Karlsruher Institut für Technologie

Flashlog: A Flexible Block-Layer Redundancy Scheme

Diplomarbeit
von

Eric Hoh

an der Fakultät für Informatik

Verantwortlicher Betreuer: Prof. Dr. Frank Bellosa

Betreuender Mitarbeiter: Dipl.-Inform. Konrad Miller

Bearbeitungszeit: 15. Februar 2010– 13. August 2010

Hiermit erkläre ich, die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Literaturhilfsmittel verwendet zu haben.

I hereby declare that this thesis is a work of my own, and that only cited sources have been used.

Karlsruhe, September 3, 2010

_____

Eric Hoh

Deutsche Zusammenfassung

Das Thema der vorliegenden Diplomarbeit ist ein Schema zur Datenreplikation. Im Gegensatz zu vielen anderen Verfahren liegt der Haupteinsatzbereich nicht bei Servern und Workstations, sondern bei privat benutzten und mobilen Computern. Einige Anforderungen, z.B. an die Sicherheit der Daten und die Kosten sind gleich. Andere Aspekte wie die ständige Verfügbarkeit der Daten ist für professionelle Benutzer essentiell während sie für private Nutzer nur eine untergeordnete Rolle spielen. Es kommen aber auch neue Anforderungen hinzu, beispielsweise das Gewicht, die Größe, der Energieverbrauch und die Robustheit der zusätzliche Hardware.

Aufgrund dieser Anforderungen haben wir ein Verfahren entwickelt, dass diesen bestmöglichst gerecht wird. Das Verfahren basiert auf der Nutzung von drei Speichergeräten, typischerweise zweier Festplatte und eines Flashspeichers. Grundlage ist ein herkömmliches Backup Schema, das die beiden Festplatten umfasst. Der Flashspeicher z.B. in Form eines USB Sticks wird dazu benutzt die Zeit zwischen zwei Backups zu überbrücken. Dabei werden alle zu speichernden Daten gleichzeitig an die primäre Festplatte und den Flashspeicher geschickt (vgl. RAID1). Aufgrund der immensen Kosten für Flashspeicher mit großer Kapazität werden ausschließlich neu Daten auf dem Flashspeicher abgelegt. Das ist ausreichend, da alle alten Daten auf dem Backup Laufwerk noch gültig sind. Bei mobiler Nutzung wird dann nur noch die interne Festplatte des Notebooks und der USB Stick gebraucht. Die Backup Festplatte kann hingegen zuhause bleiben und wird nur gelegentlich synchronisiert.

Zuerst geben wir einen Überblick über verschiedene Speicher-Frameworks, die zur Implementierung des Verfahrens genutzt werden können. Anschließend werden die Eigenschaften von Flashspeicher untersucht und es wird ein vergleichbares Verfahren vorgestellt und analysiert.

Im dritten Kapitel untersuchen wir potentielle Gründe für Datenverluste, vergleichen verschiedene häufig genutzte Redundanz-Verfahren auf Basis vorher festgelegter Kriterien und betrachten verschiedene typische Anwendungsfälle.

Im Anschluss betrachten wir verschiedene Designaspekte, die schlussendlich zu dem von uns vorgeschlagenen Verfahren führen. Darauffolgend werden weitere ausgewählte Details der Implementierung betrachtet.

Die Evaluation umfasst verschiedene Geschwindigkeitstests, eine kurze Betrachtung des Energieverbrauchs und eine sowohl qualitative als auch quantitative Untersuchung der Datensicherheit. Desweiteren werden die Kosten der Verfahren anhand eines Beispiels ermittelt. Abschließend geben wir einen zusammenfassenden Überblick über alle Testergebnisse.

Im letzten Kapitel werfen wir einen Blick auf die Zukunft und die möglichen Erweiterungen und Verbesserungen des Verfahrens.

Abstract

Data redundancy has been used by companies for many year to protect their data. The price for disk space was high and you often needed additional hard- and software which was even more expensive. This has changed in the course of time and data redundancy has become more and more interessting also for private users. But most standard redundancy schemes are designed for commercial users and not for private ones. In this thesis we want to present a redundancy scheme designed for the needs of private users. This user group is often more price-sensitive, needs less disk space, has often less experience with storage systems and has no need for increased availability. Another group of users our redundancy scheme is designed for are mobile users. Most common redundancy schemes need additional storage devices which are often impractical for mobile usage due to their size and weight. Our scheme consists of three devices, a main device, a backup device which holds a copy of all data from the main device and a log or delta device which stores all data, that has been modified since the last backup, in a log structure. Our scheme is designed to use a flash memory device like a USB memory stick as log device and therefore we have named it “Flashlog”. Such a USB memory stick has a small form factor, is leightweight and consumes very little energy. That makes it perfect for mobile usage. For desktop users the USB flash device can be replaced by faster and bigger flash devices like SSDs.

Contents

Chapter 1

Introduction

In this thesis we want to present a new redundancy scheme that is specially designed for the needs of private and mobile users, because their requirements differ a lot from the ones of commercial users which have been the target audience for most existing redundancy schemes.

1.1 Motivation

The question how to store data is as old as the computer itself. There are basic storage devices like hard disks, tapes or optical media. But they have all in common that damages to the media or device always lead to a partitial or complete data loss. One method to reduce the probability of data losses is the use of more reliable media. The problem of this method is its cost efficiency. You must pay much more for a device which is only slightly more reliable than a much cheaper one. Thus there is another common method: replication. The idea behind replication is that it is cheaper and more reliable to have multiple copies of your data on cheaper devices than having a single copy on an expensive and more reliable device. Therefore many different redundancy schemes have been invented in the course of time. Most replication schemes have been designed for conditions you can find in servers and professional workstations. We want to design a replication scheme that fits better for private and mobile users. For mobile usage it is necessary that additional hardware is small, leightweight, robust and energy-saving. For private users the price is often more important than permanent availability.

1.2 Objectives

This thesis is about the development and evaluation of a redundancy scheme which combines high safety with moderate costs. Our solution protects the data

against hardware defects like disk failures but also against software or operating errors. The soultion must not be significently more expensive than using a Backup or RAID scheme. Secondary objectives are energy saving and increasing availability and performance. It is also essential that the used hardware meet the requirements of private and mobile users mentioned above.

1.3 Idea

Our scheme consists of three devices, a main device, a backup device and a log or delta device. The main device is the drive that is already there if no replication scheme is used. The data from this drive gets copied to the backup device regularly. Up to this point this would be the regular Backup scheme, but we have the additional log device. All blocks that get written on the main device are duplicated and also added to the log. Data from the main device that have not been modified since the last backup are only stored on the main and backup device, but not on the log device. So the log device can be much smaller than main and backup device. Considering the restrictions of the mobile scenario we optimized the log device for the usage of flash memory. According to our scheme the notebook's internal HDD is used as main device, a USB flash drive as log device and an external HDD as backup device which can stay at home (Figure 1.1).

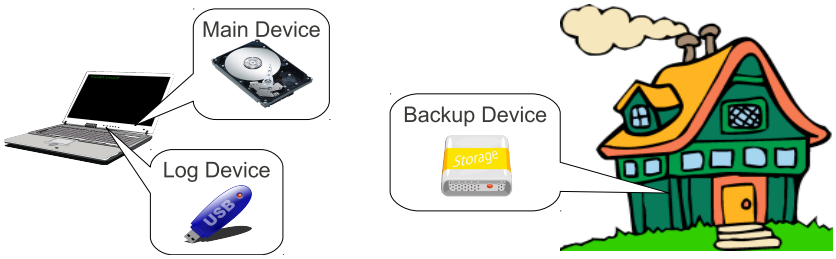

Figure 1.1: Mobile scenario; you take the notebook and the USB flash drive with you and the backup device stays at home.

1.4 Contribution

The idea of using a log device is already proposed in FEARLESS (see 2.4). Using a log structure to optimize durability and performance of flash memory is also a often used in several log sturctured filesystems. But our Flashlog scheme is the first one that combines the advantages of a space efficient log scheme with flash memory optimizations and a low overhead implementation in kernel space.

In comparison to the FEARLESS scheme we have a signifcant lower overhead regarding the needed disc space due to the block based approach and regarding the performance due to the implementation in kernel space and the additional flash memory optimizations.

1.5 Outline

In the second chapter we compare different frameworks which can be used to implement replication schemes like ours. We survey the characteristics and restrictions of flash memory and take a look on some related work. In the analysis chapter we define criteria to compare different schemes and analyse possible causes for data loss and corruption. Afterwards we take a look on the characteristics of some common replication schemes. The next chapters are about the architecture and implementation in which we present our Flashlog scheme, discuss some design decisions and explain some important implementation details. At the end we compare the scheme's performance with other schemes, check its safety in a qualitatively as well as in a quantitatively way and take a look on the costs for buying the needed hardware as well as the costs for replacing failing hardware and energy costs. After that we put all these aspects in relation and discuss the results. The last chapter is about possible extensions and improvements, architectural as well as implementation-wise.

Chapter 2

Background & Related Work

At first we present some common I/O frameworks which can be used to implement our schemes. After a brief look on a typical storage subsystem using the Linux storage subsystem as example, we take a look on some related work.

2.1 Frameworks

The I/O subsystem of current *operating systems* (OSs) is very complex. Besides the standard tasks there is a demand for extensions which can transparently transform I/O operations, e.g. data compression, encryption, replication or volume management. Many current OSs provide a framework which makes it much easier to implement such extensions. They can reside in the file layer, in the block layer or anywhere in between. In the following we take a look on some of these frameworks and at the end of this section we discuss our decisions for one of them.

ZFS

ZFS (formerly also known as *Zettabyte File System*) is a filesystem developed by Sun Microsystems and published in 2005 for their Solaris operating system. By this time there is also a implentation for FreeBSD and for Linux available. The reason why we mention it here is the fact that ZFS is not only a filesystem. Besides other useful features ZFS has an integrated volume management (zpool) and a integrated RAID system (RAID-Z). This grade of integration has advantages for guaranteeing data integrity and optimizing I/O operations in a way the single layers can not do [18]. But it also makes it more complicated to implement extensions like our Flashlog scheme. Besides the fact that Solaris is rarely used by our target audience, this is the reason why we do not have chosen ZFS as basis for our work.

FUSE

The acronym FUSE stands for *Filesystem in Userspace* and this describes what is has been designed for. FUSE can also be used for other kinds of I/O manipulations. Implementing drivers in userspace provides a great flexibility and allows the usage of libraries that are not accessible from kernel space. There are not only normal filesystems using FUSE such as *New Technology File System* (NTFS) or ZFS but also some exotic filesystems like WikipediaFS or GMailFS which map Wikipedia articles or your e-mail account on a filesystem. This flexibility also allows you to implement a replication scheme with FUSE. Fearless (2.4) is such a scheme, which is based on the same idea as our scheme, but uses a file-based approach. FUSE's biggest drawback is its performance. Due to its implementation in userspace many transistions between kernel and user space are necessary. Therefore the performance of schemes implemented with FUSE is clearly worse than comparable kernelspace implementations [5, 20].

GEOM

GEOM is FreeBSD's storage framework. It has a object oriented (from developer's perspective) and layered design (from user's perspective) and is situated in the block layer. All I/O operations going to a GEOM device can be transparently transformed by a so called GEOM module, e.g. geom_stripe (RAID0) or geom_mirror (RAID1). Each module is usually a consumer, consuming real or virtual devices as well as a provider, providing virtual devices. Different modules can also be stacked, e.g. an encryption module can be put on top of the mirror module to get confidentiality as well as safety [13].

DM

The *Device Mapper* (DM) is the same for Linux what GEOM is for FreeBSD. Naming and Implementation is different but the way how it works is similar to GEOM. The framework intercepts all I/O operations going to devices used for DM devices. These I/O operations are then redirected to the respective DM target. Targets are the equivalent to GEOM's modules, they implement the actual schemes. For simple tasks you just get an incoming request, modify it and than let DM forward it to the I/O scheduler. Using this feature you can implement a simple linear target (concatenation of multiple devices to a single big one) with only a few lines of code. As soon as you need to write the same data to multiple devices you have to take care of the I/O operations yourself. For such cases DM provides a set of convenience function, especially for creating own I/O requests and for copying data between multiple devices. On the scale of things the main

difference between GEOM and DM is the plattform they are based on, the rest are details and naming [3, 17].

