computer Science, University of Toronto
demke@cs.toronto.edu

Ashvin Goel
Dept. of Electrical and Computer Engineering, University of Toronto
ashvin@eecg.toronto.edu

Abstract

Storage systems rely on maintenance tasks, such as backup and layout optimization, to ensure data availability and good performance. These tasks access large amounts of data and can significantly impact foreground applications. We argue that storage maintenance can be performed more efficiently by prioritizing processing of data that is currently cached in memory. Data can be cached either due to other maintenance tasks requesting it previously, or due to overlapping foreground I/O activity.

We present Duet, a framework that provides notifications about page-level events to maintenance tasks, such as a page being added or modified in memory. Tasks use these events as hints to opportunistically process cached data. We show that tasks using Duet can complete maintenance work more efficiently because they perform fewer I/O operations. The I/O reduction depends on the amount of data overlap with other maintenance tasks and foreground applications. Consequently, Duet's efficiency increases with additional tasks because opportunities for synergy appear more often.

1. Introduction

Modern enterprise environments impose a host of demands on storage systems, including performance scalability, high availability and data security. In addition, they require various storage-related capabilities for meeting service-level agreements and legal needs, such as data retention, disaster recovery, data mining and storage analytics. To meet these diverse demands, storage systems rely on various types of *maintenance* tasks. These tasks run in the background, helping improve storage reliability, performance, or enabling data analysis. Common reliability and security tasks are backup and archiving [5, 9, 15, 17, 23, 27, 28, 56, 68], data scrubbing [11, 22, 37, 44, 45], write verification [47], and virus scanning [7, 24, 30, 33, 43, 55]. Performance-related tasks include data layout optimization [54], garbage collection [66] and deduplication [21]. Analysis tasks include data mining [46] and storage data analytics [14].

Maintenance tasks raise challenges for storage systems because they access a significant amount of data that does not easily fit in memory. For example, enterprises typically run full backups weekly and often more frequently [3]. Similarly, anti-virus scans in virtual machines cause I/O storms [53]. These tasks can interfere with foreground applications, which we call the *workload*, causing significant impact on their performance. Thus, administrators have to carefully schedule maintenance tasks during idle times. However, long idle times may not be available, especially with increasing data storage needs. For instance, as enterprises are moving to the cloud, data sharing occurs across time zones, and much higher consolidation ratios are observed in storage systems. As a result, workloads are losing their traditional diurnal characteristics that guarantee predictable idle periods, making it harder to meet maintenance goals.

A recent survey of 500 CIOs of medium scale organizations confirms this trend, showing that 40% of Microsoft SMB backups fail to complete within their scheduled window [64, Chart 13]. Another survey of 1200 IT professionals shows that 33% of backups routinely miss their window, while only 28% always complete on time [31, p. 3]. Ironically, over 50% of IT professionals believe that someone could lose their job if critical data was lost after a disaster [18].

Existing approaches for minimizing the impact of maintenance tasks focus on I/O scheduling, taking device characteristics into account. On hard disks, maintenance is piggybacked on workload requests [19] or performed during the seek time and rotational latency between workload requests [41, 57]. These approaches require detailed device performance characteristics to be determined, which is nontrivial in modern disks [36], and even more complicated for SSDs [1]. To be effective, they also require applications to a-priori specify the I/O requests they plan to issue. Furthermore, they need complex mechanisms for handling interblock dependencies [57], as discussed in the next section.

We propose a novel maintenance approach that prioritizes processing of data that is cached in memory. Data may be cached as a result of other maintenance tasks requesting it, or due to overlapping foreground I/O activity. This ap-

SOSP'15, October 4–7, 2015, Monterey, CA.

ACM 978-1-4503-3834-9/15/10$15.00.
DOI: http://dx.doi.org/10.1145/2815400.2815424

proach reduces the impact of maintenance in two ways. First, maintenance tasks can implicitly collaborate with each other. For example, during a full logical file system backup, data layout reorganization (such as defragmentation or garbage collection) can be performed with no additional reads. Second, data that has recently been accessed can be provided to a maintenance task. For example, a block modified by the workload can be used by an incremental backup task, avoiding an additional read. This I/O reduction helps maintenance tasks complete their work within their scheduled windows.

Our key insight is that maintenance work can be *reordered*. Maintenance tasks typically process items, such as blocks or files, in some predefined order, but they are not dependent on this order. For instance, a system administrator might require a backup to complete within a few hours. Within that period, backup of different files can be reordered without affecting the system's reliability guarantees.

We present Duet, a storage maintenance framework that provides hints to tasks about *page-level* events, such as a page being added, or modified in the page cache. Tasks use these hints to process their items out-of-order, which we call *opportunistic* work. For example, tasks can prioritize processing of files that have more pages in memory. Page-level events provide fine-grained information, helping support items of various granularities such as blocks, files, extents etc. Duet helps track completion of opportunistic work so that it is not repeated by the task.

An important goal of our work is to provide a simple programming model that minimizes changes to the tasks. We have modified five existing maintenance tasks to work with Duet. In the kernel, we have modified the scrubbing, backup, and defragmentation tasks in the Btrfs file system [49], and garbage collection in the F2fs log-structured file system [39]. These tasks required changes to fewer than 150 lines of code (LoC) each, or 2-10% of the original code. At the application level, we have modified the rsync application [59, 60], which required changes to 300 LoC, or 0.67% of its code.

We show that maintenance tasks using Duet can complete faster because they perform fewer I/O operations. This reduction depends on the amount of overlapping data accessed by the various maintenance tasks and the workloads. Tasks using Duet implicitly collaborate with each other, and are able to complete within their scheduled windows, even on busy devices. For example, when scrubbing, backup, and defragmentation are run concurrently with a workload that keeps the device busy 50% of the time, Duet-enabled tasks perform their work four times faster than the original tasks.

Our work makes three contributions. We identify that maintenance tasks can process data items in arbitrary order without affecting correctness. We provide a simple programming model that enables these tasks to perform opportunistic processing based on data cached in memory. We apply this model to several kernel and user tasks, with minimal task changes, and show the performance benefits obtained.

The rest of the paper describes our approach in more detail. Section 2 provides additional motivation for this work and compares our approach with previous work. Section 3 presents our opportunistic work model, and Section 4 describes the Duet framework that implements this model. Then, Section 5 explains how we modified five maintenance tasks to use Duet, and Section 6 demonstrates the effectiveness of the Duet-enabled tasks. Finally, in Section 7 we conclude and discuss possible avenues for future work.

2. Motivation and Related Work

We motivate this work by characterizing when and how often maintenance work is performed. A recent study of 40,000 enterprise backup systems, monitored over a span of 3 years, found that full backups are performed frequently: 28% of systems conduct one every 1-3 days, 44% perform them every 3-6 days, and only 17% of systems perform them weekly [3]. Systems that perform full backups weekly or less frequently, however, complement them with daily incremental backups. The study also reports that administrators simply use the default scheduling windows when configuring backup policies, causing backup jobs to execute in bursts. Anti-virus scans in virtual machines cause I/O storms for similar reasons [53]. These observations suggest that maintenance work is performed frequently and concurrently, where our approach is expected to have the most benefits.

Existing approaches for reducing the impact of maintenance tasks primarily focus on scheduling I/O requests [41, 57]. They require specifying the requests (in terms of sets of disk locations that will be read or written) sufficiently in advance to the scheduler, which complicates programming and limits flexibility. These approaches help amortize the cost of seeks and rotational delays but do not necessarily reduce the number of I/O operations. Consider two hypothetical tasks, one that traverses the file system in depth-first order, and the other in breadth-first order. If these tasks are run concurrently, even careful scheduling of I/O requests may not provide much benefit. Even if the two tasks traverse the file system in the same order, but are staggered in time, then the benefits of scheduling will be limited. These examples argue for out-of-order processing at the level of the maintenance application itself, which helps with both of these issues. Our approach provides hints to applications to enable efficient out-of-order processing.

With scheduling approaches, task-specific deadlock prevention mechanisms are needed for handling inter-block dependencies [57]. For example, if a block needs to be moved to a location containing live data, then the live data is copied to a persistent staging area until it can be moved to its own new location. However, deadlocks can occur if the staging area fills up with blocks that have unresolved dependencies. Our approach sidesteps this issue because it uses hints, and out-of-order processing is performed within the maintenance application rather than at the scheduler level. In particular,

Duet has no knowledge of dependencies and hence does not require staging.

Our work, which aims to reduce I/O accesses, is orthogonal to general I/O scheduling schemes, such as anticipatory scheduling [32], idle-time scheduling [20] and others [8, 26, 42]. We can use any of these to run maintenance tasks, but we found experimentally that running maintenance tasks with an idle-time scheduler [26] minimizes impact on workloads. The scheduling algorithm does not reduce the amount of maintenance work, however, so the challenges with meeting maintenance goals still persist [3, 43, 64].

Sharing work across some maintenance tasks is supported by the Simpana suite from CommVault [13], which performs a single file system pass and delivers data to "converged" backup, archive and reporting tasks that are part of the same application. Our approach supports existing maintenance tasks, that have been developed independently, both at the user and kernel level. We are also able to take advantage of data cached as a result of workload accesses.

PACMan [4] helps reduce job completion times in a cluster environment with a global cache replacement policy that coordinates data that is cached in memory across nodes. Our work focuses on long running (large) tasks, whose data may not fit in caches. Therefore, our approach aims to make the best use of the current cached data, by modifying the order in which tasks process data. However, we expect that informed cache replacement will provide us additional benefits.

