FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

State-of-the-art Garbage Collection Policies for NILFS2

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering/Internet Computing

eingereicht von

Andreas Rohner, Bsc
Matrikelnummer 0502196

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. M. Anton Ertl

Wien, 23. Jänner 2018

Andreas Rohner	M. Anton Ertl

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

State-of-the-art Garbage Collection Policies for NILFS2

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering/Internet Computing

by

Andreas Rohner, Bsc
Registration Number 0502196

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. M. Anton Ertl

Vienna, 23rd January, 2018

	Andreas Rohner	M. Anton Ertl

Erklärung zur Verfassung der Arbeit

Andreas Rohner, Bsc
Grundsteingasse 43/22

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 23. Jänner 2018

Andreas Rohner

Danksagung

Ich danke meinem Betreuer Ao.Univ.Prof. Dipl.-Ing. Dr.techn. M. Anton Ertl für seine Geduld und Unterstützung. Außerdem schulde ich Ryusuke Konishi, dem Maintainer des NILFS2-Subsystems im Linux-Kernel, Dank für die Durchsicht meiner Patches und die vielen wertvollen Verbesserungsvorschläge. Des Weiteren möchte ich mich bei Clemens Eisserer und Daniel Soukup für das Korrekturlesen der Arbeit bedanken. Der größte Dank aber gebührt meinen Eltern, die mich stets unterstützt und mir das Studium ermöglicht haben.

Acknowledgements

I would like to thank my advisor Ao.Univ.Prof. Dipl.-Ing. Dr.techn. M. Anton Ertl for his patience and support. Moreover, I owe a debt of gratitude to Ryusuke Konishi, who is the maintainer for the NILFS2 subsystem in the Linux kernel, for his review of my patches and his many valuable suggestions for improvement. I would also like to thank Clemens Eisserer and Daniel Soukup for their help in proofreading this thesis. However, my biggest thanks go out to my parents, who always supported me and made it possible for me to pursue my studies.

Kurzfassung

NILFS2 [KAS$^+$06] ist ein sogenanntes log-structured Dateisystem für Linux. Log-structured Dateisysteme schreiben sowohl Daten als auch die zugehörigen Metadaten in Form eines sequentiellen Logs. Dadurch wird die höhere sequentielle Schreibgeschwindigkeit der meisten Datenträger und Festplatten ausgenutzt. Diese Art von Dateisystem macht allerdings eine Form von Garbage-Collection notwendig. Das aktuelle NILFS2 verwendet eine sehr einfache Garbage-Collection-Strategie namens *Timestamp*. Diese Arbeit beschreibt die Implementierung von zwei zusätzlichen state-of-the-art Algorithmen (*Greedy* und *Cost-Benefit*) und vergleicht deren Performance. Des Weiteren werden in dieser Arbeit zwei Algorithmen zum Sammeln und Speichern der notwendigen Metadaten vorgestellt. Das Ergebnis ist eine Leistungsverbesserung des Dateisystems um einen Faktor 1.53 für einen Großteil der Einsatzgebiete.

Abstract

NILFS2 [KAS^{+}06] is a log-structured file system for Linux. Log-structured file systems write data and meta-data into a sequential log, to take advantage of the better sequential write performance of most storage devices. This approach necessitates some form of garbage collection, to recover the free space from the log. Currently NILFS2 uses a very simple garbage collection algorithm called *Timestamp*. This thesis describes the implementation of two additional state-of-the-art algorithms (*Greedy* and *Cost-Benefit*), and compares their performance. Additionally it discusses and compares two algorithms to gather and store the necessary meta-data used by the above mentioned garbage collection algorithms. The very promising results show a performance improvement over the Timestamp algorithm by a factor of 1.53 for most scenarios.

Contents

CHAPTER 1

Introduction

NILFS2 is a pure log-structured file system for the Linux kernel. It is heavily influenced by the original LFS developed by Rosenblum et al. [RO92] for the Sprite operating system and it offers unique and interesting features like "continuous snapshotting" [KAS$^+$06, WS11].

1.1 Motivation

Although the unique features of NILFS2 could make it a compelling choice for users, it cannot keep up with modern file systems in terms of performance [WS11, LSHC15], which is unacceptable for most users. Because of that it hasn't been very popular and the development has largely stopped.

One of the reasons for the bad performance is the inefficient garbage collector implementation. The focus during the development of NILFS2 was on the snapshotting features, and the garbage collector was added on almost as an afterthought [KAS$^+$06]. Also, continuous snapshots make the implementation of an efficient garbage collector difficult, because a block can be active in any number of snapshots and there is no easy way to determine if it is reclaimable or not [CE00]. As a result the NILFS2 garbage collector was forced to use a very simple heuristic that always selects the oldest segment for cleaning irrespective of the number of reclaimable blocks it contains. At the time, better selection policies were already well-established by other log-structured file systems, but the early design decisions of NILFS2 prevented a simple adoption.

Solving this major problem for NILFS2 could revive user interest in the file system, encourage more development, and stimulate more research into log-structured file systems in general.

1.2 Problem Statement

The crucial part in the garbage collection process is the selection of segments for cleaning. Among others, there are two well established garbage collection policies for log-structured file systems:

- Greedy: [RO92]
  Select the segments with the most free space.
- Cost-Benefit: [RO92] [KNM95]
  Perform a cost-benefit analysis, whereby the free space gained is weighed against the cost of collecting the segment.

NILFS2 currently supports only the so called *Timestamp* policy, which provides relatively poor performance and is basically a very weak form of *Cost-Benefit*. This is presumably, because the *Greedy* and Cost-Benefit policies require accurate information about how much free space is available in each segment, and NILFS2 currently does not keep track of that. So the first step in implementing new selection policies is implementing accurate free space tracking.

Additionally, NILFS2 supports a feature called “continuous snapshotting”, which automatically creates mountable checkpoints. The user can turn these volatile checkpoints into permanent snapshots at any time. The file system has to keep accurate statistics on the amount of reclaimable free space that is available in every segment. The snapshots complicate this task, because a snapshot can protect an otherwise reclaimable block, which invalidates previously gathered statistics. The number of blocks affected by a single snapshot can be huge. Consequently, an efficient algorithm to compensate for this problem is needed.

1.3 Goal of the Work

Ultimately, the goal is to implement both the Greedy and Cost-Benefit selection policies and all the tracking framework and snapshot compensation algorithms needed to get the necessary statistics about the location of the reclaimable free space in the file system.

If the statistics are not accurate, then the selection policies cannot make good selections, which harms the garbage collector’s efficiency and ultimately the performance of the whole file system.

A secondary goal is to reproduce the results of Rosenblum et al. [RO92], concerning the better bimodal distribution of the Cost-Benefit over the Greedy policy. Rosenblum et al. used a file system simulator to develop the Cost-Benefit policy and these results have to my knowledge never been reproduced by an independent researcher on an actual file system.

Another goal of this work is to develop a realistic, unbiased benchmark to test new selection policies in particular and the efficiency of the garbage collector in general.

Apart from that, the code quality should comply with Linux kernel standards, so that an inclusion into the mainline kernel is possible. The objective is not to demonstrate that the Greedy and Cost-Benefit policies perform better than the existing policy, since that fact is already well established, but to create a high quality, stable implementation that is ready for production use.

1.4 Methodological Approach

I started with an experimental approach, by developing small scale proof-of-concept implementations with the goal of identifying all the pitfalls. Then I set out to develop a realistic benchmark to properly evaluate these proof-of-concept implementations. Finally, I designed and implemented a production grade, stable, and reliable solution.

1.5 Structure of this Work

Chapter 2 gives an overview over the subject and introduces all the necessary terminology, that is used in later chapters. Chapter 3 discusses the literature on the topic. In chapter 4 I explain the design decisions for the implementation, and in chapter 5 I go into implementation details. Chapter 6 introduces the benchmark tools used, and chapter 7 shows the benchmark results. Finally, I present my conclusions in chapter 8.

CHAPTER 2

Background

This chapter tries to give the reader a broad overview of the background information necessary to understand later chapters. It introduces important terms and concepts, which are used frequently later on.

2.1 Terminology

This section introduces some of the common terms used for file systems in general and Linux file systems in particular. Later chapters rely heavily on the terms introduced here.

2.1.1 Kernel-Space, User-Space and Context-Switches

The kernel is the core of the operating system. It runs in a privileged mode and has full access to the hardware. It manages hardware resources for unprivileged programs running in user-space. Any piece of code that runs in the kernel in privileged mode is referred to as running in kernel-space.

A user-space process can call system calls to ask the kernel to perform certain privileged actions. Whenever the execution thread of a process switches between user-space and kernel-space via a system call it is called a context-switch. Context-switches are expensive operations and should be avoided as much as possible in performance sensitive applications.

2.1.2 On-Disk and In-Memory

Usually, the data structures written to persistent storage, such as a hard disk, are different from those held in main memory, when the file system is running. To distinguish between the two, the terms “on-disk” and “in-memory” are commonly used. However, the term “on-disk” does not imply any particular hardware like a hard disk.

2.1.3 Data and Meta-Data

Meta-data is data about data. In the context of a file system the data is the actual content of the files in the file system and the meta-data is everything else. The file names, directory structure, access mode, owner id, and internal data structures to support the file system would be examples for meta-data.

2.1.4 Page

The *page* is the granularity of memory used by the memory management unit (MMU) to map virtual memory addresses to physical addresses. The MMU is a hardware unit in the processor that uses the page tables, which are set up by the operating system, to perform this mapping. The page size depends on the capabilities of the MMU hardware. On current consumer hardware a page size of 4096 bytes is very common.

The Linux kernel uses one `struct page` to keep track of every memory page on the system. This structure contains information about the status of the page, where it is mapped and in which subsystem it is used. These subsystems include user-space processes, dynamically allocated kernel data, static kernel code, the page cache, etc. [Lov10, chapter 14, p. 290].

2.2 Block Device

Block devices are part of the block I/O layer which is an abstraction layer used by the kernel to represent random access devices. A *block device* is a storage device that allows random access to its content using fixed-size chunks of data [Lov10, chapter 14, p. 289]. These chunks of data are called *blocks* or *sectors*. In this thesis the term *sector* is used to avoid confusion with file system blocks. A sector is the basic unit of operation on a block device. Any read or write operation performed on a block device must be in the form of one or more sectors. The size of a sector must be a power of two and is hardware dependent. Until recently, the most common sector size was 512 bytes [Lov10, chapter 14, p. 290]. Modern devices tend to use bigger physical sector sizes like 4096 bytes. Some common examples for block devices include hard disks, SSDs, SD-Cards, and CDs.

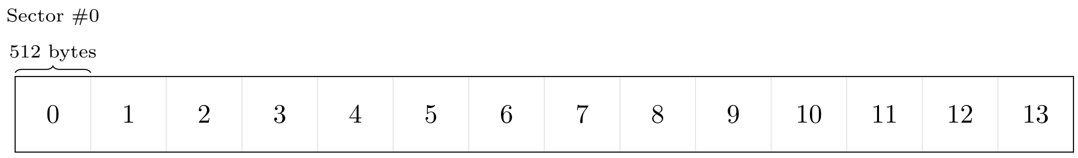


Figure 2.1: A block device with 14 sectors and $14 \times 512 = 7168$ bytes total

Figure 2.1 shows an illustration of a small block device. It can be thought of as an array of equally sized sectors which can be randomly accessed by their index. The sector size has to be a power of two and is 512 bytes in the above example. A block device can only read or write whole sectors. Any read operation smaller than that has to read a whole sector and discard the unused data. Any write operation smaller than a sector has to first read a whole sector, modify it and then write it back.

The index of a sector on the block device does not have to correspond to any particular physical location on the storage device. The firmware or driver of the device can change the mapping at any time. This can be useful for wear-leveling or to replace damaged blocks. However, these hardware details are transparent to the parts of the kernel that use block devices. For example a typical file system only sees a flat array of sectors and does not care where or how the sectors are stored.

2.2.1 Sector or Disk Block

A sector is a block on a block device which was discussed in the previous section 2.2. It usually has a size of 512 bytes. It is different from a file system block, so whenever there is a possibility to confuse the two the more specific term “sector” is used.

2.2.2 File System Block

The file system block is the basic unit for data as well as meta-data in a file system. Its size is determined by the file system and can be different from the sector size of the underlying block device. It must be a power of two, a multiple of the sector size and smaller than or equal to the page size [Lov10, chapter 14, p. 290].

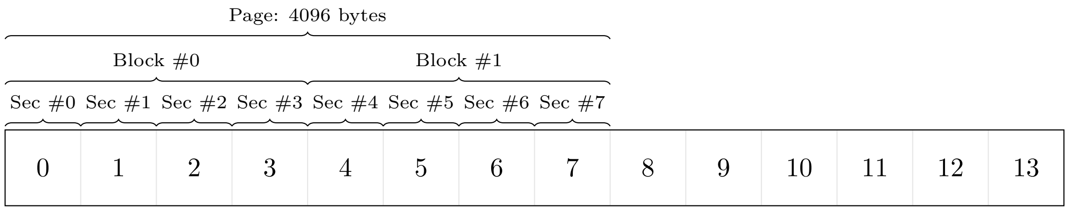


Figure 2.2: File system block mapped to a block device

The block size is a classic trade-off that depends on the kind of files stored on the file system. For example a file that is smaller than the block size, still uses at least one block. The unused extra space in the block is wasted. On the other hand if a file is very large and the block size is comparatively small, then there can be be a lot of overhead in the meta-data structures used to keep track of that file.

Most of this thesis is concerned with file system blocks, so whenever the term “block” is used, it refers to a file system block.

2.2.3 Page Cache

The *page cache* is a memory cache used by the Linux kernel to cache file system blocks. It only manages whole pages and not any smaller units like blocks or sectors. In normal operation most file systems use the page cache to read or write blocks to disk, but there is also a direct I/O mode, which circumvents it. After a block is read in from disk, it stays in the cache, until the page is needed for something else. Blocks written to the cache are not immediately written out to disk. Instead they are written back at a later time by the kernel's writeback threads.

2.3 Virtual File System

The virtual file system is an abstraction layer implemented by the Linux kernel. It sits between the various kinds of file system implementations and user-space. On the one hand, it provides an interface that is implemented by the file systems in the kernel, and on the other hand it provides a system call interface for user-space programs. Furthermore, it allows for multiple different file systems to be linked together in a single global directory hierarchy called a namespace.

All of this complexity is hidden from user-space programs. A user-space program uses VFS system calls to access the global namespace and the VFS forwards the call to the corresponding file system.

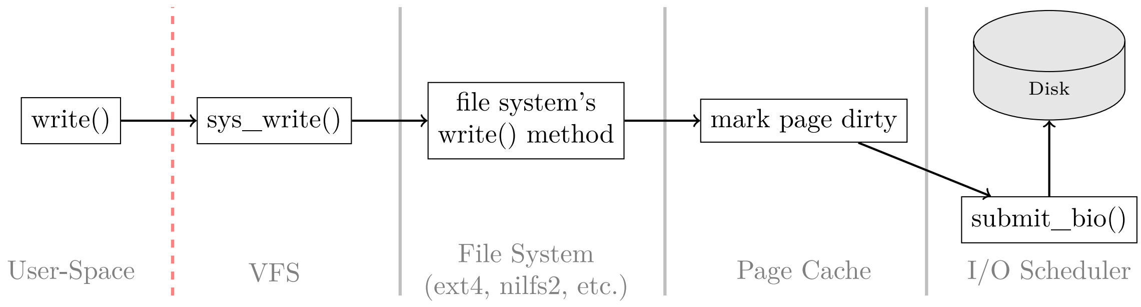


Figure 2.3: The flow of data from user-space to disk via the VFS

2.3.1 Super Block

The super block contains static information about a file system, like the block size or the addresses of meta-data blocks. Usually it is located at a known position on the underlying block device. It is the starting point for the mount operation of a file system, because the information it contains is necessary to bootstrap the file system. If the super block gets destroyed without a backup, then the whole file system is most likely lost as well.

2.3.2 Inode

An *inode* holds the meta-data for a file or directory and it points to the data blocks containing the content of the file. It is uniquely identified by the inode number. The meta-data includes such fields as owner id, group id, access mode, and more, but *not* the actual file name. Instead, the file name is stored in the data blocks referenced by one or more directory inodes. So a directory can be thought of as a special kind of file that maps file and subdirectory names to inode numbers.

2.3.3 Dentry

The dentry is an in-memory data structure used by the Linux kernel to map a file name to an inode. Additionally, multiple dentries can be connected into a tree structure to represent part of the file system directory tree in memory. Dentries are created on the fly by the VFS and cached in the so-called dcache to speed up lookups of frequently accessed files.

2.4 Sequential Writes and Storage Devices

On modern storage devices sequential writes can be up to two orders of magnitude faster than random writes. Whereas the number of one order of magnitude is often cited in the literature [OD89, MLE14, QR13] and it remains more or less true for SSDs, it is no longer correct for modern hard drives. While the sequential write speed of hard drives has improved over the years, the seek time has largely remained in the millisecond range due to physical limitations. As a result, sequential write operations can be 100 to 200 times faster than random write operations on hard disks (see figure 2.4).

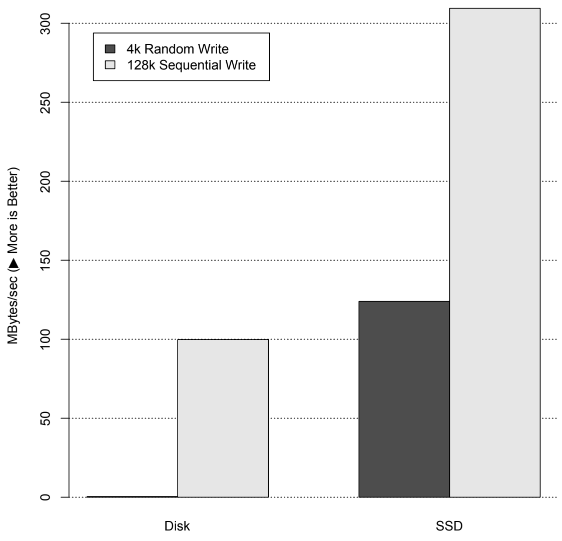


Figure 2.4: Sequential vs Random Write

This section explores some of the underlying hardware details that are the cause for these stark differences in performance. However, this section is not intended to be exhaustive. Hard disks and SSDs are hugely complicated devices that use different technologies and are subject to constant development and improvement.

2.4.1 Hard Disk

A hard disk basically consists of multiple disks, coated with a magnetic material, and a reading/writing head mounted on a movable arm. The disks, called platters, rotate rapidly at a constant speed and the head slides over them. The head uses magnetic fields to store information on the platters. A platter has multiple concentric tracks. Every track contains multiple sectors and every sector contains a chunk of data.

The head can only move from one track to the next. If it needs to read or write one specific sector it has to move to the correct track and then wait until the rotation of the platters moves the target sector underneath the head. The movement of the head and the rotational delay lead to a worst case seek time that is typically in the range of milliseconds. This physical limitation significantly slows down random write operations. Sequential writes on the other hand can fill up a whole track without any rotational delay or head movement.

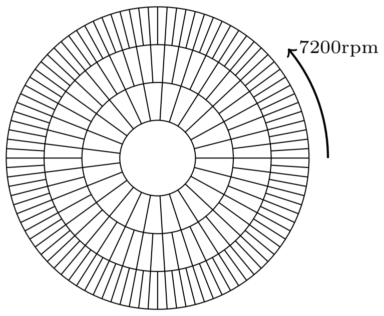


Figure 2.5: Tracks with sectors on a hard disk platter (The sector density increases on the outer tracks)

Some hard disks use a technique called *Shingled magnetic recording*, whereby newly written tracks partially overlap with previously written tracks like shingles on a roof. As a result, the tracks are narrower, the track density can be increased, and a higher disk capacity can be achieved. The reason for the overlapping tracks is that the reading head can be made smaller than the recording head. The downside of this approach is that it can slow down random writes. For example, if multiple tracks are written sequentially, then the overlapping write operations happen in the correct order and no rewrites are necessary. If, however, tracks are written randomly, then the overlapping write operation destroys previously written data, which has to be read in before the random write and then rewritten afterwards.

2.4.2 Raw NAND Flash

NAND flash is a type of flash memory that is commonly used in consumer SD cards and SSDs. It uses floating-gate transistors to store information electronically [JBLF10]. Much like the sectors on a hard disk, NAND flash can only be read and written in larger chunks. These chunks are called pages (not to be confused with MMU pages 2.1.4) and typically range from 512 to 16384 bytes in size [JBLF10]. Initially all bits in a page are set to 1, which means it is completely erased. A write operation can toggle binary 1s to 0s but not the other way around. So a page can effectively only be written once, before it has to be erased again. However, it is not possible to erase individual pages. Instead pages must be erased in bigger chunks called erase blocks [LhP06, JBLF10]. The erase block size is hardware dependent and usually very large compared to the page size (e.g. 64 - 128 pages [JBLF10, MKC$^+$12]).

The large erase block size is the reason for the slower random write performance of flash memory. If a whole erase block is written sequentially, only one erase operation is required. On the other hand, if single pages are written at random locations, multiple time-consuming erases are necessary. Additionally, the life time of NAND flash memory is limited by the number of program/erase cycles [JBLF10]. So random writes can significantly reduce the lifetime of NAND flash memory.

2.4.3 Solid State Drive (SSD)

Solid state drives have no moving parts and use integrated circuits to store information. They are compatible to hard disks and implement the same block device I/O interface. At the time of writing, most consumer SSDs use NAND flash internally and use a *flash translation layer* (FTL) to dynamically map *logical block addresses* (LBA) to *physical page addresses* on the underlying flash media [LhP06]. This allows them to significantly improve random write performance, do wear-leveling, and manage bad blocks. Additionally, the FTL has to perform garbage collection (GC) to recycle invalidated physical pages [LhP06, MKC$^+$12]. The overhead and resource requirements of the GC process heavily depends on the mapping algorithm used by the FTL. Moreover, SSDs are able to write in parallel to multiple NAND flash chips, which further complicates the FTL, its internal GC implementation, and the resulting performance characteristics. For example, Min et al [MKC$^+$12], observe that certain random write patterns can cause internal fragmentation in the FTL data-structures, which significantly reduces the performance even after the random writes have stopped. Apparently the internal GC needs time to clean up the fragmentation before the full write bandwidth can be restored.

The FTL hides a lot of the complexity of raw NAND flash, but certain patterns of random writes, still cause more GC overhead, a greater number of program/erase cycles, and as a result reduced performance and lifetime for the SSD.

2.4.4 eMMC, SD cards, and USB Sticks

eMMC, SD cards and USB sticks are similar to SSDs, but they generally use a less sophisticated FTL. Therefore the characteristics of the underlying NAND flash tend to be more visible to the user.

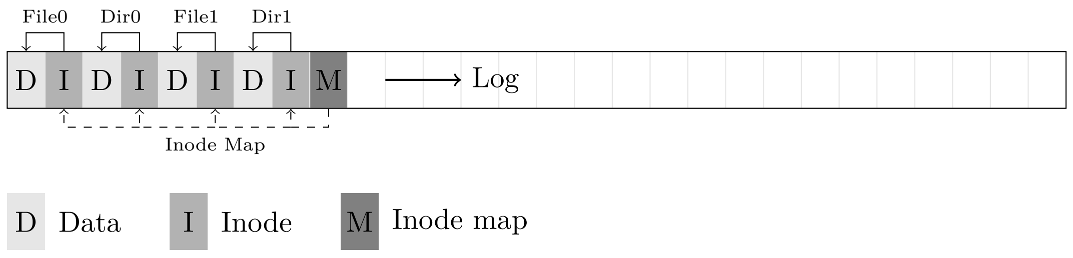


Figure 2.6: Log-structured File System: The inode map can be used to lookup the inodes, which in turn point to data blocks

2.5 Log-structured File Systems

Log-structured file systems represent their content as an append-only log of file system changes. This log is the primary on-disk data structure and contains a complete history of all write operations and meta-data changes. It can only grow sequentially in one direction. New entries are appended at the end, and existing entries are never changed or overwritten.

Ousterhout et al. [OD89] first discussed the advantages of a log-structured design, and Rosenblum et al. [RO91] first implemented a log-structured file system called LFS for the Sprite operating system. Most of their reasoning behind their design decisions is still valid to this day and many recent implementations of log-structured file systems use the same terminology and similar data-structures. This section is an introduction to LFS, since it is both representative of the file system category and the first of its kind.

2.5.1 Motivation

Rosenblum et al. [RO91] observe that both CPU and memory speeds have increased exponentially over time, while hard disks are limited by mechanical constraints (see section 2.4.1). Since most read operations can be served from memory caches, they conclude that most applications will be limited by the write bandwidth of the hard disk [OD89, RO91]. A log-structured file system caches write operations in memory, reorders them and asynchronously writes them as a sequential burst of write requests. Since hard disks are an order of magnitude faster for sequential writes (see section 2.4.1), and the write speed is the major bottleneck for applications, the performance should improve significantly over conventional file systems.

Min et al. [MKC$^+$12] apply the same logic to solid state drives (SSD), but for different reasons. SSDs and other NAND flash based storage performs an order of magnitude better with sequential writes (see section 2.4.3), because of the characteristics of NAND flash memory and the complex algorithms used in the flash translation layer (FTL). Therefore Min et al. [MKC$^+$12] create a log-structured file system called SFS that is optimized for SSDs. Furthermore, the overhead in the FTL is reduced if the file system operations are well aligned with the FTL mapping from LBA to PBA [LSHC15].

2.5.2 Indexing and Meta-Data Location

Conventional file systems (e.g. ext4 [MCB+07]) use a fixed on-disk location for meta-data. For example, the inode table is stored at a fixed location determined at file system creation time. This allows the file system to instantly lookup inode blocks, because the inode number (see section 2.3.2) can be used directly as an index into the on-disk inode table. Although the lookup is very efficient, it requires random writes to update the meta-data, which defeats the purpose of a log-structured file system.