Discussion

ZFS has all capabilities necessary to implement a replication scheme as ours, but in the end it remains a filesystem in the first place. It would be a much bigger effort to extend ZFS by such a replication scheme because it is designed as a self-contained unit. On the other hand FUSE is designed to allow the implementation of almost every kind of data storage and transformation. But the price would be a mediocre performance.

The differences between the two remaining frameworks, GEOM and DM are negligible. GEOM has a more flexible design but on the other side the flexibility DM offers is sufficient for most cases. The performance differences are insignificant if you consider the whole I/O Stack and the underneath platform and if you design and implement your DM target carefully it will be as robust as the GEOM counterpart. So in the end it is a question of personal taste and your prefered operating system. We choose DM because the Linux platform is more common and reaches a larger group of users.

2.2 The Linux Storage Subsystem

The Linux Storage Subsystem (Figure 2.1) is based on a layered design with each layer is an abstraction of the layer beneath. The uppermost layer is the *Virtual File System* (VFS), offering an abstract interface for most file I/O operations. The *filesystems* (FS) in the FS layer handle all operations on file level and map the file operations to block I/O operations for the block layer. The Device Mapper is a part of the block layer. It transforms incoming operations and then forwards them to the I/O scheduler. The I/O scheduler tries to rearrange and merge I/O operations considering the constraints given by the particular device to increase its performance [2, 12].

2.3 Flash Memory

The market share of flash memory based storage solutions are growing in comparison to the common hard disk drives. They have no moving parts. Due to this fact they are physically more robust, need less energy and have a much lower access time. Additionally they are leightweight and available in smaller form factors. But they also have some disadvantages for the users and system developers. They are

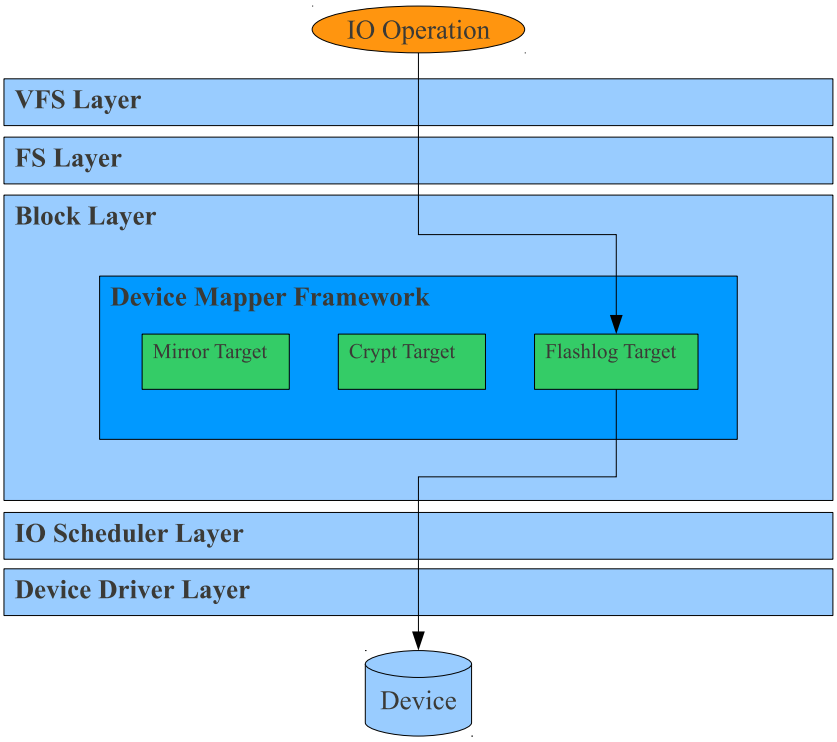

Figure 2.1: Linux I/O Subsytem

expensive and available only with small capacities. Furthermore the developers must consider their characteristics or the performance and durability will be far away from the theoretical maximum.

2.3.1 Flash Memory Layout & Restrictions

We will not discuss flash memory in every detail because there are many details which only the manufacturer knows or which are specific for a certain device. Luckily it is not necessary to know all these details. The following mentioned numbers are just typical values and can differ from device to device. In contrast to *hard disk drives* (HDDs), which only know read and write instructions, flash memory needs an additional erase instruction because writing can only be done on empty pages. Moreover, the three instructions need different amounts of time (read/write/erase: 25/200/1500μs). The finest granularity for writing (reading) to (from) flash memory is a page which has typically a size of 4096 Bytes. This value has been chosen because it is the size of a standard RAM page and it is a multiple of a sector (8 * 512 Bytes). For compatibility reasons most flash memory devices allow reads and writes with sector granularity like current HDDs. However this is

suboptimal because these sector operations must be mapped to page operations. In the worst case the flash memory controller must read (write) 4 kiB for every 0.5 kiB you want to read (write). In addition there is another restriction for writing. Pages in flash memory cells are combined to so called erase blocks of 64/128 pages. These blocks can only be written sequentially from the page with the lowest address to the page with the highest address [22]. If it is necessary to write a page with a lower address than the page before you have to copy the data into a buffer, erase the whole block and copy the modified data back. It is not only the performance which is affected by bad access patterns but also the flash memory's lifetime. MLC flash cells wear out after 1000 - 10000 write/erase cycles. In a worst case scenario a device can get unusable after some months because of worn out pages [1].

2.3.2 Wear Leveling & Other Optimizations

Most flash memory controllers have some integrated mechanisms to avoid the problems mentioned above. All these mechanisms are grouped under the heading Wear Leveling. Most devices have reserve pages which can replace faulty pages. Additionally the controller can copy data to reserve pages/blocks and exchange them with the old pages/blocks instead of erasing/rewriting the data. This mechanism leads to evenly distributed writes among all pages and increases the performance because the erasing operation can be accomplished in the background. These are only two of many mechanisms used in current controllers. The manufactures do not give many details because these mechanisms are very important for the performance and durability of their flash devices and they do not want to share their knowledge with competitors.

Wear Leveling can reduce the negative impact of bad access patterns but they cannot be completely avoided without the help of the layers above, namely the filesystem (in our case the Flashlog target). The filesystem can reduce the number of write/erase cycles by filling blocks sequentielly. It can also use the trim command to free blocks which are not used anymore. Without using the trim command every page is marked as being in use after a while. In this case only the small number of reserve pages can be used by the wear leveling algorithms. Many current *Solid State Disks* (SSDs) also supports NCQ (Native Command Queuing), which allows the drive to resort commands and avoids unnecessary erases/rewrites [9].

2.4 FEARLESS

Flash Enabled Active Replication of Low End Survivable Storage (FEARLESS) is a replication scheme based on the same idea like ours; replicating only mod-

ified data. In contrast to our approach Fearless uses a file based approach. The advantage is, that you can rescue every modified file, even if the main and backup device fails. On the other side you have to copy each file to the log device, even if only a single byte has been modified. Fearless is implemented using the FUSE framework which leads to a signifcant overhead. According to the developers of Fearless "FUSE_XMP has less than 30% of the performance of the underlying ext3 file system." [15]. In comparison to FUSE_XMP which is only a FUSE wrapper for the standard filesystems in the kernel Fearless has a overhead of 1% (read) / 29% (write). Flashlog has similar overheads but does not suffer from FUSE (DM has a very small overhead). The write overhead is in both cases caused by the additional writes to the log which can be relativized if the smaller log device is faster than the main device, e.g. a HDD as main device and a SSD as log device. Another difference is our optimizations for flash memory as log device. The developers of Fearless indeed propose using flash memory for the log device but they do not mention any optimizations. A reason for that could be the decision for the file based approach that makes such optimizations difficult to implement.

Chapter 3

Analysis

Before we analyse the pros and cons of the redundancy schemes, we must clarify which criteria to use. Furthermore we look into the causes of data losses and corruption and determine which schemes can protect against which kinds of problems. Finally we discuss some use cases typically for our target audience.

3.1 Comparison Criteria

There are different aspects about redundancy schemes that can be used as criteria to score them. Four typical criteria are safety, reliability, availability and costs. Besides these very generic criteria there are many others, which are often only interessting for some users, e.g. the mobile scenario mentioned in the introduction has further requirements regarding the size and energy consumption of the used devices.

Safety is probably the most important aspect. You can make some statistical analysis to measure safety, but that is not the whole truth. You can calculate failure probabilities of disks or RAID arrays but it is hard to calculate the probability that a user accidentally deletes important data or the probability that someone spills a glass of water over the computer. An additional qualitative analysis is necessary for a realistic rating of data protection.

Best safety is useless if the data gets corrupted and so reliability is almost as important as safety. The usual way to protect data against corruption are checksums and *Error Correction Codes* (ECC). Most redundancy schemes do not provide any additional protection against data corruption, but there are some advanced filesystems like ZFS and btrfs which implement such checksums so that this extra protection can be used with every scheme.

Availability is often more important for professional users than for private users. For a private user it is mostly just annoying if his data are temporarily not

available but a professional user loses money for every minute in which he cannot work due to not available data.

The price aspect is important for any user, professional and private ones and it is the only criteria of this set that can not be achieved by stacking different schemes.

3.2 Causes for Data Losses or Data Corruption

There are many reasons which can lead to data loss or data corruption. Almost every piece of hard- or software can be the cause of storage problems. But there are also external factors such as power outages or hacker attacks that can compromise your data.

Most obvious are problems with the storage subsystem itself. Almost every redundancy scheme protects your data against the failure of your storage devices. For some schemes such as RAID5 a controller failure can be even more problematic than a failure of the disk itself, because multiple drives are involved if a controller fails and RAID5 can compensate only a single device failure.

A less obvious cause would be a broken CPU, RAM or cable. These causes can be even nastier because they can happen without being noticed for a long time. Schemes that replicate all I/O requests immediately replicate also the errors. Better solutions are schemes which store older versions of every file or block. Snapshots (see 3.3.5) can be used to add such a behaviour to other schemes.

Another category are problems that are caused intentionally, e.g. by a virus or a trojan horse. The most sensible protection for this case is a good security system (virus scanner, firewall, etc.), but also simple backups can help to reduce the damage if the security system fails.

Sometimes the problem is not the computer, but the user himself, e.g. if he clicks the wrong button and deletes important files. For such a case snapshots and backup copies on external media are sufficient to protect the data.

3.3 Analysis Of Different redundancy Schemes

We present different typical and some less typical redundancy schemes in this section. After an explanation how they work we discuss their strengths and weaknesses.

3.3.1 RAID 1

Redundant Array of Independent disks Level 1 (RAID1) replicates all incoming writes on multiple devices. Read operations are usually performed on multiple device concurrently. Either different chunks are read from different devices to increase the performance or the same chunks are read from all devices to avoid data corruption [16, 19]. RAID1 can be implemented in software as part of the operating system (e.g. Linux' *Multiple Device* framework), as part of the controller's driver (aka Fake-RAID or Host-based RAID) or completely in hardware. The hardware implementation has the advantage to be completely OS independant, but it increases the already high costs and if the controller fails you must often replace it with a controller of the same type or at least from the same vendor to recreate the array.

3.3.2 RAID 4/5/6

RAID Level 4, 5 and 6 are all parity based redundancy schemes. If you have n disks in your RAID4 array you can fill n-1 disks with data. The last disk is used to store parity data. A RAID5 has no dedicated parity disk but distributes the parity data among all disks of the array. A RAID6 array uses the double amount of parity data and can compensate two disk failures instead of one [16,19]. All this RAID levels share some characteristics, they need additional computing power to calculate the parity data, they have an increased read performance, but their write performance is on a par with a single disk or worse. The failure of one (RAID6: two) disk(s) lead to a drastic performance degradation. With all disks intact chunks can be read from different disks, but with a degraded array some of the chunks have been lost and must be recalculated with the help of the parity information and all other chunks that have been used to calculate the parity information in the first place. All three RAID levels can be implemented in hard- and software, but due to their need for computational power a hardare RAID controller with a dedicated XOR unit is recommendable. Some RAID6 implementations have another advantage that is often forgotten. They can detect and correct bit errors. Almost all modern hard disks use ECC to prevent such errors, so that this is nothing you have to worry about as long as you have only a few disks. The probability of such errors are often stated to be 1 in 10^{14} for consumer devices and 1 in 10^{15} for enterprise devices, which is one bit error in about 11/14 TB. The biggest devices currently available can store 3TB, so that you only need an RAID5 array with 5-6 disks to get such a problem, statistically speaking. That might not yet be a problem for usual private users but in the not so far future it could become one.