The database community has investigated data-driven approaches, with systems actively scanning in-memory data and invoking interested queries [2, 62], or pushing data onto processors and allowing any interested computation to process it [6]. These approaches take advantage of declarative database queries to perform out-of-order processing. We use polling and hints so that tasks can perform out-of-order processing, instead of using a pure data-driven approach. The latter may be harder to retrofit in existing, imperative tasks that impose specific ordering requirements.

3. Opportunistic Work Model

Our aim is to enable one or more maintenance tasks to execute concurrently, with minimal impact on the foreground workload. To do so, we leverage the property that maintenance work can usually be reordered. Maintenance tasks typically process items, such as data blocks or files, in a predefined order but they are not dependent on this order. Our Duet framework provides tasks with hints about cached data. Tasks can use these hints to opportunistically process cached data out-of-order, reducing the total I/O required to meet their goals.

Our work has two sub-goals. First, to encourage adoption, we would like to provide a simple programming model that minimizes changes to the tasks. This requirement introduces several challenges. We would like to reuse existing maintenance tasks rather than writing them from scratch. These tasks are developed independently, so rewriting each task to explicitly collaborate with every other task in the system is unreasonable, both due to the large number of possible combinations of tasks, and the effort needed to add new tasks. We should also not require tasks to specify maintenance I/O a-priori, because it is too onerous on the developer, and constrains the ability to adapt tasks to changes in the system. Finally, tasks operate at several granularities (e.g. blocks, files, extents, segments), and we need to easily support all of them. Our second goal is to design a framework that supports a variety of maintenance tasks and scales with the number of tasks running on the system.

Next, we provide an overview of Duet and how it meets these goals. The design of Duet is described in more detail in Section 4.

3.1 Overview of Duet

Duet hooks into the page cache and provides notifications about page-level events to maintenance tasks, such as a page being added, removed, dirtied or flushed (written back to storage) from the page cache. We leverage page-level events because data is cached at page-size granularity when it is read from and written to storage. Tasks use the Duet API to poll for these events at appropriate times, such as before each item is processed. We provide a polling interface because it avoids complications with handling asynchronous events. The tasks then use their own criteria to decide how and when to act upon the events. The result is that tasks perform out-of-order processing of cached items, reducing I/O operations. Duet helps track work completion so that tasks operate once on each item, either opportunistically or during normal operation.

While Duet events occur at the page granularity, a task, such as defragmentation, may operate at a different granularity, such as extents. For example, the task may require all pages of an extent to be in memory before defragmenting it, but the task will receive notifications when any of these pages are brought into memory by other applications. Instead of encumbering Duet with inter-page dependencies, the Duet events help tasks keep track of data available in memory, so that they can perform opportunistic processing. For example, the defragmentation task can use the Duet events to build a priority queue based on the extents with the most pages in memory. It can then use this queue to prioritize its processing, requesting from storage any additional pages of that extent that are needed to complete its operation. This approach avoids pinning pages in memory and the related issues that arise under memory pressure [57].

Algorithm 1 shows a simple example of a Duet file task, based on the Defragmentation and Rsync file tasks that we adapted for Duet (see Table 3). The `sid` parameter is a session id, and the Duet API calls are shown in bold, as described in Section 3.2. The original task runs a loop, calling `pick_next_file` (line 6) to choose files in some predetermined order, and then invoking `process_file`

```
1: function example_task(sid, pqueue)
2:    while (1) do
3:       // Process files opportunistically
4:       handle_queued(sid, pqueue)
5:       // Resume normal processing order
6:       (inode, path) = pick_next_file()
7:       if (!inode) then return
8:       handle_file(sid, inode, path)
9:
10: function handle_queued(sid, pqueue)
11:    // Process files opportunistically
12:    while (1) do
13:       prioqueue_update(sid, pqueue)
14:       inode = prioqueue_dequeue(pqueue)
15:       if (!inode) then return
16:       duet_get_path(sid, inode, path)
17:       handle_file(sid, inode, path)
18:
19: function prioqueue_update(sid, pqueue)
20:    // Fetch events, update priority queue
21:    while (1) do
22:       num = duet_fetch(sid, items, max)
23:       if (num == 0) then return
24:       prioqueue_enqueue(pqueue, items, num)
25:
26: function handle_file(sid, inode, path)
27:    // Skip if processed opportunistically
28:    if (duet_check_done(sid, inode)) then
29:       return
30:    // Process file as usual
31:    process_file(path)
32:    duet_set_done(sid, inode)
```

Algorithm 1. An example Duet task.

(line 31) to process the chosen file. The `handle_queued` (line 4) function performs opportunistic processing. It uses `prioqueue_update` (line 13), which fetches pending page events from Duet (line 22) and updates a priority queue of inodes (line 24). The priority queue is sorted by some task-specific criteria, such as the number of pages the inodes have in memory. Next, the `handle_queued` function dequeues the highest priority inode (line 14) and processes it opportunistically. The `handle_file` code is modified to check whether inodes have been processed already and to mark them as processed. This processing repeats, aggressively fetching page events again because the page cache may have changed, until there are no more items in the priority queue.

Our approach helps meet the goals described earlier. It does not require tasks to specify their pending work to Duet a-priori. In particular, a task can change the work it performs while it is running, without informing Duet. For example, a defragmentation task in a copy-on-write file system can simply ignore an overwritten file that it was planning to defragment. Duet also does not need to know about subtle dependencies that exist in the task, such as a page *A* needing to be processed before another page *B*. With our best-effort approach, the task can simply ignore inopportune events, such as page *B* being available in memory before page *A*. While our approach requires tasks to be modified, they do not need to explicitly collaborate or be aware of each other.

```
int duet_register(path, notification_mask)
int duet_deregister(session_id)
int duet_fetch(session_id, item_array, count)
int duet_check_done(session_id, item_id)
int duet_set_done(session_id, item_id)
int duet_unset_done(session_id, item_id)
int duet_get_path(session_id, inode_num, path)
```

Table 1. The Duet API

3.2 Duet API

Duet supports maintenance tasks operating at either the block or the file system layer. Block layer tasks, or *block tasks*, operate on data blocks, while file system layer tasks, or *file tasks*, are aware of files and directories.

Table 1 shows the Duet API. Maintenance tasks call `duet_register` to start using Duet. In the `path` parameter, block tasks specify a device file, while file tasks specify a directory (which we call the *registered* directory). The `notification_mask` consists of the types of events that are of interest to the task, as shown in Table 2 and explained later in the section. The `duet_register` call returns a `session_id` that is used in the rest of the Duet calls. The task ends the session when its work is complete by calling `duet_deregister`, which releases all Duet session state.

The heart of the Duet API is the `duet_fetch` system call that provides notifications to tasks about page-level events. These notifications are returned in `item_array`, up to a maximum of `count`, similar to the `read` system call. The fetch call returns any events that have occurred but have not yet been returned by previous calls to fetch. Block tasks receive notifications for any page-level events occurring on the device, while file tasks receive them for page-level events on *all* the files or directories located within the registered directory and its sub-directories.

Table 2 shows that tasks can register to be notified about page events or changes in page state. *Event notifications* are triggered when a page is added, removed, modified, or flushed from the cache. *State notifications* are emitted when the existence or modification status of a page *changes* in the page cache. For example, if a task registers for Exists notifications, and a page is removed and re-added between two consecutive fetch operations, then the page is considered to have reverted back to the same state, i.e. it exists in the cache, and an event is not generated on the next fetch call.

An item in `item_array`, returned by `duet_fetch`, consists of a tuple (`item_id`, `offset`, `flag`), that corresponds to a given page. For block tasks, the `item_id` is the block number, while for file tasks it consists of the inode number uniquely identifying a file or directory. The `offset` is only used for file tasks, and corresponds to the logical off-

Event	State Change	Description
Added	Exists	Page added in cache
Removed	¬ Exists	Page removed from cache
Dirtied	Modified	Dirty bit set for page
Flushed	¬ Modified	Dirty bit cleared for page

Table 2. Event and State-based Notifications

set within the file. The `flag` field consists of six bits, one for each event and state notification type shown in Table 2. This field identifies only the page events that have not yet been made available to the task via fetch operations. For example, suppose a page is added, a fetch operation occurs, and then the page is removed. The next fetch call will return an item for the page with only the removed bit set in the item `flag`, informing the task that the page has been removed.

To track whether items have been processed, tasks use the `duet_*_done` calls (Table 1), as described in more detail in Section 4. Finally, file tasks use `duet_get_path` to translate an item's inode to a path relative to the registered directory. We provide this call, rather than returning file paths in `duet_fetch` for two reasons. First, tasks will generally invoke `duet_get_path` once per file, but `duet_fetch` is called much more frequently because it operates at page granularity. Second, `duet_get_path` serves as the *truth* for our page cache hints [38]. When it fails, it indicates that the file is no longer cached, allowing tasks to back out of opportunistic processing that may not be worthwhile. Section 5 shows how tasks use these primitives.

3.3 Discussion

Our initial design for Duet was event-driven, with tasks being informed as I/O requests moved through the page cache, file system, and block layers in the storage stack. While this approach provided comprehensive control over storage processing, we found it tedious to implement and use. The implementation challenges arose from requiring changes across the storage layers, while in practice we found that page-level events are sufficient for maintenance tasks.

The event-driven approach is hard to use because existing maintenance tasks are not designed to process data made available at arbitrary times, either because of synchronization issues with on-going processing, or because of dependencies with unavailable data. Our current polling-based approach is easier to retrofit in maintenance tasks because it allows them to poll for events and perform out-of-order processing at suitable times. However, polling makes no guarantees of data availability between fetching and processing an item. For this reason, we take advantage of the page cache, which provides sufficient time to detect and exploit synergies between tasks. A side benefit is that our current approach does not require any file-system specific changes.