In LFS most meta-data is also written to the log, so the location of the blocks constantly changes with every meta-data update. For example, figure 2.6 shows one block of the inode map as part of the log. The inode map contains a map from inode numbers to block addresses, as is indicated by the dashed arrows in figure 2.6. The addresses of the inode map blocks are stored at a fixed location on disk called the checkpoint region (see section 2.5.7). So the inode lookup process has to perform the following steps:

- Lookup the inode map block in the checkpoint region, which resides at a known location
- Lookup the inode number in the inode map to get the corresponding inode block
- Read the inode from the inode block

The checkpoint region is relatively small and the extra level of indirection reduces the number of random writes that are necessary for an update.

Example: `/dir0/file0`

This section demonstrates how LFS uses its data structures to find the first block of the file `/dir0/file0`. Directories are represented by inodes and can be thought of as special files that contain a map from human readable names to inode numbers (see section 2.3.2). To get the block of an inode, the inode map is used as shown above. The process is divided into the following steps:

1. Lookup the inode for the root directory `/`, which has a known inode number, in the inode map to get its block address
2. Use the block address to read the inode
3. Use the index data structure of the inode to get its data block addresses
4. Get the data blocks of the root inode
5. Lookup the inode number of `dir0` in the directory map contained in the data block

6. Repeat steps 1 through 4 for the inode number of `dir0`
7. Lookup the inode number of `file0` in the directory map contained in the data block
8. Repeat steps 1 through 3 for the inode number of `file0`
9. Read the first data block of `file0`

2.5.3 Free Space Management

The management of free space is *the* crucial part of any log-structured file system. Since new data can only be appended at the end of the log, it will eventually reach the end of the disk, where it has to wrap around and find new free space. However, it cannot simply overwrite the oldest part of the log, because some of those blocks may be still in use by the file system. The free space becomes fragmented, which means that there are small patches of free blocks interspersed with occupied blocks. There are basically two approaches to dealing with this problem: Threading and Copying [RO91].

Threading means that the new data is written to the fragmented free blocks. It is threaded in-between the live blocks. If the new data is threaded at the level of individual blocks, then write requests would no longer be sequential, which defeats the purpose of a log-structured file system.

Copying means that a garbage collection process copies the live blocks to the top of the log to create a larger sequential area of free space for the log to grow into. Copying adds an extra cost, which reduces the overall performance of the file system.

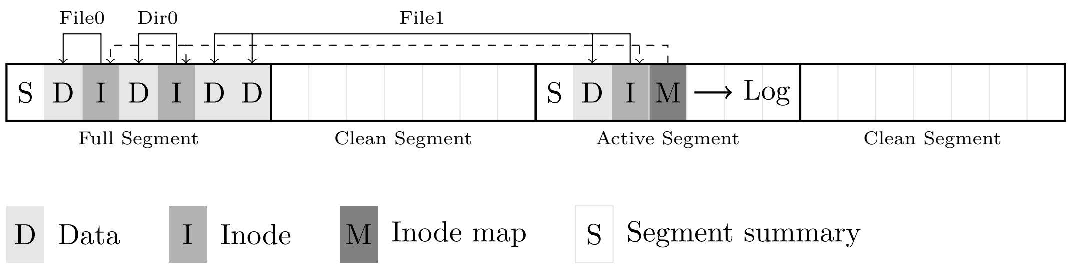


Figure 2.7: Log-structured File System with Segments

LFS uses both methods at different levels to get the best trade-off. It divides the disk into large chunks, called segments. The segments are always written sequentially from beginning to end and they are large enough to take full advantage of the sequential write bandwidth of the disk. However, the order in which the segments are selected for writing is *not* sequential.

Furthermore, a garbage collection process copies live blocks to the end of the log. It defragments the free space and creates more clean segments for the log to grow into.

So LFS uses threading at the segment level and copying at the block level. The file system F2FS [LSHC15] also uses threading at the block level when the disk utilization is high and the cost of copying is greater than the cost of doing slow random writes.

Segment Summary Block

Every segment starts with a segment summary block. It contains the following information for every block in the segment:

- Inode number the block belongs to
- The offset of the block in the file

Using these two numbers the garbage collector can determine if a block needs to be copied or not. It can look up the inode from the log and get the address of the block at the given offset. If that address points to a location in the current segment, then this block is in use by the file system and has to be copied. If, however, the address points to somewhere else, then the block was moved to another location and the old copy in the current segment can be regarded as free space.

Partial Segments

A segment can contain multiple segment summary blocks, because of partial segments. If there is not enough data to fill a whole segment, a partial segment is created.

2.5.4 Garbage Collection

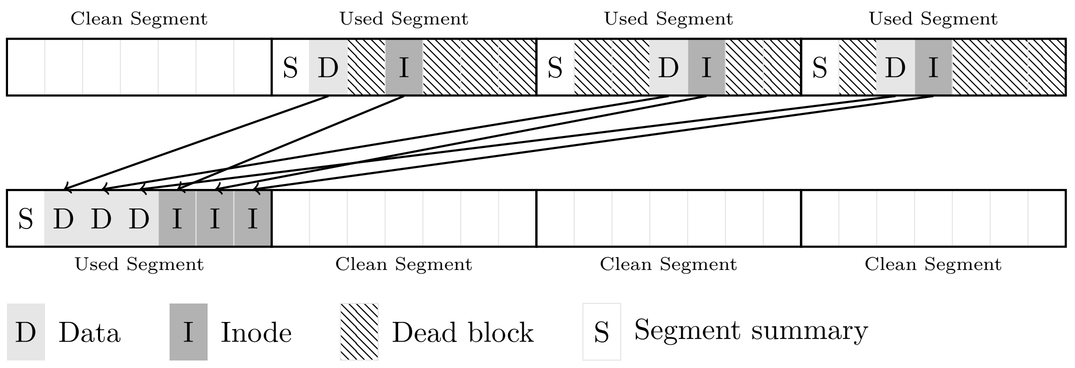


Figure 2.8: Garbage Collection Process

The garbage collector (GC) is an essential part of any log-structured file system. Without it, the file system would quickly run out of free space, because it cannot overwrite the fragmented free blocks locked up inside of old segments. Figure 2.8 shows the basic two steps of the garbage collection process. First it selects multiple used segments with lots of dead blocks, then it copies the live blocks elsewhere and marks the segments as clean again. Since the dead blocks are no longer needed, they are simply discarded. The free space previously occupied by dead blocks is now reusable by the file system.

Dead Block A block that is no longer referenced by any inode in the file system. This happens when a file is truncated, deleted or partially overwritten. Dead blocks contain no useful data, but they lock up free space that is unavailable to the file system.

Live Block A block that is referenced by an inode. The garbage collector has to copy it and update the reference in the inode to the new location.

Clean Segment A segment with no file system data. The free space in clean segments is available to the file system.

2.5.5 Selection Policies

The crucial part in the garbage collection process is the selection of segments for cleaning. The file system performs best, when most of the disk bandwidth is available for writing new data as opposed to copying existing blocks. Since the GC has to copy live blocks from a segment, it is important to select segments for cleaning where the cost of doing so is minimal. One approach is to select segments with the smallest segment utilization u, where $0 \leq u \leq 1$. As it turns out this is not the best policy [RO91].

The reason for this non-intuitive finding is that blocks have different so called *hotness*:

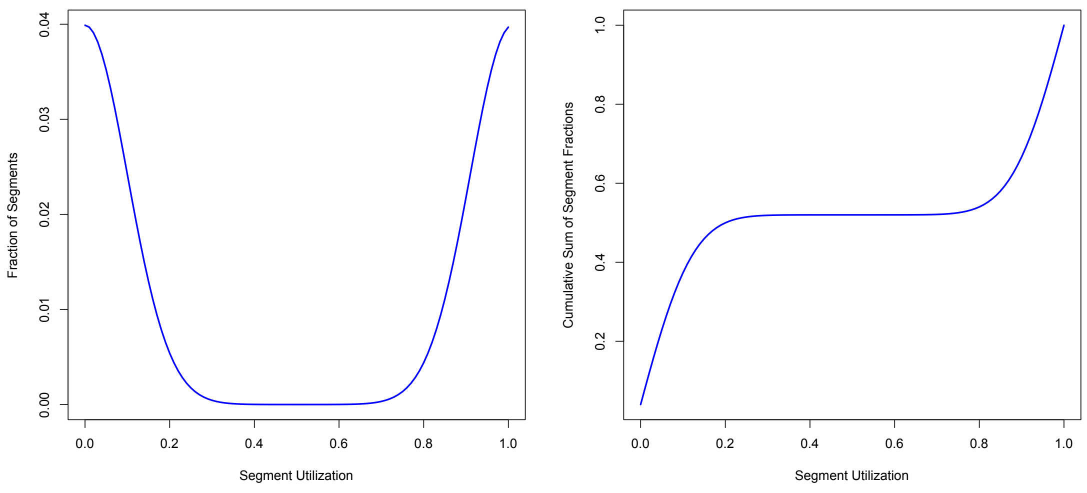
Hot blocks are part of files that are overwritten frequently or have a high likelihood of being deleted soon after creation. For example, temporary files or documents that are frequently edited contain hot blocks. Also file system internal meta-data blocks that change frequently could be considered hot.

Cold blocks are part of files that have stabilized over a longer period of time. These files are mostly read and rarely written. For example, read only documents, media files, or program executables and libraries are made up of cold blocks.

Since hot blocks have a high probability of being invalidated in the near future, copying them is a waste of disk bandwidth. Most of them will become dead blocks soon after the copy, which forces the GC to clean the segment again. It is better to wait longer, avoid unnecessary copies, and let the hot blocks die.

On the other hand cold blocks are very unlikely to die. So after cold blocks have been copied to a segment, it is likely to remain stable with a high segment utilization for a long time. As a result the GC does not have to treat that segment again for a long time. Copying cold blocks pays for itself.

The worst case scenario is when a segment contains exactly 50% hot and 50% cold blocks. The hot blocks die quickly and tie up a lot of free space, which forces the GC to clean the segment again and needlessly copy the cold blocks.



(a) Standard Deviation of 0.1 (b) Cumulative Sum

Figure 2.9: Bimodal Distribution Example

The best case scenario is when most segments are either mostly empty or completely full. So the segment utilization u should be either $u \approx 0$ or $u \approx 1$. This is called a bimodal distribution of the segment utilization. If a policy is successful it should produce a bimodal distribution.

The following selection policies have been used in various log-structured file systems:

Timestamp [**KAS+06**] The oldest segments are selected. This policy is only used by NILFS2.

Greedy [**RO91**] The segments with the lowest segment utilization are selected. This policy is not optimal, because of the different hotness of the blocks.

Cost-Benefit [**RO91, SBMS93, KNM95, JHT+07, LSHC15**] Perform a cost-benefit analysis, whereby the free space gained is weighed against the cost of collecting the segment. The age of the segment is used as an estimate for its hotness. This policy is used in LFS and the more modern file system F2FS.

Cost-Hotness [**MKC+12**] Cost-Hotness is related to Cost-Benefit. The difference is that the hotness is directly measured instead of estimated by the age of the segment.

2.5.6 Cost-Benefit

This section describes the Cost-Benefit policy in detail, since it is the main focus of this thesis. The Cost-Benefit policy selects the segments with the highest ratio of benefit to cost. The benefit is composed of the amount of free space in the segment and the hotness of the data. The hotness is estimated by the age of the segment. In short: The older the segment, the colder the data, the greater the benefit from copying it. The free space generated can be calculated by $(1 - u)$, where u is the segment utilization. The costs are conservatively estimated to be $1 + u$. The segment has to be read in as a whole, which costs 1, and the live blocks u have to be written out again.

$$\frac{benefit}{cost} = \frac{free\ space\ generated \times age\ of\ data}{cost} = \frac{(1-u) \times age}{1+u}$$

Segment Usage Table

The segment usage table is a data structure used by LFS to provide the statistical information needed by the Cost-Benefit policy. It keeps track of the number of live blocks and the last modification time for every segment. The blocks containing these data are written to the log and can be found through the checkpoint region. The lookup process works exactly like the lookup of inodes in the inode map (see section 2.5.2).

2.5.7 Crash Recovery

After a system crash, the file system is potentially in an inconsistent state. To recover from this state, the last on-disk changes have to be scanned and corrected. In case of a log-structured file system the location of the last write operations are always at the end of the log. Therefore only a few blocks at the end of the log have to be scanned, which speeds up the recovery time. LFS uses two techniques to deal with crashes: *checkpoints* and *roll-forward*.

Checkpoints

A checkpoint represents a consistent file system state. It marks a position in the log where all meta-data has been written to disk and it can be used to recover the file system at that point. Checkpoints are stored in a checkpoint region, which is a fixed known location on disk that is reserved for the checkpoint data. LFS has two checkpoint regions, which are never modified at the same time. If one checkpoint region gets corrupted during a crash, the other can take over.

The checkpoint region contains the addresses to the inode map and the segment usage table, a timestamp and a pointer to a location in the log where the file system is in a consistent state. This information is enough to bootstrap the inode map and mount the file system.

Roll-Forward

LFS uses a technique called *roll-forward* to recover as much data as possible. It scans the log from the latest checkpoint on-wards, and reapplies the operations from to log to the recovered file system.

2.6 Introduction to NILFS2

NILFS2 [KAS$^+$06, WS11] is a pure log-structured file system for Linux, and it has been part of the mainline kernel since version 2.6.30. In contrast to other log-structured file systems, like the original LFS [RO91] or F2FS [LSHC15], it does not use any update-in-place, on-disk data structures. Only the super block is frequently updated to point to the end of the log, everything else is written directly to the log.

2.6.1 Crash Recovery

One of the design goals of NILFS2 is to prevent data loss in the event of a system crash. Much like LFS (see section 2.5.7) it uses the checkpoints as the starting point for the recovery process and then performs a roll-forward to recover as much of the partially written segments as possible. The log-structured design ensures that blocks are never overwritten and all data is written out sequentially. As a result, the roll-forward algorithm is relatively simple and data loss is minimized.

The only data structure that is not protected by checkpoints is the super block, because it resides at a fixed location and is not part of the log. The super block is needed to bootstrap the file system and start the recovery process. Since NILFS2 has to update the super block fairly frequently, it is very important to make sure that it doesn't get corrupted. To that end NILFS2 stores a second copy of the super block at the end of the block device. If one copy of the super block gets corrupted, NILFS2 restores it automatically from the backup copy.

2.6.2 Snapshots

The main selling point of NILFS2 is a feature called "continuous snapshotting" [WS11]. While LFS (see section 2.5.7) is limited to two checkpoints, NILFS2 supports an arbitrary number of checkpoints. NILFS2 creates a checkpoint automatically every 30 seconds, before the file system is unmounted, and whenever some variant of the `fsync()` function is called. The garbage collector automatically deletes the oldest checkpoints, when disk space has to be reclaimed. So depending on the size of the disk, there is always a large number of automatically created checkpoints available.

At any time the user can turn any of these checkpoints into snapshots [WS11]. A snapshot is nothing more than a checkpoint marked as permanent, so that the garbage collector can no longer delete it automatically. Additionally, snapshots can be mounted as read-only file systems. This way it is possible to recover accidentally deleted files, create backups, and restore the file system to a previous state.

2.6.3 Segments and Disk Layout

NILFS2 uses the term “segment” for three slightly different concepts. Figure 2.10 illustrates the three usages and their relation to one another.

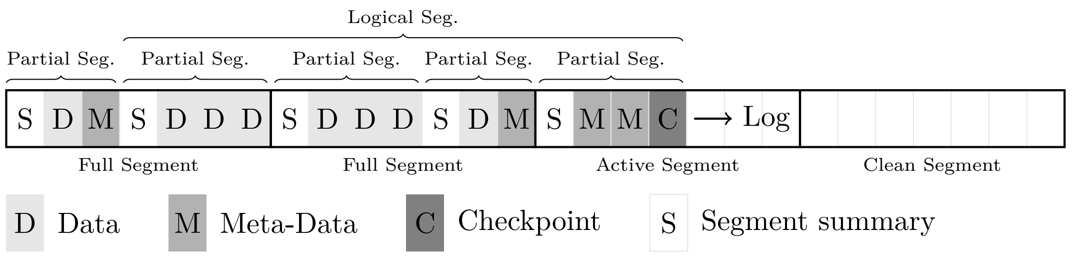


Figure 2.10: NILFS2 disk layout

Disk Segment NILFS2 splits the disk into equally sized segments that are large enough to take advantage of the higher sequential write speed of the disk. These segments are always written sequentially from beginning to end. The concept is identical to the segments used in LFS (see section 2.5.3). The garbage collector works on the level of whole segments. Every segment has a segment number, which can be used as an index to calculate the starting address of the segment.

Partial Segment The partial segment is the basic unit of writing in NILFS2. Any data that has to be written out is packaged in the form of a partial segment. It contains a segment summary block and payload data. The segment summary contains among other things the inode number of every payload block. This information is used by the garbage collector to determine which blocks are live and which blocks are dead. A disk segment can contain multiple partial segments, but they must not span across its boundary.

Logical Segment A logical segment represents a consistent file system state and it can span across multiple disk and partial segments. All blocks necessary to recover the file system are written out within the logical segment. So all the meta-data files and their B-tree nodes are in a consistent state, and optionally a checkpoint is created at the end.

2.6.4 Logical Segment Construction

The creation of logical segments is a multi-stage process. The order in which the stages are executed is important for file system consistency. So it is not possible to proceed to the next stage, before the current stage has finished.

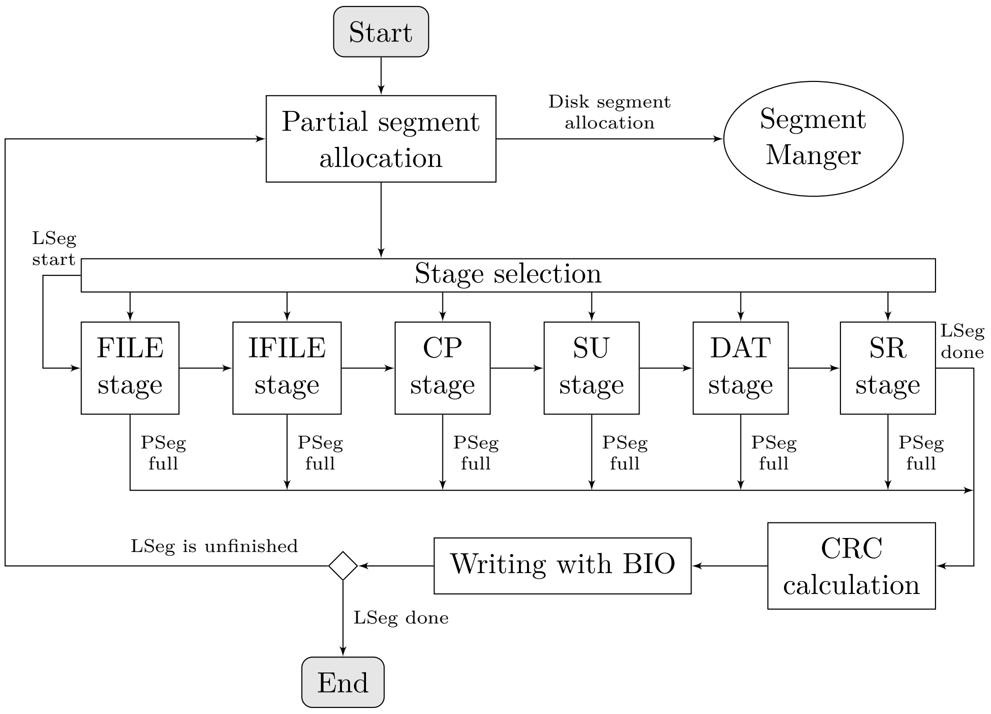


Figure 2.11: Segment Construction Flow Chart

Figure 2.11 shows the whole segment construction process with minor omissions. The segment construction begins with the allocation of a new segment buffer, which is the in-memory representation of a partial segment. The segment buffer is mapped to an on-disk location within the current disk segment. If there is no space left, a new disk segment is allocated for the buffer.

The current state of the process is maintained in a single data structure. Among other things it records the current stage, which allows the stage selection activity to continue the last unfinished stage. The stages lookup and collect dirty pages from the page cache and add them to the segment buffer. When the buffer is full, i.e. it has reached the end of the current disk segment, the stage is interrupted temporarily to write out the buffer. After that the segment construction is restarted and continues with the previously interrupted stage. The stages are named after the kind of file they treat. So first regular files are written out, then the different kinds of meta-data files (see section 2.6.6), and last the super root block.

The super root block contains the inodes of the meta-data files and is needed to mount the file system. However, not every logical segment has a super root. Only logical segments that contain a new checkpoint and therefore have to be mountable get a super root at the end.

2.6.5 Inode Implementation

An inode in NILFS2 is 128 bytes in size. Apart from the normal file attributes, it has space for up to 7 direct block addresses. For bigger files the space of the 7 addresses is reused as the root of a B-tree. The leaves of the B-tree contain the block addresses.

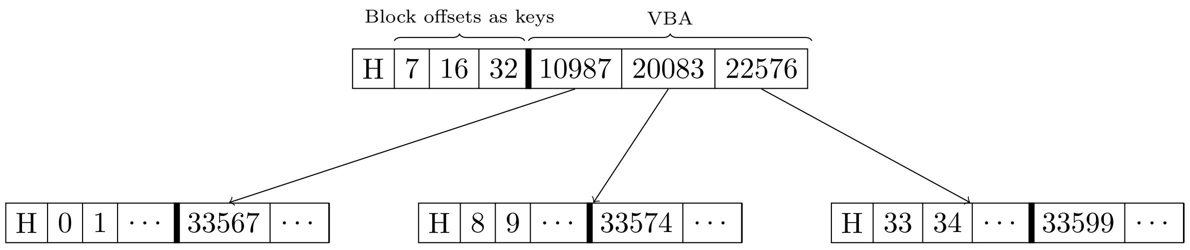

Block offsets as keys
VBA
H | 7 | 16 | 32 | 10987 | 20083 | 22576
H | 0 | 1 | ··· | 33567 | ···
H | 8 | 9 | ··· | 33574 | ···
H | 33 | 34 | ··· | 33599 | ···

Figure 2.12: NILFS2 B-Tree

Figure 2.12 shows an example of a B-tree as it is used in NILFS2. The root node is part of the inode and has only seven 64 bit entries available. Subsequent nodes are made up of a whole file system block, so the maximum number of entries depends on the block size. For example, a block size of 4096 bytes would allow for $\frac{block\ size - header}{key\ size - ptr\ size} = \frac{4096-16}{16} = 255$ entries. The first entry is occupied by the node header, which contains various flags, the number of entries, and the level in the tree. A node can be partially filled, which is indicated by the number of entries field in the header. The keys are block offsets into the file and the pointers are virtual block addresses (VBA) that need to be resolved through the DATFILE (see section 2.6.7). The leaf nodes are always on level 0 and they contain virtual block addresses to the actual data blocks.

2.6.6 Meta-Data Files

While LFS stores the addresses to the inode map and the segment usage table at a fixed location on disk, NILFS2 uses hidden files for all its meta-data:

IFILE The IFILE contains all inodes in the file system and it is itself represented by an inode. If any file in the file system changes, its inode changes, which forces the IFILE to change, which in turn changes the root of the B-tree in the IFILE inode. So the inode of the IFILE is enough to save the state of the whole file system at a particular point in time. This allows for very small and efficient checkpoints. Unlike the CPFILE, which is a flat array, the IFILE uses bitmaps to efficiently allocate inodes.

CPFILE The CPFILE is a sparse file that contains checkpoints. Every checkpoint has a unique number. This number is directly used as an index into the CPFILE. Since checkpoint numbers must not be reused, the CPFILE grows continually with the age of the file system. Every new checkpoint increases the size of the CPFILE.

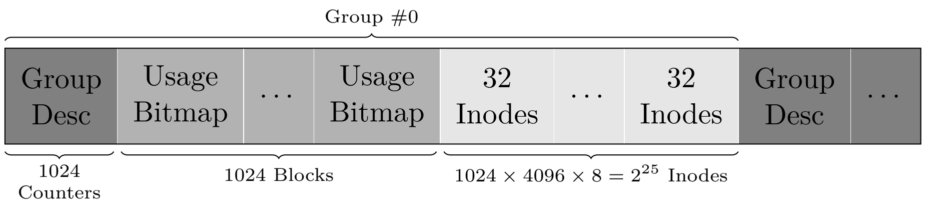


Figure 2.13: Content of the IFILE with 4K blocks

However, empty blocks are removed from the file, which creates a sparse file with a large hole at the beginning.

SUFILE The SUFILE is comparable to the segment usage table of LFS. It has an entry for every segment on the disk. The entries contain a timestamp and the number of blocks in the segment. The SUFILE, much like the CPFILE, is a flat array of SUFILE entries. The segment number is used as an index into that array.

DATFILE The DATFILE contains DAT entries. A DAT entry maps virtual block addresses (VBA) to device block addresses. Every file in the file system uses virtual addresses, which have to be resolved through the DATFILE. The DATFILE uses bitmaps to efficiently allocate and reuse DAT entries.

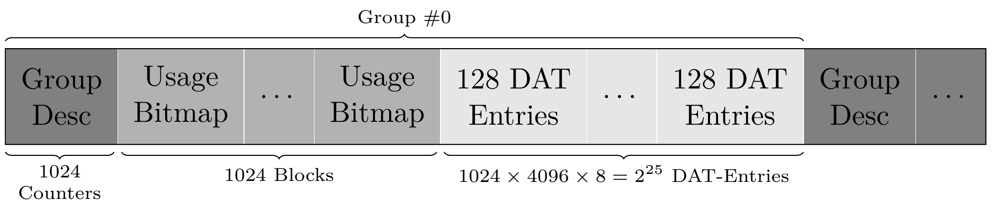


Figure 2.14: Content of the DATFILE with 4K blocks

Every checkpoint contains a different version of the IFILE, which represents the state of the file system at the time of the creation of the checkpoint. Only the IFILE is stored in the checkpoints. The other meta-data files do not need to be versioned, so they are stored in a separate block in the log called a “super root”.