3.3.3 Backups

Backups are possibly the oldest form of replication. They can be done on different levels. You can copy every single block/sector from your hard disk to another medium (cloning) or you can work on filesystem level and copy your data file by file. Working on filesystem level makes it easy to replicate only selected parts of your data by choosing specific directories or files. In contrast to RAID systems the backup scheme does not replicate the data immediatly after they have been produced but after it has been triggered manually. This can be a disadvantage as well as an advantage. If a disk fails you can only restore the state at the time of the last backup. But you can also restore it if your data gets deleted by a virus or a software bug which is not possible with a RAID system because the RAID "replicates" also the data deletion immediatly. Many advanced backup tools allow incremental backups. That means all data are copied at the first time and after that only the differences between this copy and the latest state are saved. This method allows the user to restore different versions of the data without needing too much additional disk space.

3.3.4 Online Backups

An online backup is a special form of the normal backup described above. The difference between them is, that you do not need a backup disk because the the data are transfered to a backup server in the internet. Some providers of such servers support so called *continuous data protection* (CDP). That means that any modification to the data is immediately transfered to the backup server [7]. Such services often supports storing different revision of your data. A further advantage is the physical separation between your local storage devices and the backup servers. In a best case scenario the provider mirrors your data on servers in different data centers. The providers charge a monthly fee for their service. For smaller capacities the prices are often moderate, but if you need more space using such a service can get very expensive. Furthermore you need a fast internet connection, which is often a problem, especially for the backup process, because many private users' internet access (e.g. tv cable, DSL) is asymmetric with much higher download than upload speed.

3.3.5 Version Control System / Snapshots

Version Control Systems (VCSs) are not replication schemes in the first place but in some cases they can be used for it. Their main purpose is the management of different versions of the same data and they are typically used for software development. Normally you initialize a repository (a place where all your data

get stored), add the files you want to track to this repository and commit the data whenever you have changed the data significantly [14]. If you place your repository on another drive as the actually data or even on another computer you can restore not only the last state of your data but the state at every commit. The different versions are only stored as diff against their predecessor and so they are very space efficient. As mentioned before these software is mostly used for source code and therefore most VCSs are optimized for text files and not for binary files.

Snapshots are similar to VCSs, but on filesystem or block level. Snapshotting is mostly a feature of filesystems or volume managers. If you trigger a snapshot the system creates a new virtual block device containing exactly the same data as the source.[1] If someone tries to write onto either the origin or snapshot device the blocks get copied in a new place and modified there (*Copy-On-Write*) [10]. Snapshots are often used in combination with a backup scheme. Making a copy of a device in use can result in an inconsitent backup due to the fact that the data can be modified while the copy is in progress. This problems can be solved with a snapshot. You create a snapshot which is an atomic operation. Afterwards you can backup the snapshot while using the origin device as usual.

You can use snapshots also for other purposes, such as software testing. You create a snapshot, install the software to test and if you do not like the software you can just roll back the snapshot. You can also use snapshots as mechanism to prevent data loss caused by mistakes of users.

3.4 Typical Use Cases

In this section we take a deeper look into some typical use cases. We analyse them to find out what requirements and restrictions are inherent in them.

3.4.1 Mobile Usage

In the last years the trend goes from desktop computers towards notebooks and recently towards netbooks. These computers can be very practical, but they also have some limitations. In most cases you cannot build in an additional hard disk to create a RAID array and you do not have a permanent internet connection to make an online backup. You could use an external hard disk but that also has some disadvantages, you would have an additional item to carry and if you want to use your notebook in a train, a bus or another vehicle with limited space you would have to find a place for it. Furthermore such a external hard disk needs an additional power supply or at least it consumes some watts from the notebook's

[1] The snapshot device actually does not contain any data at the beginning. The mechanism is similar to what filesystems use to provide hardlinks.

battery. In this case the most important requirement is that the needed hardware has a small form factor, is leightweight and consumes only little energy.

3.4.2 Backup++

This scenario is most interesting for people already using backups to safe their data. A backup is very simple and safe method to protect your data, but it does not protect your data all the time. If your main device fails you can only restore the state at the time of your last backup run. That can be even more nasty if you make your backups irregularly or at wide intervals. In this scenario the key point is that the user wants a protection also in the time between two backup runs. The additional space needed for the backup itself already doubles the costs so that it is important to keep the additional costs for this extra protection low.

3.4.3 Online Backups

Assuming you use an online backup service with CDP. Depending on the amount of data you write per time unit you need a fast internet connection. If you write too fast and your disk fails before anything is transfered, parts of your data get lost. For this use case we need a persistent buffer that protects the data until they got transfered to the backup server.

3.4.4 Persistent Write Cache

This use case differs from the other ones because this time the goal is not data protection but improving the I/O performance or saving energy. Assuming you already have some kind of data replication you can try to improve your I/O performance by using a very fast device for the replicated data or you can try to reduce the power consumption by using a very energy-saving device. In the second case energy can be saved because all modern HDDs have power-saving mechanisms. If the drive is not used for a while it activates a sleep mode and saves energy. If your replication device can handle some requests you can lengthen the time in which the main disk can sleep. The Linux kernel implements a mechanism called laptop mode which delays writes to increase this time. The bigger this delay is the bigger is the damage if your computer crashes before the data can be written, but if the delay is too short it does not safe enough energy and wears your disk out due to a high number of spin ups and downs. Flash memory based devices do not have this problem because they do not have any moving parts. Instead of delaying write operations you can store them on a flash device. Such a scheme was proposed by Fabian Franz to make Linux' laptop mode safer [8]. But also the Flashlog implements such a behaviour (asynchronous replication mode; see 5.6).

Chapter 4

Proposed Solution

In this chapter we take a look on some aspects of a redundancy schemes, present and discuss the alternatives and finally chose one of the options. At the end of this chapter this leads us to the solution we want to propose and which is then described in detail in the next chapter.

4.1 Design Decisions

4.1.1 Stacking of existing schemes vs. new full integrated scheme

The easiest way to create a new redundancy scheme is stacking different existing schemes. All RAID schemes with two digits in their level are such schemes. A RAID10 or RAID1+0 is an striped array consisting of mirrored arrays [16]. But there are also other possibilites, e.g. you can use a RAID scheme together with a backup scheme or with Snapshots. The RAID protect your data against device failures, the backup schemes against everything else. The problem with this combinations is that you often get more redundancy than you need. This would not be a problem, but this additional space must be paid, too. So we decided to take some characteristics of RAID1, Backup and Snapshots, mix them up and create a new integrated scheme out of it. From RAID1 we take the protection against device failures, from the Backup scheme we take the protection against other kinds of failures and from the Snapshot scheme we take its space efficiency.

4.1.2 Replication vs. parity-based redundancy

Most current redundancy schemes are based either on replication or on calculating parity data. Parity-based schemes need additional computing power, but they are very space efficient and therefore very popular. Nevertheless we choose an

approach based on replication for our scheme. The reason for that is the focus on private and mobile users. At least three independant disks are necessary to implement an efficient parity based scheme. For typical capacities of private and mobile users a replication-based scheme is cheaper even if the overhead is bigger because two mid-size hard disks for the replication scheme are cheaper than three smaller disks needed for the parity-based scheme. For capacities up to 1TB replication-based schemes are cheaper. If you take the energy and replacement costs into account partiy-based schemes are more expensive even for capacities up to 2TB.

4.1.3 File vs. block layer

A redundancy scheme can be implemented on different layers. The common approaches are the file-based and the block-based ones. Using the file-based approach has the advantage that it is very easy to specify only certain parts of your data for backup. These schemes are often implemented in userspace so that you can use them even if you do not have root privileges. The problem is that most schemes requires that you copy the whole file, even if only a single byte has been modified. The approach can be considerably slower if you have to copy many small files. These schemes are also often not transparent for the user, that means you must start the replication manually. A transparent, file-based approach is Fearless (see 2.4), but using FUSE leads to a signifcant performance overhead.

Block-based approaches are usually implemented in the kernel and work transparently. This leads to a low performance overhead, but the kernel must be customized if the scheme is not in the official kernel. Besides the performance overhead also the overhead regarding the needed disk space is lower because you work with a finer granularity. You do not need to copy the whole file, but only the modified blocks. We want to create a scheme that works transparently with a low overhead regarding disk space as well as performance. So we choose the block-based approach.

4.1.4 Magnetic vs. flash-based storage

If you want a mass storage device there are currently three wide-spread technologies: magnetic, optical and flash-based devices. Optical storage media are cheap, but slow, unreliable and only available with small capacities. Magnetic devices are also cheap, but they are signifcantly faster and more reliable. They are available with capacities up to some TB. Flash memory is even faster and theoretically also more reliable. They have a small form factor and consume only little energy. The problem is their price which is about 40x higher than the price of HDDs. We need both technologies to create a feasible solution for the use cases mentioned

before. Only flash memory has the attributes needed for the mobile scenario and only HDDs have the required capacity.

4.2 The Flashlog scheme

These reflections leads us to a integrated scheme, which is implemented as part of the block layer, using replication and HDDs as well as flash memory. Due to the prices for flash memory we use as few of it as possible. For desktop usage the flash memory can be replaced by a standard HDD or, if you want to reduce the energy consumption, by a notebook HDD.

The idea behind our scheme is that every piece of data is stored on two different devices for every point in time, not less and not more. The basis is a normal backup scheme. We use a RAID1-like mechanism to improve the safety between two backup runs. We do not have to sync the two devices at the beginning because we already have the backup device. The second device must only store data that gets modified between two backup runs and so we use a device with a smaller capacity for that, namely a flash-based device. Studies show that the average working set sizes are only 2% of the capacity over a 24h period [21]. More interessting for us would be the total amount of written data, but the working set size give us a rough estimation for this numbers and with that the needed amount of flash memory. Furthermore we need a data structure to store the incoming write requests for the main device on the much smaller flash memory device. We choose a log structure because with this we can optimize the scheme to improve the flash device's performance and durability (see 5.2).

Chapter 5

Implementation

While the last chapter's topic was the way that leads us to this scheme, this chapter's topic is the scheme itself. First we give you a more detailed overview of our implementation and then we take a look on some key aspects of the implementation.

5.1 Overview

We implemented our proposal as a DM target (Figure 5.1). Accordingly it is situated in the block layer. The upper layers transform any I/O request into block I/O objects (BIO objects). These BIOs contain all necessary information such as the destination sector, the block size and a memory address which is the source or destination for the read / write operations. Normally these BIOs are directly forwarded to the I/O scheduler but if the destination is a DM device the corresponding DM target's map function is called and gets the BIO object. The map function can then remap the request which means that the existing BIO is just modified and then resubmitted or it can take care of anything itself. In our case the map function uses remapping for read requests. Write requests get queued and then handled by the I/O thread which does the actual writing (Figure 5.2).

The Flashlog target consists of three respectively four threads. The main thread initializes all data structures, implements the DM interface, reserves and frees the devices and starts/stops the other threads as needed.

The I/O thread is responsible for distributing all write BIOs onto the underlying devices. Writing to the main device is easy. The thread just modifies the BIO's device attribute from the DM device to the main device. It is a bit more difficult to write on the log device. The thread must allocate free space on the log device from the ringbuffer, write the metadata and than modify the BIO object to redirect the write operation to the new sector on the log device.