While Duet does not change the file access control model because `duet_fetch` doesn't provide any file data, it can leak information about pages in memory. For block tasks, we require them to be able to access their block device. For file tasks, we use file permissions to return events for files that are accessible to the task, but we currently do not take path-based access control into account.

Note that tasks using direct I/O will not benefit from Duet because they bypass the page cache. However, the maintenance tasks that we have examined do not use direct I/O. Tasks such as databases that use direct I/O cache data themselves, and we plan to apply the Duet API to such caches at the user level.

Finally, there are some similarities between Duet and the Linux Inotify mechanism [29, 35] that reports file-level accesses to applications. Inotify is used by applications such as the file manager, desktop search utilities, and for file synchronization (e.g. Dropbox, Google Music Manager). While Inotify focuses on file-level accesses, Duet is designed to track file data in memory. As a result, Duet provides page-level information, which is finer grained than Inotify's file-level information, allowing better prioritization for out-of-order processing. Similarly, Duet provides information about when data is flushed and evicted. Unlike Duet, Inotify does not support watching directories recursively, so adding watches to each sub-directory can take significant time for large directories, and is race prone. On the other hand, Duet does not inform tasks about file metadata changes (e.g. permissions, extended attributes).

4. The Duet Framework

This section describes the design and implementation of the Duet framework.

4.1 Framework Design

Duet hooks into the page cache modification routines and gets control when a page is added or removed from the page cache, or when a page is marked dirty or flushed. When these page cache events occur, Duet is passed a page descriptor and an event type, such as a page being added, removed, etc. Duet traverses the list of sessions, examining the notification mask registered by each session to determine whether the event type is of interest. If so, we determine whether the page is *relevant* to the session by checking whether it belongs to the correct device for a block task, or whether it lies within the registered directory for a file task. If the page is relevant, and has not been marked done, we update its set of pending events.

Duet maintains an item descriptor for each relevant page with pending events. The item descriptor contains the same information that a fetch call returns, i.e. an item of type (`item_id`, `offset`, `flag`), as described in Section 3.2. An item descriptor contains pending events when one or more bits are set in its flag field. A fetch call returns item descriptors with pending events, and marks the descriptors up-to-date by clearing their flag fields.

When a session is registered, we scan the page cache and initialize an item descriptor for each relevant page. The

flag field is set to indicate that the page is present (and possibly dirty). This scan serves two purposes. First, it is required for state notifications to be generated correctly, such as the Exists state from Table 2. Second, a scan allows a task to immediately take advantage of pages in the page cache (by invoking fetch after session registration).

While item descriptors help track pending events, Duet also needs to track relevant pages so that events can be generated efficiently. To do so, Duet maintains per-session state consisting of either one or two bitmaps. For block tasks, Duet maintains a single done bitmap that tracks completed work. The bitmap stores one bit for each block on the device. Tasks use the duet_*_done functions shown in Table 1 to check, set, and reset bits in the bitmap. When the task marks a block as done, the associated item descriptor is marked in the bitmap. Future events on the page are then ignored until the done bit is unset by the task. Although tasks decide when some work is complete and can track completed pages themselves, informing Duet avoids tracking and generating events for completed pages.

For file tasks, Duet maintains two bitmaps, done and relevant. Both bitmaps store a bit for each inode (i.e., for each file or directory) in the file system that contains the registered directory. When a file is marked in the done bitmap, the item descriptors for all the associated pages of the file are marked up-to-date and future events on the file are no longer tracked. We use file-level marking because these tasks operate at file granularity.

File tasks are only interested in files or directories located within the registered directory. We use the relevant bitmap to ensure that fetch only returns events on these relevant objects. When any page of a file (or directory) is added to the page cache for the first time, we traverse its path backwards to detect whether the file lies within the registered directory.[1] This check would be expensive if applied on every page access, so after the first access we use the relevant bit to determine whether the corresponding inode is relevant. If the inode is not relevant, we immediately mark the file as done, thus avoiding tracking the file pages or generating any events for the file in the future. Otherwise, we mark the relevant bit, and consult it on page cache events before generating notifications for the file.

Duet also needs to handle files and directories being moved into, or out of, the registered directory. We detect that a file is moved into the registered directory at the VFS layer and initialize item descriptors for all pages of the file in a manner similar to the page scan performed during session initialization. When a file is moved out of the registered directory, we set the Removed bit and clear the Exists bit for all existing pages of the file, marking the file as done. After the next fetch, Duet will ignore the file.

[1] This operation is relatively efficient in our Linux implementation, which maintains a directory entry cache that pins in memory the directory path leading to a file page.

Directory renames are trickier, because they require handling all files and directories under the renamed directory. Duet deals with directory renames by resetting the relevant and done bitmaps for all files other than the files that have already been processed, i.e. have both bits set. This approach avoids the need to traverse the renamed directory, and it guarantees that tasks will not receive unnecessary events for processed files. However, it requires rechecking file relevance when the files are accessed again.

4.2 Implementation

Our implementation of Duet consists of three components: a Linux kernel module, hooks in the Linux page cache, and a library for user and kernel tasks, implemented for Linux 3.13. Our implementation consists of 1700 lines of code for the first two components, and 1000 lines for the library.

While the item descriptors of different sessions are logically independent, we reduce memory requirements by keeping a single item descriptor per page for all sessions. The merged item descriptor consists of the item_id, offset, and an N-byte array for storing the flag fields for up to a maximum of N concurrent sessions. This maximum value can be configured at module load time. With this implementation, we allocate a descriptor when any session has pending events on the page, and deallocate it when no session has pending events on the page.

The merged descriptor implementation allows using a more efficient single, global hash table to look up the descriptors. We use the item_id and offset as the hash key, and then use the session id (which ranges between 0 and $(N-1)$) to index into the flag array.

Note that an item descriptor with pending events will remain allocated even if the corresponding page is deallocated from the page cache. For tasks that only subscribe to event notifications, the descriptor is only deallocated when it is marked up-to-date by a fetch call. Thus, item descriptors can grow over time if a task does not issue fetch calls. To counter denial of service, we limit the number of item descriptors per session and drop new events when this limit is reached. Note that this issue did not affect our Duet tasks because they invoke fetch calls many times per second, as explained in Section 6.4.

When a task registers for state notifications (e.g. Exists), a page can also be marked up-to-date when the corresponding events cancel each other, such as when a page is added and subsequently removed from the cache. As a result, the maximum number of item descriptors are bounded by ($2 \times$ *max. number of pages in page cache*). This bound would be reached if all existing pages are relevant, and they are removed and subsequently replaced by new pages between fetch calls. With this bound, events are never dropped.

We use a red-black tree to dynamically allocate portions of the relevant and done bitmaps, to represent ranges that have marked bits, and deallocate them when all their bits are unmarked or when the session terminates. This limits

memory consumption when tasks are interested in small, localized chunks of a device or file system.

Duet allows synergies to be detected between block and file tasks. For example, a block device could be mounted as a file system. In this case, we would like to provide any block task operating on the device with page events for files accessed by file tasks, or applications operating on the file system. However, the translation of a page's file offset to a block number is a filesystem-specific operation, and Duet is filesystem agnostic. Fortunately, many file systems in Linux (e.g. Btrfs, Ext2/3/4, XFS, F2fs) implement this translation through the FIBMAP ioctl call [40, 61]. We use this functionality, when implemented, to inform block tasks of file-level accesses on the same device. A similar API exists in Windows [48]. In the event that a page does not correspond to a block yet (e.g. due to delayed allocation [16]), the page is left to be returned by a later fetch operation.

Duet ignores pages that are not backed by files because they are not useful for maintenance tasks. It further provides both file and directory pages to file tasks. However, our current file tasks ignore the directory pages.

Finally, the Duet library is used by both in-kernel and user-level tasks. It implements a priority queue for storing Duet events that are fetched using the Duet API. Through this library, tasks can access the Duet API (Table 1), and the priority queue primitives shown in Algorithm 1. Our current implementation uses a red-black tree for the priority queue.

5. Applications

This section describes how applications use the Duet framework. Table 3 shows three block and two file tasks that we have modified to work with Duet. We have modified the existing in-kernel scrubbing, backup, and defragmentation utilities available in the Btrfs copy-on-write file system [49]. We have also modified the in-kernel garbage collector used by the log-structured F2fs file system [39]. These modifications were made in Linux 3.13. Finally, we have changed version 3.1.1 of the Rsync user-level application [59, 60]. Table 3 describes the order in which these applications normally process items and the changes we have made to them for Duet. Next, we describe these changes in more detail.

5.1 File System Scrubbing

To protect against data loss due to silent data corruption [10, 11, 37], commercial storage systems rely on scrubbing [44, 52]. A scrubber is a background process that periodically scans data and verifies its correctness using checksums. While scrubbing is commonly performed at the block layer, the Btrfs scrubber operates within the file system protecting against a wider variety of errors [37]. In Btrfs, a checksum is stored for every file system block, updated on a block write, and verified on a block read to ensure that applications receive correct data. The scrubber reads all allocated file system blocks on a given device sequentially and verifies them against their checksums for correctness.

Our opportunistic scrubber relies on the semantics of the file system's read and write operations to reduce maintenance work, while providing the same reliability guarantees as the original scrubber. The opportunistic scrubber receives notifications when a page is Added or Dirtied in the page cache. When a page is added, we mark the relevant block number as scrubbed, since Btrfs verifies data correctness during the read operation. On the other hand, checksums are not verified on a write request, so we unmark the bit for the dirty page block, indicating that the new checksum needs to be re-verified.