2.6.7 Data Address Translation (DAT)

As noted in section 2.6.2, NILFS2 supports an arbitrary number of mountable snapshots and checkpoints, which are automatically created every few seconds. A block can be active in multiple or even all of the checkpoints. This creates a problem for the garbage collector, because it has to move live blocks from the victim segments to the end of the log. However, if the location of a block changes on the disk, all references to it have to be updated. This would be impractical in the case of NILFS2, because it would require a complete scan of the file system every time a block is moved.

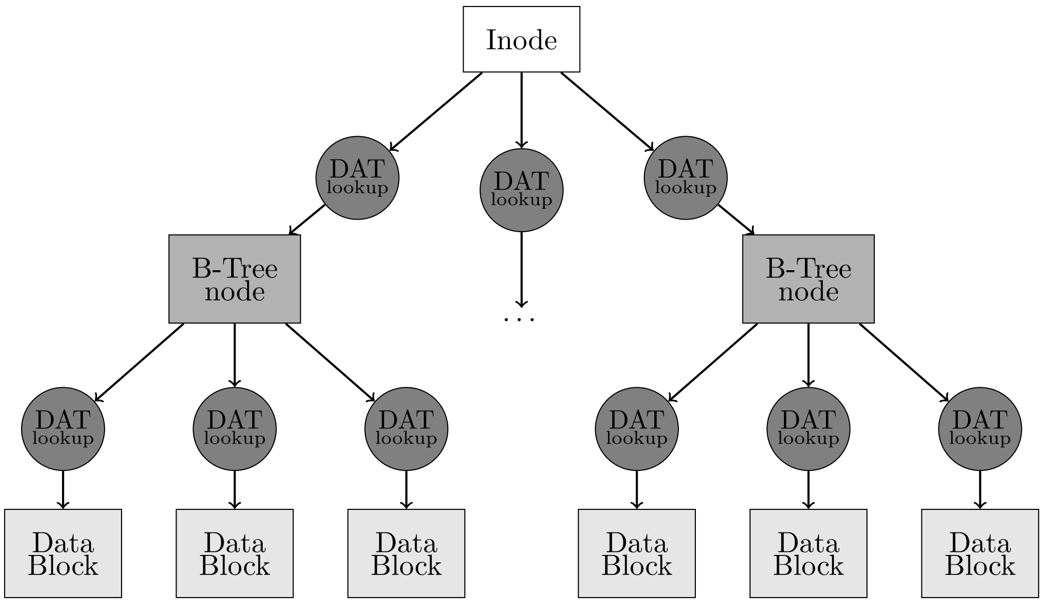


Figure 2.15: DAT lookups on every level

NILFS2 solves this problem with an extra level of indirection called the data address translation layer (DAT). Instead of referencing the disk blocks directly, the inodes and the B-tree store virtual block addresses (VBA), which point to an entry in the DATFILE. If a block needs to be moved, only the DAT entry is updated and the VBA stays the same. Since the rest of the file system knows nothing about the physical location of the block and references it only through the VBA, no further updates are necessary.

The DATFILE not only contains the mapping from VBA to physical block addresses, but also the checkpoint number the block was allocated in and the checkpoint number when it was deleted. The garbage collector uses this extra information to determine the liveliness of a block, without having to scan every checkpoint.

Figure 2.15 shows a simplified read operation of a file. Since every block reference is a VBA, DAT lookups occur on every level of the file system, which has a significant performance impact. The IFILE, where the inodes are stored, the B-tree nodes, and the actual data blocks all use virtual addresses, which have to be resolved through the DAT layer. Additionally, if a file is truncated, all the DAT entries of all its blocks have to be updated with the current checkpoint number, so that the garbage collector knows that these blocks are dead. In fact the only file that does not use the DAT layer is the DATFILE itself.

2.6.8 User-space Garbage Collection

NILFS2 uses a garbage collector called `cleanerd` that runs entirely in user-space. The behaviour of the `cleanerd` process can be tweaked through a configuration file, which is usually called `/etc/nilfs-cleanerd.conf` [WS11]. The user can adjust, among other things, the conditions under which the GC starts and stops its operation, as well as its aggressiveness.

Protection Period

Another important setting in the configuration file is the so called “protection period”, which protects a checkpoint from being deleted for a certain period of time after its creation. So all changes to the file system are guaranteed to be reversible for at least the time specified in the protection period. After that, the GC is allowed to delete checkpoints.

However, the user has to be aware of this setting and adjust it manually in certain situations. For example, deleted files occupy disk space until the protection period has passed and the GC had time to collect the corresponding segments. So if the user needs to quickly free up disk space, he or she has to manually override the protection period and force the GC to start. Most file system benchmarks write huge amounts of data, delete it, and assume that the disk space is immediately available again. This is not true for NILFS2, and it is therefore challenging to tweak the GC to work well with standard benchmarks.

Selection Policy

The selection policy can be set by the user in the configuration file. It uses the segment usage data as input, evaluates every segment based on a heuristic, and selects the next best segment for cleaning. The stock NILFS2 implementation only supports the Timestamp policy (see section 2.5.5).

Collection Process

The GC sleeps for most of the time. Periodically it wakes up and checks the amount of free space available. If it falls below a certain value set by the user in the configuration file, the GC starts the collection process, which can be divided into four steps:

1. The selection of victim segments for cleaning. The GC uses the `ioctl()` system call to retrieve all of the segment usage information stored in the SUFILE. The selection policy uses this information to evaluate every segment and select the most promising ones for cleaning.

2. Read the segment summary blocks of the victim segments. The GC reads the segment summary blocks directly from the block device in user space and thereby circumvents the kernel file system entirely. This is perfectly safe, because NILFS2 never accesses the segment summaries once they are written out. Every block in a segment is listed in its segment summary with additional information like the inode and checkpoint number it belongs to.

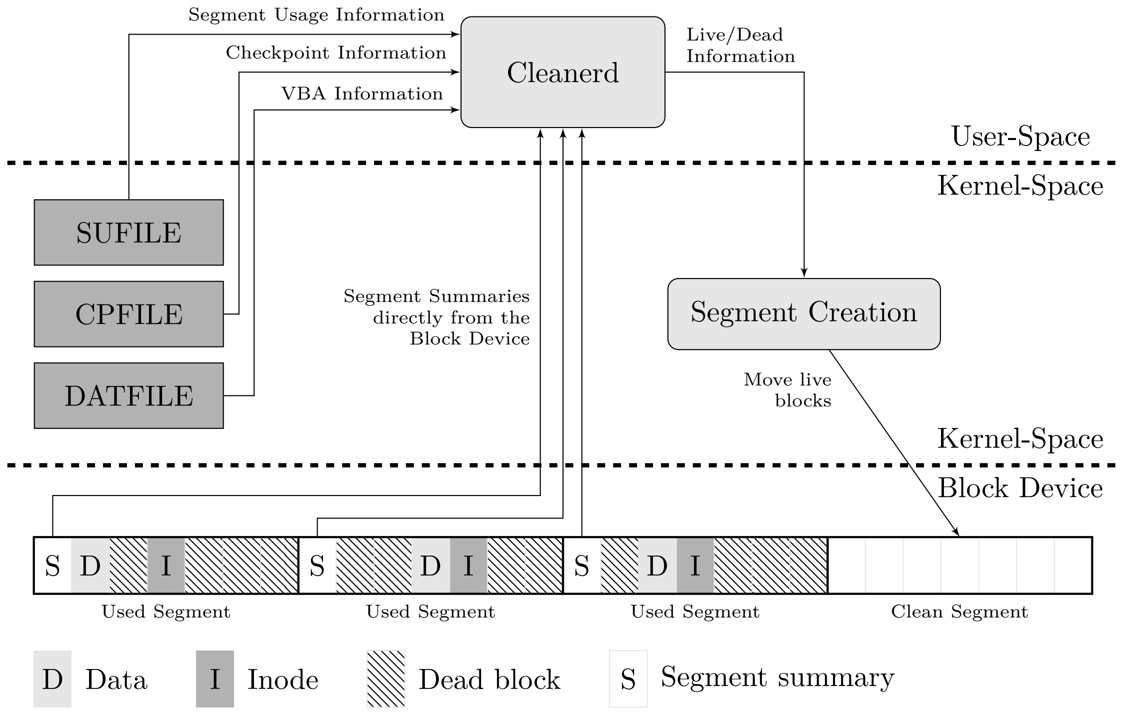


Figure 2.16: Collection Process

3. Determine if blocks are live or dead. The GC uses the `ioctl()` system call to get the DAT entries for all blocks in the segment. The DAT entry contains the checkpoint number of the time when the block was created and deleted. This information combined with a snapshot list is enough to determine the liveliness of the block.

4. Send the gathered information to the kernel. The rest of the work happens in the kernel. The live blocks are read in and marked as dirty, the dead blocks are discarded, and the victim segments are marked as clean at the end.

2.7 File System Aging

File system aging is a technique to artificially age a file system by simulating a long term workload on it [SS97]. The simulation can be accelerated to be faster than real time, so that the file system experiences weeks or months of usage in a few hours. After the aging process the file system is in a state similar to the real system the workload was recorded on.

A lot of file system performance problems do not appear on an empty file system, because it takes time for the fragmentation of files and free space to accumulate. Therefore benchmarks run on an empty file system can give misleading results. Aging can help to create a better starting point for benchmarks in a reproducible manner.

2.7.1 Replay of File System Traces

File system traces are recordings of file system operations from production systems over months or years. A recording contains a list of operations on the file system, but not the actual data that was read or written. As a result the recordings are small and highly compressible, compared to the full disk image. Additionally, the file names are anonymized, to protect the users of the file system. This allows for these traces to be shared online for research purposes. For example, the IOTTA Repository [Rep11], has a public collection of freely available I/O traces for file system research purposes.

These traces can be used to create realistic and repeatable benchmarks, by replaying the traces at an accelerated speed onto the file system under test. The effect is a simulated but realistic aging of the file system. Since the garbage collector for a log-structured file system relies on heuristics for its operation, it is especially important to test it with an authentic and unbiased benchmark. Synthetic benchmarks, on the other hand, could introduce a bias because of the heuristics and assumptions used in their design.

There are several different types of file system traces. They differ mostly in the level of detail and the method of recording:

NFS Traces Traces recorded by analyzing the network traffic from a NFS (Network File System) server. Although the traces are easy to record by capturing the network traffic, they do not contain requests that are satisfied by client-side caches and never sent over the network. There are also problems with lost packets, latencies introduced by the network, and inconsistencies in the traces. However, the advantage of NFS traces is, that they can be passively captured through a mirror port on the network switch, which leaves the machine under test undisturbed [Bla92, ELMS03].

System Call Traces Traces that capture system calls from user-space similar to the Linux tool `strace`. These traces have to be captured on the same system the applications run on, which can influence the performance of the file system. The overhead of user-space capturing tools can be up to 100 times worse than capturing in kernel-space [PBS16]. The traces also do not contain lower-level operations, such as memory-mapped reads and writes [JWZ05].

VFS Traces Traces recorded on the level of the VFS (see section 2.3). The recording and replay requires a special VFS layer, but the accuracy and performance is much better than traces recorded on higher levels [JWZ05].

Block I/O Traces Traces recorded on the block device (see section 2.2) level.

CHAPTER 3

Previous Work

The *Greedy* [RO92] and *Cost-benefit* [RO92] [KNM95] policies were originally suggested by Rosenblum und Ousterhout [RO92] for the use with Sprite-LFS. Although the recent trend in log-structured file system development seems to point towards write-time hot data separation, these policies are still used in modern file systems [LSHC15]. The *Cost-benefit* policy basically tries to separate hot and cold data at garbage collection time by using the age of the segment as a simple measure of its update likelihood [MLE14].

Seltzer et al. [SBMS93] implemented a log-structured file system for UNIX. Much like NILFS2 they used a user-space garbage collector to provide more flexibility and easier configuration. This implementation also revealed a fundamental problem with log-structured file systems. In case of high disk utilization the garbage collection overhead becomes very high, because the segments have no time to accumulate dead blocks. This result was further examined in a subsequent paper of Seltzer et al. [SSB+95] and was the motivation for the hole-plugging algorithm proposed by Matthews et al. [MRC+97].

Jambor et al. [JHT+07] created a log-structured file system implementation for Linux that coincided with the publication of an early version of NILFS2. However, unlike NILFS2 it was not subsequently added to the mainline Linux kernel. Their design was inspired by the UNIX implementation of Seltzer et al. [SBMS93] and both implementations share some major features. Among other things, they both use a user-space garbage collector and an inode file called the *ifile* to store meta-data. Unlike NILFS2 this version of LFS only supports a single snapshot. As long as the snapshot exists, the garbage collector simply does not clean any segments older than the snapshot.

As a natural extension to *Cost-benefit* Min et al. [MLE14] introduce the *Cost-hotness* policy. They use NILFS2 as a starting point and remove all the snapshotting features. Furthermore they add eager hot and cold data separation at segment creation time, whereby blocks are ordered into groups by an *iterative segment quantization* algorithm. Min et al. define the *hotness* of a block or file generally as $\frac{write\ count}{age}$, whereby newly created blocks inherit the hotness of the file they belong to. Using this definition, the blocks are proactively sorted into hotness groups before they are written out, so that blocks with similar hotness end up in the same segment. Additionally, the garbage collector uses the hotness of the blocks to calculate the overall segment hotness, which is the basis for the victim segment selection by the *Cost-hotness* policy. Since the directly measured hotness is a much better estimate for the update likelihood than the age of the segment, the *Cost-hotness* policy is superior to *Cost-benefit*. For the *Cost-hotness* policy the tracking of free space is also required, but by removing all snaphotting features from NILFS2 the implementation becomes fairly trivial.

Qiu et al. [QR13] propose a new file system called NVMFS, which uses non-volatile DIMMs in combination with a normal consumer SSD. These non-volatile DIMMs are as fast as normal DRAM, but can serve as a permanent data storage. NVMFS uses NVRAM to store meta-data and other hot data, which is frequently accessed and updated, and migrates the cold data to the SSD using a log-structured approach. So the NVRAM acts as both a cache for frequently accessed data and permanent storage for meta-data. The data blocks are also grouped according to their update likelihood and they use a variation of the *greedy* policy for the garbage collector. In contrast to the approach presented in this paper, NVMFs does not allow snapshots and therefore does not have to compensate for inaccurate meta-data.

Hu et al. [HEH$^+$09] propose a new *Windowed-greedy* policy for the flash translation layer of SSDs. Instead of comparing the usage of all segments and choosing the best ones, which is computationally expensive, the *Windowed-greedy* policy only considers a certain constant number of the oldest segments. Since the oldest segments are most likely to be empty, the policy can save resources and still achieve a nearly optimal result. They test their theoretical analysis on a SSD-simulator written in Java.

Wang and Hu [WH02] propose the WOLF reordering write buffer for log-structured file systems. WOLF reorders blocks at segment creation time according to their update likelihood or hotness. Their approach is quite similar to the one of Min et al. [MLE14] in that blocks are grouped into different segment buffers before they are written to disk. But instead of using the hotness information for the garbage collection process, they use a combination of the *Cost-benefit* and *Hole-plugging* policies.

The F2FS file system [LSHC15] is a rapidly maturing log-structured file system for Linux. It is specifically designed to perform well on modern flash storage devices and uses a technique called *adaptive logging* to solve the problem of extremely high garbage collection overheads in cases of high file system utilization. In other words, it switches between sequential writes and random writes depending on the file system status. This design is somewhat similar to the adaptive hole-plugging algorithm proposed by Matthews et al. [MRC$^+$97].

To my knowledge there currently does not exist any log-structured file system that supports both snapshots and efficient garbage collection policies at the same time. The current research seems to focus on hot data tracking rather than support for continuous snapshots like in NILFS2.

CHAPTER 4

Design

As shown in section 2.6.8 NILFS2 uses a user-space garbage collector. The selection policies are implemented in the user-space daemon called `cleanerd`, but they rely on accurate segment usage data, which has to be collected in kernel-space.

Since the stock NILFS2 file system only supports the Timestamp policy (see section 2.6.8), it does not track the number of live blocks at all. However, the tracking of live blocks is a non-trivial issue for a continuous snapshotting file system, where blocks can be active in any number of checkpoints and snapshots.

This chapter describes the design decisions and algorithms used to implement the live block tracking as well as the new selection policies for the user-space tools.

The kernel modifications can be divided into several parts. Firstly, the on-disk format of the SUFILE has to be changed to include additional fields. Then a memory cache for the usage updates is needed to increase performance and to reduce the locking contention in the SUFILE. Thirdly, the live block tracking functionality has to be hooked into the file system at several points. Additionally, blocks collected by the garbage collector have to be handled differently, because the protection period has to be taken into account. Lastly, and most importantly, the problem of snapshots must be handled. To solve the latter problem, I created three different algorithms and evaluated each separately.

4.1 On-disk Format Extension of the SUFILE

As described in section 2.6.6, the SUFILE holds segment usage information. Every segment has one entry in the SUFILE. It is the natural place to store the number of live blocks for segments. Table 4.1 shows the stock fields in the segment usage entry.

To support all the features described in subsequent sections, three more fields are necessary. This changes the on-disk format of the SUFILE, but NILFS2 meta-data files have support

Type	Name	Description
`__le64`	`su_lastmod`	Timestamp when the last partial segment was written to this segment. Bascially a creation timestamp.
`__le32`	`su_nblocks`	The total number of blocks in the segment. *Not* the number of live blocks.
`__le32`	`su_flags`	Flags indicating the state of the segment.

Table 4.1: Segment Usage Entry (NILFS2 Stock)

for extensions like this. So there are no compatibility issues and a file system with the new disk format is fully backwards compatible with older drivers. Nevertheless, a feature compatibility flag was added to indicate the on-disk format change. Table 4.2 describes the three new fields.

Type	Name	Description
`__le32`	`su_nlive_blks`	The number of live blocks.
`__le32`	`su_nsnapshot_blks`	Blocks protected by a snapshot.
`__le64`	`su_nlive_lastmod`	Last time the field number of live blocks was modified.

Table 4.2: Segment Usage Entry (Extension)

`su_nlive_blks` Tracks the number of live blocks in a segment. Its value should always be smaller than or equal to `su_nblocks`, which contains the total number of blocks in the segment.

`su_nsnapshot_blks` Contains the number of blocks in a segment that are protected by a snapshot. The value is meant to be a heuristic for the GC and is not necessarily always accurate.

`su_nlive_lastmod` Is necessary because of the protection period used by the GC. It is a timestamp, which contains the last time `su_nlive_blks` was modified. For example if a file is deleted, its blocks are subtracted from `su_nlive_blks` and are therefore considered to be reclaimable by the kernel. But the GC additionally protects them with the protection period. So while `su_nlive_blks` contains the number of potentially reclaimable blocks, the actual number also depends on the protection period. To enable GC policies to prefer segments with unprotected blocks, the timestamp in `su_nlive_lastmod` is necessary.

4.2 Cache for SUFILE Updates

Access to the SUFILE is protected by a single reader/writer semaphore. Consequently an arbitrary number of readers, but only a single writer can gain access at the same time. This design is fine for the stock NILFS2 implementation, because the SUFILE is mostly read by the user-space garbage collector and only ever written by the segment construction thread.

However, the tracking of live blocks involves lots of small asynchronous writes whenever a block gets overwritten or deleted. Most of the time the `su_nlive_blks` field has to be decremented for every block individually. For instance, if a large file is deleted, the `su_nlive_blks` field is decremented by one for every block in the file. This means that the SUFILE semaphore has to be acquired and released at least once for every single deleted block. As a result the access pattern changes dramatically. The SUFILE experiences mostly small writes and comparatively few reads. Additionally, there is a potential deadlock if blocks from the SUFILE itself are deleted. A naive implementation would effectively turn the SUFILE semaphore into a global lock for the whole file system.

To overcome this challenge, I introduced a cache for the live block updates. This cache accumulates the changes in-memory and is flushed to the SUFILE at segment creation time. The locking of the cache is optimized specifically for maximum concurrency of write operations.

4.3 Hook Tracking Functions into File System

Since the stock NILFS2 implementation does not track live blocks at all, there is no simple straight forward way to count the number of live blocks. Instead, the tracking functions have to be hooked into the file system at several different places. Whenever a block gets overwritten or deleted the SUFILE cache has to be decremented for the corresponding segment.

The DATFILE offers a good opportunity to track most files in the file system, because, whenever a block dies, it records the current checkpoint number for said block. The purpose of this feature is to protect snapshots from the GC, but it can be adapted to also track the number of live blocks. However, the above approach does not work for the DATFILE itself, which has to be tracked by several additional hooks in other places.

Since the hooks have to be inserted all over the file system, the SUFILE cache cannot assume that any locks are held during its execution. More importantly it cannot assume that any locks are not already taken, which could lead to a deadlock. As a result the locking scheme for the SUFILE cache has to take the different execution environments into account.

4.4 Recheck Live Blocks and Protection Period

The user-space garbage collector uses the concept of a so called protection period. The protection period protects a newly created checkpoint from automatic deletion for a certain amount of time. See section 2.6.8 for more details.

It is not the segments that are protected, but the checkpoints, so it does not matter when a block was created or in which segment it is located, but in which checkpoint it died. As a consequence protected blocks can occur in any segment, and they have to be treated as live blocks by the garbage collector.

However, the kernel knows nothing about the protection period and the blocks that are considered protected by the garbage collector are dead to the kernel. So the kernel has to recheck the work of the garbage collector.

4.5 Handling Snapshots

As described in section 2.6.2, NILFS2 creates checkpoints automatically in certain intervals and the GC can delete existing checkpoints automatically to free up space. Consequently, simple checkpoints do not influence the live block tracking. However, the user can turn any checkpoint into a snapshot at any time, which prevents the GC from deleting it. This way snapshots keep deleted blocks alive, which can mislead the GC selection policy into selecting sub-optimal victim segments. To make matters worse, the user can also turn a snapshot back into a checkpoint at any time.

However, the segment selection heuristics presented in the literature [RO91, KNM95, LSHC15, MKC$^+$12] rely on information about the segment utilization, which changes constantly when snapshots are used [CE00]. Jambor et al. [JHT$^+$07] solved this problem by supporting only a single snapshot and preventing the garbage collector from collecting any segments that were created before the snapshot. So a snapshot prevents all modification of segments that existed prior to the snapshot itself. These locked up segments may contain reclaimable free space, which cannot be reused by the file system until the snapshot is removed and the garbage collector has processed them. This single snapshot seems to be intended as a temporary tool for online backups. Unfortunately, this approach does not work well with the unlimited number of permanent snapshots supported by NILFS2. The file system would quickly run out of free space, because every new snapshot would lock up more and more segments, thereby shrinking the space available for new data and the GC.

I have implemented three different algorithms to solve this issue. Each algorithm can be activated with a feature flag when the file system is created. This section describes their design.

4.5.1 Implementation 1: Prevent Starvation

This is the simplest solution to the problem. It accepts that snapshots introduce inaccurate live block counts that mislead the selection policies of the GC. Therefore after a snapshot was created or removed the performance of the GC degrades for a certain period of time. This degradation is not permanent, however, because every time the GC treats a segment, it corrects any invalid live block counts. So after some time the performance should recover to its previous level. In essence this algorithm uses the operation of the GC to gradually correct the information in the SUFILE.

There is only one problem with this approach and that is starvation of segments. The selection policy decides which segments the GC visits next based on the information in the SUFILE. If that information erroneously suggests that most blocks in a certain segment are live, then it will never be cleaned and the free space it holds is permanently lost to the user.

The following steps will lead to starvation of a segment:

1. A new segment is created.
2. The user turns a checkpoint into a snapshot.
3. Most of the blocks in the segment get overwritten or deleted.
4. The selection policy chooses the segment for cleaning because of its low number of live blocks. This number is misleading, however, because in actuality most blocks are protected by the snapshot created earlier. So the GC corrects the number of live blocks and includes the protected blocks. This adjustment is necessary to prevent the selection policy from selecting the same segment again immediately.
5. The user doesn't need the snapshot any more and converts it back into a checkpoint.
6. The blocks in the segment are now reclaimable free space and no longer protected, but the GC will never attempt to clean that segment again. The information in the SUFILE suggests, that it has a high number of live blocks, which will never be corrected.
7. The free space in the segment is lost and the result is starvation.

It is important to note that actual starvation is only the worst-case scenario and it would be rather rare in practice. Most of the time these segments would only lock up a lot of free space for an unusually long time. Eventually they would be chosen by the selection policy if the free space in the file system is scarce enough. Nevertheless, the problem has to be dealt with, because these starving segments would accumulate over time and eventually render the file system unusable.

The solution for this problem involves two steps:

1. Mark the SUFILE entries that were updated by the GC and note the number of blocks protected by snapshots. As shown in table 4.2 the SUFILE entry has a field called `su_nsnapshot_blks` for this purpose. It is not managed by the kernel and its value does not have to be accurate at all. It contains the number of snapshot protected blocks the GC found the last time it scanned the segment. A value other than zero also functions as a marker for a potentially starving segment.

   There are two ways to get this information from the GC to the kernel. Firstly through the `ioctl()` system call used to update the segment usage information in case of a late cleaning abort (see section 5.1). This way the GC can directly write to the SUFILE entry.

   Secondly through a new flag for the `nilfs_vdesc` data structure. The flag marks blocks that are protected by a snapshot. On the kernel side the flag is passed on to the `buffer_head` data structure to get the information to the segment construction phase. During segment construction the flagged blocks are counted and written to the `su_nsnapshot_blks` field of the SUFILE entry for the new segment.

2. Call the function `nilfs_sufile_fix_starving_segs()` after a snapshot was turned back into a checkpoint. This would be after step 5 in the starvation example above. The function scans through all SUFILE entries looking for entries with a high value of `su_nsnapshot_blks`. The field `su_nsnapshot_blks` acts both as a marker and as an indicator of the severity of the problem. If the fields `su_nlive_blks` and `su_nsnapshot_blks` are both bigger than half of the segment size, it reduces both values to that size. Basically there is a cutoff limit of 50% for all potentially starving segments.