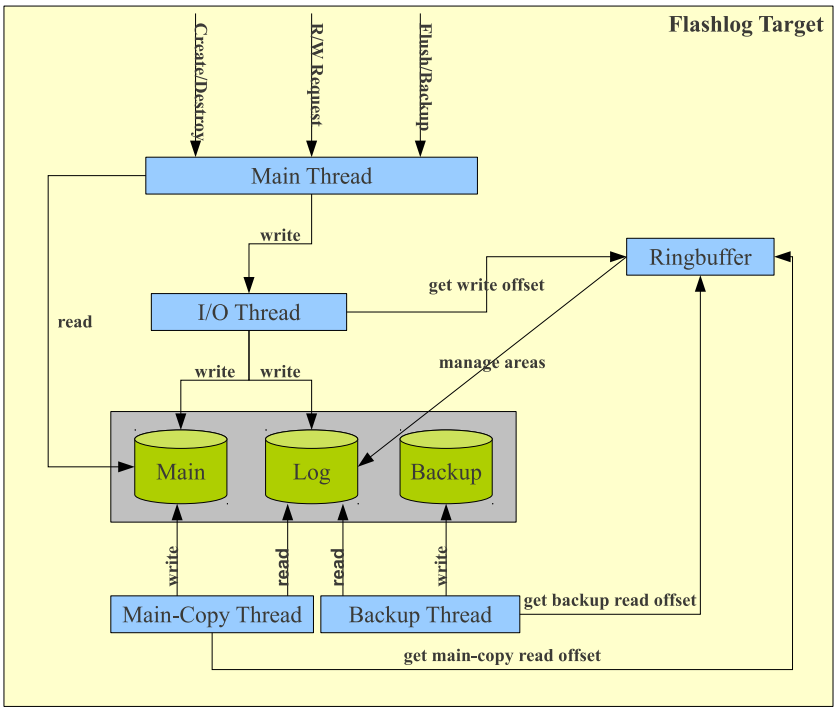

Figure 5.1: The main thread implements the DM interface, initializes all data structures and starts/stops the other threads, the I/O thread distributes the writes onto the underlying disks and the two copy threads copy entries from the log to the main/backup device. The ringbuffer manages the disk space on the log device and stores the current read/write positions for the other threads.

5.2 Log Device Layout

The naive approach would be writing metadata followed by the associated data back to back. This approach works well in memory and even on a mechatronical hard disk drive, but not on a flash memory based device due to its many restrictions. The metadata of a single entry has 24 Bytes and many Linux filesystems use 4 kiB as smallest granularity for writes. That means almost every entry would cross a page boundary and therefore would lead to writing two complete pages. The following entry would even more problematic because you would have to modify an already written page. But you cannot modify a written page on flash memory without erasing it before. Erasing a single page on a flash memory device means erasing the complete block of 64 or 128 pages and rewriting all other pages. This would be necessary for every single entry to the log.

As you can see this naive approach would not only be very slow, but it would also wear your flash memory out in no time. In our design (Figure 5.3) we divide

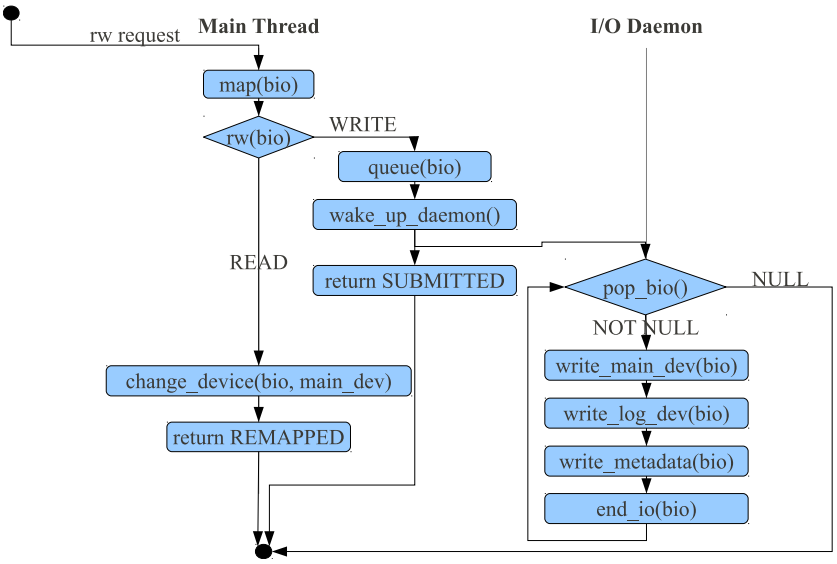

Figure 5.2: Algorithm for incoming block I/O requests in synchronous mode. The DM framework calls the map function. Reads are directly forwarded to the I/O scheduler. Writes are queued and and then handled by the I/O thread.

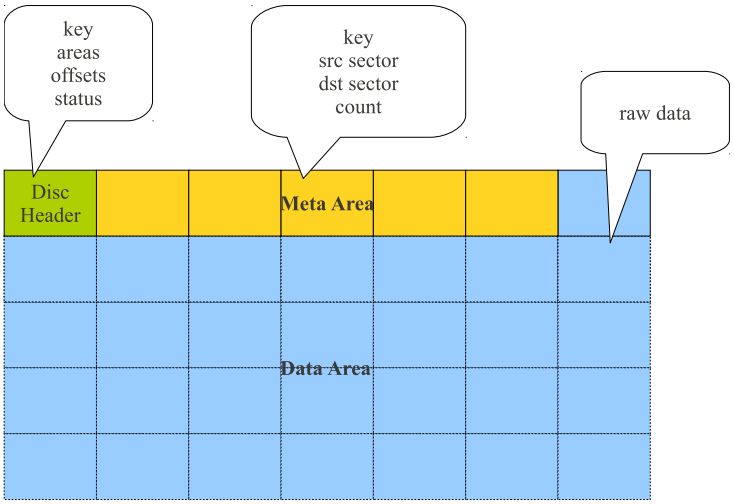

Figure 5.3: The log device is divided into three parts, the disc header, the meta area and the data area. The disc header stores the general state of the target, the boundaries of the areas, which parts of the areas are in use and the read/write pointers. The meta area is the "table of contents" of the data area. It contains information how blocks from the main device are mapped to the log device. The data area stores only raw data blocks.

the space on the flash memory device in three parts (areas), a disk header, a meta area and a data area. The disk header is small (1MiB) and stores only some status data. The other areas are for metadata / data and their size can be configured. The preset for the area sizes is:

$$n_{meta} = \frac{size_{page}}{size_{meta\_datum}} = \frac{4096B}{24B} \approx 170$$

$$size_{data\_area} = \frac{n_{meta}}{n_{meta}+1} * n_{pages} = \frac{170}{171} * n_{pages}$$

$$size_{meta\_area} = \frac{1}{n_{meta}+1} * n_{pages} = \frac{1}{171} * n_{pages}$$

This preset reserves enough space for the meta area even if every single write operation has the smallest possible size. If you know that most writes would have a bigger size you can reduce the number of pages reserved for the meta area. The implementation requires that the area sizes are chosen as multiples of 1MiB. This constraint ensures that the areas are not only page aligned but also block aligned (block size is usually 256 kiB or 512 kiB) if used on a complete disk or an aligned partition. The reason for this is flash memory's restriction regarding the write order in blocks.

Metadata are not written for each incoming write operation because this would mean erasing a block 170 times before the first page is filled. Instead we collect the metadata in memory and then write a complete page. Additionally we use not only one meta page but several because the computer can issue many more write requests than any storage device can handle. In such a case we have two choices. We can either write meta pages synchronously which is very slow or we can use multiple memory pages as buffer. We implement a combination of both. The user can define a max. number of buffer pages. If they are all filled a synchronous write is performed and the last page is reused afterwards.

5.3 (A)synchronous Replication

This section is about the topic when an I/O request is deemed to be completed (Figure 5.4). This is easy if you only write to a single drive, but for a scheme that writes to two drives you have the choice. You can consider an I/O request as finished after it has been written to at least one device (async), or you can wait until it is written on all devices (sync). Both variants can be useful so we decided to implement both and let the user decide which one fits his needs.

Synchronous replication is safer, but slower (approximately as slow as the slower of both devices).

Asynchronous replication can be used to speedup writings (e.g. by using a fast SSD as log device), but a crash after the data has been written to one device but not to the other one can lead to data loss and data corruption.

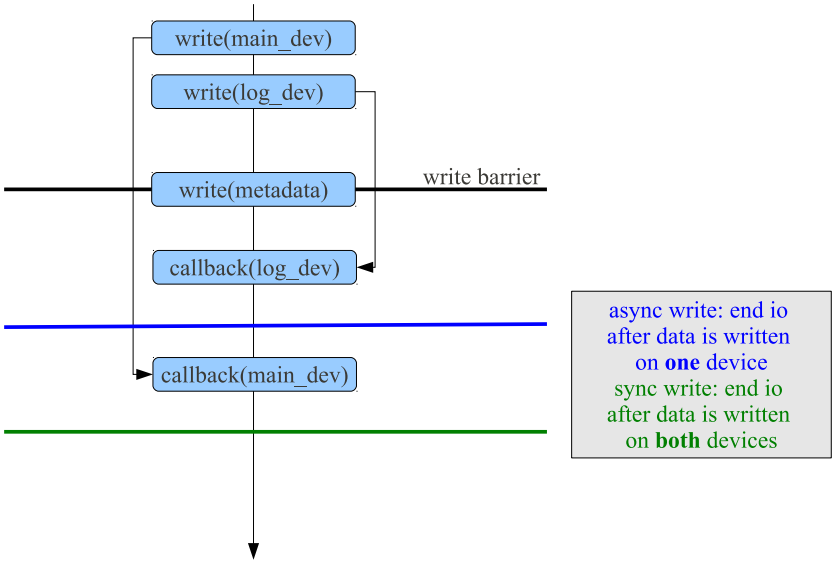

Figure 5.4: This diagram illustrates the difference between synchronous and asynchronous repliaction. It also shows how write barriers are used to guarantee write ordering between data and metadata writes.

5.4 Resync/Backup

For safety reasons it is necessary to resync the data from the main or log device with the backup device regularly. The main device as well as the log device stores all data needed to resync the backup device. Resyncing the backup device with the log device is faster because the log device stores only the modified blocks and so you must copy considerably less data. But there are cases in which you must use the main device, e.g. before the first resync, after a log device failure or after a log overflow.

Both variants can be implemented in user space as well as in kernel space. If implemented in kernel space the data can be backed up while the target is running (online). The user space implementation works offline, i.e. with the target stopped, to prevent any data inconsistencies. We decided to implement the resync between main and backup device in user space because it is used very rarely and so doing it offline is no problem. You can just use something like the unix command dd for this purpose. For the backup using the log device as source we use

a kernel space implementation, because having to stop the target whenever you want to do a backup would be very annoying.

5.5 Write Barriers

Without any precautions write orders can be reordered by the I/O scheduler or by the storage device itself. This can be useful for hard disk drives to reduce the number of head movements and by association improve the performance and durability of the device. Also flash based drives can use reordering to avoid unnecessary erase or read-modify-write cycles (see 2.3.1). This is unproblematic as long as there are no two write operations for the same sectors. In this case reordering would lead to data corruption because the newer data would be written first and then they would be overwritten by the older data [6].

In the past you would peform the first operation, wait until it is finished and then perform the second operation. This method is safe but slow. The write barrier mechanism is a faster alternative. The filesystem can set a write barrier flag for an operation if the underneath layers support it. Every operation that has been performed before (after) the flagged operation must then be completed before (after) the barrier request.

If the incoming BIO's barrier flag is set, we carry it over to the new BIO for the main device, but not for the log device. The log device do not overwrite old data but just appends new ones. Not the position of the data on the device but the position in the metadata is crucial for the order in which data get restored and we guarantee that all requests get processed (including writing the metadata) in the same order they have come in. This is easy because metadata get collected in memory in a single thread and there is no reordering for memory operations[1].

As mentioned above we can ignore barrier flags for writing data to the log device, but it is crucial to ensure the ordering of data writes in relation to metadata writes (Figure 5.4). The data writes must be completed before we can perform metadata writes. Without this determination a crash can lead to data corruption if the metadata says "on page x are the data y", but the crash occures before the data have been written on page x.