5.2 Snapshot-based backup

Btrfs is a copy-on-write file system that supports taking fast, file-system snapshots. All data and metadata in the snapshot is shared with the live file system until blocks are updated in the live system. Btrfs provides backup tools that allow taking a consistent backup using a read-only snapshot.

Our opportunistic backup tool exploits copy-on-write sharing because read operations to the live data may access data shared with the snapshot that is being backed up. By registering the backup session with Duet for the Exists notifications, we are informed of pages that currently exist in the page cache and their corresponding block numbers.

To perform opportunistic processing, the backup tool locks a page, checks that it is not dirty and then copies it to a private buffer. Next, it checks that the page has not been modified since the snapshot using back-references in Btrfs, and then unlocks the page. Finally, the data from the private buffer is sent out-of-order to the backup storage.

5.3 File Defragmentation

Due to the copy-on-write nature of Btrfs, any write to a file stores the new data in unused blocks. This layout reorganization causes fragmentation, especially for small random writes. Btrfs allows defragmenting a file by merging small extents with logically adjacent ones. The existing Btrfs tool allows defragmenting one file at a time at the user level. We have reimplemented this tool in the kernel to speed up defragmentation for multiple files and directories. Our in-kernel implementation uses metadata prefetching during namespace traversal, speeding it up by a factor of 10. We use this implementation as the baseline for our experiments.

Our opportunistic defragmenter monitors Exists notifications to track files that have data in memory, and prioritizes those files with the highest fraction of pages in memory compared to their size, similar to the example in Algorithm 1.

5.4 Garbage Collection

F2fs is a log-structured file system [50], designed to perform well on flash storage [39]. F2fs groups blocks in segments. When a block is updated, it is appended to the log, and its previous version becomes invalid (in some segment). Segments with many invalid blocks are cleaned by a background garbage collector that copies the remaining valid blocks in

Task	Type	Notification Mask	Processing Order	Duet Modifications
Scrubber	Block task	Added ∨ Dirty	Processes blocks by Btrfs extent key	Recently read blocks are not scrubbed
Backup	Block task	Exists	Processes files by inode number	Backup in-memory blocks out of order
Defragmentation	File task	Exists	Processes files by inode number	Prioritize files with blocks in memory
Garbage collector	Block task	Exists ∨ Flushed	Uses cost function to pick victim out of a segment group	Cost function adjusted to take into account blocks already in memory
Rsync	File task	Exists	Traverses directory hierarchy in depth-first order	Out-of-order transfer of files with pages already in memory

Table 3. Tasks adapted to use Duet and their characteristics

the segment to the log, freeing the segment for logging future writes. The garbage collector prioritizes segment cleaning using a cost function based on the amount of data that needs to be moved and the segment's age. It runs when the device is idle, cycles through 4096 segments at a time (instead of all segments on the device), and cleans one segment with the minimum cost (i.e. the most invalid blocks).

Our opportunistic garbage collector modifies the cost function to account for the number of valid blocks of a segment that are cached because these blocks save read operations. During cleaning, a segment's blocks are synchronously read from storage, and marked dirty in memory for asynchronous writeback. We conservatively weigh both read and write operations equally, and change the number of blocks that need to be moved from $valid\_blocks$ to $valid\_blocks - cached\_blocks/2$ in the cost function.

The garbage collector also monitors Flushed notifications. When a block is flushed to disk, it is mapped to a new segment, and its copy in the old segment is invalidated. On a flush event, we adjust the in-memory counters for both the old and new segments. Interestingly, the notion of completed work does not apply to the garbage collector because a segment can always become dirty again, and so the Duet done primitives are not used.

5.5 Rsync Application

Rsync is a widely-used user-level tool for synchronizing the contents of a source and a destination directory. It uses data checksums to find differences between source and destination files, sending only the updated data blocks. Rsync consists of three processes that communicate via sockets and pipes. The *sender* process is responsible for traversing the directory hierarchy at the source and sending the file metadata to the *receiver* process, which passes it to the *generator* process. The generator calculates file checksums and sends them to the sender, which then generates its own checksums to detect updated blocks. Finally, updated data is sent to the receiver, which updates the destination files.

The opportunistic rsync uses the Exists notifications to track files that have data in memory, prioritizing files with the highest number of pages in memory, similar to the example task shown in Algorithm 1. It ensures that the metadata for a file is sent once in the first step, either opportunistically or during normal operation.

5.6 Lessons Learned

This subsection outlines the most important lessons we have learned when adapting tasks for Duet.

Some tasks may require operating on a consistent view of at least a portion of the device or filesystem. For example, a file backup task may require that backed up files represent a consistent version of the data, unaffected by partial updates. Duet does not provide any such guarantees to tasks, apart from hints on data availability. The backup task we examined relies on the ability of Btrfs to take filesystem snapshots, to ensure backup consistency. Alternatively, a backup task could use file locking to ensure consistency at the file-level and leverage Duet events to prioritize files with in-memory pages, similar to our defragmentation task.

Tasks should not assume that data will be available (or unmodified) after being notified about an event. This helps avoid races and inconsistencies. For example, our backup task locks a page before checking its dirty status and whether it belongs to the snapshot, as described earlier in Section 5.2.

Maintenance tasks may consume CPU and memory resources while running, which could affect the performance of workloads. In our experience, maintenance tasks make moderate use of these resources, as they are usually bottlenecked on I/O. Thus, an I/O scheduler capable of assigning low priority to maintenance I/O works well [20, 26, 42]. Furthermore, maintenance work is usually partitioned in small chunks that can be scheduled around workloads. For example, rsync processes files in 32KB chunks. Overall, Duet is not dependent on the way that maintenance work is scheduled or partitioned, allowing tasks to individually regulate their impact on workloads.

6. Evaluation

This section evaluates the benefits of Duet. Our evaluation has three goals. First, we evaluate the ability of Duet to reduce I/O when a maintenance task runs together with a foreground workload. Second, we evaluate the I/O reduction when maintenance tasks are run concurrently, which implicitly enables them to collaborate on shared data. Third, we evaluate the overhead of Duet.

Section 6.1 describes our experimental setup. Section 6.2 and Section 6.3 quantify the I/O reduction when running maintenance tasks individually and concurrently. Section 6.4 evaluates the overhead of Duet using microbenchmarks. Section 6.5 concludes our evaluation by discussing the effect of other parameters on our approach.

6.1 Experimental Methodology

We have chosen to use the Filebench benchmark [25] as the foreground workload in our experiments. Filebench is a widely used benchmark that allows us the flexibility to change most aspects of its workload, allowing us to evaluate Duet for a range of workload characteristics.

An alternative would be to use real traces of file system activity for the workload, but we found few publicly available traces that can be replayed accurately [67]. Moreover, the traces did not contain sufficient information needed for our evaluation. For instance, existing traces do not provide information on files that are *not* accessed (or the corresponding fraction of the file system), but we need this information for some maintenance tasks. Using existing traces also does not allow for adjusting any workload parameters.

6.1.1 Workload Characteristics

Three workload characteristics have the most impact on our opportunistic approach: data overlap, read-write ratio and workload I/O rate. Next, we describe how we vary these characteristics to study their effect in our experiments.

Our approach reduces the I/O footprint of a maintenance task when the data accessed by the task *overlaps* with the data accessed by other ongoing maintenance tasks or with foreground workloads. While there is potential for high data overlap between maintenance tasks running concurrently, the data overlap with workloads depends on both the type of maintenance task and the workload. Many tasks, such as incremental backups and garbage collectors, tend to access hot areas of the file system, so data overlap with the workload is expected to be high. For tasks that access all data on a device, such as scrubbing, the data overlap will vary depending on the workload since there is high variability in the fraction of device data accessed across workloads [12, 51].

By default, Filebench uses a uniform distribution to pick the files it operates on, which gives it high coverage of the file system, i.e. a large percentage of the files get accessed, creating high data overlap with maintenance work. We modified Filebench in two ways to vary the amount of data overlap. First, we limit the data coverage of Filebench to different fractions of the overall file system. Second, we analyzed the Microsoft Production Build Server trace [34] and extracted traces of file events for three different storage devices. Figure 1 shows that the file access distributions of the Microsoft traces are highly skewed compared to Filebench's uniform distribution policy. We have modified Filebench to pick files using the Microsoft distributions, and we show results for both the uniform and the skewed file access distributions.

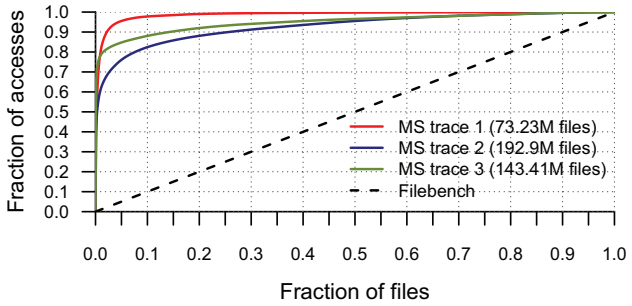

Figure 1. File access distributions for Microsoft traces and the Filebench benchmark

The *read-write ratio* of workload operations also impacts opportunistic processing because maintenance tasks react differently to updates. For example, a file may be further fragmented or defragmented due to a write, and sharing with the backup snapshot is broken when a block is updated. We use Filebench with three of its default workload personalities. The *fileserver* personality is a write-heavy workload, with a read-write ratio of 1:2. The *webproxy* personality is more read-heavy, with read-write ratio of 4:1. Finally, the *webserver* personality is a read-mostly workload with a 10:1 read-write ratio, with all write operations appending data to a single log file. Note that maintenance operations are useful even on systems that run read-mostly workloads. For example, scrubbing can help detect and repair data corruption caused by hardware failure, and similarly, backup can help with recovery from software bugs, administrative errors or security incidents.