   A lower value of `su_nlive_blks` makes the segment more attractive for the GC. As a result the segments are more likely to be chosen by the GC in the future and will not starve.

The value of 50% for the cutoff limit is a trade-off between the resulting performance degradation and the amount of free space locked up in starving segments. In the worst-case scenario, the GC starts to re-examine potentially starving segments, when the file system has 50% of free space left. The ideal value for this percentage will likely vary depending on the file system usage pattern. A dynamic approach whereby the cutoff limit depends on the free space left in the file system could be advantageous and could be a topic for future work.

This solution is not perfect, because there can be more than one snapshot and every time one of them is deleted, all SUFILE entries are scanned and potentially reduced in value. This conservative strategy is necessary, because there is not enough information to determine which blocks are protected by which snapshot. So there will be a significant performance decrease every time the user creates or deletes a snapshot until the GC has corrected the SUFILE entries.

The big advantage of this algorithm is its scalability. Although the SUFILE has to be scanned in its entirety, it is quite small compared to the corresponding file system and it should reside mostly in the page cache since the GC has to scan it regularly to select victim segments. In other words its performance will scale well with the rest of the file system and it is unlikely to become a bottleneck at certain file system sizes.

4.5.2 Implementation 2: Scan DATFILE for Dead Blocks

Whereas the previous algorithm uses a Laissez-faire approach, this one tries to actually correctly track the blocks protected by snapshots and update the corresponding SUFILE entries. Whenever a checkpoint is turned into a snapshot or back, any block in the file system could be affected, because a snapshot protects all blocks that were alive when the checkpoint was created. Since live blocks can be anywhere on the disk, and they are constantly moved around by the GC, literally all usage counters could be in need of an update. To solve this issue, this implementation scans the whole DATFILE for blocks in need of an update, whenever the user creates or deletes a snapshot.

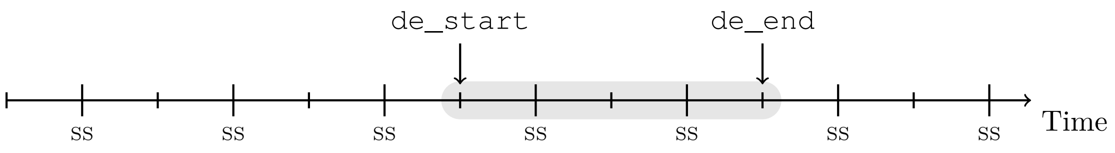


Figure 4.1: Block lifetime

Figure 4.1 shows the lifetime of a block from its creation, marked with `de_start`, to its deletion, marked with `de_end`. The points on the X axis represent checkpoints over time, whereby some checkpoints are also snapshots. Any snapshot that falls within the interval between `de_start` and `de_end` of a block, prevents that block from being collected by the GC. So if there is any snapshot in that interval, the block has to be counted as alive. Additionally, the algorithms presented in this chapter have to take the likely possibility of multiple snapshots per block into account.

DATFILE Extension

With a few exceptions, the DATFILE contains an entry for every block in the file system. As shown in section 2.6.7 it is used to translate virtual into physical block addresses. Apart from that, it contains information about the life cycle of every block in the form of the creation (`de_start`) and deletion (`de_end`) checkpoint numbers. This is most of the information needed to update the segment usage counters. The physical block address can be used to calculate the segment number, and the checkpoint numbers determine if a block belongs to a certain snapshot or not.

In addition to that the update history has to be tracked for every block, because one block can be influenced by multiple snapshots. For example if a dead block belongs to two checkpoints and one of them is turned into a snapshot, then the usage counter of the corresponding segment is incremented. However, if the second checkpoint is also turned into a snapshot, the counter must not be incremented again. So the DAT entry must be able to indicate if the corresponding segment counter was previously incremented or decremented. Fortunately, the DAT entry has an unused reserved field called `de_rsv` that can be adapted for this purpose, which has the added benefit of not requiring an on-disk format change.

Type	Name	Description
`__le64`	`de_blocknr`	Physical block number
`__le64`	`de_start`	Checkpoint number where the block was created
`__le64`	`de_end`	Checkpoint number where the block was deleted/overwritten
`__le64`	`de_ss`	One of the snapshots the block belongs to. This value is in-between `de_start` and `de_end`.

Table 4.3: Fields in the DAT Entry

Table 4.3 shows the fields of the DAT entry with the field `de_rsv` renamed to `de_ss`. The new field can hold either a snapshot number or two special snapshot numbers used as flags:

`NILFS_ENTRY_INC` Is equal to zero, which is never used for actual checkpoints or snapshots. It indicates that the corresponding segment usage counter for this block was previously incremented and does not need to be incremented again. It is the initial value for newly created DAT entries.

Snapshot Number Same as `NILFS_ENTRY_INC` but it also contains the actual snapshot number responsible for the increment. As shown in figure 4.1, the lifetime of a block can span over multiple snapshots, and its segment usage counter must not be decremented before all of them are deleted. The snapshot number in this field is used to lookup neighboring snapshots from the sorted list in the CPFILE (see secton 4.5.2). If one of the neighboring snapshots falls within the lifetime of the block, it is still protected and its usage counter must not be decremented.

`NILFS_ENTRY_DEC` Is equal to special checkpoint number `NILFS_CNO_MAX`, which is also never used for actual checkpoints. This flag indicates that the corresponding segment usage counter was decremented and must not be decremented again, but it can be incremented.

Whenever the usage counter of a segment changes and the `de_ss` field does *not* contain a snapshot number, the marker flags have to be adjusted. For example, if the GC moves a block from one segment to another it also updates the usage counter of the target segment. Therefore the `de_ss` field has to be set to `NILFS_ENTRY_INC` to make it consistent with the new state. Furthermore, if a block is deleted or overwritten, the usage counter of its segment is decremented and its `de_ss` must be set to `NILFS_ENTRY_DEC`.

Helper Functions

Algorithm 4.1 introduces a few helper functions that are used by algorithms in later sections. Although they are quite simple, their descriptive names help with the readability of the more complex algorithms that use them.

Algorithm 4.1 Snapshot Helper Functions

```
function ISDEAD($cno$)
    return $cno \neq CNO\_MAX$
end function

function ISMARKER($ss$)
    return $ss = ENTRY\_INC \vee ss = ENTRY\_DEC$
end function

function CONTAINS($start$, $end$, $cno$)
    return $cno \geq start \wedge cno < end$
end function

function INCUSAGE($segnum$)
    Access the SUFILE cache and increment the usage counter for $segnum$
end function

function DECUSAGE($segnum$)
    Access the SUFILE cache and decrement the usage counter for $segnum$
end function
```

ISDEAD(*cno*) This function takes a checkpoint number and compares it to the special value `NILFS_CNO_MAX`, which is not a real checkpoint number and only used as a marker. If the `de_end` field has the value `NILFS_CNO_MAX`, it is alive. Conversely, if it has any other value it must be dead.

ISMARKER(*ss*) This function takes a snapshot number from the `de_ss` field and tests if it is one of the two marker values `NILFS_ENTRY_INC` or `NILFS_ENTRY_DEC`.

CONTAINS(*start*, *end*, *cno*) This function returns *true* if the checkpoint number falls within the interval of $[start, end)$. Since the checkpoint numbers are guaranteed to increase strictly monotonically, a snapshot that falls within this interval protects the corresponding block.

INC/DECUSAGE(*segnum*) The SUFILE cache described in section 4.2 provides these functions. The segment number can be calculated from a physical block address by a simple division.

After Snapshot Creation

After the user has turned a checkpoint into a snapshot, the blocks that were alive at the time the checkpoint was written are protected from the GC.

From the perspective of the GC protected blocks cannot be discarded and must be treated the same as live blocks. So protected blocks increase the segment cleaning costs the same way live blocks do. Since the GC selection policies use the segment usage counters to estimate the cleaning cost of a segment, it is important to increment the counters by the number of resurrected blocks. To that end the whole DATFILE has to be scanned for these kinds of blocks.

Algorithm 4.2 DAT Entry Selection: Snapshot Creation

```
function SELENTRY($start$, $end$, $ss$, $oldss$)
    return ISDEAD($end$) $\wedge$ ISMARKER($oldss$) $\wedge$ CONTAINS($start$, $end$, $ss$)
end function
```

Algorithm 4.2 shows the function that pre-selects DAT entries for further processing. Firstly, an entry has to be dead and not protected by another snapshot. Secondly, its life-span has to cover the newly created snapshot. One block can belong to many snapshots, but there is no need to know all of them. It suffices to know, that it belongs to one or more snapshots. So blocks that belong to another snapshot are ignored.

Algorithm 4.3 Scan DATFILE After Snapshot Creation

```
 1: $ss \Leftarrow$ Created snapshot
 2:
 3: for all entries $de$ in the DATFILE do
 4:     $segnum \Leftarrow de.de\_blocknr \div segment\ size$
 5:     $start \Leftarrow de.de\_start$
 6:     $end \Leftarrow de.de\_end$
 7:     $oldss \Leftarrow de.de\_ss$
 8:
 9:     if SELENTRY($start$, $end$, $ss$, $oldss$) then
10:         $de.de\_ss \Leftarrow ss$                        ▷ Update DAT entry
11:
12:         if $oldss = ENTRY\_DEC$ then                ▷ Was decremented before
13:             INCUSAGE($segnum$)
14:         end if
15:     end if
16: end for
```

Every time the user turns a checkpoint into a snapshot, Algorithm 4.3 is executed. It receives the checkpoint number of the new snapshot as an input parameter and scans every DAT entry in the DATFILE. Every unprotected, dead entry that belongs to the new snapshot, is marked as such by setting the `de_ss` field to the new snapshot number.

If the segment usage counter for an entry was previously decremented, indicated by the `NILFS_ENTRY_DEC` marker, it is incremented again.

After Snapshot Deletion

The algorithm that is executed after the deletion of a snapshot is more complex than the one for snapshot creation, because a block can be protected by an arbitrary number of snapshots. Consequently, the segment usage counters cannot be simply decremented for every block that belongs to a snapshot. Instead the algorithm has to make sure that no other snapshot falls within the range of the interval $[start, end)$ of the DAT entry.

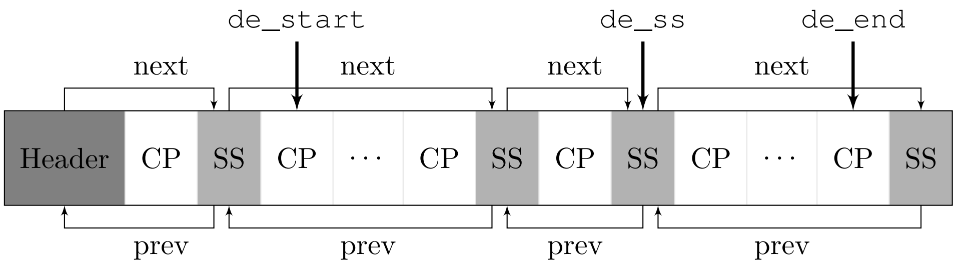


Figure 4.2: CPFILE with Snapshot Linked List

Fortunately, the structure of the CPFILE, where the checkpoints are stored, facilitates this operation by providing a sorted linked list of snapshots. Figure 4.2 shows the CPFILE and its relatively simple structure. It consists of a header followed by a simple flat array of fixed-size checkpoints. The checkpoint numbers can be used directly as an index into this array. To guarantee that the checkpoint numbers increase strictly monotonically, old entries are never reused and new entries are appended at the end.

There is no fundamental difference between a checkpoint and a snapshot. Snapshots are just specially flagged checkpoints that are inserted into a sorted linked list. The linked list is an optimization to quickly find all the snapshots among the ordinary checkpoints. The fact that it is sorted by checkpoint number turned out to be very useful for this implementation.

Algorithm 4.4 DAT Entry Selection: Snapshot Deletion

```
function SELENTRYDEL($start$, $end$, $ss$, $oldss$)
    return ISDEAD($end$) $\wedge$ (ISMARKER($oldss$) $\vee$ $oldss = ss$) $\wedge$
        CONTAINS($start$, $end$, $ss$)
end function
```

Algorithm 4.4 shows the selection function for the snapshot deletion. The only difference to the version used for the snapshot creation (see algorithm 4.2) is that DAT entries marked with the deleted snapshot are also selected.

Algorithm 4.5 Scan DATFILE After Snapshot Deletion

```
1: $ss \Leftarrow$ Deleted snapshot
2: $prev \Leftarrow$ Snapshot previous to $ss$
3: $next \Leftarrow$ Snapshot next of $ss$
4:
5: for all entries $de$ in the DATFILE do
6:     $segnum \Leftarrow de.de\_blocknr \div segment\ size$
7:     $start \Leftarrow de.de\_start$
8:     $end \Leftarrow de.de\_end$
9:     $oldss \Leftarrow de.de\_ss$
10:
11:    if SELENTRYDEL($start$, $end$, $ss$, $oldss$) then
12:        if CONTAINS($start$, $end$, $prev$) then
13:            $newss \Leftarrow prev$                          ▷ Found another snapshot
14:        else if CONTAINS($start$, $end$, $next$) then
15:            $newss \Leftarrow next$                          ▷ Found another snapshot
16:        else if $\neg$ISDEAD($end$) then
17:            $newss \Leftarrow ENTRY\_INC$                    ▷ No adjustment needed
18:        else
19:            $newss \Leftarrow ENTRY\_DEC$                    ▷ No other snapshot in range
20:        end if
21:
22:        if $oldss = newss$ then                          ▷ Abort if nothing has changed
23:            continue
24:
25:        $de.de\_ss \Leftarrow newss$                         ▷ Update DAT entry
26:
27:        if $newss = ENTRY\_DEC$ then                     ▷ No longer protected by snapshot
28:            DECUSAGE($segnum$)
29:        else if $oldss = ENTRY\_DEC$ then                ▷ Was not protected but is now
30:            INCUSAGE($segnum$)
31:        end if
32:    end if
33: end for
```

Algorithm 4.5 is executed every time a snapshot is deleted by the user. It gets the checkpoint number of the deleted snapshot as well as the previous and next snapshot from the linked list as input.

It scans every DAT entry of the DATFILE and uses the function shown in algorithm

4.4 to select entries in need of an update. Figure 4.2 shows an example of a DAT entry with its fields `de_start`, `de_end` and `de_ss` pointing to checkpoint numbers in the CPFILE. If the previous or the next snapshot lie within the interval of $[start, end)$, then the corresponding block is still protected by another snapshot and the segment usage counter must not be decremented. If on the other hand both the previous and the next snapshot are outside of said interval, then the block is unprotected and can be decremented.

Support for the Late Abort GC Feature

As described in section 5.1 the GC has the ability to abort an inefficient segment cleaning and instead update the segment usage information directly in place. After such an update, the marker flags in the `de_ss` field of the corresponding DAT entries might be wrong. To compensate for that, I introduced a new `ioctl()` system call `SET_INC_FLAGS`, which is used by the GC to flip the `de_ss` value from `NILFS_ENTRY_DEC` to `NILFS_ENTRY_INC`.

4.5.3 Implementation 3: Scan Whole DATFILE

The third implementation is an extension of the second implementation described in section 4.5.2. It differs only in the two selection functions shown in algorithm 4.6:

Algorithm 4.6 DAT Entry Selection Functions

```
function SELENTRY($start$, $end$, $ss$, $oldss$)
    return ISMARKER($oldss$) $\wedge$ CONTAINS($start$, $end$, $ss$)
end function

function SELENTRYDEL($start$, $end$, $ss$, $oldss$)
    return (ISMARKER($oldss$) $\vee$ $oldss = ss$) $\wedge$ CONTAINS($start$, $end$, $ss$)
end function
```

Instead of only looking at dead blocks, this implementation also scans live blocks. As a result more DAT entries need to be scanned and updated, but the tracking accuracy is significantly increased. In fact with this approach the segment usage counters are always accurate and all corner cases are covered.

It is more important to increment the usage counters for dead blocks, since values that are too low mislead the GC selection policy, but in the following scenario the treatment of live blocks is also important:

1. Write `File1`
2. Create snapshot #1
3. Delete `File1`

File1 is protected by snapshot #1 and alive at the time the snapshot is created. If the DAT entries of File1 are not marked as protected, then the usage counter will be decremented when File1 is deleted. So *Implementation 2* creates invalid usage values if protected files are deleted and *Implementation 3* works in all cases.

So, the great advantage of this method over *Implementation 2* is that it always keeps the usage counters up-to-date with perfect accuracy. On the one hand, this improves the performance of the garbage collector, but on the other hand, it increases the time, and the number of write operations needed to create or delete a snapshot. There is also an issue with scalability, because the number of DAT entries that have to be scanned increases linearly with the size of the file system. While *Implementation 2* tries to tweak this trade-off between accuracy and performance in the direction of performance, *Implementation 3* tries to achieve perfect accuracy.

4.6 Add Selection Policy Framework

Although the parser for the configuration file supports multiple selection policies, there is no framework or interface to implement the policies. The timestamp policy simply sorts the segments according to their age and selects the oldest ones, which is directly implemented in the cleanerd daemon without any abstraction or framework.

So I decided to introduce a clean and simple interface for selection policies. Listing 4.1 shows the two function pointers a policy has to implement.

Listing 4.1: Selection Policy Data Structure

```
struct nilfs_selection_policy {
        int (*evaluate)(const struct nilfs_sustat *sustat,
                        const struct nilfs_suinfo *si,
                        int64_t prottime, int64_t now,
                        long long *imp);
        int (*compare)(const void *a, const void *b);
};
```

evaluate() This function evaluates one segment and calculates its importance as a candidate for cleaning. The importance is an integer used to compare and sort segments. The function returns 1 to indicate success or an error code otherwise. Apart from the segment usage information it takes various statistics and timestamps as parameters.

compare() This function compares the importance of two segments. It is used to sort segments according to their importance. The reason for it being part of the selection policy is, that the integer value of the importance can have different meanings for different policies. Some require the segments to be sorted in ascending and some in descending order.

For example, the Timestamp policy simply sets the importance to the last modified timestamp of the segment and uses a compare function that leads to an ascending order. This way the segments with the smallest importance are selected first. The segments with the smallest importance have the smallest timestamp and are therefore the oldest segments.

4.6.1 Add Support for Greedy/Cost-Benefit Policies

Using the policy framework introduced in section 4.6, I implemented the original Timestamp policy as well as two new policies called Greedy and Cost-Benefit. These are well known policies for log-structured file systems, which were used by Rosenblum et al. [RO91] for their implementation of LFS.

Greedy The Greedy policy is very simple. It subtracts the number of live blocks from the total number of blocks in the segment. The result is the number of dead blocks in the segment, which is directly used as the importance value. Furthermore, it sorts the segments in descending order, so that the segments with the biggest number of dead blocks are selected first.

Cost-Benefit [RO91] The Cost-Benefit policy, as the name suggests, performs a cost-benefit analysis, whereby the free space gained is weighed against the cost of collecting the segment. The basics of Cost-Benefit are described in section 2.5.6.

4.7 Benchmarks

4.7.1 File System Aging

The general concept of file system aging is described in section 2.7. Aging is ideal for benchmarking the garbage collector, because it simulates a real world workload and creates a realistic segment usage distribution. The usage and age of the segments are the primary inputs for the selection policy.

I chose to use NFS traces for the aging benchmark, because they are readily available from the IOTTA Repository [Rep11] and come in a simple human readable text format. There is also a more advanced storage format called DataSeries [AAMV09]. It is a general purpose compressed binary format for structured serial data that is suitable for NFS traces. It supports the separation of the data into different tables like a relational database. Anderson [And09] shows that the DataSeries format is more efficient for capturing and analyzing NFS traces than the text format used by Ellard [ES03]. Nevertheless, the format is very complicated to use and there are to my knowledge no replay tools available.

Although the difficulties with NFS traces are well known and researched [Bla92, ELMS03, ZCcCE03, TV14], the publicly available open source tool Nfsreplay [nfs08] was unable to handle the errors and inconsistencies in the NFS traces from the IOTTA Repository [Rep11] without crashing. Moreover, it is designed to benchmark NFS servers and not file systems. Thus it requires a running NFS server to work, which is impractical for an automated benchmark. On top of that, the project seems to be abandoned since 2008, when the last release was published.

The tool TBBT [ZCcCE03] is not open source and not publicly available, it requires extensive preprocessing and an NFS server to work. Similarly, the proprietary replay tool ParaSwift [TV14] uses a sophisticated modeling approach to handle the increasing size of traces generated by modern workloads, but it is not freely available.

So after many failed attempts to modify Nfsreplay, I decided to write my own replay tool from scratch. My replay tool is a stand-alone console application, consumes the compressed text format directly, corrects for errors and inconsistencies on the fly, supports all NFS protocol versions, and is able to handle arbitrarily large traces, by continuously removing the oldest directory entries after a maximum amount of memory is consumed. It is fully open source and available on GitHub [nfs17].

CHAPTER 5

Implementation

This chapter describes the implementation details for the algorithms discussed in chapter 4.

5.1 Late Abort of Inefficient Cleaning Operations

This section describes a feature I have implemented in the course of this thesis as a first step to optimize the garbage collector for NILFS2. In addition, I have submitted the patches to the mainline kernel and they have been accepted. It can be enabled and disabled in the GC configuration file with the `set_suinfo` flag.

Since this feature is now enabled by default in NILFS2 all the benchmarks presented in this thesis are run with it enabled. As a result, the performance of the stock Timestamp policy is significantly better than it would have been in its original version.

Depending on the selection policy (e.g. Timestamp) and the file system usage, segments with a high percentage of live blocks can be selected for cleaning. In that case all live blocks have to be copied to a new segment and very little free space is gained. This consumes a lot of disk bandwidth and is quite inefficient. The Timestamp policy of the stock NILFS2 is especially vulnerable for this problem, because it always selects the oldest segments regardless of the number of live blocks.

At a certain point during the collection process, the user-space garbage collector has an opportunity to abort the cleaning process. After it has selected the victim segments and collected all the block information, it can determine the ratio of live blocks to free space gained and decide to not go through with the cleaning. This is not ideal, because it has already spent a lot of computational and disk resources to get to this point, but it can still save a lot by aborting.

To avoid that the exact same segments get selected again immediately, they have to be marked in the SUFILE. For this purpose the SUFILE entries of the victim segments are

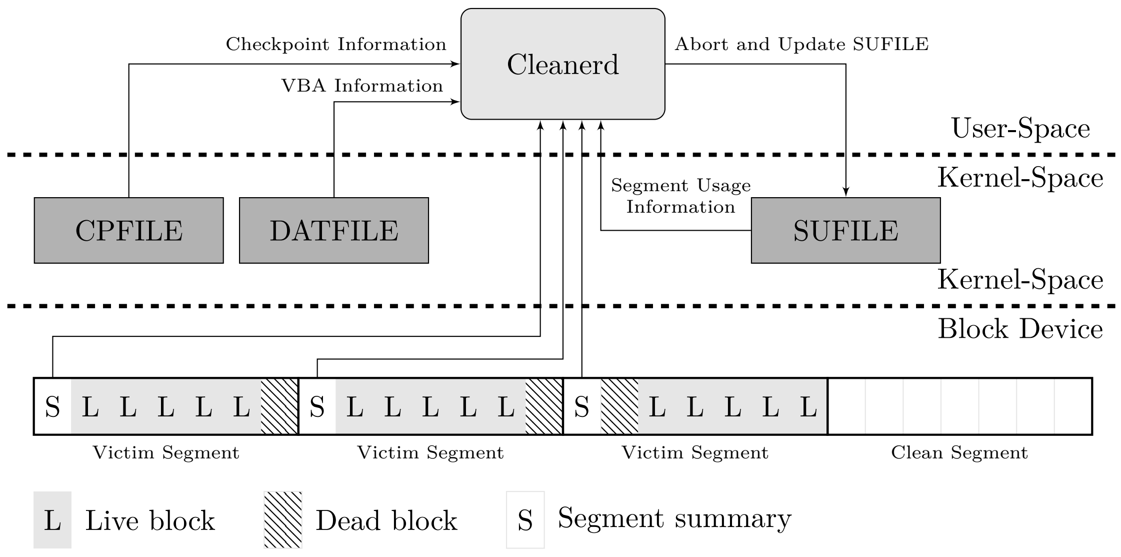


Figure 5.1: Late Abort for Full Segments

updated with the current timestamp. So old segments are turned in-place into young segments. To support this feature the user-space GC needs a way to tell the kernel which SUFILE entries to update. Since there is already a `GET_SUINFO` ioctl system call its pendant is called `SET_SUINFO`.

5.2 Implementation of the SUFILE Cache

This section describes the implementation of the SUFILE cache introduced in section 4.2. This part is crucial for the performance of the whole file system, because nearly every modification to the content of the file system results in a change to the live block counters in the SUFILE. So an efficient cache that can handle high levels of concurrency, introduces minimal overhead, and avoids dead-locks is very important.

5.2.1 Groups and Nodes

The entries of the SUFILE are divided into groups of currently $2^6 = 64$ members. All updates to the `su_nlive_blks` field for a group can be stored in one cache node. A node basically consists of an array of $2^6 = 64$ counters. One for every member of the corresponding group. Additionally, the nodes contain a `rcu_head`, to be able to use the node with an RCU callback, a group index, and a dirty flag. Listing 5.1 shows the C data structure of a cache node.

Listing 5.1: Cache Node Data Structure

```
struct nilfs_sufile_cache_node {
        s32 values[NILFS_SUFILE_CACHE_NODE_COUNT];
        struct rcu_head rcu_head;
        unsigned long index;
        bool dirty;
};
```

The data type for the values is a signed 32-bit integer. Negative values decrement the corresponding `su_nlive_blks` field and positive values increment it. The group index can be directly derived from the segment number by dividing it with the group size: $group = segnum >> 6$. After a modification of the values, the dirty flag is set to true. A dirty node contains changes that have to be flushed to the SUFILE.