[1] Actually memory operations can be reordered by the compiler or the CPU but they guarantee that their reordering has no effect on the result so we can just assume that their is no reordering at all

5.6 Asynchronous Replication

In asynchronous mode data are primarely written on the log device. Data are written back to the main device if a flush is triggered or before a read operation because reads can only performed on the main device and to make sure that the latest data are read it is necessary to resync the main device before reading from it. For this reason it is crucial to resync regularly or the latency can get enormous, e.g. if you have written gigabytes of data without any read in between. A following read has to wait until all these data have been written back to the main device.

There is another way how async. replication can be implemented. You can implement the async. mode like the sync mode with the only difference that the BIO is ended after the first drive has written the data. This method avoids an additional read on the log device to copy the data to the main device but is has other disadvantages. The main device cannot be spun down as long as there are any incoming reads or writes. Moreover the upper layer can free the buffer memory after a BIO has been ended but before the the data have been written on both devices. So it is necessary to copy the data to another buffer before the writes are issued. This problem becomes more serious if the log device is much faster than the main device (or the other way round). In a worst case scenario (mechatronical HDD as main device, a fast SSD as log device and small random writes) the big throughput gap between main and log device can get enormous due to the fact that log is always written sequentielly, even for random writes, so that the faster SSD gets a further advantage over the slower HDD. All data, that have been written to the SSD but not yet to the HDD must be kept in memory.

Our method needs more reads on the log device but can reduce the active time of the main device, is easier to implement and does not need additional memory. On a fast device like a SSD the additional reads are not a problem, but for slower devices like USB flash drives the synchronous mode is probably the better choice, at least if you have high I/O workloads.

5.7 Area Management

As mentioned before we separate data and metadata on the log device into areas. We use two modified ring buffers to manage these areas (Figure 5.5). Each ringbuffer has one writer (I/O thread) and up to two readers (backup thread and for async. replication mode main-copy thread). All methods are lock protected because all three threads can try to access and modify the data at the same time.

There are two functions per reader and writer, get_offset(size) and inc_offset(size). The first one checks if there is enough space left in the area and returns the offset for the next read/write. This is necessary because the writes/reads do not always

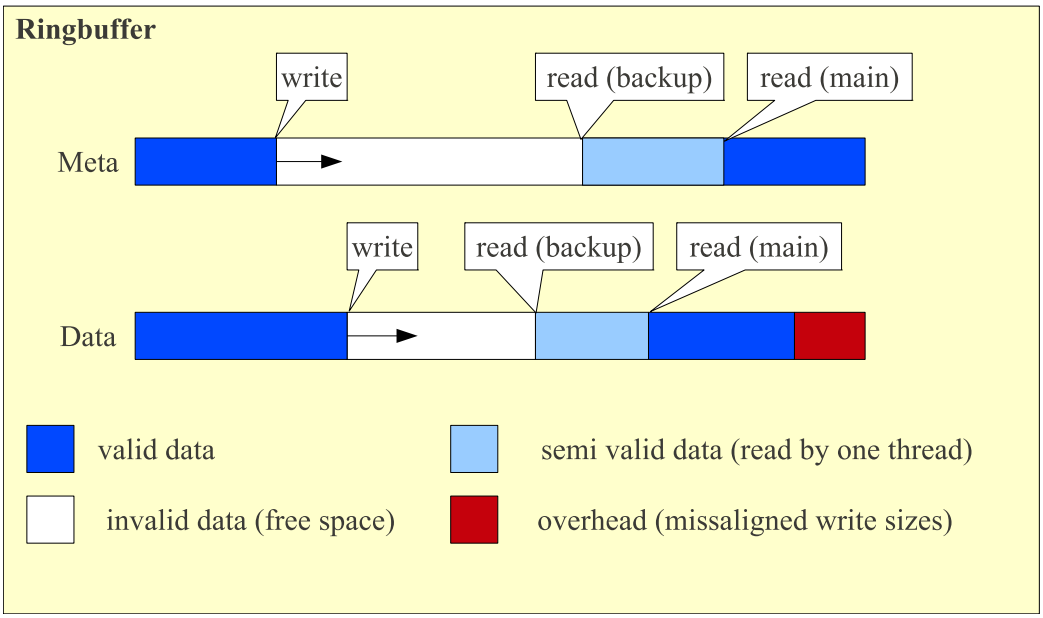

Figure 5.5: Structure of the ringbuffer, which is used to manage the areas on the log device.

have the same size and we do not split requests. If we are just before the end of the area it is possible that a small request would fit at the end of the area, but a bigger one must be relocated to the start of the area. If this happens we cannot use the space at the end of the area until both read pointers have passed. This will never happen for the meta area because the meta area is page-aligned and metadata are always written page by page.

The second function increases the offset after the read/write has been triggered. We do not wait until the data has finally written to disk because this leads to write after write problems due to the I/O thread writing asynchronously. On the other hand this can lead to read after write problems, e.g. if the backup thread tries to read a log entry that has already been processed by the I/O thread but not yet written. There are two mechanisms which together prevent this problem. The I/O workqueue is always flushed before the main-copy or backup thread is started. Additionally both threads only copy data that has been available at the point in time when they have been triggered. Write after read problems can not occur because the copy threads work synchronously.

5.8 Buffering and Flushing

For this section we must differentiate between data and metadata. Metadata are always buffered because their size (24 Byte per entry) is to small to write them one by one. That would not only cause a big overhead but it would also be very bad for the flash memory's lifetime. Metadata are written at page granularity. For performance reasons they are written asynchronously and so we have to keep some reserve pages in stock. If this reserve is not sufficient the I/O thread automatically switches to synchronous writing and reuses the same page afterwards.

In synchronous replication mode only the metadata must be flushed. The user data are not buffered in any way so flushing is not necessary. In asynchronous mode a flush must not only write the metadata back, but it triggers a resync between log device and main device.

Chapter 6

Evaluation

In this chapter we present the results of the practical tests, discuss the results of the qualitative and quantitative analysis of the schemes' safety, examine their speed, energy consumption and costs.

The performance of the schemes is measured on two levels, with and without filesystem. The tests without any filesystem show the potential of the schemes while the tests with filesystem represent more practical results and show how efficient the schemes are under real world conditions. Our focus lays on the write performance because Flashlog just forwards read requests so that its performance does not differ much from the performance of the raw device.

Safety is nothing you can measure directly, so we do a statistical analysis of the schemes' probability of failure and in addition we discuss which schemes is helpful in which scenario, because failure probabilities are only useful for failures of the storage devices themselves, but not for other kinds of failures.

The schemes need different kinds and amounts of hardware so that they also have different costs, especially with growing capacities. We compare the schemes on the basis of costs per size without considering the other aspects, because we would have to do too many additional assumptions due to the many requirements of the different use cases.

The energy consumption of the schemes is measured for some real world workloads and compared with the energy consumption of a single drive.

6.1 Test Setup

Our test system is based on a Intel Core i7 920 CPU running with 2.67GHz on a MSI X58 Mainboard, 3GiB of DDR3 RAM. The operating system is Ubuntu 10.04 Server Edition 32Bit with the Ubuntu 2.6.32-22 generic kernel. We uses the storage devices stated in 6.1. If not mentioned otherwise we use the two HDDs

for RAID1 and as main + backup device of the Flashlog scheme. For RAID5 we use the two HDDs + the SSD.

type	name	capacity	comment
3.5” HDD	Seagate Barracuda 7200.10	80GB	7200rpm
3.5” HDD	Western Digital WD RE3	500GB	7200rpm
SSD	Intel X25-E Extreme	32GB	SLC flash memory
USB flash device	SanDisk Cruzer Contour	8GB	

Figure 6.1: storage devices used for the tests

6.2 Benchmarks

We have performed several benchmarks to find out how well the different devices and schemes perform with certain workloads. At first we compare the time for doing an initial and incremental backup. Then we compare the scheme's sequential performance, which is important if you often work with big files like video or music files. Finally we compare the random I/O performance. This kind of workload is the most common one, e.g. program files, config files, websites, sourcecode, ...

6.2.1 Initial Backup

In this benchmark we assume we have data on the main device and an empty backup device[1] We measure the time the schemes need to get into a state in which the data are fully protected. The replication method for the Flashlog scheme has no influence on the initial backup so there is only one entry for Flashlog in the diagram (Figure 6.2).

Unsurprisingly Flashlog and RAID1 show a similar performance. Both schemes copy all sectors sequentielly from one device to the other. In contrast to both of them the RAID5 can benefit from the fast speed of the SSD. The performance of rsync depends on the number and size of the files and of course on the used filesystem. This dependencies reduce the effective performance. In a best case scenario it can be as fast as Flashlog or RAID1 which copy the data sequentielly and with optimal block size. The diagram shows rsync's speed with ten 128 MiB files. Its performance would be much worse if you copy many very small files.

[1] Not all RAID implementations support carrying preexisting data over to a newly created array. The time for resyncing an array after a device failure is comparable, because the same work has to be done.

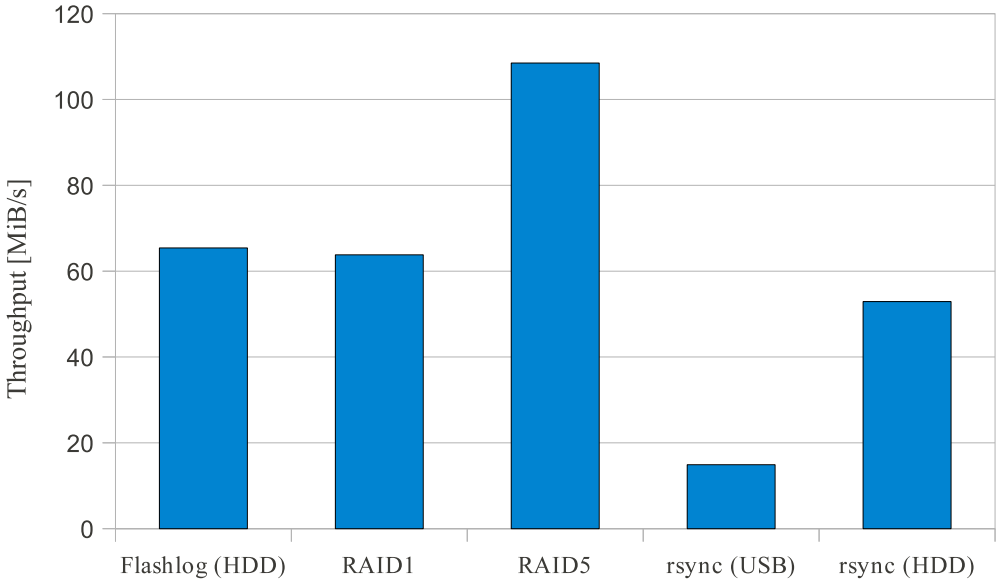

Figure 6.2: Initial Backup Performance

6.2.2 Incremental Backup

This test shows how long it takes to update an old backup (Figure 6.3). RAID schemes always update the ”backup” on-the-fly so that they never need additional time for this. For rsync the time strongly depends on the way to find out which files has been modified and again on the average file size. We use the checksum mode because it is safer than just checking date and size.

Flashlog loses performance in comparison to the initial backup. This is caused by a suboptimal implentation of the backup daemon which generelly writes the data in the same order and in the same sizes as they were originally issued. That means if the same block has been modified multiple times the backup daemon does the same on the backup device. Furthermore the backup daemon always uses synchronous writing for each continuous block of data. This is not such a big problem for data that have mainly been written sequentielly, but it is very slow for data written randomly, especially on devices with low random write performance like USB flash drives. Nevertheless it is usually signifcantly faster than just copying all data because you write only the modified data.