Finally, the *workload I/O rate* affects the opportunistic processing performed by Duet tasks. Increased workload I/O rate creates more opportunities for synergy with maintenance work, but it reduces the overall time that the storage device remains idle. When maintenance tasks are run at idle times to reduce their impact on workloads, they need enough idle time to complete their work. In our experiments, we control workload I/O by using rate-limiting commands available in Filebench to throttle its bandwidth.

6.1.2 Evaluation Metrics

Our evaluation uses three metrics: I/O saved, maximum utilization, and speedup. The first metric measures the maintenance *I/O saved* by Duet. The second metric takes into account that when tasks only run at idle periods, they may not complete under high device utilization. We define *device utilization* as the percentage of time during which foreground I/O requests keep the device busy, when *no* maintenance tasks are being run. This metric is reported as the `%util` statistic of the `iostat` tool. We profiled each Filebench personality with different levels of throttling (and no maintenance load) to achieve a given device utilization, and report results for utilization values ranging from 0-100%, in 10% intervals. The *maximum utilization* is the highest device utilization at which maintenance work can still be com-

Metric	Description
I/O saved	$\frac{\text{Maintenance I/O saved with Duet}}{\text{Total maintenance I/O performed without Duet}}$
Maximum utilization	Maximum device utilization *by the foreground workload*, at which maintenance work completes by the end of the experiment
Speedup	$\frac{\text{Task completion time without Duet}}{\text{Task completion time with Duet}}$

Table 4. Duet evaluation metrics

pleted in 30 minutes, by the end of the experiment. Finally, when tasks run with normal IO priority, such as Rsync, we run Filebench unthrottled, and measure the *speedup* of the maintenance task. Table 4 summarizes these metrics; higher values are always better.

6.1.3 Experimental Setup

We conduct our experiments for 30 minutes on a file system populated with 50GB of data, during which maintenance tasks run concurrently with the workload. At this rate, maintenance can be run weekly on a 16TB storage system. Each experiment is run three times, and every data point in our plots is an average across these runs. Generally, there is low variability across the runs, and so we omit the error margins. Otherwise, we show 95% confidence intervals. We use CFQ, the default Linux I/O scheduler that supports I/O prioritization. Our in-kernel tasks issue their maintenance I/O requests at Idle priority. These requests are serviced only after the device has remained idle for some time. We have measured the latency of Filebench workloads, both without and with one or two maintenance tasks running concurrently, at various device utilizations, and found that there is insignificant impact on workload latency. As an example, the webserver workload latency at 50% device utilization, without any maintenance task is $11.67 \pm 0.12ms$. When scrubbing, it is $11.60 \pm 0.25ms$, and with backup it is $11.82 \pm 0.16ms$.

All experiments are run on HP ProLiant DL160 Gen8 servers, equipped with Intel Xeon E5-2650 CPUs with 8 cores, and 300GB SAS drives running at 10K RPM. While the machine has 32GB of DDR3 RAM, we boot it with 2GB of memory to have a realistic page cache size compared to our working set of 50GB. We examine the effect of the page cache size on our approach in more detail in Section 6.5.

6.2 Running Single Tasks

This section evaluates the ability of Duet to perform maintenance work opportunistically. We evaluate Duet with five different maintenance tasks, while varying the data overlap between the maintenance and foreground work.

Scrubbing We implemented opportunistic scrubbing by modifying 75 of the 3500 lines of code of the Btrfs scrubber. Our evaluation with different Filebench workloads shows that, as expected, the I/O saved with opportunistic scrubbing

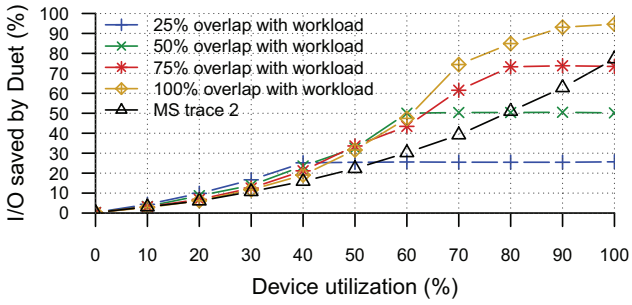

Figure 2. I/O saved when the scrubbing task is run together with the webserver workload

increases with higher device utilization and the data overlap between the workload and the scrubber. Figure 2 shows the results for the webserver workload. As utilization and data overlap increase, more data gets accessed by the workload, avoiding the need to scrub it. Beyond a certain utilization, the I/O saved reaches an upper limit, equal to the data overlap. At this point, the scrubber skips scrubbing all shared data because the workload accesses it before the scrubber processes it as part of its sequential scan, showing that Duet allows exploiting any available synergy between I/O tasks.

Savings decrease for more write-heavy workloads. Recall that we unset the `done` bit for updated blocks, if they have not already been scrubbed in the course of normal (sequential) scrubbing. The webproxy performs similarly to the webserver because its write operations mainly append data to files, allowing read operations to significantly reduce scrubbing work. However, the write-intensive fileserver workload has 40% of the IO savings compared to the other two workloads, since any file can be overwritten, so the opportunistic savings are lower. When the skewed file access distribution is used, the results are similar, but savings are decreased by 15-30%. This decrease is small, despite the majority of accesses being directed to a small fraction of the files, because it is sufficient for a file to be read once to be considered scrubbed.

By reducing the required I/O, Duet allows scrubbing to complete faster. Table 5 shows the maximum device utilization at which scrubbing completes in a 30 minute interval. Note that for normal scrubbing to complete, the device must not be busier than 70% (column 4), regardless of the workload, because the amount of work remains constant. With Duet, the maximum utilization increases with the data overlap. Devices can be busier, from 70% to 100% (column 5), depending on the characteristics of the workload.

Backup We implemented opportunistic backup by modifying 140 of the 4900 lines of code of the Btrfs backup tool. The backup tool processes files in the order of their inode numbers, and each file is processed fully before moving to the next one. This results in more random accesss than scrubbing, and so the backup requires almost twice the amount of time needed for scrubbing. This extra time allows the backup task to interact longer with the foreground workload, result-

Workload, Read-Write ratio	Overlap with maintenance	File access distribution	Scrubbing		Backup		Defragmentation	
			Baseline	Duet	Baseline	Duet	Baseline	Duet
Webserver, 10:1	25%	Uniform	70%	80%	40%	50%	40%	40%
	50%	Uniform	70%	80%	40%	60%	40%	40%
	75%	Uniform	70%	90%	40%	70%	40%	50%
	100%	Uniform	70%	100%	40%	100%	40%	60%
	100%	MS trace	70%	80%	40%	60%	40%	60%
Webproxy, 4:1	100%	Uniform	70%	90%	40%	90%	50%	70%
	100%	MS trace	70%	80%	40%	50%	50%	60%
Fileserver, 1:2	100%	Uniform	70%	80%	40%	60%	60%	70%
	100%	MS trace	70%	70%	40%	50%	50%	60%

This table shows the maximum utilization (in 10% intervals) at which each maintenance task can still complete its work in a 30 minute interval. Higher values mean that the foreground workload is able to utilize the device more. Higher read-write ratios, higher data overlap and a uniform distribution of file accesses improves opportunistic processing, allowing higher maximum utilization.

Table 5. Maximum utilization with and without Duet for Btrfs maintenance tasks

ing in more opportunities for I/O savings. Therefore, the I/O saved reaches its upper limit at a much lower device utilization compared to scrubbing. For example, Figure 3 shows that with 25% overlap, the maximum I/O saved is reached at 20% utilization versus 40% for scrubbing (see Figure 2).

With the additional seeks, the baseline can tolerate a *maximum utilization* of 40%, which is close to half that for baseline scrubbing (columns 6 and 4 in Table 5). Duet reduces random I/O, allowing backup to complete on devices with 50-100% utilization (25-150% busier than the baseline).

When a block is updated, it gets copied to a new location and is no longer shared with the backup snapshot. Therefore, subsequent reads to the same file offset do not benefit the backup task, as they no longer refer to the snapshot data. As a result, the I/O saved decreases with decreasing read-write ratio. This effect applies both to writes that append and overwrite data, as well as to deletions and re-creations of files. Webproxy, which includes file append, delete, and create operations shows the impact of breaking sharing with the backup snapshot in this way. It yields 80% of the I/O savings of webserver, while fileserver, which also breaks sharing by overwriting files, yields up to 40% of the IO savings of webserver.

Defragmentation We implemented opportunistic defragmentation by modifying 95 of the 1200 lines of code of the Btrfs defragmenter. The defragmenter merges small, logically adjacent extents by bringing them into memory, and then writing them back to storage as part of the same transaction, thus creating a single, larger extent. The total I/O required to defragment a file consists of the number of pages read and then written, which is twice the number of pages in the new extent. Recall that we reduce this I/O by prioritizing files that have more pages in memory. Therefore, the I/O saved is the sum of the number of pages in memory when an extent is processed and the number of pages that were already marked dirty by the workload. The former do not require I/O and the latter will be flushed soon anyway.[2] Our experiments are performed on a 10% fragmented file system.