The cache nodes are allocated on-demand using a so-called `kmem_cache`, whenever a segment in a group needs an update. A `kmem_cache` facilitates efficient memory allocation, by pre-allocating a reusable pool of nodes. It is a standard memory allocation technique in the Linux kernel.

5.2.2 Radix Tree

The cache nodes are organized in a radix tree. The Linux kernel supports a special variant of radix tree that uses an `unsigned long` as key and an arbitrary pointer as value. The properties of the Linux radix tree are ideal for storing the SUFILE cache nodes, because the group index can be the key, while the cache nodes are the values. Moreover, a radix tree is most efficient if the key space is contiguous without large holes, which applies to the group index ($0 \leq group \leq (nsegments >> 6)$).

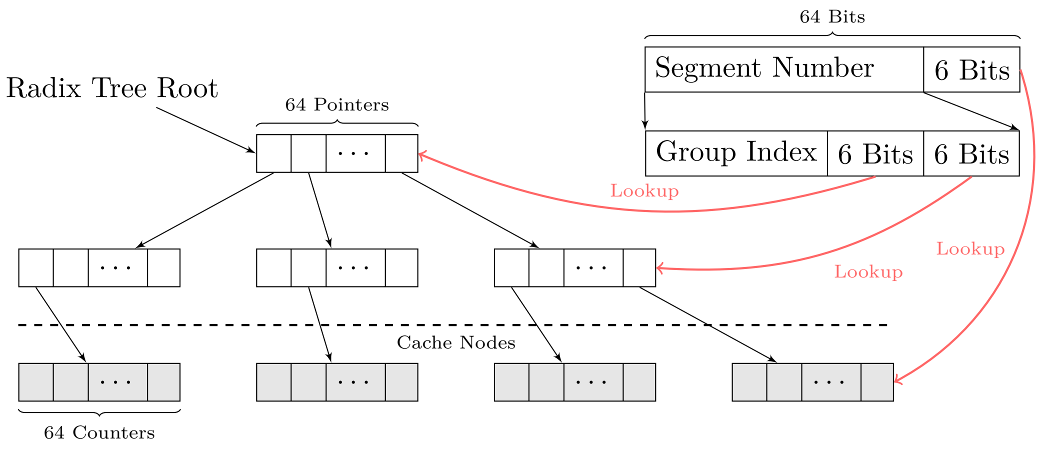


Figure 5.2: Radix Tree Node Cache

A radix tree node has 64 pointers, whereas every pointer either points to another tree node or a value. The lookup procedure splits the key into 6 bit chunks. For every level of the tree a different chunk is used to index the array of pointers in its nodes. So the key can be thought of as an index into a huge flat array and the pointers in the nodes represent a certain range of this array. For instance, the pointers in the root node each represent exactly $\frac{1}{64}$ of the total array, the pointers on the first level represent $\frac{1}{64} \times \frac{1}{64} = \frac{1}{4096}$ and so on. As a result the 6 bit chunks of the index can be used directly to index the nodes. Figure 5.2 shows the lookup of a segment number in the SUFILE cache.

5.2.3 Locking Scheme: Update and Insert

The locking scheme of the SUFILE cache is optimized to allow for maximum concurrency. The access to the radix tree is handled differently from the locking of the cache nodes.

The cache nodes use a global *blockgroup lock*, which is stored in a support memory structure of the SUFILE. A blockgroup lock is a simple extension of a normal spin lock. Instead of using a single spin lock it works on a whole array of spin locks. The size of the array is determined by the kernel at compile time. The blockgroup lock uses a simple algorithm to spread out the load among the array of locks. To get at one of the locks, the user has to supply a group identifier. This identifier modulo the size of the array is used as an index into the array. So a specific group always gets the same lock, but some groups share a lock. It is a compromise between having a single global lock as opposed to one lock per cache node.

The lookup operation for the radix tree is protected by the *Read-Copy-Update (RCU)* synchronization mechanism. RCU is similar to a traditional reader/writer semaphore in that it allows an arbitrary number of readers and a single writer. The difference is that RCU only works with pointers and the readers do not directly synchronize with the writers. Instead the writers keep copies of old versions of objects around, until no reader uses them anymore. The advantage of this approach is extremely fast read access.

For example, when a writer needs to update an object, it creates a new object, initializes it completely, and eventually switches a global pointer from the old to the new version. At the same time multiple readers are allowed to access the global pointer. Some see the new object and some see the old, but all have a consistent view of the world. Since both objects are in use, the writer cannot immediately free the old version. Instead it has to wait until all the readers using the old object have exited their critical sections.

Listing 5.2: SUFILE Cache Locking

```
// Start RCU critical section
rcu_read_lock();

// Protected by RCU
node = radix_tree_lookup(radix_tree, group_index);

// Node access protected by block group lock
lock = bgl_lock_ptr(block_group_lock, group_index);
spin_lock(lock);
/* Update the counters in node */
spin_unlock(lock);
```

```
// End RCU critical section
rcu_read_unlock();
```

However, the writer does not have to keep track of all the readers, because it can deduce their state indirectly from the state of the CPU. The readers are not allowed to sleep within a critical section. So after every CPU has performed at least one context switch, all critical sections using the old object must have finished and it is safe to free it.

Instead of waiting, the writer can also register a callback to free the old object. This way the writer can continue without interruption and the RCU subsystem automatically calls the callback next time the CPU state indicates it to be safe.

Additionally, the insert and delete operations on the radix tree must be synchronized with a spin lock, because they change the structure of the tree and RCU cannot synchronize writers.

Listing 5.2 shows a simplified locking sequence for the SUFILE cache.

5.2.4 Locking Scheme: Flushing

The cache is flushed by iterating over all dirty cache nodes in the radix tree. Any counter with a value other than zero is added to the corresponding SUFILE entry.

Since the locking is heavily optimized for fast updates to the cache, the locking required to flush the cache is very complicated. The main reason for this is that a function that writes to the SUFILE must be able to sleep, because reading and writing blocks is a slow operation that may put a process to sleep. Consequently it cannot hold any spin or RCU locks during its execution, and because pointers to nodes must not be accessed outside of an RCU critical section, it cannot access the nodes it needs to flush.

One possible solution would be to release the RCU lock before a call to a function that might sleep and reacquire it later. However, this would invalidate all pointers from the previous critical section, because they might have been freed in the interim. Consequently, the nodes would need to be looked up again in the radix tree after every sleep.

As it turns out the complicated solution outlined above is not necessary for the SUFILE cache, because there is a third level of locking protecting the nodes from being freed outside of the RCU critical section.

Table 5.1 shows operations performed on the cache and the locks that protect them. As noted in section 5.2.3 a spin lock protects the radix tree from concurrent updates. So the “Insert Tree” and “Remove Tree” operations use the “Radix Tree Lock”. The lookup of nodes in the radix tree is protected by RCU. Since the “Update Node” and “Flush Tree” operations lookup nodes but do not modify the tree, they only need RCU protection.

However, as stated above the “Flush Node” operation cannot be called inside a critical section. In fact such a call would be a serious bug that would threaten the stability

Operation	RCU	BG	Radix Tree Lock	SUFILE Sem.
Insert Tree			✓	
Update Node	✓	✓		
Flush Tree	✓			✓
Flush Node	Missing!	✓		✓
Remove Tree			✓	✓
Free Node	✓			

Table 5.1: Operations Protected by Locks

of the whole system. Fortunately, the SUFILE semaphore can be used to bridge the protection gap. It is needed anyway to get access to the SUFILE and it ensures that the flush operation can never run concurrently with the remove operation.

This is a good compromise because nodes are flushed in the collection stage and removed at the end of segment construction. These two operations always run one after the other in the same thread. So there is no need for them to run concurrently and there would be no performance gain.

5.2.5 Flush Cache Flavors

As shown in section 2.6.4 the segment construction process goes through several re-entrant stages. To handle all the error conditions of this process, which can be interrupted at almost any point, three different flavors of the flushing operation are necessary:

Full Flush This is in a way the simplest case. Every entry of every cache node in the cache is written to its corresponding SUFILE entry.

Mark Flush This operation does not modify the cache or the SUFILE at all. It only marks the blocks of the SUFILE that would be modified by a real flush as dirty. This is useful for the collection stage of the segment construction process, where dirty blocks are collected and added to a new segment. So the mark flush marks SUFILE blocks for inclusion in the next segment, but it does not change any values. In case of an error the stage can be safely aborted and no harm is done.

Write Flush This operation is the opposite of Mark Flush. It does not mark any new blocks as dirty, but it can write to previously dirtied blocks. It is executed very late in the segment construction process, because it changes the SUFILE blocks as well as clears the cache. Since it is a destructive operation, it cannot be used in the block collection stage, which can be interrupted and reiterated many times.

5.2.6 Circular Dependency with the DATFILE

The SUFILE uses virtual block addresses, which have to be translated into physical addresses. Therefore, any change to the SUFILE causes updates to the DATFILE, which in turn cause segment usage updates to the SUFILE. This circular dependency becomes problematic when a checkpoint has to be written to disk. A checkpoint is a point where the whole file system is in a consistent state. Because the SUFILE has to be written before the DATFILE, the segment usage updates from the DATFILE are lost and the file system is no longer consistent.

My solution to this problem is to run the propagation of the DATFILE blocks and the Mark Flush for the SUFILE cache in a loop, until an equilibrium is reached where no new blocks are dirtied. Of course there has to be a limit on the number of iterations and there is no guarantee, that the equilibrium will be reached within the limit. In that rare case the last cache updates have to be written out later or may be lost if the file system is immediately unmounted. These small and rare inaccuracies should not impact the performance of the file system.

The loop is only possible, because Mark Flush does not change any values and can be called multiple times.

5.3 Hook DATFILE Operations

As discussed in section 2.6.7 NILFS2 uses *virtual block addresses (VBA)* instead of *physical block addresses (PBA)* for most files. A VBA maps to an entry in the DATFILE. A DAT entry contains the PBA and two checkpoint numbers, `de_start` and `de_end`. `de_start` is the checkpoint number when the block was created and `de_end` when it died. Every time a VBA is accessed, it has to be translated through the DAT layer into a PBA. This extra level of indirection is necessary to allow the garbage collector to move blocks around efficiently. Instead of updating every reference in every checkpoint and snapshot, it can simply update the DAT entry to point to the new location.

The garbage collector needs `de_start` and `de_end` to check if a block is protected by a snapshot or not. Since the checkpoint numbers are guaranteed to increase strictly monotonic over time, the start and end checkpoint form an interval $[start, end)$. If a snapshot falls within this interval, the block is protected.

So NILFS2 already tracks the lifetime of blocks. When a block is deleted or overwritten, the `de_end` field of its DAT entry is set to the current checkpoint. My implementation extends this existing mechanism to also decrement the usage counter of the corresponding segment. The segment number can be calculated from the PBA, which is directly available in the `de_blocknr` field of the DAT entry.

5.4 Hook B-tree Operations to Track the DATFILE

The only file that does not use VBAs is the DATFILE itself. To track the DATFILE a different mechanism has to be used.

Log-structured file systems suffer from the so called “wandering tree problem”. Every time a block is updated, it is copied and as a result its address changes, which in turn causes the B-tree node that references that address to change, which causes the nodes above to change all the way up to the root node and the inode. To mitigate the problem somewhat, NILFS2 delays the B-tree updates to the segment creation phase.

My implementation hooks into the B-tree propagation function, which is executed during segment creation. At this point the old and new address of the updated block is available. The old address can be directly used to calculate the segment number, which is then used to decrement the corresponding segment usage counter.

However, the method described above only covers block updates, but not block deletions. Only the garbage collector can trigger the deletion of DATFILE blocks, so to support this feature, another function has to be extended. The garbage collector initially calls the function `nilfs_dat_freev()` to free DAT entries. If all entries in a block are freed, the block is deleted. Eventually the B-tree is updated to reflect this change. At that point the old block address is available and it is the ideal location to decrement the segment usage counter.

The B-tree code is ideal to add the tracking functions, because all updates to the DATFILE necessitate changes to its B-tree and all the necessary block addresses are directly available.

5.5 Recheck Live Blocks and Protection Period

As described in section 4.4 the kernel has to recheck the work of the GC to avoid invalid tracking numbers and to account for the user-space protection period.

To that end, I have implemented the function `nilfs_dat_is_live()`, which looks up the DAT entry for a specific block, and checks if it is live or not.

Since the victim segments of the garbage collector are cleared anyway, it is not necessary to decrement their usage counters. It is, however, necessary to adjust the usage counters of the newly created segment, because it may already contain dead blocks, which were protected from being garbage collected by the user-space protection period. So every block coming from the GC is checked during segment construction with `nilfs_dat_is_live()`.

Since the DAT lookup for every GC block is quite expensive, I added the following optimization. I introduced a new flag for the `nilfs_vdesc` data structure, used by the GC to convey block information to the kernel. The flag is set for every dead block that is kept alive by the protection period. On the kernel side the flag is passed on to the `buffer_head` data structure to get the information to the segment construction phase. During segment construction, the marked blocks do not have to be checked again with `nilfs_dat_is_live()`.

5.6 Extension of the User-Space Utilities

The user-space utilities use the same header files as the kernel driver, so many of the modifications needed to extend the SUFILE are identical to the ones described in section 4.1. Additionally, the tool `lssu`, which displays the content of the SUFILE, was extended to show the new fields added to the SUFILE entry.

The feature compatibility flag `sufile_live_blks_ext` was introduced to indicate the on-disk format change. It is enabled by default, but it can be explicitly disabled on the command line: `mkfs.nilfs2 -O ^sufile_live_blks_ext`. The flag cannot be changed after the file system is created, because changing the on-disk format of the SUFILE on an existing file system is not supported.

Because of the on-disk format change, the `mkfs.nilfs2` program needed to be modified quite extensively. Although the design of NILFS2 meta-data files support a dynamic entry size, the original version of `mkfs.nilfs2` uses a constant for the SUFILE entry size. So the entry size had to be changed to be dependent on the state of the `sufile_live_blks_ext` feature flag and all the address calculations for SUFILE entries had to be updated.

5.7 Additional Flags for `nilfs_vdesc`

The GC uses a data structure called `nilfs_vdesc` to collect and process information about a particular virtual block in the file system. It holds information gathered directly from the segment summary block as well as the checkpoint numbers from the corresponding DAT entry and additional flags. It is not only used internally to determine if a block is dead or not, but also directly copied to kernel-space.

To support the various algorithms and the live block tracking I added two additional flags to the `nilfs_vdesc` structure:

NILFS_VDESC_SNAPSHOT_PROTECTED The block is protected by a snapshot. This information is used in the kernel as well as in the GC to calculate the number of live blocks in a given segment. A block with this flag is counted as alive regardless of other indicators.

NILFS_VDESC_PERIOD_PROTECTED The block is protected by the protection period of the user-space GC. The block is actually dead, but for the moment protected. So it has to be treated as if it were alive and moved to a new free segment, but it must not be counted as live by the kernel. This flag indicates to the kernel that this block should be counted as dead instead.

The original `nilfs_vdesc` structure already has a field `vd_flags`, but it is not usable for bit flags, and changing that would break backwards compatibility. Therefore I decided to re-purpose the padding field and rename it to `vd_blk_flags`.

Unfortunately older versions of the user-space tools do not initialize the padding field to zero. So it is necessary to signal to the kernel if the new `vd_blk_flags` field contains usable flags or just random data. Since the `vd_period` field is only used in user-space, and is guaranteed to contain a value that is greater than zero, it can be used to give the kernel a hint. So by setting `vd_period.p_start` to zero, the user-space tools can signal to the kernel, that they support extra bit flags and that it is safe to use the `vd_blk_flags` field.

These precautions are necessary, because it is very common to have a system with a more recent kernel and old user-space tools or vice versa. Table 5.2 shows all combinations of kernel and user-space versions and the resulting compatibility issues.

	Old Kernel	New Kernel
Old User-Space	No issues	`vd_period.p_start` $>$ 0 so `vd_blk_flags` is ignored
New User-Space	`vd_blk_flags` is ignored	`vd_period.p_start` $=$ 0 so `vd_blk_flags` is interpreted

Table 5.2: User-Space Utilities Compatibility Table

5.7.1 Extension of the **`SET_SUINFO`** Ioctl

As described in section 5.1 the GC can abort the cleaning of a segment, if the free space gained is very small compared to the cost of copying all the live blocks. An integral part of that feature is the `SET_SUINFO` ioctl, which allows the GC to update SUFILE entries directly without moving any blocks.

Because of the SUFILE extension described in sections 4.1 and 5.6, the `SET_SUINFO` ioctl has to be extended as well to support the newly added fields. Apart from that the actual values sent to the kernel via the ioctl have to be calculated in user-space.

To this end I added the `nilfs_count_nlive_blks()` fucntion to the GC. It simply loops through the `vdescv` and `bdescv` vectors and counts the live blocks belonging to a certain segment. Here the new `vdescv` flags introduced earlier in section 5.7 come in handy. If `SNAPSHOT_PROTECTED` flag is set, the block is always counted as alive. However, if it is not set and `PERIOD_PROTECTED` is set instead it is counted as dead.

If the kernel either doesn't support the `SET_SUINFO` ioctl or doesn't support the `set_nlive_blks` flag, it returns `ENOTTY` or `EINVAL` respectively and the corresponding options are disabled and not used again.

CHAPTER 6

Methodology

This chapter describes the benchmark tools that were developed for the purpose of testing the new segment selection policies.

6.1 Replay Tool for NFS Traces

As stated in section 4.7.1 there is currently no reliable replay tool for NFS traces. The tools that are available, were either designed to test NFS implementations or to analyze the traces. None seemed usable as a standalone benchmark utility. So I decided to write my own tool with automated benchmarks in mind. This section describes the implementation of this tool.

My primary design goals were to make it read the compressed text format of the traces directly, gather information about the file system on the fly without any pre-processing, replay the traces directly even without full path information, and gracefully handle errors and inconsistencies.

6.1.1 File Format

NFS traces do not capture file system operations, but the network packets exchanged between client and server. Every line in the text format of the trace directly corresponds to a raw network packet of the NFS protocol. As a result there are a lot of errors, inconsistencies, lost packets, aborted transactions, lost connections with clients, and clients that use old NFS protocol versions.

The file format is a simple human readable text format. Every line represents a network packet and contains multiple fields separated by a space character. The first eight fields have a fixed meaning. They are followed by a list of attributes, whereby the first token is the name and the second its corresponding value. The number of attributes depends on the kind of operation. Listing 6.1 shows the format of one line and listing 6.2 shows an example of a read operation.

Listing 6.1: NFS Trace File Format

```
<Timestamp> <SenderId> <ReceiverID> <Transport Protocol>
<ProtocolVersion> <TransactionID> <Opcode> <OperationName>
<Attributes>
```

Listing 6.2: NFS Trace Example Line

```
999316802.796958 31.03f2 30.0801 U C3 fcdc4183 6 read
fh 6189010057570100200000000051d72d2239de06d24400006189010057570100
off 59e000 count 2000
```

6.1.2 File Handle

File handles appear in the attribute list under various names (e.g. fh, fh2) depending on the operation. The read operation shown in listing 6.2 has one file handle with the attribute name “fh”. It is a 64-byte long, hex-encoded string that uniquely identifies an inode from a certain file system on a certain disk at a specific time. However, that file system is no longer available and the file handle on its own cannot be resolved to any particular file, directory or path.

It is also important to note that a file handle is only unique for a certain amount of time, because file systems tend to reuse inode numbers. So if for example a delete operation does not appear in the trace because of packet loss, another file with the same inode number and file handle can appear later and the replay tool cannot distinguish between the two. All the resulting conflicts and inconsistencies must be handled properly by the replay tool.

6.1.3 Reconstruction of the Original File System

Since the NFS protocol is stateful, the operations in the traces do not come with their full context, like the full path name of the target. Instead they reference the target of an operation by its file handle. As a result the replay tool has to reconstruct the state of the original file system, by extracting information from certain operations that occur in the traces.

For example, the `LOOKUP` operation can be used to discover the file name associated with a file handle. By using these operations from the trace, the replay tool is able to reconstruct the directory hierarchy of the original file system and create a mapping from file handles to paths. Nevertheless, some file handles reference files that do not appear in the traces at all, so the reconstruction is not perfect [ELMS03].

6.1.4 Pipeline Design

The replay tool uses a pipeline design, meaning there are no pre-processing steps for the trace files. The trace files are read in line by line, each line is parsed and immediately passed on to the next stage of the pipeline. The tool reconstructs the directory structure of the original file system on the fly and simply uses the file handle as a file name if no path information is available. This design is very similar to to the approach of Talwadker et al. [TV14], though much less sophisticated.

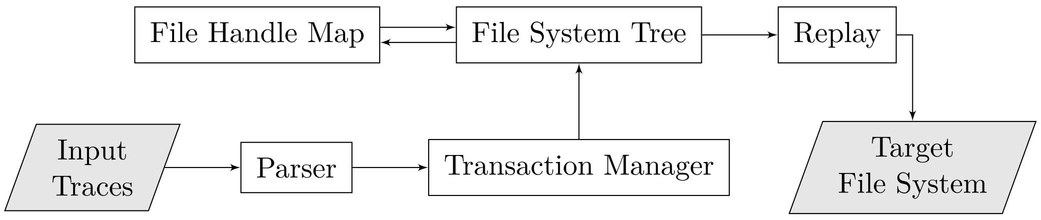


Figure 6.1: NFS Replay Tool Flow Chart

The pipeline consists of the following stages also shown in figure 6.1:

Parser The parser reads in the input file line by line and parses the packets into an internal data structure.

Transaction Manager Since the traces do not represent finished operations, but network packets, the replay tool has to track NFS transactions. A transaction consists of a request followed by a response with the same transaction id. The response has a status field that indicates if the operation was successful or not. Since multiple clients can be connected to one server at the same time, the transactions overlap and intersperse each other.

The transaction manager keeps track of the transactions that appear in the trace, matches the request and response packets, and forwards successful transactions to the next stage. It also has to discard unfinished transactions after a timeout period.

File System Tree An in-memory reconstruction of the original file system. It is expanded and contracted on the fly with each operation performed on it. It also keeps track of the state of every tree node. Every node contains a timestamp when it was last accessed and a flag indicating if it exists in the target file system. Not every operation is replayed on the target file system. For example the `LOOKUP` operation only provides path information and has no impact on the target file system.

File Handle Map As the name suggests it maps file handles to tree nodes in the file system tree. A file handle can map to multiple tree nodes, because in the original file system a single inode can be referenced by multiple directory entries.

GC Since both the file handle map and the file system tree have to be stored in memory, they are limited in size. Periodically a garbage collector iterates through all entries and removes the oldest ones.

Replay Performs Linux system calls on the target file system based on the NFS operation type.

6.1.5 Compression

The text format of the traces is highly compressible. Usually a compression ratio of 33 : 1 can be achieved with modern tools. The replay tool is designed to be a console application that accepts its input through `STDIN`. So the traces can be stored in a compressed format, decompressed on the fly, and piped to the replay tool on the console.

6.1.6 Acceleration

The traces are replayed as fast as possible. There is no throttling implemented in the replay tool. However, to make sure that the operations reach the disk and are not simply handled by the page cache, the target file system is synced periodically based on the timestamps in the traces. The user can specify the number of trace minutes between the forced syncs. This value does not refer to real time minutes, but the accelerated time measured by the timestamps in the traces.

6.1.7 Accuracy

Because of the accelerated replay and the artificially introduced file system syncs, the accuracy is not particularly good. As a result the replay tool is not well suited for accurate performance analysis of particular applications. But it is ideal for generating a realistic high throughput workload to compare the performance of the NILFS2 garbage collector.

6.2 Hot And Cold Access Simulator

The simulator is a simple command line tool written in C. It is designed to reproduce the bimodal distribution of segment utilizations reported by Ousterhout et al. [OD89] in the original LFS paper.

The simulation used by Ousterhout et al. can be divided into the following three steps. Firstly, it writes a certain number of small 4k files to disk until a target disk utilization is reached. Secondly, it overwrites these files using one of two random access patterns until the disk is full and no clean segments are left. Thirdly, it starts the cleaning process using different victim segment selection algorithms, and then repeats step two until the overall write cost stabilizes.

The following random access patterns were used:

uniform Every file is equally likely to be overwritten.

hot-cold The files are divided into two groups called *hot* and *cold*. 90% of the files are in the cold and 10% in the hot group. The file access is modeled in reverse, so that 10% of the time the cold group is accessed and 90% of the time the hot group. In short, 90% of writes go to 10% of the files. This simulates a simplified form of locality.

My simulator implementation behaves exactly like the original implemented by Ousterhout et al. The only difference is, that it does not control the GC and it does not measure the write cost to determine when to stop. Instead the GC is managed by a test script that controls both the simulator and the GC. The simulator is designed to stop without an error if the disk is full. So the test script allows the disk to fill up until the simulator terminates, then it activates the GC and waits until 20% of the disk is free again, stops the GC and restarts the simulation. This process can be repeated an arbitrary number of times, but after 24 iterations about 40% of the cold files should have been overwritten, which matches the results reported by Ousterhout et al for the original simulation.

6.3 Dbench

Dbench [dbe17] is a well known file system benchmark. It is designed to simulate typical network file system workloads and replay them on a target NFS server or directly on a target file system. The tool comes with a set of default workloads, but it is also possible to generate new workloads from recorded NFS traces. Dbench simulates multiple clients running in parallel. Each client executes a workload in an infinite loop until the simulation is stopped. The user can select among other options the workload, the number of clients and the runtime.

The difference between Dbench and my NFS replay tool discussed in section 6.1 is that Dbench cannot parse NFS traces in text format, but is designed to use a general purpose network capture format. Furthermore, it has to pre-process the traces into workload files before replaying them. Since the uncompressed traces are hundreds of gigabytes in size, only a small portion can be transformed into a workload file.