Rsync is slower compared to the initial backup. The reason is that we choose a worst case scenario in which all files have been completely changed. Using modification time and filesize instead of checksums to determine if a file has been modified is less safe but much faster, especially if most files has not been modified at all. There are plans for the btrfs filesystem to allow tools to read the filesys-

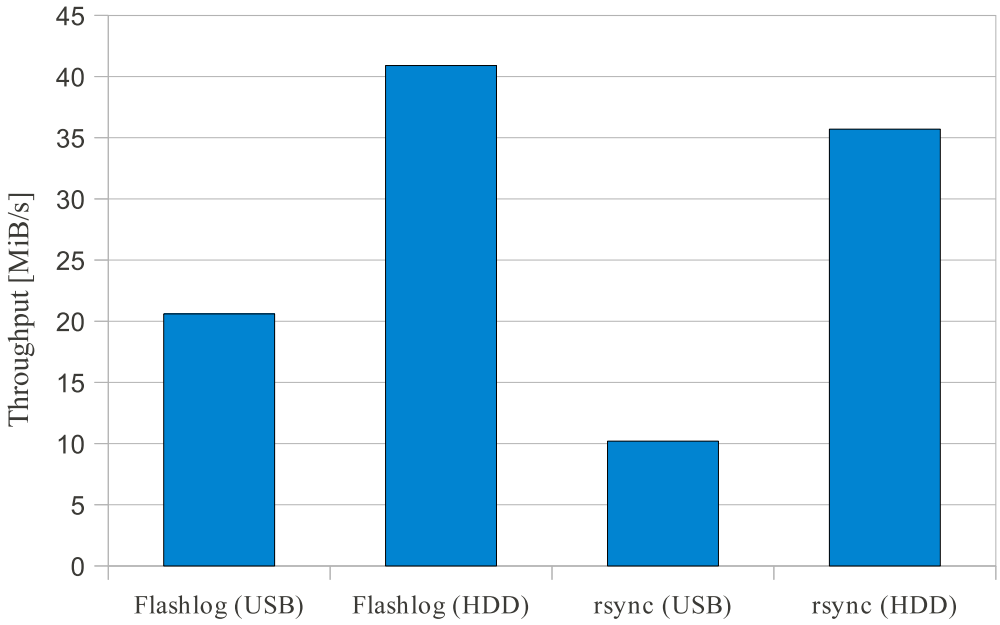

Figure 6.3: Incremental Backup Performance

tem's internal generation number for any file and directory which can then be used instead of time / filesize and checksums [4].

6.2.3 Sequential I/O

In this section we measure the sequential performance of the different replication schemes. We take a look on the raw write performance of the devices without filesystem, then we measure their speed using Bonnie++ on a ext4 filesystem with default settings.

Raw Write

The first test is a write test on a raw device without any filesystem. We use the dd tool for this test to copy 2 GiB of data in 4k blocks from /dev/zero to the test device (Figure 6.4).

As expected the SSD is the fastest device followed by Flashlog in asynchronous mode and the SSD as log device. The gap between them is mainly caused by the additional writes for the metadata and the disc header. Flashlog in synchronous mode is as fast as the slowest of its drives (main or log, backup device performance is only relevant for backup/restore time). This leads to a similar performance for HDD only, Flashlog with HDD+SSD and RAID1. This similarity shows that both

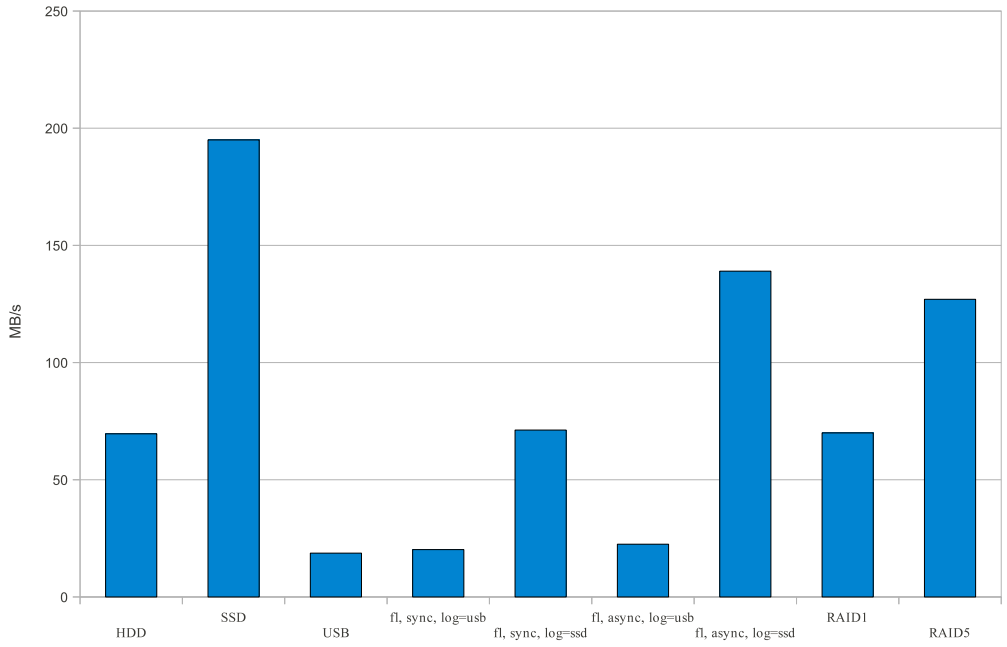

Figure 6.4: Raw write performance of the single drives, the Flashlog scheme in sync and async mode with different log devices and RAID levels 0 and 5 measured with dd

Flashlog and RAID have only a minimal overhead. As soon as the USB flash drive gets involved the performance drops to the level of the flash drive alone.

Bonnie++

Our next benchmark tool is Bonnie++. We use a buffer size of 512 MB and a write size of 1 GB. Each test device contains a single ext4 partition filling the complete device. The partition is formatted and mounted without any additional options.

The results (Figure 6.5) are a bit worse than the raw write results what is not unexpected due to the filesystem overhead. The only noticeable difference between write and rewrite is with the Flashlog device in asynchronous mode. This anomaly can be explained with additional read operations that have been occured during the write test which triggers a resync between the log and the main device. In the following rewrite test these reads are probably handled by the filesystem cache. This write back theoretically decreases the performance by 50% of the log device because every byte that has been written to the log device must also be read for the copy operation. The gap is bigger for the HDD/SSD combination due to the fact that in this case the HDD is the bottleneck because its maximal write speed is less than 50% of the SSDs read speed.

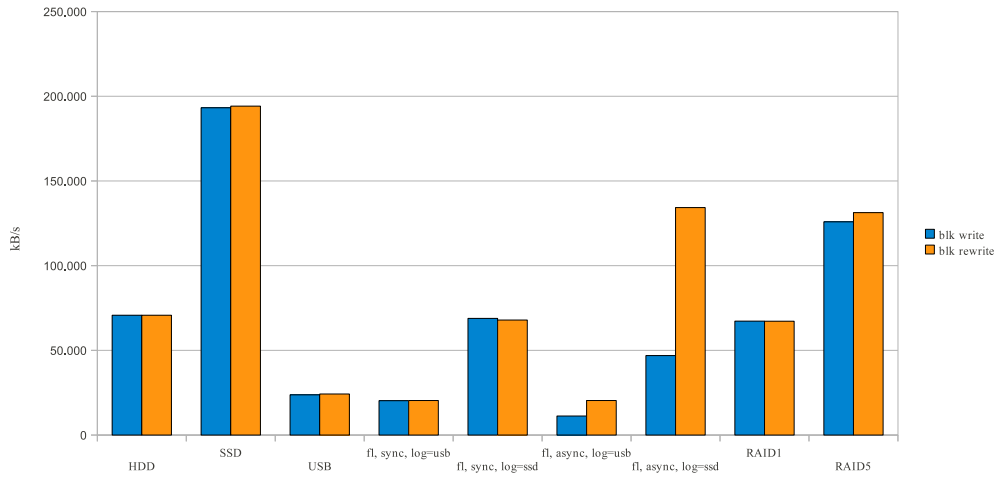

Figure 6.5: Sequential write/rewrite performance of the single drives, the Flashlog scheme in sync and async mode with different log devices and RAID levels 0 and 5 using the ext4 filesystem measured with Bonnie++

Scheme	CPU Load	Scheme	CPU Load
HDD	8%	Flashlog (sync, log=USB)	2%
SSD	18%	Flashlog (sync, log=SSD)	7%
USB	2%	Flashlog (async, log=USB)	1%
RAID1	8%	Flashlog (async, log=SSD)	5%

Figure 6.6: Bonnie++ CPU-Usage

The CPU usages of the schemes (Figure 6.6) correlate strongly with their performance. The results for the rewrite tests equals the numbers above with one exception. The Flashlog (async, log=SSD) scheme has an increased CPU usage during the rewrite test which is caused by the higher write speed due to the missing write back before read.

File Copy

This is a simple test to compare the wattage of a single SSD with a Flashlog array consisting of the same SSD with a USB flash drive as log device (Figure 6.7). Ten 100MiB files are copied from another device to the test device/array.

In this test the curves are completely different, because this test is completely I/O bound, so that it is not very astonishing that the SSD is much faster than the Flashlog array using the USB flash drive. But you can also see the effect of the async. replication mode. For a SSD as main device this does not help to save

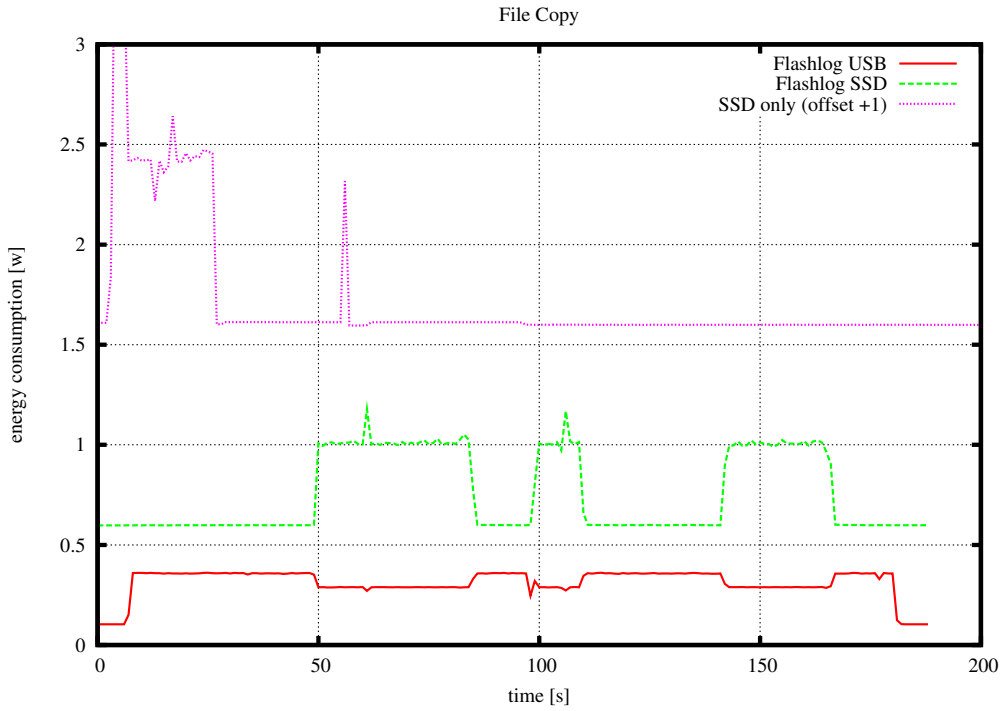

Figure 6.7: Energy consumption while copying 10 100MiB files from a separate disk onto the SSD / Flashlog array using the SSD as main device and the USB flash drive as log device

any energy, but in another constellation (HDD as main device) the results could be different, performance-wise as well as regarding the energy consumption. Unfortunately we had problems with the instrumentation of the HDD so that this comparison must be done another time.

TV capturing

This test scenario (Figure 6.8) is optimal for the Flashlog scheme. We write data, coming directly from the satellite into a big file. There is no need for any reads.

While the single disk run permanently because of the regularly incoming writes, the same disk can sleep most of the time because the USB flash drive stores all incoming data in the meantime.