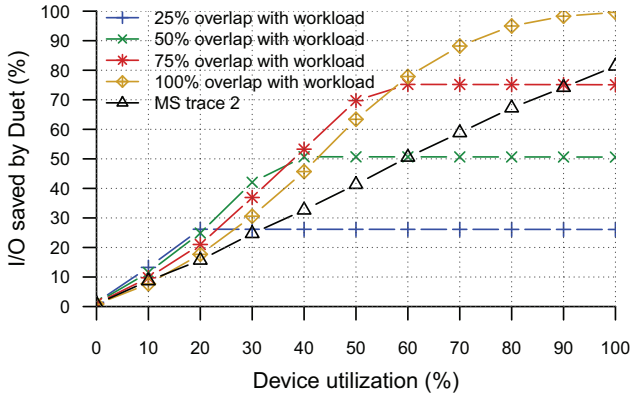

Figure 3. I/O saved when the backup task is run together with the webserver workload

The I/O saved results with defragmentation are similar but smaller than the scrubbing and backup results shown in Figure 2 and Figure 3. For read-heavy workloads such as webserver, we only save on read accesses, which are close to 50% of the total I/O. On the other hand, a workload appending data to files, such as webproxy, can also save I/O by defragmenting files with dirty pages in memory. This benefit does not apply to a write-heavy workload like fileserver, which overwrites files thus defragmenting them, reducing the total work performed by the maintenance task. In this case, the savings are available for only those files that the opportunistic defragmentation task processed. Similar to scrubbing and backup, we find that using the skewed file access distribution reduces the I/O saved by 15-30%.

The maximum utilization results for defragmentation are similar to the previous two tasks, and shown in the last two

[2] The page could be modified again between the time we flush it as part of the defragmentation process, and the time when it was originally planned to be flushed. We cannot account for this case.

Workload, device utilization	Segment cleaning time	
	Baseline	Duet
Fileserver, 40%	17.1 ± 3.5ms	16.1 ± 2.6ms
Fileserver, 50%	17.0 ± 1.1ms	10.6 ± 2.4ms
Fileserver, 60%	16.4 ± 1.4ms	9.0 ± 3.6ms
Fileserver, 70%	15.8 ± 1.0ms	7.9 ± 2.8ms

Table 6. Segment cleaning time with and without Duet

columns of Table 5. The difference between the baseline and Duet is less pronounced compared to the other workloads, because most defragmentation writes still need to be performed. As a result, we improve upon the baseline by at most 50% across different workloads.

Garbage collection The opportunistic garbage collector was implemented by adding 150 lines to the 1400 lines of code of the F2fs in-kernel garbage collector. Our aim is to clean segments faster by selecting segments with cached blocks. Reducing the segment cleaning time is crucial when the file system is running out of clean segments. In that case, F2fs transitions to overwriting invalid blocks in scattered segments. When that happens, we have measured a 57% increase in filebench latency, and 29% increase in device utilization. However, even when there is no pressure for clean segments, speeding up cleaning time enables consuming less idle time or cleaning more segments.

We have used the fileserver workload for these experiments because it is the only workload that overwrites and deletes existing blocks. We present our results when the fileserver is run between 40% and 70% device utilization. At lower utilization, the garbage collector does not run, and at higher utilization, there is not enough idle time for the garbage collector to run. Table 6 shows the average cleaning time for a segment, with and without Duet. Duet improves cleaning performance at higher utilization, when cleaning is most needed. Performance improves because more segment blocks are cached in memory, and the opportunistic garbage collector picks segments requiring fewer read operations.

Rsync We implemented opportunistic Rsync by modifying 300 of the 45000 lines of code in the Rsync application. We evaluated opportunistic Rsync by running it locally, copying 50GB of data between two disks, while running Filebench on the source device during the transfer. Rsync is used locally for various tasks, such as when performing snapshot-based backups [65], synchronizing data across VMs, and for copying data when upgrading devices [63].

The total I/O required to synchronize a file between the source and destination folders includes reading all of its data at both the sender and the receiver side to produce checksums, and writing the updated data blocks on the receiving side. In our experiments, the destination folder is initially empty, so the files are not checksummed. Instead, their data is sent to the receiving side and the I/O operations required per file are twice the number of data blocks of the file, for reading and writing each once. Similar to our previous experiments, we find that read I/O can be reduced proportional to the data overlap between the workload and Rsync. Similar to defragmentation, write I/O cannot be saved, so with 100% overlap we can save 50% of the total I/O.

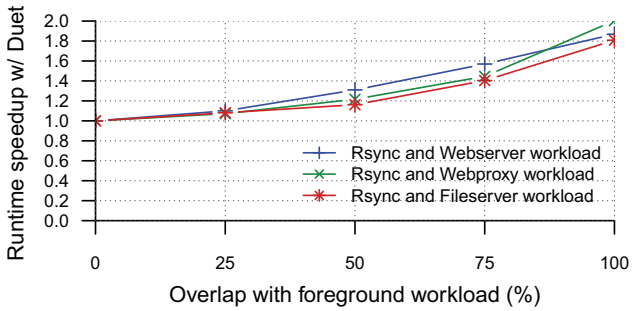

Figure 4. Runtime speedup with different overlap for Rsync

In the previous experiments, the maintenance task is run at a lower priority, and thus the Filebench workload is run throttled to allow the maintenance task to make progress. Rsync, however, runs at normal I/O priority, affecting Filebench throughput by up to 27%. Thus, in this experiment we run Filebench unthrottled and measure the speedup of Rsync. Figure 4 shows the results for the webserver workload. It shows that the speedup increases with higher data overlap, with Rsync completing twice as fast at 100% data overlap. This speedup reduces the time period during which Rsync impacts the workload.

6.3 Running Multiple Tasks Together

Today, maintenance tasks are run in isolation to avoid interference and slowdown. This section shows that when Duet tasks are run concurrently, I/O savings increase due to higher data overlap, enabling them to complete their work faster.

Scrubbing and Backup In this experiment, we run scrubbing and backup together with the different Filebench workloads. Figure 5 shows the results for the amount of I/O saved for the webserver workload. With Duet, data accesses by either the backup task, or the scrubber, benefit the other task. As a result, even when Filebench is not run (0% utilization), Duet reduces the total I/O needed to complete maintenance work by at least 50%. Similar to previous results, higher device utilization and higher data overlap increase I/O savings further. The results for other workloads are similar to the results discussed previously in Section 6.2, with more write-intensive workloads resulting in lower savings.

The significant work reduction allows us to also complete maintenance work on busier devices. Figure 6 shows the maintenance work completed at various device utilizations. While the baseline tasks fail to complete maintenance work beyond 30% device utilization, Duet allows 70-90% maximum utilization.

Scrubbing, Backup, and Defragmentation We also experimented with combining three maintenance tasks. As shown in Figure 7, roughly 55% of maintenance I/O is needed when

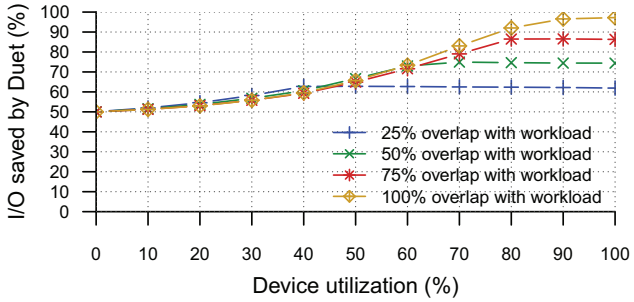

Figure 5. I/O saved when the scrubbing and backup tasks are run together with the webserver workload

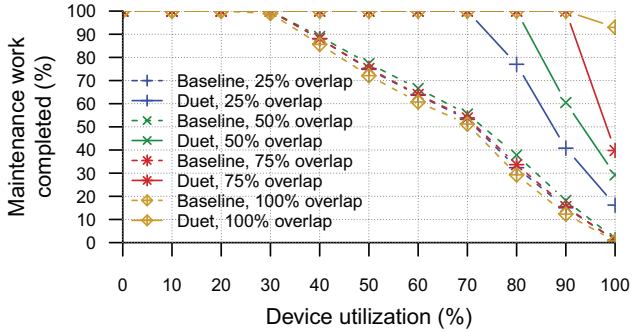

Figure 6. Maintenance work completed when scrubbing and backup are run together with the webserver workload

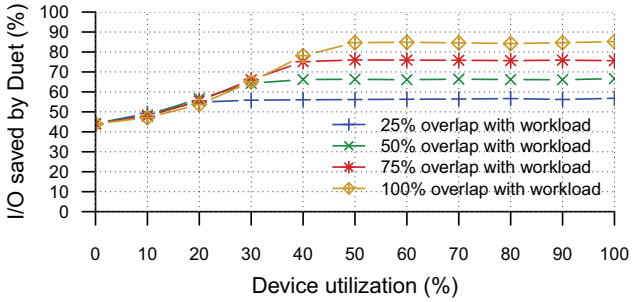

Figure 7. I/O saved when scrubbing, backup, and defragmentation are run together with the webserver workload

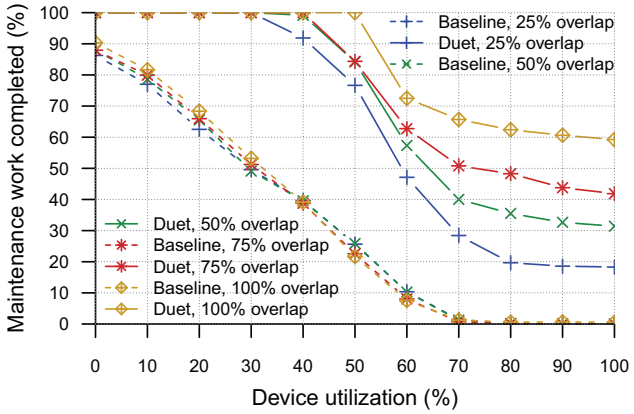

Figure 8. Maintenance work completed when scrubbing, backup, and defragmentation are run together with the webserver workload. Without Duet, maintenance work fails to complete even when the device is idle.

the tasks run without Filebench, as all three are accessing the same data. This I/O includes one pass over the file system, and the write requests for the defragmentater, which cannot be saved. The maximum I/O saved with the read-only webserver workload is roughly 80%, which is almost all maintenance work, other than the writes needed for defragmentation. More write-intensive workloads perform worse, but still achieve I/O savings up to 60%.