6.4 IOzone

IOzone [ioz17] is a file system benchmark designed to identify performance problems at certain file sizes and record lengths. To that end it measures the performance for all combinations of file and record size in a certain range and produces a three-dimensional performance map. On the X-axis is the file size on the Y-axis the record size and on the Z-axis the measured performance. The record length is simply the granularity at which the file is read or written.

IOzone gives very repeatable results, since, unlike Dbench (see section 6.3), it does not use any concurrency.

CHAPTER 7

Benchmark Results

7.1 Test System

All benchmarks were performed on an AMD Phenom II X6 1090T six core processor with a Samsung 850 EVO 250GB SSD and a conventional 1TB 7200RPM hard disk. To make the results comparable and to limit the run-time of the benchmarks, both drives were partitioned with a test partition of 100 GB. All benchmarks were run on these test partitions. In the following graphs and figures the hard disk is abbreviated to “Disk” and the SSD is called “SSD”.

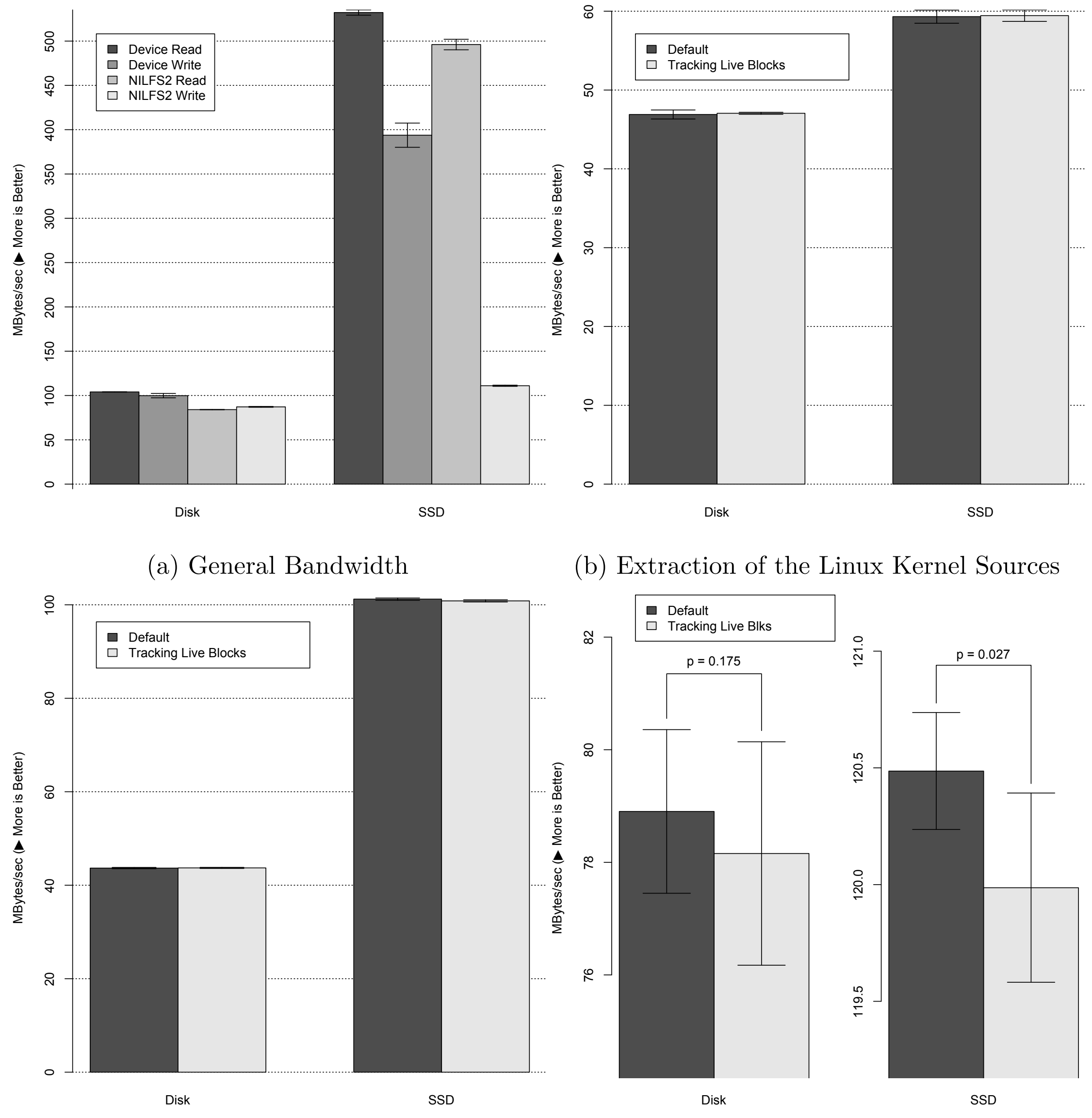
Figure 7.1a gives a general indication of the disk bandwidths the test system is capable of. These numbers can be used to put the other results in this chapter into perspective.

7.2 Measure Overhead

Since the SUFILE extension discussed in section 4.1 doubles the SUFILE entry size, and the live block tracking leads to significantly more updates to the SUFILE, the file system performance should decrease slightly. This overhead should be measurable with standard file system benchmarks.

Figure 7.1b shows a very simple benchmark, which uses a kernel source archive of a recent Linux kernel and extracts it on a clean NILFS2 file system volume. Although, there should be a very slight performance decrease due to the overhead of the live block tracking, the measurement is not accurate enough to show this, and the difference remains within the margin of error.

Figure 7.1c shows the results of a 10 minute Dbench run using the built-in `client.txt` workload and 6 worker threads. The value shown is a measurement Dbench calls “Throughput”, which is the total amount of data per second written to disk. There is a slight difference in the expected direction, but it is *not* statistically significant.



(a) General Bandwidth (b) Extraction of the Linux Kernel Sources

(c) Dbench Throughput (d) IOzone Bandwidth (32GB/1MB)

Figure 7.1: Disk and File System Bandwidth on the Test System

7.2.1 SUFILE Blocks Written

Although the throughput and disk bandwidth is hard to measure accurately enough to get a significant difference, the actual number of blocks written can be measured exactly.

Figure 7.2 shows the overhead of the live block tracking in blocks written to disk. Positive numbers indicate an overhead and negative numbers an improvement. The measurements were taken using the IOzone benchmark on a clean file system and the

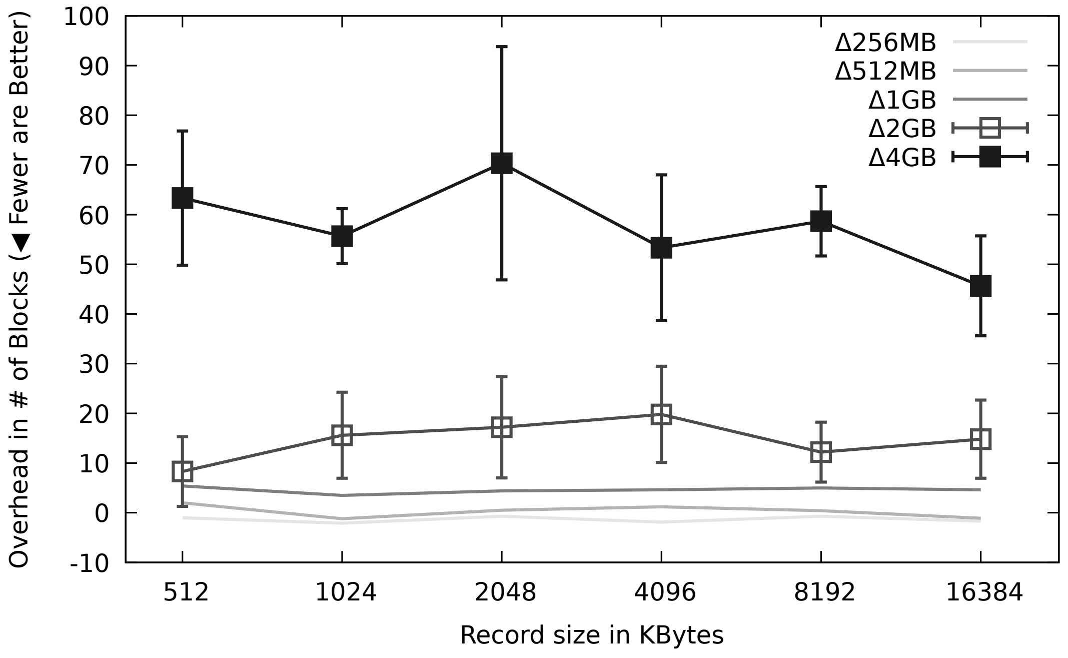


Figure 7.2: IOzone Written Blocks Overhead

`/proc/diskstats` file, which records among other things the number of blocks written to a block device.

For small files below 2 GBs in size, there was no measurable overhead. Surprisingly, the results for the 256MB file show a slight improvement, indicated by a dip below the zero line, rather than an overhead. This is probably due to the high variability of the results. After the mark of 2GB, however, a clear trend emerges. The overhead is independent of the record size and it grows disproportionately with the file size. A doubling in file size causes a six-fold increase of the overhead. The overhead is caused both by the bigger SUFILE entries and by the more frequent updates to the SUFILE. However, it is relatively small compared to the payload data and it is upwards bound by the size of the SUFILE.

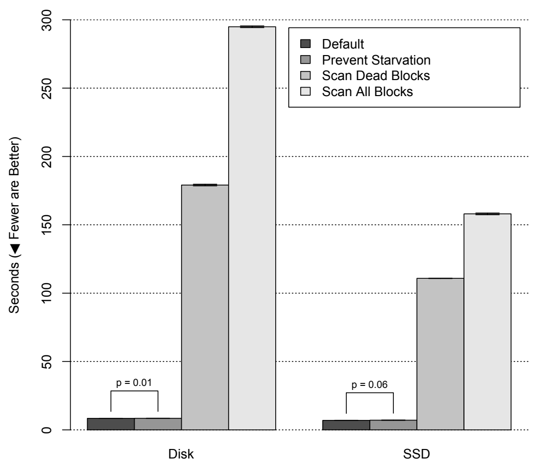
7.2.2 Bandwidth/Throughput

Since the IOzone benchmark gives very repeatable results and the overhead is higher for bigger files, I tried to measure the bandwidth overhead using a file size of 32 GBs and a record length of 1MB. Figure 7.1d shows the results. There seems to be a small difference, but the results are not statistically significant.

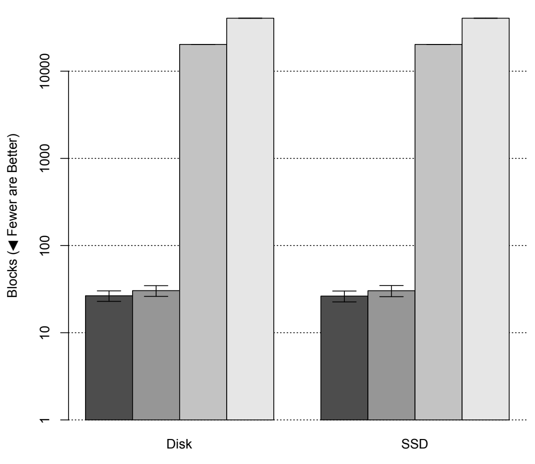
7.3 Snapshot Creation/Deletion Cost

As described in section 4.5, I have implemented three different algorithms to handle the tracking of snapshots, all of which add additional costs at snapshot creation and deletion time. To measure the overhead of each algorithm I created a simple benchmark script, which performs the following steps:

1. Create a 5 GB test file on a clean NILFS2 volume
2. Create another 5 GB test file
3. Create a snapshot to protect the files
4. Delete the second test file, which is protected by the snapshot
5. Turn the snapshot into a checkpoint and back again
6. Repeat step 5 in a loop



(a) Creation Time for 100 Snapshots



(b) Snapshot Size in Blocks

Figure 7.3: Snapshot Overhead

The first test file simulates 5 GB worth of live blocks, and the second test file the same number of dead blocks. The loop runs for 100 iterations and figure 7.3a shows the results:

Prevent Starvation The first algorithm simply scans the comparatively small SUFILE for starving segments. As a result it performs nearly as well as the original algorithm used by the stock NILFS2. In fact the difference is so small that it lies within the margin of error and is therefore not statistically significant.

Scan Dead Blocks The second algorithm has to scan the whole DATFILE every time a snapshot is created or deleted. As a result the performance suffers significantly compared to the first two methods. Creating and deleting 100 snapshots takes more than 170 seconds on a hard disk, which is over 21 times worse than both the stock and starvation algorithm. However, a snapshot creation time of 1.7 seconds is probably still acceptable for most users, because snapshots are usually created manually and rarely.

Scan All Blocks The third algorithm also scans the whole DATFILE, but compared to the second algorithm it has to write out the modified DAT entries for live as well as dead blocks. Consequently, the performance is significantly worse. 100 snapshots take over 290 seconds.

Figure 7.3b shows the cost of creating a single snapshot measured in file system blocks. It uses a logarithmic scale, because the stock and starvation algorithms wouldn't be visible otherwise. While the stock and starvation snapshot creation write only a few blocks, the scanning algorithms write three orders of magnitude more to disk. The snapshot creation/deletion cost scales linearly with the file system size. So a bigger file system entails a proportionally bigger cost, which could cause problems with very large file systems on comparatively slow disks.

It should be noted that this benchmark represents an extreme case. Usually the first snapshot causes higher costs, because it has to modify almost all DAT entries in the file system. Subsequent snapshots would only have to modify the DAT entries not covered by the first snapshot.

7.3.1 Page Cache Memory Usage

Figure 7.4 shows the size of the page cache in MBs during the snapshot overhead benchmark. The page cache was flushed before the benchmark was run, and the benchmark itself only creates and deletes snapshots. As a result, the page cache contained mostly the meta-data files of the NILFS2 file system and should therefore be a good indicator of the memory usage of the various algorithms under test.

The Prevent Starvation algorithm increases memory usage only slightly compared to the unmodified default implementation. Although, it has to scan the whole SUFILE every time a snapshot is deleted, the impact on the memory usage is quite small. There are two reasons for this. Firstly, the garbage collector also has to scan the whole SUFILE regularly, so that it resides in the page cache anyway, and secondly, the SUFILE is quite small. Moreover, the SUFILE does not have to fit into main memory for the algorithm to work, because it only scans 32768 SUFILE entries at a time before it releases the transaction lock. By releasing the transaction lock every 32768 entries, it gives the file system a chance to write out dirty pages, which in turn allows the Linux kernel to free these pages. Consequently, the Prevent Starvation algorithm scales very well in terms of memory usage.

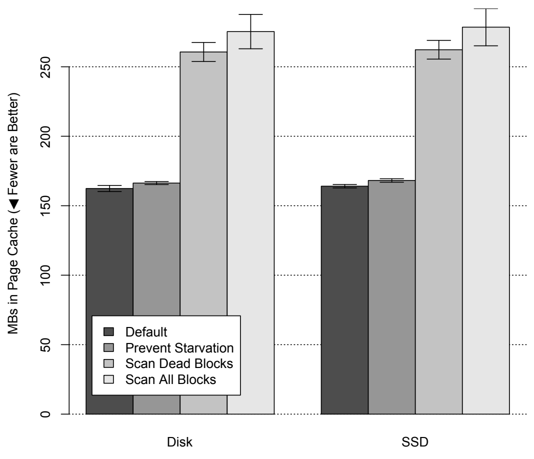


Figure 7.4: Page Cache Size in MBs

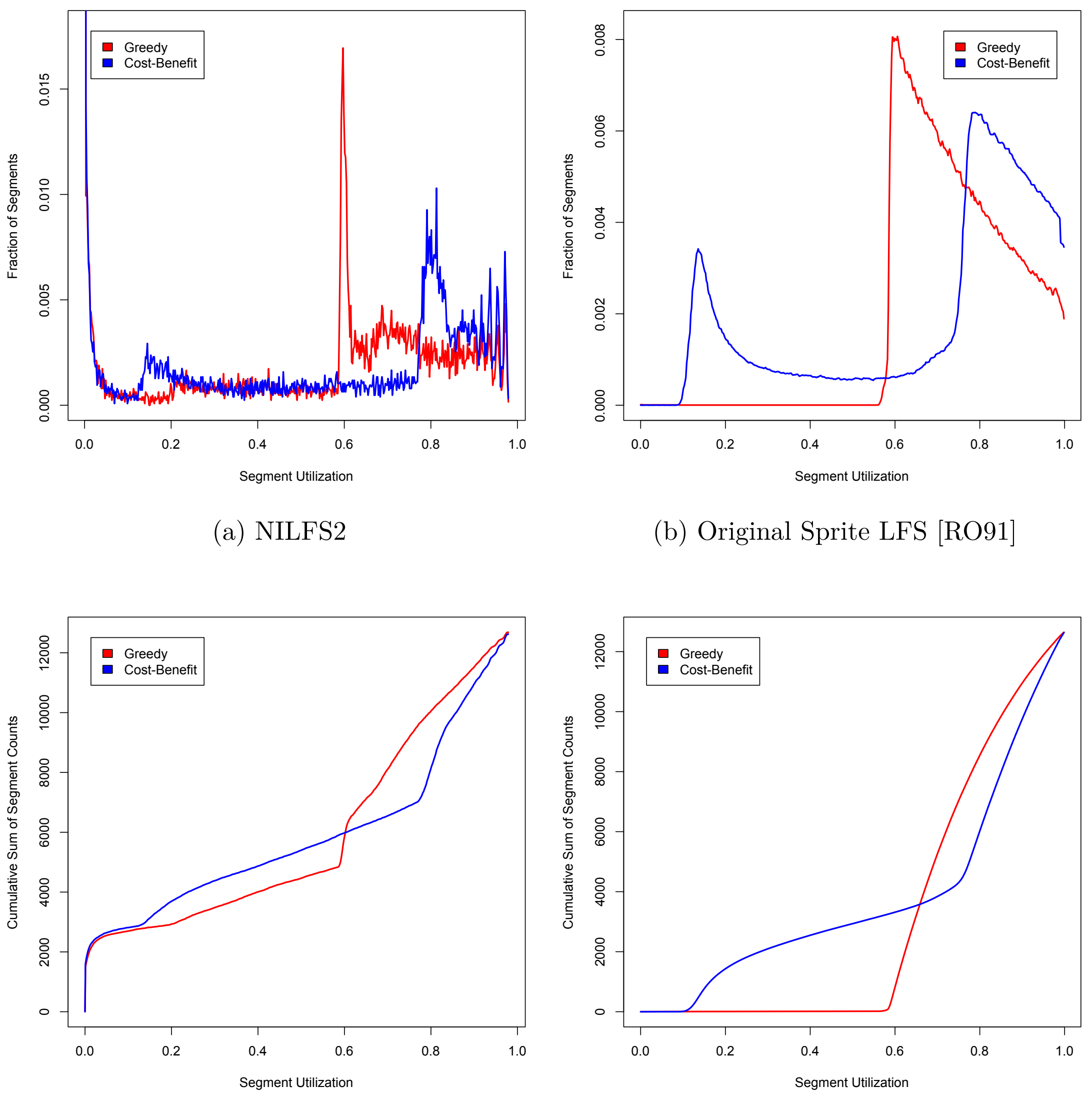
The Scan Dead Blocks and Scan All Blocks algorithms, however, use substantially more memory, because they have to scan and modify the whole DATFILE, which is much bigger than the SUFILE. The DATFILE contains one entry for every block in the file system. So for a 100 GB NILFS2 volume with a block size of 4 KB it can grow up to 800 MB. This value grows linearly with the size of the volume, so a 1 TB volume has a DATFILE of 8 GB:

$$\frac{Volume\ Size}{Block\ Size} \times Entry\ Size = \frac{100\ GB}{4\ KB} \times 32\ Bytes = 800\ MB \tag{7.1}$$

In contrast to the Prevent Starvation algorithm, the transaction lock cannot be released during the scan of the DATFILE. So in the worst case scenario the whole DATFILE has to fit into memory or the algorithms will fail. However, this is only a flaw in the current implementation and not in the algorithm itself. With some effort, it would be entirely possible to implement a version of these algorithms that doesn't suffer from this design flaw. It would have to perform the scan asynchronously in the background and persistently store intermediate results when it releases the transaction lock. This way the scan can be safely resumed at a later time, even in the event of a crash.

7.4 Reproduce Bimodal Distribution

One of the goals of this thesis was to implement the Cost-Benefit segment selection policy and use it to reproduce the bimodal distribution reported by Rosenblum et al. [RO91] in the original LFS paper. To that end, I have developed a hot and cold data simulator similar to the one used by Rosenblum et al. The implementation of the simulator is described in section 6.2.



(a) NILFS2 (b) Original Sprite LFS [RO91]

(c) NILFS2 (Cumulative Sum) (d) Original Sprite LFS (Cumulative Sum)

Figure 7.5: Bimodal Segment Distribution

Figure 7.5a shows the distribution of the segment utilization for the Greedy and Cost-Benefit policies in NILFS2 after 24 hot-cold simulator runs. Every run consists of two phases, namely the simulation of the write pattern and the cleaning of the GC. The two phases always follow each other and never overlap. The simulator is designed to create a hot and cold access pattern, so that 90% of the writes go to 10% of the files. This benchmark is very similar to the one used by Rosenblum et al. [RO91] for the Sprite LFS file system, whose results are shown in figure 7.5b.

While the Greedy policy produces a segment distribution where most of the segments have an utilization of around 60%, the Cost-Benefit policy achieves a much better bimodal distribution. The best-case scenario for a log-structured file system is when most segments are either completely full or completely empty and no free space is tied up in partially filled segments. A good bimodal distribution approaches this best-case scenario. On the other hand, the worst-case scenario is when all segments are exactly 50% full, because then the maximum amount of free space is tied up and unusable to the file system.

The results clearly show, that the Greedy policy approaches the worst-case and the Cost-Benefit policy the best-case scenario.

Due to the less controlled environment of a real file system implementation with lots of meta-data overhead and a user-space cleaner with a wide range of options, the results in figure 7.5a are not as clear compared to the ones of Rosenblum et al. [RO91] shown in figure 7.5b. Nevertheless, the results of Rosenblum et al. are clearly reproducible with a modern file system and their conclusions remain valid.

The data used to draw the graph in figure 7.5b were extracted from a histogram shown in the original paper by Rosenblum et al. [RO91]. Although the paper doesn't say how the graph was generated, it is possible to deduce from the area under the curves, that they used a histogram with about 512 bins. So to make my results comparable with this data, I also used 512 bins for all segment utilization histograms in this thesis.

Figures 7.5c and 7.5d show the same data as figures 7.5a and 7.5b, but as a cumulative sum of the number of segments in each bin of the histogram. This results in a much cleaner and easier to interpret graph, whereby the ideal bimodal distribution rises quickly for the low segment utilization values, indicating a high number of empty segments, then plateaus off, indicating very few partially filled segments, and finally rises again at the end, indicating a high number of full segments. In other words, the steeper the curve rises at the start and the end, and the flatter the plateau is in the middle, the better it resembles an ideal bimodal distribution. The total number of segments in the NILFS2 benchmark was 12624. Unfortunately, this number is not available for the historical data shown in figure 7.5d, so I simply used the same value to make the two graphs comparable.

7.5 Selection Policy Evaluation

In this section I evaluate the newly added segment selection policies, Greedy and Cost-Benefit, and compare them to the stock Timestamp policy. Furthermore, I try to reproduce the existing results of Rosenblum et al. [RO91], by comparing the Greedy and Cost-Benefit policies against each other. All of the benchmarks presented here were run on a clean NILFS2 file system on a 100 GB partition filled with 20% static data. The GC was configured with the default NILFS2 settings, except for the cleaning interval, which had to be shortened to allow the GC to keep up with the benchmarks.

All benchmarks in this section were performed on the hard disk as well as the SSD and the results were mostly equivalent. However, since the SSD is much faster, the effect of the selection policy was less pronounced. So if both graphs show the same thing, I chose the hard disk graph.

7.5.1 Garbage Collector Efficiency Graph

To compare different segment selection policies, it is necessary to measure the quality of any particular selection. The goal of any selection policy is to select the segments with the fewest number of live blocks, because that way less blocks have to be copied and more free space is gained. To approximate that notion of quality, I devised a measure I call “garbage collector efficiency”. It can be calculated in the following way:

Every time the GC cleans a segment, it logs the number of live blocks it had to copy. This value can be used to calculate the efficiency of the collection process. The fewer blocks have to be copied, the more efficient it is. For example, if the GC cleans 4 segments with a total of 8188 blocks and there are only 200 live blocks that have to be copied to a new location, the resulting efficiency is $(1 - \frac{200}{8188}) * 100 = 97.56\%$.

For example, figure 7.6 plots the GC efficiency over time to compare the different policies. The vertical, dashed lines mark the time when the benchmark terminated. After that the GC keeps working, until it reaches its goal of 20% free space in the file system.

7.5.2 Synthetic Benchmarks

I chose the synthetic benchmarks Dbench [dbe17], described in section 6.3, and Fsmark [fsm17] to evaluate the selection policies. Unfortunately, both proved to be unsuitable for the task, because they were designed to measure the performance of a file system under high load and high concurrency conditions. They were not designed to produce a realistically aged file system.

For example, figure 7.6 shows the GC efficiency for the Fsmark benchmark. Fsmark generates a uniform segment utilization of about 60% across all segments. If all segments have about the same utilization, then they are equally good candidates for cleaning and the selection policies are irrelevant. The only reason why the Timestamp policy is slower, is because it has to work its way through the 20% of static data, which the others can avoid. So Fsmark is completely unsuitable to benchmark GC selection policies.

The results for Dbench shown in figure 7.7 are even worse, because it uses only very short workloads and cleans up after itself. As a result the segment utilization is almost 0 for all segments, and any segment selection is equally as good as any other. The Timestamp policy has a few dips in performance, because it regularly encounters the 20% of static data, which the other policies can avoid.