6.2.4 Random I/O

We use iozone for the random I/O tests. There are two important parameters for this test, the record length and use or not use of kernel I/O buffering. The record

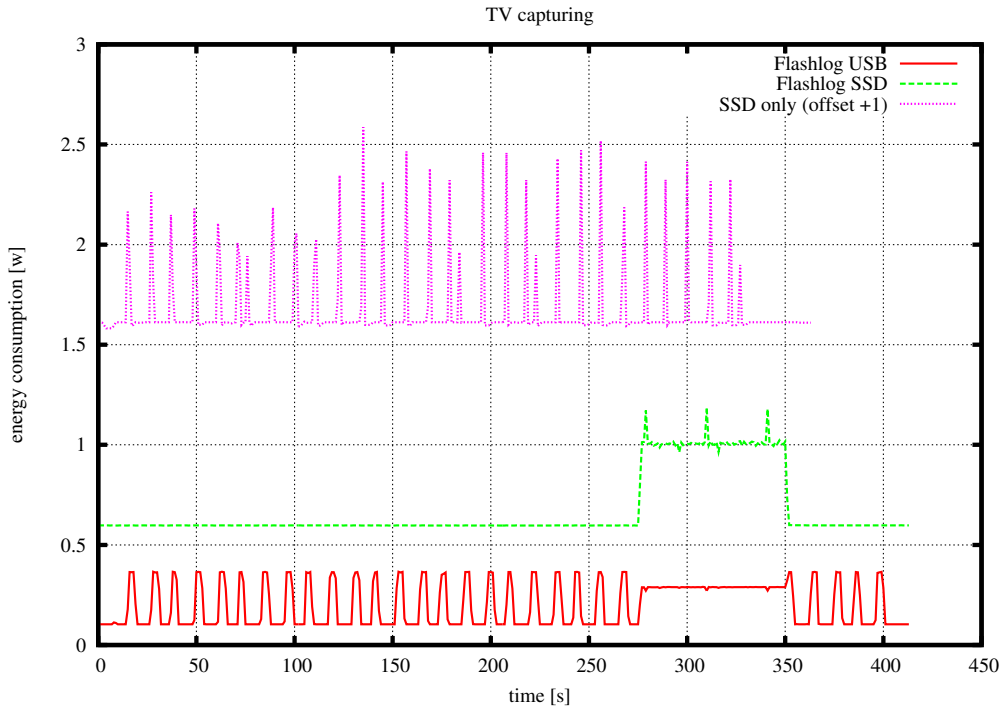

Figure 6.8: Energy consumption while capturing a TV stream onto the SSD / Flashlog array using the SSD as main device and the USB flash drive as log device

length varies from 4kB up to 16MiB.

The results for random read (Figure 6.9) demonstrate the fundamental advantage of flash memory based storage devices, especially with small record sizes where even the USB flash drive outperforms the HDD by a factor of nine. The Flashlog scheme just forwards read requests so that its performance is the same like the single HDD. RAID1 and RAID5 distributes read requests on their underlying disks so that they can almost double their read performance in comparison to a single HDD.

The write results (Figure 6.10) are similar to the read results for most schemes. The SSD is faster than the others and the USB flash drive is slower. The Flashlog scheme (log=USB) is more than 20x faster than the USB flash device alone, although Flashlog must write even more data on the USB flash device. The reason for that is that Flashlog writes the data as log and therefore the data are always written sequentially even for random write requests. This effect cannot be noticed if the Flashlog schemes uses the SSD as log device because in this case the HDD is the limiting factor.

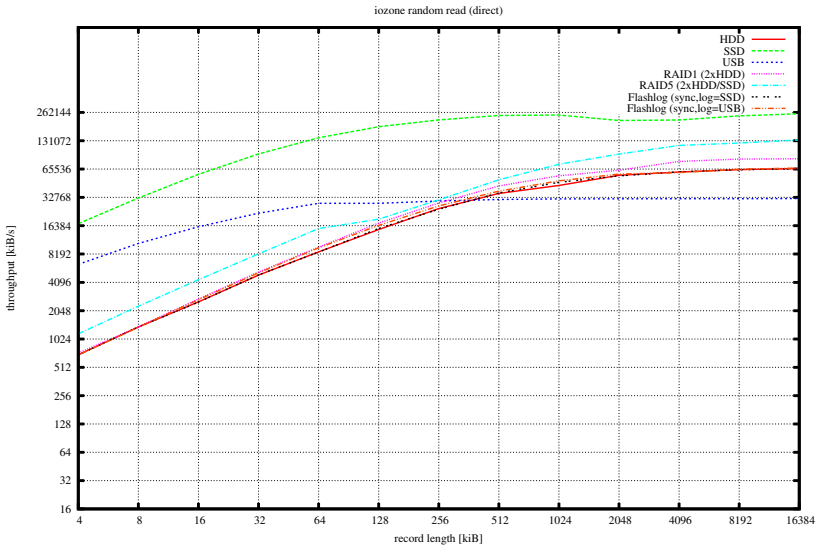

Figure 6.9: Random read performance of the single drives, the Flashlog scheme and the RAID levels using different record lengths; measured with iozone using -I flag (disabled kernel buffering)

Kernel Compilation

For our next test we let the computer compile the Linux kernel. Compiling is mainly limited by the CPU, so that it give us a rather low load scenario with random patterns with mixed reads and writes. In Figure 6.11 we compare the wattage of a stand-alone SSD with a Flashlog array consisting of the SSD as main device and the USB flash drive as log device in asynchronous replication mode.

Such mixed workloads are unfavourable for the asynchronous mode because whenever a read occurs we have to resync main and log device so that we do not have any advantages. For such workloads we can only hope, that the caching mechanism can handle some reads so that the we can at least let the main device idle for a bit longer. As you can see this seems to work two or three times during our test. Although we recommend using the synchronous mode if you often have such workloads because it is safer and the amount of energy you would save with async. mode is minimal.

6.3 Reliability

In this section we want to compare the safety of common replication schemes. We evaluate the protection the schemes can offer in several situations like a device failure, a controller failure, ... Additionally we do a statistical failure analysis on

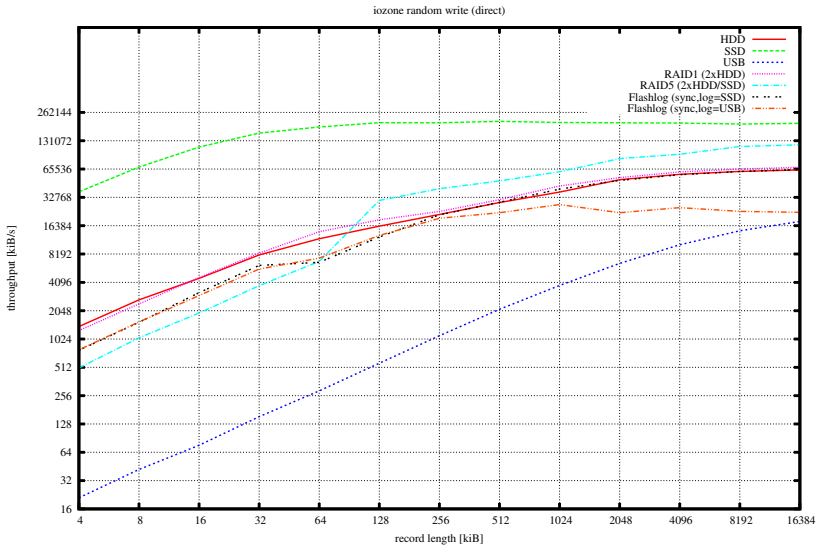

Figure 6.10: Random write performance of the single drives, the Flashlog scheme and the RAID levels using different record lengths; measured with iozone using -I flag (disabled kernel buffering)

the basis of the failure probabilities of the single devices.

6.3.1 Qualitative Analysis

In this section we present several scenarios in which data losses can occur. Then we evaluate which replication schemes can prevent these, and if they can, to which extent.

Power Failure

In this scenario we assume a failure of the power supply. This kind of failure is mainly a problem for the filesystem and the caching mechanisms of the I/O subsystem. As long as the replication schemes do not buffer any data there is nothing they can do. Flashlog buffers a small amount of metadata so that the log device can be out of sync after such a failure. In this case a full backup is required to guarantee data integrity. If Flashlog is used in async. mode this is not necessary because all writes are written to the log first and copied to the main device afterwards. RAID systems normally do not buffer any data so that they behave like a single HDD in such a situation. The same applies to the Backup scheme. In some rare cases if the filesystem gets broken and cannot be repaired the Flashlog and Backup schemes have advantages because both have a consistent

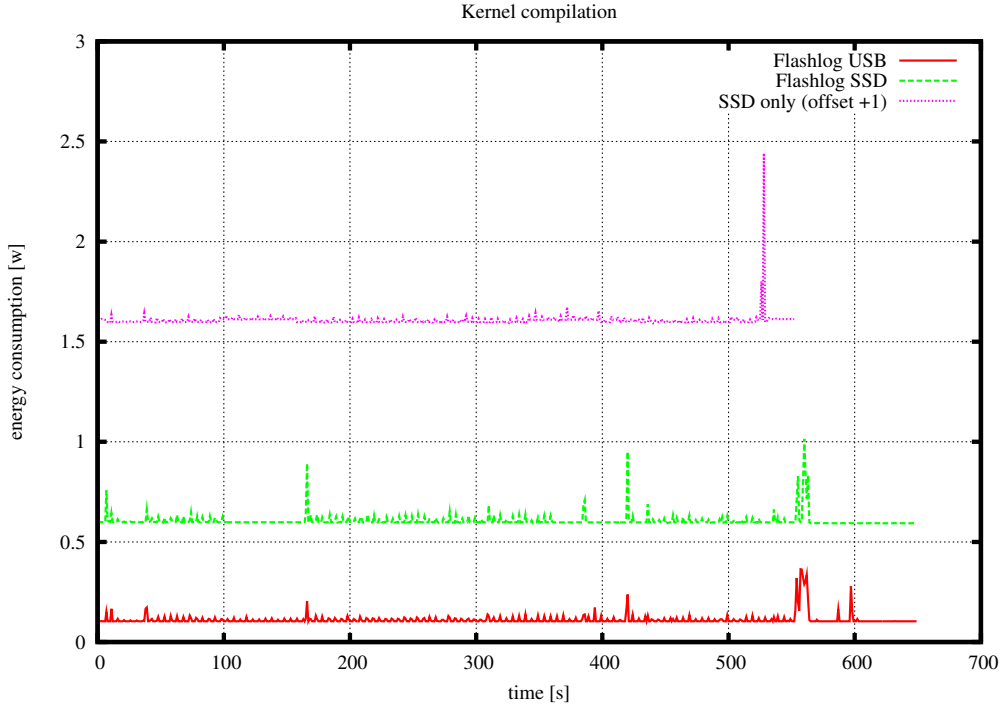

Figure 6.11: Energy consumption while compiling the Linux kernel on the SSD / Flashlog array using the SSD as main device and the USB flash drive as log device

state on their backup drives. This whole problem can be prevented by using a *copy-on-write* (COW) filesystem like ZFS or btrfs. If your filesystem does not have this feature you can make regular snapshots. But the best solution would be a redundant *power supply unit* (PSU) together with an *uninterruptible power supply* (UPS) to prevent such situation.

Kernel Panic

This scenario is similar to the one before. If the kernel panics the only thing you can do is using the reset button which leads to the same situation as before, but this time neither a redundant PSU nor a UPS can help.

Device Failure

This is the kind of failure for which most replication schemes are designed for. The RAID schemes do not only increase the safety in these situation, but also the availability. Flashlog has the same safety, but no increased availability. Backups cannot prevent data loss completely, but at least reduce the damage in most cases.

Snapshots do not help for this kind of failure because the snapshots are on the same device as the data.

Controller Failure

This kind of failure is less common and normally less risky than the failures above. If all drives of an array are connected to the same failing controller the effect is the same as for the power failure. After replacing the controller everything is fine. If the devices are connected to different controllers than it can become more serious, because all devices connected to the failing controller would suddenly disappear. This can easily lead to a broken RAID array. In the best case the array is usable after controller replacement, in the worst case the data must be reconstructed manually which can get very expensive.

It is even worse, if the controller does not drop out completely but corrupts the data. In this case only non-realtime replication schemes can help. Backups on external media reduces the amount of data loss. Besides external drives are often connected to other controllers (e.g. USB), so that in case of Flashlog even writes initiated after the controller failure can be valid as long as they are not based on corrupted data from other drives.