Figure 8 shows the maintenance work completed at different device utilizations. Duet completes all maintenance work even with 50% device utilization, which the baseline can complete only 25% of the work even on an idle device.

6.4 Performance Overhead

CPU overhead To determine the overhead of Duet, we run a simple file task that registers the root directory of the file system with Duet, and either remains idle, or fetches events periodically in 10, 20 and 40ms intervals, sleeping in between. We chose these intervals because they are close to the typical Rsync fetch interval, which is 20ms. To generate page events, we run the webserver workload unthrottled on the file system, which generates roughly 12 page events/ms. We estimate the CPU available to applications by running a program that spins in a tight loop at low priority, and then measure the loop counter value periodically. Based on the counter value, the CPU overhead of using Duet is roughly 0.5-1.5%, as shown in Figure 9. State-based notifications have slightly lower overhead because events can be merged. The fetch frequency does not change the number of events that need to be copied to user space and thus has a small effect on overhead.

Memory overhead Duet maintains item descriptors for pages and bitmaps for up to N concurrent sessions. For $N = 16$, an item descriptor requires 32 bytes (inode number, offset, 16-byte flag array and hash node). With state notifications, the worst case memory overhead is 1.5% ($\frac{32*2}{4096}$), as explained in Section 4.2. In practice, fetch is called often enough that a buffer of 256 items does not fill up. At most, such a buffer would require 2.3KB, since we only return the flag variable for one session, and no hash node.

Item bitmaps are dynamically allocated when the range that the bitmap represents contains both set and unset bits. In the worst case, block tasks will use 1 bit per device block, and file tasks will use 2 bits per inode. In our experiments, when scrubbing a fully utilized disk with 100% overlap with the workload, the bitmap required 1.47MB, while the worst case estimate for 50GB of data is 1.56MB.

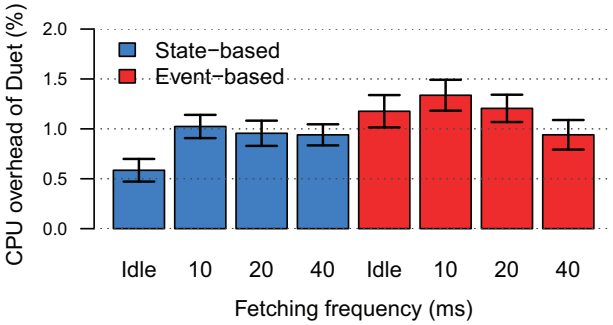

Figure 9. CPU overhead of Duet

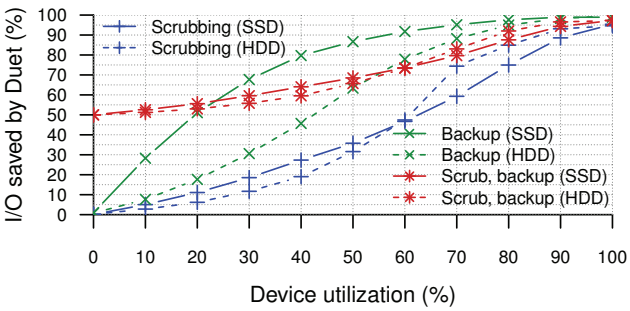

Figure 10. I/O saved on a solid-state drive

6.5 Effect of Other Parameters

Storage device type The results presented so far have been based on a hard drive setup. To evaluate the efficiency of Duet on faster devices, we experimented with a consumer-grade solid-state drive (Intel SSD 510 120GB), and found that for this model, the I/O savings and the work completed did not change qualitatively. For example, the scrubber completes in half the time, but the throughput of the workload is also much higher, resulting in the same number of accesses and savings, as shown in Figure 10. With backup, we achieve higher savings on the SSD. The reason is that the backup tool issues 64KB random reads, and the random read performance of our Intel 510 SSD and our enterprise 10K hard drive is roughly similar, about 21 MB/s [58]. Hence the default backup time is similar on the hard drive and the SSD. However, the workload is more sequential and has higher throughput on the SSD. This allows more data overlap and thus the Duet-enabled backup achieves higher I/O savings.

I/O prioritization Our in-kernel maintenance tasks were run at lower priority, which has minimal impact on the workload. We also experimented with the Linux Deadline I/O scheduler, which does not allow prioritizing different streams of I/O. We find that without I/O prioritization, workload requests are slowed down significantly when a maintenance task is running. Maintenance work finishes faster but the workload issues fewer data requests and thus the I/O saved is reduced. Hence, Duet works better when maintenance tasks run at low priority.

Cold data placement We define cold data as the part of the file system that is not accessed by the workload but requires maintenance. We find that the physical placement of this data on the storage device does not affect performance, even when cold data is separated from the data accessed by the workload. Since maintenance I/O occurs when the device has been idle, additional seeks occur only when switching between maintenance tasks and workloads.

Page cache size We also modified the ratio of the page cache size to the file system size. This ratio is expected to affect the workload's performance, but it may also affect the maintenance task. With a larger cache, more synergies and thus more I/O saving are expected. In our experiments, the page cache size is roughly 2% of the file system data accessed by the task. Surprisingly, changing this ratio had a marginal effect on our results. We believe that maintenance tasks access significant amounts of data, and hence it is the out-of-order processing, rather than cache locality that provides the most benefits.

7. Conclusion and Future work

We take advantage of the loose requirements on work ordering in maintenance tasks: maintenance work, by definition, runs in the background, and its processing can be generally reordered without affecting the correctness, reliability or performance guarantees provided by the task.

We have presented a model that allows storage maintenance to be performed opportunistically based on data cached in memory. Maintenance tasks perform out-of-order operations on this data, reducing the total I/O needed. We designed and built Duet, a framework that provides notifications about page-level events to tasks. This granularity works well for both the block or file granularity processing performed by maintenance tasks, requiring relatively small changes in them for performing out-of-order processing. Our evaluation shows that opportunistic maintenance tasks require less I/O and complete faster, and the benefits increase when tasks run concurrently.

Our work suggests that maintenance work does not have to be relegated to a maintenance window, which is hard to schedule because idle times are unpredictable and may not be sufficient for the work needed. Instead, maintenance work should be done at low priority, continuously and synergistically with other workloads to minimize its impact.

Acknowledgments

We thank the undergraduate students that helped us improve Duet's design by contributing to its evaluation: Abdi Dahir, Max Holden, Pranay U. Jain, and Suvanjan Mukherjee. We also thank our shepherd, Steven Swanson, and the anonymous reviewers for their invaluable feedback. We especially appreciate the feedback from both faculty and students of the CSL and SysNet groups at the University of Toronto, especially Michael Stumm, Ding Yuan, and David Lie. This work is supported by the NSERC Discovery Grant program.

References

[1] AGRAWAL, N., PRABHAKARAN, V., WOBBER, T., DAVIS, J. D., MANASSE, M., AND PANIGRAHY, R. Design Tradeoffs for SSD Performance. In *USENIX Annual Technical Conference* (2008), pp. 57–70.

[2] AGRAWAL, P., KIFER, D., AND OLSTON, C. Scheduling Shared Scans of Large Data Files. *Proc. VLDB Endow. 1*, 1 (Aug. 2008), 958–969.

[3] AMVROSIADIS, G., AND BHADKAMKAR, M. Identifying Trends in Enterprise Data Protection Systems. In *USENIX Annual Technical Conference* (2015), pp. 151 – 164.

[4] ANANTHANARAYANAN, G., GHODSI, A., WANG, A., BORTHAKUR, D., KANDULA, S., SHENKER, S., AND STOICA, I. PACMan: Coordinated Memory Caching for Parallel Jobs. In *USENIX Conference on Networked Systems Design and Implementation (NSDI)* (2012).

[5] ARCSERVE. arcserve Unified Data Protection. http://www.arcserve.com, May 2014.

[6] ARUMUGAM, S., DOBRA, A., JERMAINE, C. M., PANSARE, N., AND PEREZ, L. The DataPath System: A Data-centric Analytic Processing Engine for Large Data Warehouses. In *ACM SIGMOD International Conference on Management of Data* (2010).

[7] AVG TECHNOLOGIES. Anti-Virus Guard (AVG) 2015.0.5645. http://www.avg.com, December 2014.

[8] BACHMAT, E., AND SCHINDLER, J. Analysis of Methods for Scheduling Low Priority Disk Drive Tasks. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (2002).

[9] BACULA SYSTEMS. Bacula 7.0.5. http://www.bacula.org, July 2014.

[10] BAIRAVASUNDARAM, L. N., GOODSON, G. R., PASUPATHY, S., AND SCHINDLER, J. An Analysis of Latent Sector Errors in Disk Drives. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (2007).

[11] BAIRAVASUNDARAM, L. N., GOODSON, G. R., SCHROEDER, B., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. An Analysis of Data Corruption in the Storage Stack. In *USENIX Conference on File and Storage Technologies (FAST)* (2008).

[12] BHADKAMKAR, M., GUERRA, J., USECHE, L., BURNETT, S., LIPTAK, J., RANGASWAMI, R., AND HRISTIDIS, V. Borg: Block-reorganization for self-optimizing storage systems. In *USENIX Conference on File and Storage Technologies (FAST)* (2009).

[13] COMMVAULT SYSTEMS. Get Smart About Big Data : Integrated Backup, Archive & Reporting to Solve Big Data Management Problems. http://www.commvault.com/resource-library/5445a1123fab55441d000eae/commvault_vol3_issue1.pdf, July 2013.