However, Dbench has a lot of options and can use pre-processed NFS traces. It might be possible to use it to age a file system and evaluate the selection policies, but I did not

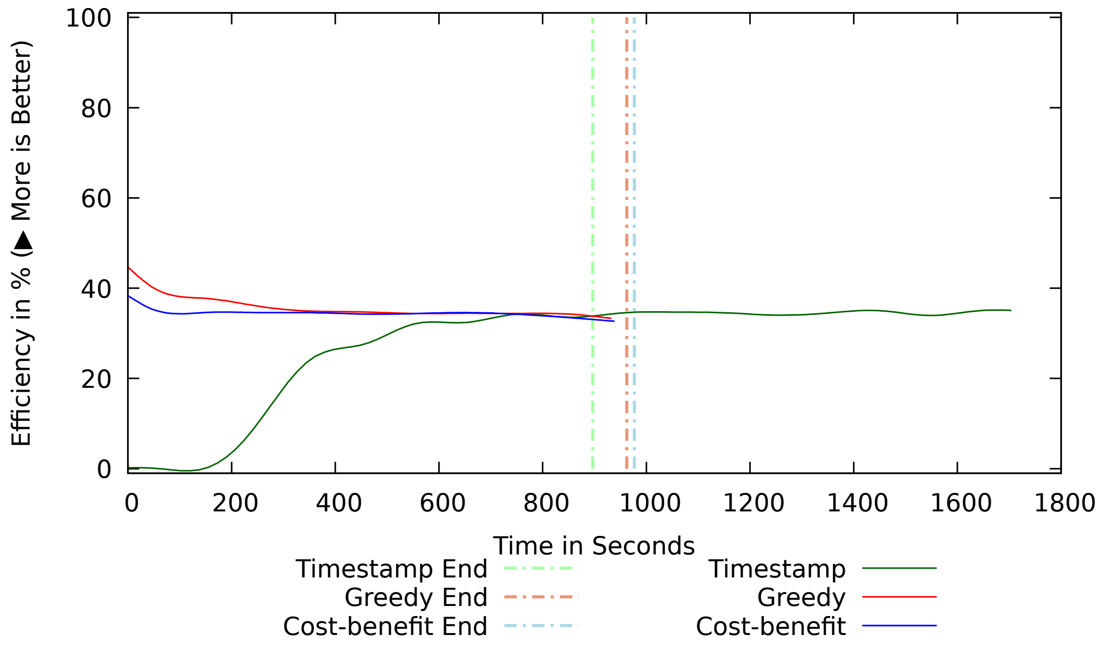


Figure 7.6: Fsmark GC Efficiency with 20% Static Data (Disk)

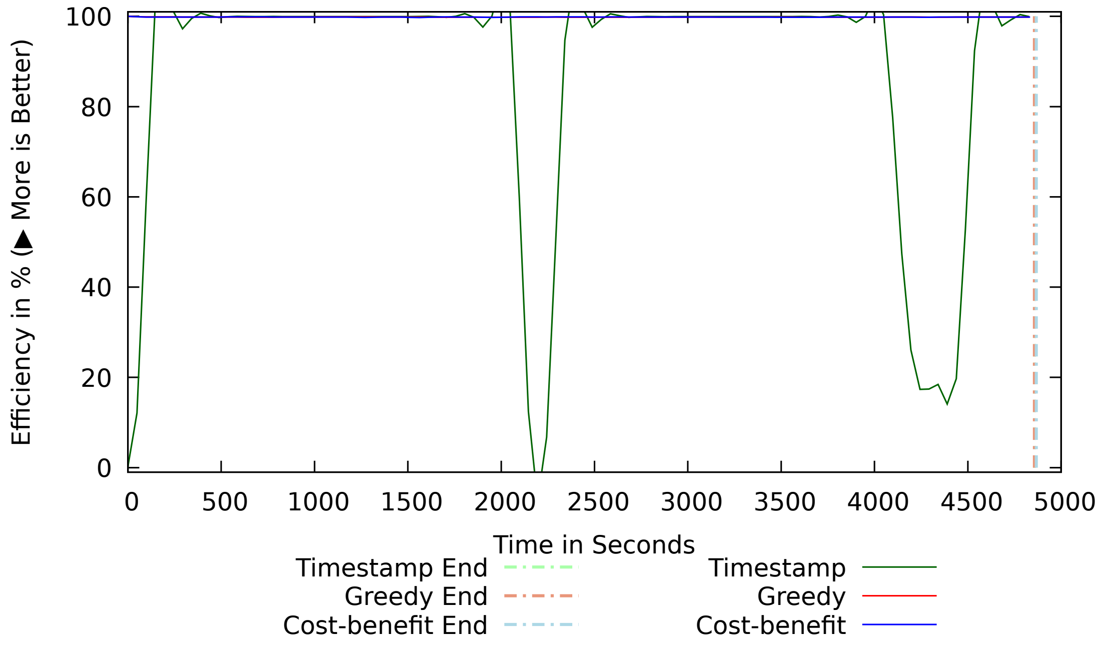


Figure 7.7: Dbench GC Efficiency with 20% Static Data (Disk)

explore this route in this thesis, because the Replay benchmark described in section 7.5.3 was good enough.

7.5.3 Replay Benchmark

The replay benchmark is a shell script that uses the NFS replay tool, discussed in section 6.1, to replay a NFS trace on a 100 GB test partition.

Trace Characteristics

The trace used is a freely available, historic NFS trace from the IOTTA repository [Rep11] called "lair62". It is from the Harvard SOS trace set recorded in the year 2001 on a Harvard NFS server. Although it contains several months worth of file system operations, it can be replayed in a few hours on modern hardware. The trace is quite old, but it is small enough to be replayed in a few hours and the resulting files fit neatly into the 100 GB test partition.

The "lair62" trace, like all other traces from the IOTTA repository, was anonymized before publication to protect the personal data of the people using the traced machine. However, some well-known file and directory names were kept in the clear, because they don't contain any personal information and they greatly help with the analysis of the traces. Using these well-known names it is possible to guess the function of some of the directories. For example, a directory containing files like `.Xauthority` or directories like `.ssh`, `.cache`, or `.gnome` is very likely a home-directory on a Unix machine.

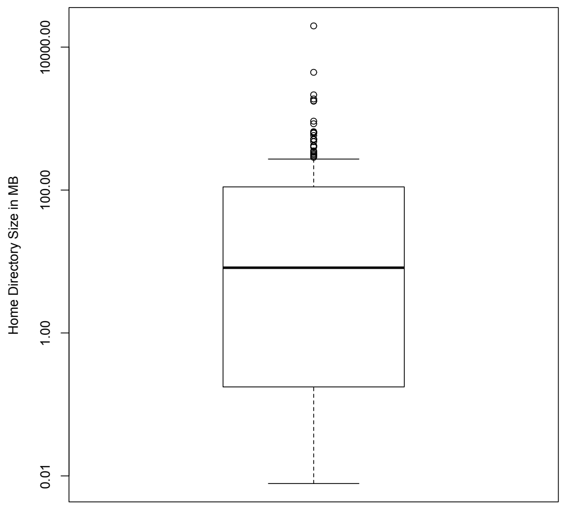


Figure 7.8: Home Directory Sizes of the "lair62" Trace

Using the method above I was able to determine that the "lair62" trace contains the home directories of about 188 users. The home directories have an average size of 257 MB, the median is 8.2 MB, and the total size of all files contained in the trace is 47.3 GB. Most users accessed their home directory very rarely, so that most of the traces come from only a hand full of users. Figure 7.8 shows a boxplot of the home directory sizes on a logarithmic scale. There are quite a few outliers, who contribute most of the data.

NFS Operation	Number Performed
Remove	5, 090, 646
Link	450, 681
Lookup	138, 545, 846
Rename	452, 322
Write	80, 006, 160
Create	4, 446, 401

Table 7.1: Trace Statistics for “lair62”

The average file size is 249.3 KB, but the median is 3.7 KB, so most files, apart from a few big outliers, are very small. Table 7.1 shows the statistics collected by the replay tool.

Shell Script

The shell script performs several preparation and data collection steps. The following list shows the steps in the order they are executed:

1. The NILFS2 volume is filled up with a 20 GB dummy file that is intended to simulate static, rarely changing data, such as archives, images or video files. So 20% of the data on the partition is static and never changes.
2. The replay tool runs on the partition.
3. After the actual tests have terminated, the benchmark waits until the garbage collector reclaims 20% of the free space in the file system. This step ensures a fair comparison between the selection policies, because the file system is in a defined state when the benchmark finishes.

Results

Figure 7.9 shows the results for the Greedy, Cost-Benefit and Timestamp policy. The benchmark was run three times under the exact same conditions, with the only difference being the selection policy.

The Timestamp policy consistently performs worse than the other two policies. Since it always selects the oldest segment regardless of the number of live blocks, it has to move more blocks around, which reduces its efficiency, clogs the bandwidth of the disk, and slows down the benchmark. Compared to the other policies the run-time is almost double.

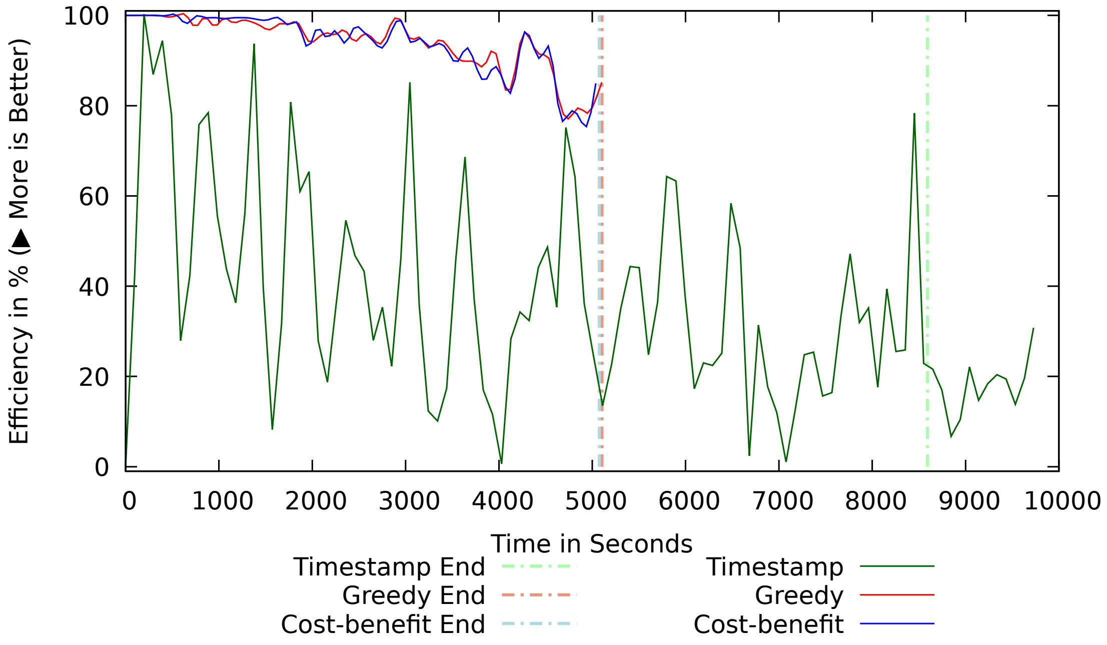


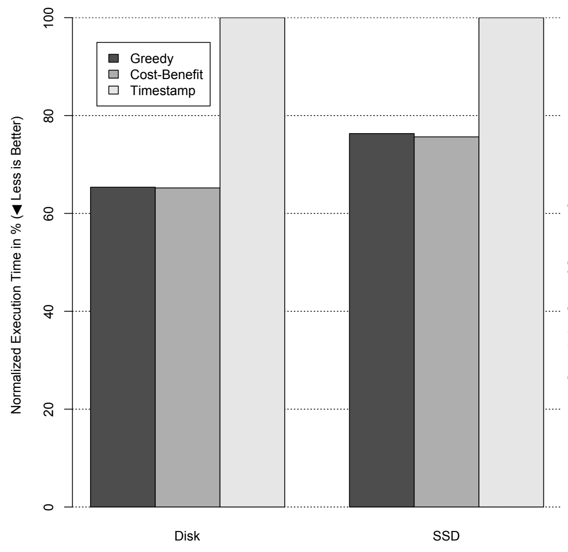
Figure 7.9: NFS Replay GC Efficiency with 20% Static Data (Disk)

In addition to that, it cannot avoid the segments where the static data is stored, because during the benchmark they repeatedly become the oldest segments and are therefore selected to be cleaned. These segments are almost full and moving them costs a lot of disk bandwidth while almost no free space is gained.

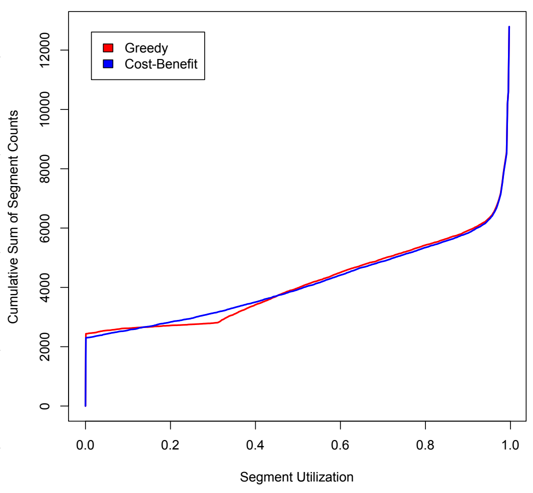
Fortunately, the GC can abort the cleaning of these segments, once it is clear that very little free space can be gained from it. This wastes a lot of resources reading the segment summary blocks and gathering the necessary information, but it is better than moving a full segment. Although I implemented the late abort feature in the course of this thesis, it is not part of the selection policy evaluation, because it has been accepted into the mainline kernel and is enabled by default in NILFS2. Without it the performance of the Timestamp policy would be even worse.

The performance of Greedy and Cost-Benefit is very similar, but it seems the latter is slightly better. In contrast to the Timestamp policy, Greedy and Cost-Benefit can take the number of live blocks into account, which enables them to avoid the static data segments entirely, and to make much better selections overall. The result is a higher efficiency and a much faster run-time. The extra cost of tracking the live blocks and the bigger SUFILE entry size is more than compensated for by the much more efficient GC.

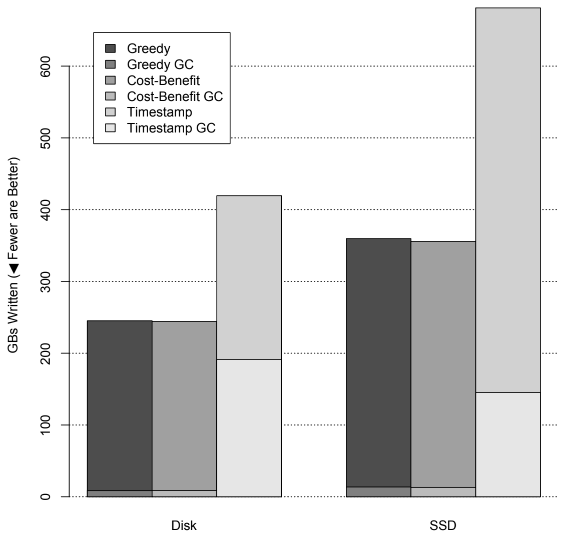
Figure 7.10a shows the normalized execution time of the replay benchmark in percent. Both Greedy and Cost-Benefit are compared relative to the Timestamp policy, which is set to 100%. The performance improved by a factor of 1.53 on the hard disk and 1.32 on



(a) Normalized Execution Time



(b) Segment Utilization (Disk)



(c) Total Data Written

Figure 7.10: NFS Replay Statistics

the SSD compared to the Timestamp policy. In the best case the Cost-Benefit policy is 0.67% better than Greedy, which is not a significant difference.

This result was unexpected, because the Cost-Benefit policy produced a much better bimodal distribution than the Greedy policy using the hot and cold data simulation described in section 7.4. In fact figure 7.10b shows an almost perfect bimodal distribution for both policies. Furthermore, both policies showed a consistently high efficiency of over

90% for most of the benchmark (see figure 7.9). This implies that the benchmark did not stress the GC enough to distinguish between the two policies. In other words, there were always enough almost empty segments for the policies to choose from. It is very difficult to find a benchmark that works with the comparatively slow Timestamp policy and challenges the other policies at the same time.

Figure 7.10c shows the total amount of data written to disk, and the percentage of the total that was written by the GC as a stacked bar graph. Using the Greedy and Cost-Benefit policies, the GC took up between 3.5% to 3.8% of the total amount written to disk. On the other hand, the Timestamp policy used 45.6% with the hard disk and 21.3% with the SSD. In other words, the GC with the Timestamp policy used up between a quarter and half of the disk bandwidth moving blocks around. This bandwidth is not available for the actual benchmark, which explains the longer run-time shown in figure 7.10a.

Figure 7.10c also shows, that the SSD had much more data written to it than the hard disk. This seems to be an artefact of the way the NFS replay benchmark works. Since the file system is synchronized periodically and not after every single operation, the much faster SSD can write out almost all individual changes, while the slower hard disk accumulates more of the write operations in the page cache. For example, if there are 10 write operations on a single block, then the SSD is fast enough to write out 10 blocks, while the hard disk is so slow that the 10 write operations hit the page cache and the block is written out only once.

7.5.4 Replay Benchmark With Less Free Space

Since the Timestamp policy ignores the number of live blocks and simply cleans the oldest segment, it can cause bottlenecks during the benchmark. Although it is only 34% slower on average, the performance varies hugely over the course of the benchmark. Sometimes it isn't able to free up enough space in time, which terminates the replay tool with a `ENOSPC` error. To compensate for this variability, a large enough reservoir of free segments has to be available.

So to compare the Timestamp policy with the other policies, the benchmark discussed in section 7.5.3 had to use a larger test partition than would have been necessary for the amount of data written by the replay tool. This allowed the Timestamp policy to run successfully, but it did not stress the Greedy and Cost-Benefit policies enough to cause a significant difference between the two.

This benchmark reduces the partition size to 58 GB and uses no static data. So while the previous replay benchmark had $100 - 20 - 47.3 = 32.7$ GB of extra free space, this benchmark works with only $58 - 47.3 = 10.7$ GB. As a result the GC gets executed sooner and the selection policies have fewer easy choices.

Figure 7.11 shows the results for both the SSD and the hard disk in one graph. In the first half of the benchmark there are lots of easy choices for the selection policies, so

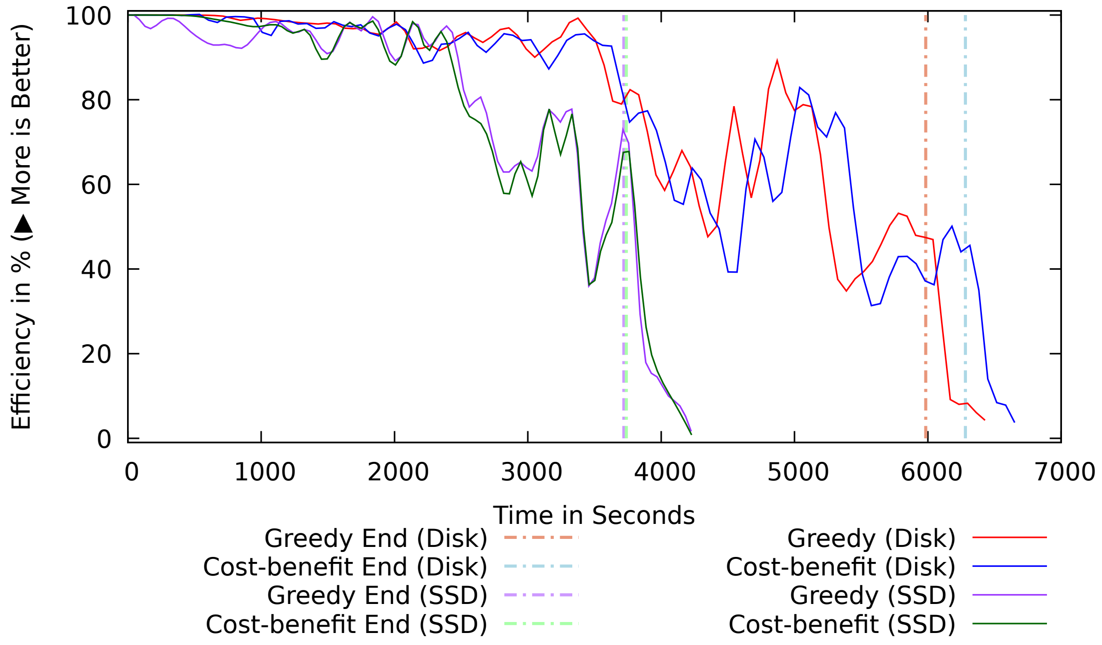
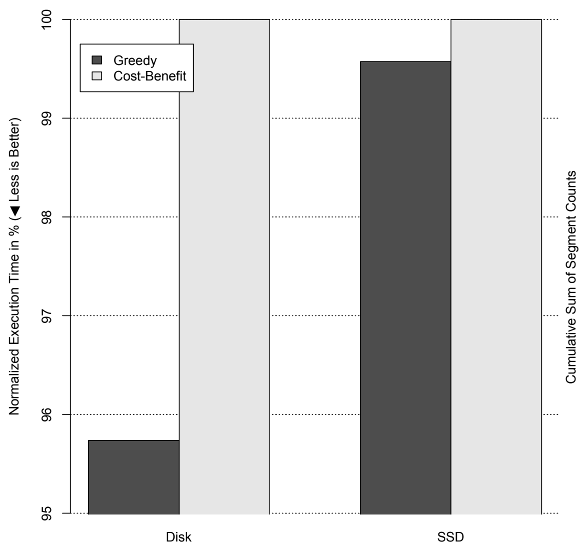
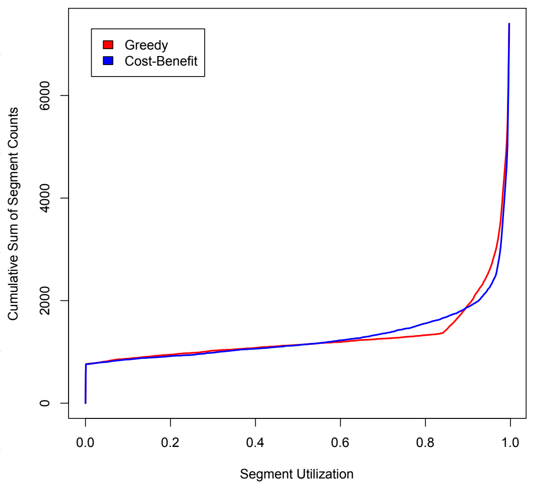


Figure 7.11: NFS Replay GC Efficiency on 58 GB partition



(a) Normalized Execution Time



(b) Segment Utilization (Disk)

Figure 7.12: NFS Replay Statistics on 58 GB partition

the efficiency is consistently above 90%. As the file system fills up, the efficiency drops dramatically, because the segments have to be cleaned sooner and more often to create the free space needed for new files.

The Greedy policy seems to have a clear advantage over Cost-Benefit in this scenario. It is almost 5% faster on the hard disk, despite having a less optimal segment distribution shown in figure 7.12b. It seems like Greedy performs better, when the file system is nearly full and there is very little free space available. In that case the segments have to be cleaned frequently and they don't have time to age. As a result, all segments contain hot data and their relative age is irrelevant. The assumptions made by the cost-benefit analysis do not apply any more and it is more efficient to use a greedy algorithm.

The excessive garbage collection overhead in situations with high disk utilization was also noted by Seltzer et al. [SBMS93] during the testing of their log-structured file system for UNIX. They later expanded on that finding in another paper [SSB+95], which seems to have been the motivation for the development of the *hole-plugging* algorithm proposed by Matthews et al. [MRC+97]. Jambor et al. [JHT+07] also found high garbage collection overhead during the evaluation of their implementation of LFS for Linux. This was especially true for uniform random write operations under high disk utilization conditions.

However, they did not compare the Greedy against the Cost-Benefit policy in this scenario, because only the Cost-Benefit policy was available in their implementations. A selection policy that takes the file system utilization into account and switches from a cost-benefit to a greedy algorithm at a certain threshold should be able to outperform both the Greedy and Cost-Benefit policies.

7.6 Evaluation of Live Block Tracking in the Presence of Snapshots

This section evaluates the three different snapshot tracking algorithms discussed in section 4.5. For this purpose I modified the replay benchmark described in section 7.5.3 by adding a shell script that creates and removes randomly selected snapshots.

7.6.1 Parallel Replay Benchmark

The parallel replay benchmark is identical to the replay benchmark, with the only difference, that a snapshot script is executed in a sub-shell in parallel.

Every five minutes the snapshot script selects a checkpoint and turns it into a snapshot. Additionally, it keeps a list of the last three snapshots it created and turns the oldest one back into a checkpoint. So at any time during the benchmark there are three active snapshots and they change every five minutes. The idea behind this script is, to simulate a user creating and deleting snapshots for backup purposes.

7.6.2 Prevent Starvation

Figure 7.13 shows the GC efficiency results for the Prevent-Starvation algorithm run on the hard disk. The results for the SSD are very similar. The efficiency is about 10%

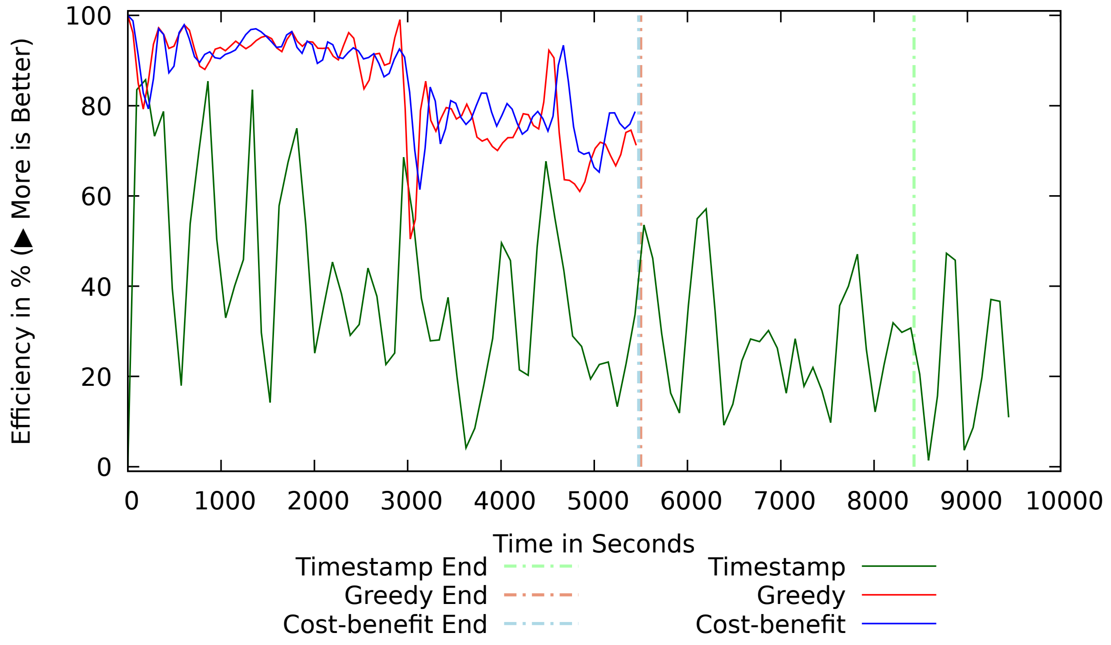


Figure 7.13: NFS Replay Parallel GC Efficiency for Prevent-Starvation (Disk)

lower than the replay benchmark without snapshots discussed in section 7.5.3, and the run-time is about 10% longer. The Cost-Benefit policy performs noticeably better than the Greedy policy.

7.6.3 Scan Dead Blocks

Figure 7.14 shows the results for the Scan-Dead-Blocks algorithm. Both the run-time and the efficiency come very close to the reference results without any snapshots. The Cost-Benefit policy performs slightly better than the Greedy policy, and the Timestamp policy seems to be unaffected by the snapshots.

7.6.4 Scan All Blocks

Figure 7.15 shows the results for the Scan-All-Blocks algorithm. Although the variability of the efficiency is higher compared to the Scan-Dead-Blocks algorithm, the run-time is very similar. Surprisingly, the Greedy policy is clearly better than the Cost-Benefit policy.

The reason for this could be the much higher number of meta-data updates generated by this algorithm. As discussed in section 4.1, the SUFILE extension includes the field `su_nlive_lastmod`, which contains a timestamp indicating the last time the live block counter was updated. The Cost-Benefit selection policy uses this timestamp to avoid recently updated segments. The reason for this is the protection period, which could

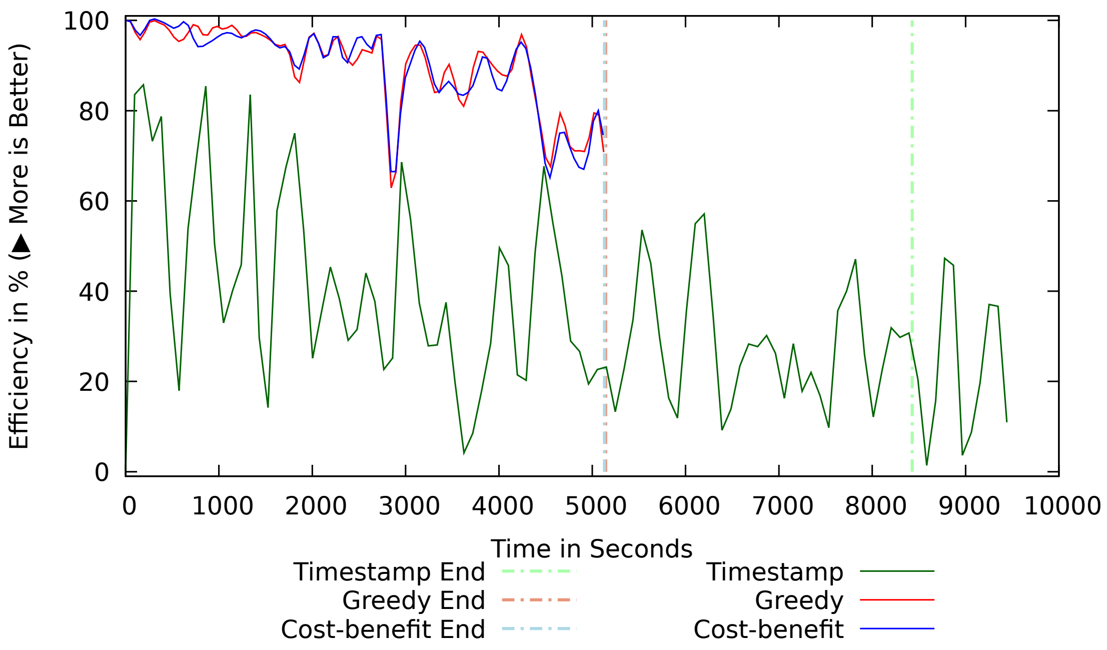


Figure 7.14: NFS Replay Parallel GC Efficiency for Scan-Dead-Blocks (Disk)

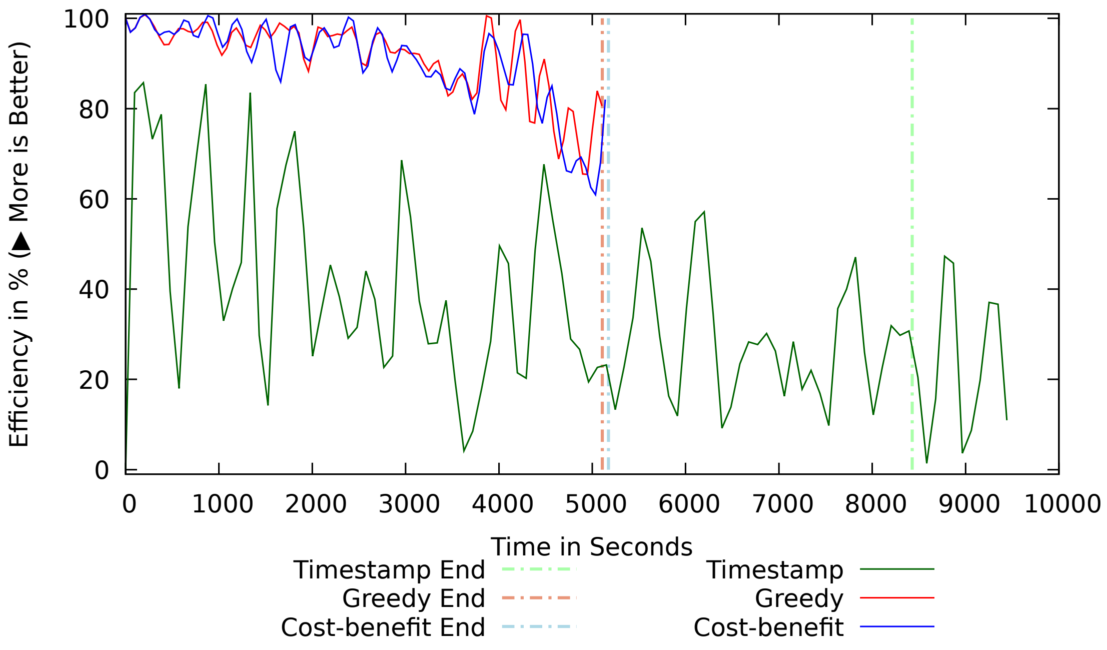


Figure 7.15: NFS Replay Parallel GC Efficiency for Scan-All-Blocks (Disk)

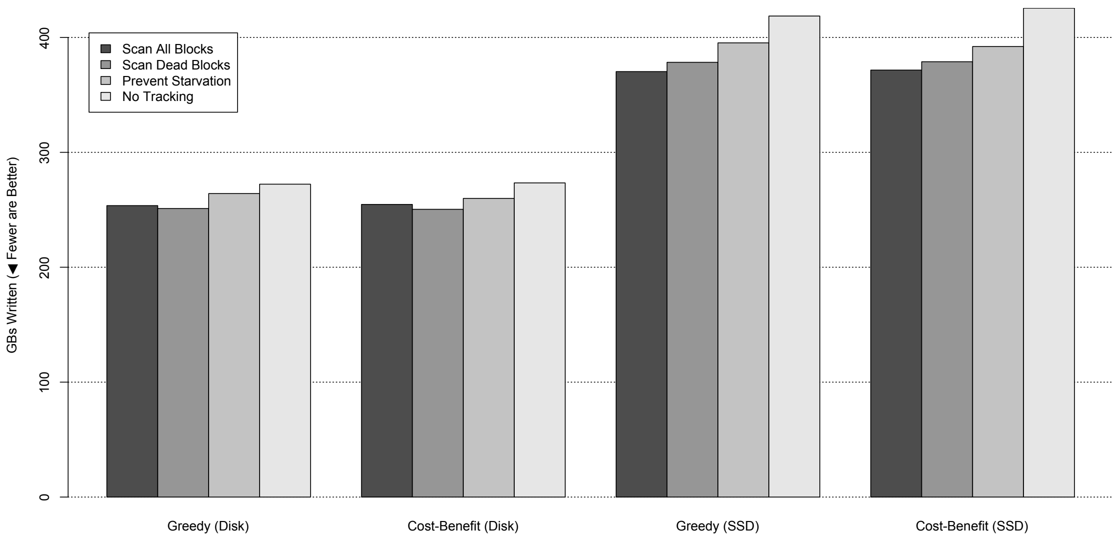
potentially protect a large number of blocks in those segments. Since the DAT file is spread across the whole partition, and the Scan-All-Blocks algorithm potentially has to update the whole DAT file, a higher proportion of segments will have a recent timestamp. This in turn could exclude good selection candidates for the Cost-Benefit policy. However, the `su_nlive_lastmod` timestamp does not have to be updated for the DAT file, because DAT file blocks are not protected by the protection period. So this could be fixed in a future implementation.

7.6.5 Algorithm Comparison

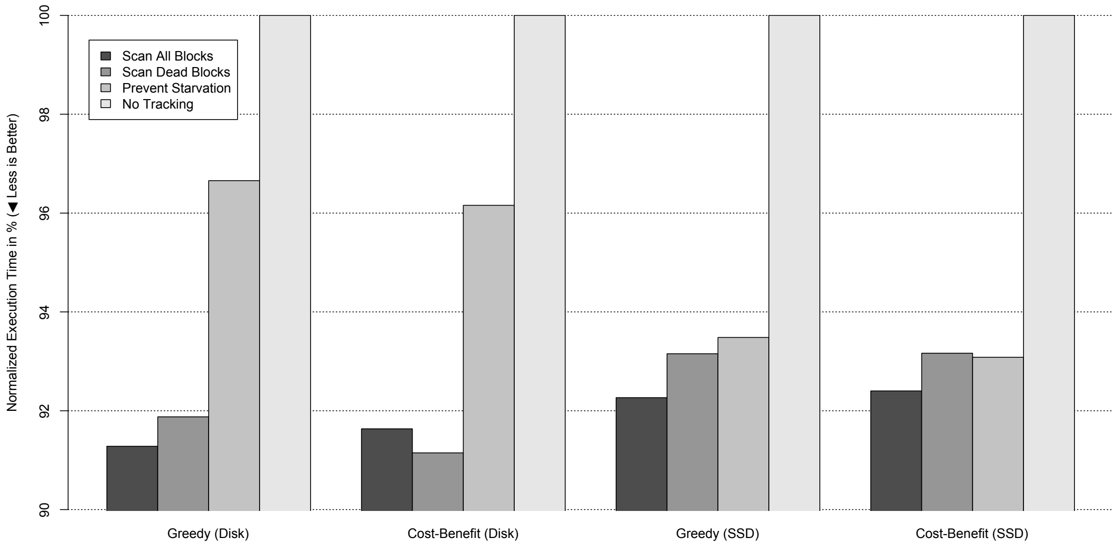
Figure 7.16a shows the data written to disk for all algorithms on the hard disk as well as the SSD. Despite the large snapshot creation/deletion costs discussed in section 7.3, the Scan-Dead-Blocks and Scan-All-Blocks algorithms write less data to disk. It seems like the better GC performance due to the the higher live block tracking accuracy, more than compensates for the higher snapshot creation/deletion costs.

Figure 7.16b shows the normalized execution time in percent, whereby the highest value for the disk and SSD was set to 100%. Using no snapshot tracking algorithm had the worst performance across the board. The Prevent-Starvation algorithm comes in second being about $4 - 7\%$ better than doing nothing. Scan-All-Blocks algorithm seems to be better than the Scan-Dead-Blocks algorithm, but only by a fraction of one percent. It may be necessary to redesign the benchmark to distinguish between those two algorithms better.

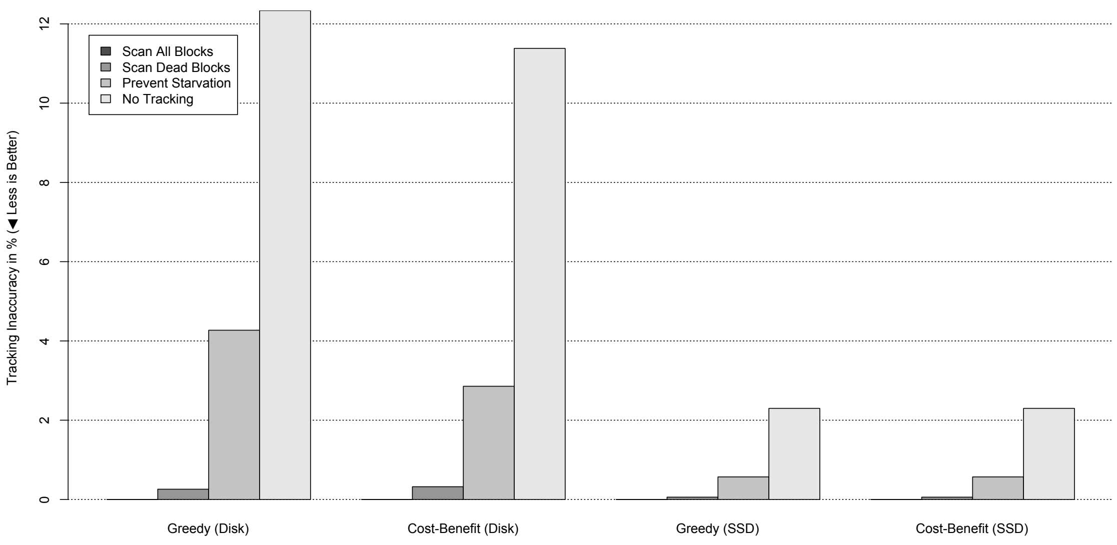
Figure 7.16c shows the block tracking inaccuracy introduced by the snapshots in percent of all blocks in the file system. In other words, it is percentage of unaccounted blocks in the file system. The Scan-All-Blocks algorithm introduced almost no inaccuracy at all, which puts it at almost exactly 0%. The inaccuracies introduced by the Scan-Dead-Blocks algorithm are also very small. They range from 0.5‰ for the SSD to 3‰ for the hard disk. Using no snapshot tracking at all has the worst accuracy and the Prevent-Starvation algorithm lies somewhere in the middle.



(a) Data Written



(b) Normalized Execution Time



(c) Tracking Inaccuracy

Figure 7.16: NFS Replay Parallel Statistics

CHAPTER 8

Conclusion

In this thesis I have presented an efficient implementation of two new selection policies for the garbage collector of NILFS2. Both policies perform significantly better than the default Timestamp policy in all benchmarks. The only advantage of the Timestamp policy is that it predictably deletes the oldest checkpoints first, which preserves more checkpoints and therefore allows the user to restore data from further in the past in case of an accidental deletion. However, using the protection period the other policies can also guarantee a certain time-frame in which the data is safe from the garbage collector.

Furthermore, I was able to reproduce the results of Rosenblum et al. [RO91] by showing that the Cost-Benefit policy generates a better bimodal distribution of segment utilizations than the Greedy policy. However, I could not reproduce the link between a good bimodal distribution and better performance. In fact, the Greedy policy performed significantly better in some benchmarks than the Cost-Benefit policy, despite having a worse bimodal distribution. This was especially true when the file system utilization was high. The Cost-Benefit policy may be better in the long run, because of the better bimodal distribution, but if free space is scarce, it pays to use a greedy algorithm. In the latter scenario the segment turn-over is so high, that the benefits of the Cost-Benefit policy never materialize. These results confirm the findings of Seltzer et al. [SBMS93, SSB+95] and Matthews et al. [MRC+97].

One possible solution for the above dilemma, may be a policy that takes the file system utilization into account and switches from a cost-benefit to a greedy algorithm at a certain threshold. This could be easily implemented in NILFS2. However, it would be difficult to create a good benchmark that simulates both scenarios accurately.

The overhead created by the live block tracking was barely measurable and more than compensated for by the hugely increased garbage collector performance. Although the live block tracking implementation is technically challenging for a file system that was not designed for it, it is ultimately worth it, because it is the basis for any sophisticated selection policy. The Greedy and Cost-Benefit policies are comparatively simple and only the first step. If the live block tracking were to be added to the mainline kernel, other researchers could benefit from this and create more advanced selection policies.

However, the live block tracking only works flawlessly if no snapshots are used. I have implemented and benchmarked three different algorithms to compensate for that problem. They are a trade-off between the live block tracking accuracy and the overhead they produce.

The Prevent-Starvation algorithm has almost no overhead, but it also produces the worst inaccuracies. A certain level of inaccurate live block counters is not a problem, because the garbage collector corrects the values over time. The only effect is a temporary decrease in garbage collector performance.

The Scan-Dead-Blocks algorithm has several orders of magnitude higher overhead for snapshots than the Prevent-Starvation algorithm, but it also produces a much higher accuracy and overall better file system performance. While the Prevent-Starvation algorithm shifts most of the work to the garbage collector, the Scan-Dead-Blocks algorithm actively corrects the live block counters at snapshot creation time. As the benchmarks have shown, this seems to be the more efficient approach, but it increases the time it takes to create or delete a snapshot. Furthermore, this delay scales linearly with the size of the file system, so that it could take several seconds to create a snapshot on a huge file system. The current implementation blocks the user-space process until the snapshot operation has finished, but it is entirely possible to delay this task and let the garbage collector do it asynchronously in the background.

The Scan-All-Blocks algorithm is an extension to the Scan-Dead-Blocks algorithm. It doubles the overhead, but it achieves perfect accuracy and even better performance in the benchmarks. It seems to be more efficient to actively correct all live block counters as soon as possible rather than shifting the work to the garbage collector.

However, there is no clear winner among these algorithms, because they are a trade-off between accuracy, overhead, and the spread of the workload. Ultimately, it depends on the particular use case. If really fast snapshots are important, then Prevent-Starvation is the better choice. If snapshots are rare and peak performance is important, then one of the other two algorithms is preferable. To accommodate this fact, I have added three file system flags that allow the user to choose between the algorithms. As a candidate for inclusion into the mainline kernel I would choose the Prevent-Starvation algorithm, because it is the simplest and it scales well with large file systems.

List of Figures

List of Tables

List of Algorithms

Bibliography

[AAMV09] Eric Anderson, Martin Arlitt, Charles B. Morrey, III, and Alistair Veitch. Dataseries: An efficient, flexible data format for structured serial data. *SIGOPS Oper. Syst. Rev.*, 43(1):70–75, January 2009.

[And09] Eric Anderson. Capture, conversion, and analysis of an intense NFS workload. In *Proccedings of the 7th conference on File and storage technologies*, FAST '09, pages 139–152, Berkeley, CA, USA, 2009. USENIX Association.

[Bla92] Matt Blaze. NFS tracing by passive network monitoring. In *Proceedings of the USENIX Winter 1992 Technical Conference*, pages 333–343, 1992.

[CE00] Christian Czezatke and M. Anton Ertl. Linlogfs - a log-structured filesystem for Linux. In *Proceedings of FREENIX Track: 2000 USENIX Annual Technical Conference*, page 77–88, 2000.

[dbe17] Dbench website, 2017. `https://dbench.samba.org/` [Online; accessed 04-July-2017].

[ELMS03] Daniel Ellard, Jonathan Ledlie, Pia Malkani, and Margo Seltzer. Passive NFS tracing of email and research workloads. 2003.

[ES03] Daniel Ellard and Margo Seltzer. New NFS tracing tools and techniques for system analysis. In *In Proceedings of the Annual USENIX Conference on Large Installation Systems Administration*, pages 73–85, 2003.

[fsm17] fs_mark website, 2017. `https://sourceforge.net/projects/fsmark/` [Online; accessed 21-July-2017].

[HEH$^+$09] Xiao-Yu Hu, Evangelos Eleftheriou, Robert Haas, Ilias Iliadis, and Roman Pletka. Write amplification analysis in flash-based solid state drives. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, SYSTOR '09, pages 10:1–10:9, New York, NY, USA, 2009. ACM.

[ioz17] Iozone website, 2017. `http://iozone.org/` [Online; accessed 04-July-2017].

[JBLF10] William K. Josephson, Lars A. Bongo, Kai Li, and David Flynn. DFS: A file system for virtualized flash storage. *Trans. Storage*, 6(3):14:1–14:25, September 2010.

[JHT+07] Martin Jambor, Tomas Hruby, Jan Taus, Kuba Krchak, and Viliam Holub. Implementation of a Linux log-structured file system with a garbage collector. *SIGOPS Oper. Syst. Rev.*, 41(1):24–32, January 2007.

[JWZ05] Nikolai Joukov, Timothy Wong, and Erez Zadok. Accurate and efficient replaying of file system traces. In *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies - Volume 4*, FAST'05, pages 25–25, Berkeley, CA, USA, 2005. USENIX Association.

[KAS+06] Ryusuke Konishi, Yoshiji Amagai, Koji Sato, Hisashi Hifumi, Seiji Kihara, and Satoshi Moriai. The Linux implementation of a log-structured file system. *SIGOPS Oper. Syst. Rev.*, 40(3):102–107, July 2006.

[KNM95] Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda. A flash-memory based file system. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*, TCON'95, pages 13–13, Berkeley, CA, USA, 1995. USENIX Association.

[LhP06] Seung-Ho Lim and Kyu ho Park. An efficient NAND flash file system for flash memory storage. *Computers, IEEE Transactions on*, 55(7):906 – 912, july 2006.

[Lov10] Robert Love. *Linux Kernel Development.* Addison Wesley, Upper Saddle River, NJ, third edition, 2010.

[LSHC15] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2FS: A new file system for flash storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 273–286, Santa Clara, CA, 2015. USENIX Association.

[MCB+07] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux Symposium*, June 2007.

[MKC+12] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. SFS: Random write considered harmful in solid state drives. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, pages 12–12, Berkeley, CA, USA, 2012. USENIX Association.

[MLE14] C. Min, S. W. Lee, and Y. I. Eom. Design and implementation of a log-structured file system for flash-based solid state drives. *IEEE Transactions on Computers*, 63(9):2215–2227, 9 2014.

[MRC+97] Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello, Randolph Y. Wang, and Thomas E. Anderson. Improving the performance of log-structured file systems with adaptive methods. *SIGOPS Oper. Syst. Rev.*, 31(5):238–251, October 1997.

[nfs08] nfsreplay, 2008. `http://www.gelato.unsw.edu.au/IA64wiki/nfsreplay` [Online; accessed 11-June-2017].

[nfs17] NFS trace replay repository, 2017. `https://github.com/zeitgeist87/nfstrace-replay` [Online; accessed 06-June-2017].

[OD89] John Ousterhout and Fred Douglis. Beating the I/O bottleneck: a case for log-structured file systems. *SIGOPS Oper. Syst. Rev.*, 23(1):11–28, January 1989.

[PBS16] Thiago Emmanuel Pereira, Francisco Brasileiro, and Livia Sampaio. A study on the errors and uncertainties of file system trace capture methods. In *Proceedings of the 9th ACM International on Systems and Storage Conference*, SYSTOR '16, pages 14:1–14:11, New York, NY, USA, 2016. ACM.

[QR13] Sheng Qiu and A.L.N. Reddy. NVMFS: A hybrid file system for improving random write in nand-flash SSD. In *Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on*, pages 1–5, 2013.

[Rep11] IOTTA Repository. Historical section : Network file system traces, 2011. `http://iotta.snia.org/historical_section` [Online; accessed 15-December-2013].

[RO91] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *SIGOPS Oper. Syst. Rev.*, 25(5):1–15, September 1991.

[RO92] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, February 1992.

[SBMS93] Margo I. Seltzer, Keith Bostic, Marshall K. McKusick, and Carl Staelin. An implementation of a log-structured file system for UNIX. In *Proceedings of the Usenix Winter 1993 Technical Conference, San Diego, California, USA, January 1993*, pages 307–326, 1993.

[SS97] Keith A. Smith and Margo I. Seltzer. File system aging – increasing the relevance of file system benchmarks. *SIGMETRICS Perform. Eval. Rev.*, 25(1):203–213, June 1997.

[SSB+95] Margo Seltzer, Keith A. Smith, Hari Balakrishnan, Jacqueline Chang, Sara McMains, and Venkata Padmanabhan. File system logging versus clustering:

A performance comparison. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*, TCON'95, pages 21–21, Berkeley, CA, USA, 1995. USENIX Association.

[TV14] Rukma Talwadker and Kaladhar Voruganti. ParaSwift: File I/O trace modeling for the future. In *28th Large Installation System Administration Conference (LISA14)*, pages 128–141, Seattle, WA, 2014. USENIX Association.

[WH02] Jun Wang and Yiming Hu. WOLF — a novel reordering write buffer to boost the performance of log-structured file systems. In *Proceedings of the FAST 2002 Conference on File and Storage Technologies*, page 47–60, 2002.

[WS11] Yuanting Wei and Dongkun Shin. NAND flash storage device performance in Linux file system. In *Computer Sciences and Convergence Information Technology (ICCIT), 2011 6th International Conference on*, pages 574–577, 2011.

[ZCcCE03] Ningning Zhu, Jiawu Chen, Tzi cker Chiueh, and Daniel Ellard. An NFS trace player for file system evaluation, 2003.