[14] COMMVAULT SYSTEMS. Data Analytics - Overview. http://documentation.commvault.com/hds/v10/article?p=features/reports/web_reports/data_analytics.htm, 2014.

[15] COMMVAULT SYSTEMS INC. CommVault Simpana 10. http://www.commvault.com/simpana-software, April 2014.

[16] Allocate-on-flush (Delayed Allocation). http://en.wikipedia.org/wiki/Allocate-on-flush.

[17] DELL INC. Dell NetVault 10.0. http://software.dell.com/products/netvault-backup, May 2014.

[18] DIMENSIONAL RESEARCH. The state of IT recovery for SMBs. http://axcient.com/state-of-it-recovery-for-smbs, Oct. 2014.

[19] DIMITRIJEVIC, Z., RANGASWAMI, R., AND CHANG, E. Y. Systems Support for Preemptive Disk Scheduling. *IEEE Transactions on Computers 54*, 10 (2005), 1314–1326.

[20] EGGERT, L., AND TOUCH, J. D. Idletime Scheduling with Preemption Intervals. In *ACM Symposium on Operating Systems Principles (SOSP)* (2005).

[21] EL-SHIMI, A., KALACH, R., KUMAR, A., OLTEAN, A., LI, J., AND SENGUPTA, S. Primary Data Deduplication – Large Scale Study and System Design. In *USENIX Annual Technical Conference* (2012).

[22] EMC CORPORATION. Data Integrity on VNX (White Paper). http://www.emc.com/collateral/hardware/white-papers/h11181-data-integrity-wp.pdf, November 2012.

[23] EMC CORPORATION. EMC NetWorker 8.2. http://www.emc.com/data-protection/networker.htm, July 2014.

[24] ESET. ESET NOD32 Antivirus 8.0.304. http://www.eset.com, October 2014.

[25] Filebench v1.4.9.1, September 2011. http://filebench.sourceforge.net.

[26] GOLDING, R., BOSCH, P., STAELIN, C., SULLIVAN, T., AND WILKES, J. Idleness is not sloth. In *USENIX Annual Technical Conference* (1995).

[27] HEWLETT-PACKARD COMPANY. HP Data Protector 9.0.1. http://www.autonomy.com/products/data-protector, August 2014.

[28] IBM CORPORATION. IBM Tivoli Storage Manager 7.1. http://www.ibm.com/software/products/en/tivostormana, November 2013.

[29] Inotify. http://en.wikipedia.org/wiki/Inotify.

[30] INTEL SECURITY GROUP. McAfee AntiVirus Plus. http://www.mcafee.com.

[31] IRON MOUNTAIN. Data Backup and Recovery Benchmark Report. http://www.ironmountain.com/Knowledge-Center/Reference-Library/View-by-Document-Type/White-Papers-Briefs/I/Iron-Mountain-Data-Backup-and-Recovery-Benchmark-Report.aspx, 2013.

[32] IYER, S., AND DRUSCHEL, P. Anticipatory Scheduling: A Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O. In *ACM Symposium on Operating Systems Principles (SOSP)* (2001), pp. 117–130.

[33] KASPERSKY LAB. Kaspersky Anti-Virus 15.0.2. http://www.kaspersky.com/kaspersky/_anti-virus, February 2015.

[34] KAVALANEKAR, S., WORTHINGTON, B., ZHANG, Q., AND SHARDA, V. Characterization of storage workload traces from production Windows Servers. In *IEEE International Symposium on Workload Characterization* (2008).

[35] KERRISK, M. Filesystem notification series. https://lwn.net/Articles/605313, July 2014.

[36] KREVAT, E., TUCEK, J., AND GANGER, G. R. Disks Are Like Snowflakes: No Two Are Alike. In *USENIX Workshop on Hot Topics in Operating Systems (HotOS)* (2011).

[37] KRIOUKOV, A., BAIRAVASUNDARAM, L. N., GOODSON, G. R., SRINIVASAN, K., THELEN, R., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEU, R. H. Parity Lost and Parity Regained. In *USENIX Conference on File and Storage Technologies (FAST)* (2008).

[38] LAMPSON, B. W. Hints for computer system design. In *ACM Symposium on Operating Systems Principles (SOSP)* (1983), pp. 33–48.

[39] LEE, C., SIM, D., HWANG, J., AND CHO, S. F2fs: A new file system for flash storage. In *USENIX Conference on File and Storage Technologies (FAST)* (Feb. 2015), pp. 273–286.

[40] LOIZIDES, C. Journaling-Filesystem Fragmentation Project. http://www-stud.rbi.informatik.uni-frankfurt.de/~loizides/reiserfs/fibmap.html, 2004.

[41] LUMB, C. R., SCHINDLER, J., GANGER, G. R., NAGLE, D. F., AND RIEDEL, E. Towards Higher Disk Head Utilization: Extracting Free Bandwidth From Busy Disk Drives. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2000).

[42] MI, N., RISKA, A., LI, X., SMIRNI, E., AND RIEDEL, E. Restrained Utilization of Idleness for Transparent Scheduling of Background Tasks. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (2009).

[43] MIRETSKIY, Y., DAS, A., WRIGHT, C. P., AND ZADOK, E. AVFS: An on-access anti-virus file system. In *USENIX Security Symposium* (2004).

[44] OPREA, A., AND JUELS, A. A Clean-slate Look at Disk Scrubbing. In *USENIX Conference on File and Storage Technologies (FAST)* (2010).

[45] ORACLE CORPORATION. Oracle Solaris ZFS Administration Guide: Checking ZFS File System Integrity. http://docs.oracle.com/cd/E23823_01/html/819-5461/gbbwa.html, 2012.

[46] RIEDEL, E., FALOUTSOS, C., GANGER, G. R., AND NAGLE, D. F. Data Mining on an OLTP System (Nearly) for Free. In *ACM SIGMOD International Conference on Management of Data* (2000).

[47] RISKA, A., AND RIEDEL, E. IRAW: Idle read after write. In *USENIX Annual Technical Conference* (2008).

[48] RODDY, M. Adventures in Luserland: Finding Disk Sectors Associated with File Records. http://www.wd-3.com/archive/luserland.htm, 2003.

[49] RODEH, O., BACIK, J., AND MASON, C. Btrfs: The Linux B-Tree Filesystem. *ACM Transactions on Storage 9*, 3 (2013), 9:1–9:32.

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

[51] RUEMMLER, C., AND WILKES, J. A trace-driven analysis of disk working set sizes, Apr. 1993.

[52] SCHWARZ, T. J. E., XIN, Q., MILLER, E. L., LONG, D. D. E., HOSPODOR, A., AND NG, S. Disk Scrubbing in Large Archival Storage Systems. In *Proc. of IEEE MASCOTS* (2004).

[53] SLOANE, G. Enabling VMware vShield Endpoint in a VMware Horizon View Environment (White Paper). http://www.vmware.com/files/pdf/techpaper/vmware-horizon-view-vshield-endpoint-antivirus.pdf, 2014.

[54] SON, S. W., CHEN, G., AND KANDEMIR, M. Disk Layout Optimization for Reducing Energy Consumption. In *Proc. of the 19th International Conference on Supercomputing* (2005).

[55] SYMANTEC CORPORATION. Norton Security 22.0.2. http://us.norton.com/norton-security-antivirus, October 2014.

[56] SYMANTEC CORPORATION. Symantec NetBackup 7.6.0.4. http://www.symantec.com/backup-software, November 2014.

[57] THERESKA, E., SCHINDLER, J., BUCY, J., SALMON, B., LUMB, C. R., AND GANGER, G. R. A Framework for Building Unobtrusive Disk Maintenance Applications. In *USENIX Conference on File and Storage Technologies (FAST)* (2004).

[58] TOM'S HARDWARE. As-ssd 4k random read. http://www.tomshardware.com/charts/ssd-charts-2014/AS-SSD-4K-Random-Read,Marque_fbrandx14,2784.html, 2014.

[59] TRIDGELL, A., AND MACKERRAS, P. The rsync algorithm. Technical Report TR-CS-96-05, Australian National University, June 1996.

[60] TRIDGELL, A., MACKERRAS, P., AND W., D. rsync 3.1.1. https://rsync.samba.org/, June 2014.

[61] TSO, T. Ext2 Filesystems Utilities. http://e2fsprogs.sourceforge.net/, Dec. 2010.

[62] UNTERBRUNNER, P., GIANNIKIS, G., ALONSO, G., FAUSER, D., AND KOSSMANN, D. Predictable Performance for Unpredictable Workloads. *Proc. VLDB Endow. 2*, 1 (2009), 706–717.

[63] VAN WINKEL, J. A look at rsync performance. http://lwn.net/Articles/400489/, 2010.

[64] VANSON BOURNE. Virtualization Data Protection Report 2013 – SMB edition. http://www.dabcc.com/documentlibrary/file/virtualization-data-protection-report-smb-2013.pdf, 2013.

[65] VENEZIA, P. Why you should be using rsync. http://www.infoworld.com/article/2612246/

data-center/why-you-should-be-using-rsync.html, 2013.

[66] WANG, J., AND HU, Y. WOLF – A Novel Reordering Write Buffer to Boost the Performance of Log-Structured File System. In *USENIX Conference on File and Storage Technologies (FAST)* (2002).

[67] WEISS, Z., HARTER, T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Root: Replaying multithreaded traces with resource-oriented ordering. In *ACM Symposium on Operating Systems Principles (SOSP)* (2013), pp. 373–387.

[68] ZMANDA INC. Amanda 3.3.6. http://amanda.zmanda.com, July 2014.