CPU or RAM failure

CPU or RAM failures can normally not be detected by any replication scheme. These hardware parts use *error correction codes* (ECC) on their own to detect and correct failures. If this does not work all drives that has been used in the meantime are possibly corrupted and cannot be trusted. Checksums on application level can help to detect such problems and non-writable snapshots, backups and Flashlog's backup device can reduce the amount of data loss.

Accidental Deletion

If the problem is neither the hardware nor the software but the user most schemes cannot help. RAIDs replicate all operations, even if they were issued accidentally. Snapshots, Backups and Flashlog can again reduce the amount of data loss, but the best option would be being more careful.

Viruses, Trojan Horses

This problem is similar to the one above with the difference that this time the problems are caused intentionally. Snapshots which are stored on the same device as the data can get compromised. You are on the safe side if you have a backup

copy which is not accessible from the compromissed computer. Backups and Flashlog meet this demand.

Comparison & Results

The Flashlog scheme is not always the best choice, but it increases the safety in each case. Table 6.12 shows for which kind of failures each scheme can offer any additional protection.

The RAID scheme's only advantage is the increased availability in case of a device failure. For controller problems the scheme can be a solution, but it can also cause problems, if the array is damaged due to too many offline disks. If the controller corrupts the data the scheme detects the problems in the best case. In the worst case the corrupted data get replicated. The Backup scheme cannot fully protect the data, but it can always be used as a fallback. Flashlog combines this behaviour with the full protection in case of a device failure.

scheme	power supply	kernel panic	device	controller	CPU/RAM	user error
RAID	-	-	++	+/-	-	-
Backup	+	+	+	+	+	+
Snapshot	+	-	-	-	+	+
Flashlog	+	+	++	+	+	+

Figure 6.12: Summary of the quantitative analysis; ++ full protection, + partial protection, - no protection

6.3.2 Quantitative Analysis

The firstbsection describes the key data like kind of failure, failure rates, etc. and in the second part we compare the probability of data losses for the different schemes.

Assumptions

We can do this kind of analysis only for device failures because for other types of failures we do not have enough data (failure rates of controllers, CPU, RAM, cables, etc.) or the probabilities strongly depends on the user (how often do you get a virus or how often do you delete important data accidentally?). We need values for the *Mean Time To Failure* (MTTF) and *Mean Time To Repair* (MTTR) for the single devices. Those numbers can only be obtained by testing lots of disks over a longer period of time. We do not have enough devices and time and so we

just take typical values as stated by the manufacturers. These are the same values[2] as used in [15].

Device	MTTF
HDD	300,000h
USB flash drive	50,000h
SSD	2,000,000h

Figure 6.13: Assumed MTTFs

You should keep in mind, that these values are not very realistic as shown in [23], but they are the best we have and should be good enough to compare the schemes. Another interesting point in that paper is the fact that there has been no significant differences between high-end disks with *Small Computer System Interface* (SCSI) or *Fibre Channel* (FC) interface and much cheaper devices with *Serial Advanced Technology Attachment* (SATA) interface regarding their failure rates.

Calculation

$$
\begin{aligned}
n &:= \text{Usable Disk Space} \\
\lambda &:= \text{Failure Rate} \\
\mathrm{REPAIR}_{x,y} &:= \text{“disk x fails while disk y is in repair”} \\
\mathrm{FIRST}_{x} &:= \text{“first failure: device x”} \\
P(\mathrm{REPAIR}_{\mathrm{HDD,USB}}) &\approx \lambda_{\mathrm{HDD}} * MTTR_{\mathrm{USB}} \\
P(\mathrm{FIRST}_{\mathrm{HDD}}) &= \frac{\lambda_{\mathrm{HDD}}}{\lambda_{\mathrm{HDD}} + \lambda_{\mathrm{Log}}} \\
&= \frac{\mathrm{MTTF}_{\mathrm{Log}}}{\mathrm{MTTF}_{\mathrm{Log}} + \mathrm{MTTF}_{\mathrm{HDD}}}
\end{aligned}
$$

[2]The main reason for failures of USB flash drives is not the flash memory itself but the USB interface. According to the USB specifications a standard USB connector must function for at least 1500 plug/unplug cycles [24]. This limit is much earlier reached than the flash memories write limit if the USB flash device is plugged/unplugged regularly. Flash memory cards like Secure Digital (SD) or Compact Flash (CF) cards have often a specified MTTF of 1mio+ hours.

$$
\begin{aligned}
\lambda_{\text{RAID1}} &= (2n * \lambda_{\text{HDD}}) * P(\text{REPAIR}_{\text{HDD,HDD}}) \\
\Rightarrow MTTF_{\text{RAID1}} &= \frac{\text{MTTF}^2_{\text{HDD}}}{2n * \text{MTTR}_{\text{HDD}}} \\
\lambda_{\text{RAID5}} &= ((n+1) * \lambda_{\text{HDD}}) * (n * P(\text{REPAIR}_{\text{HDD,HDD}})) \\
\Rightarrow \text{MTTF}_{\text{RAID5}} &= \frac{\text{MTTF}^2_{\text{HDD}}}{n * (n+1) * \text{MTTR}_{\text{HDD}}} \\
\lambda_{\text{Flashlog}} &= (\lambda_{\text{HDD}} + \lambda_{\text{Log}}) \\
&* \ [P(\text{FIRST}_{\text{HDD}}) * P(\text{REPAIR}_{\text{Log,HDD}}) \\
&+ \ P(\text{FIRST}_{\text{Log}}) * P(\text{REPAIR}_{\text{HDD,Log}})] \\
\Rightarrow \text{MTTF}_{\text{Flashlog}} &= \frac{\text{MTTF}_{\text{Log}} * \text{MTTF}_{\text{HDD}}}{\text{MTTR}_{\text{Log}} + \text{MTTR}_{\text{HDD}}}
\end{aligned}
$$

Results

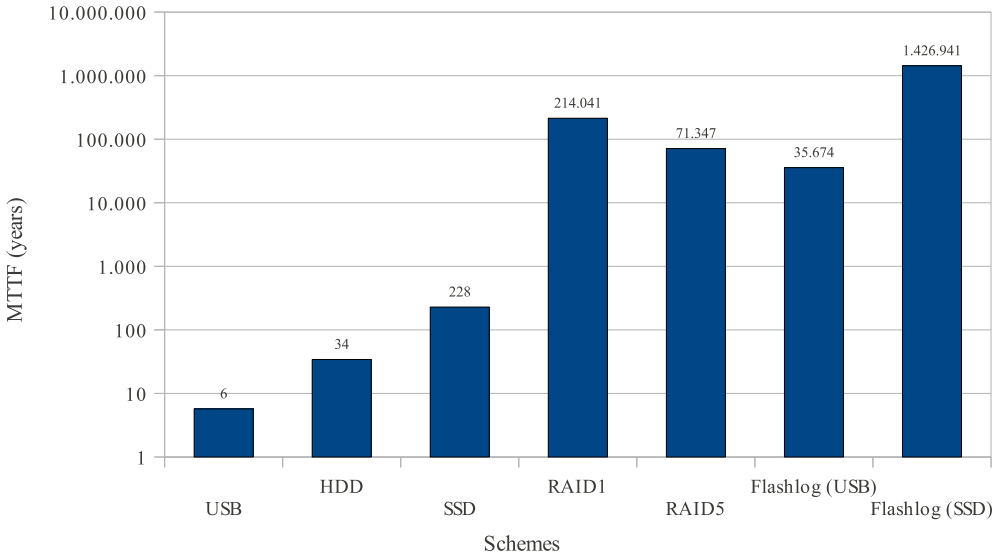

Figure 6.14: MTTF of the single drives, the Flashlog scheme using a SSD or USB flash drive as log device and the RAID levels 1 and 5

Looking at the numbers (Figure 6.14) you could think all these schemes provide more safety than anybody would ever need, but this not completely true. The problem is the way the manufacturers calculate the MTTF. They take lots of disks and count how many of them fail within a quite short time [11], but for most

customers the other way round (few disks, long time) would be the much better metric. Both methods would not differ, if the failure rate would be constant over time and according to the manufacturers this is true, at least for a "usual" time of usage[3], but as shown in [23] the failure rate increases even from the first year on. The average failure rate for 5 years old disks is more than eight times higher than stated. According to these numbers the realistic MTTF for these drives would be around 120,000 - 150,000 hours instead of 1-1.5 million if you consider the life-time of a single drive. The situation gets even worse if you use the device longer than the assumed 3-5 years.

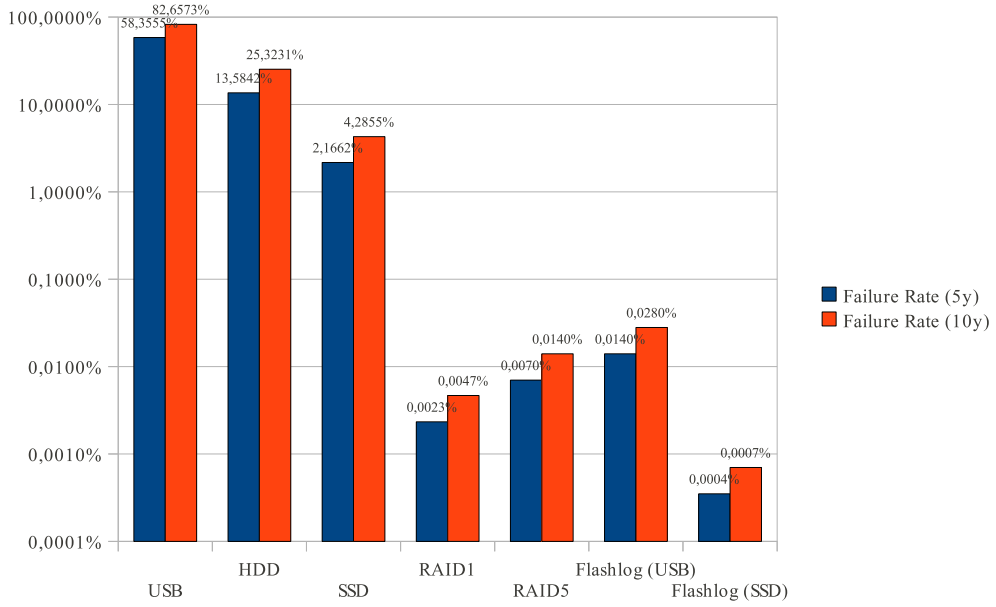

Figure 6.15: Probability that a device/array fails within 5y/10y

The resulting failure rates (Figure 6.15) are very low for all replication schemes, because two or more devices which should work for 300,000 hours must fail within a small timeframe (MTTR) to cause a failure of the array. But there is again some facts that are not considered for this calculation. The usual formulae assume that device failures are independant, but as shown in the above mentioned paper this is not true, e.g. in a RAID array the devices are stressed in the same way, the temperature and humidity are the same and the devices themselves are often from the same batch. Also the statiscal analysis in that paper shows that drive failures are not as independant as assumed.

[3]The *bathtub curve* is a often used model for the distribution of hardware failures over time. After an increased failure rate at the beginning (so called infant mortality), the model assumes a constant failure rate for the typical usage time. After that the failure rate grows due to aging effects

Summarizing we say that the failure rates are theoretically very low for all redundancy schemes, but considering the aspects mentioned above the safety is good enough for private users, but not much more.

6.4 Costs

In this section we calculated the costs for the schemes. We do not calculate the so called *total cost of ownership* (TCO) because these numbers would be more interesting for companies but not for our target audience, private users. Besides it is hard to quantify the costs for certain events, e.g. what is the monetary value for lost family photos or how much money do you lose if you cannot use your computer for some hours? Because of that we concentrate on the more obvious costs which are also more relevant for our target audience. This would be the costs for buying the necessary hardware, replacing it if it fails and energy costs. All costs are calculated for an effective capacity of 1TB. Furthermore we make assumptions as shown in Table 6.16.

price for 3.5" HDD 1TB	50.00 €
price for 3.5" HDD 500GB	35.00 €
price for 2.5" SSD 64GB	150.00 €
price for USB flash drive 16GB	25.00 €
price per 1kWh	0.20 €
energy consumption HDD	6W
energy consumption SSD	1.5W
energy consumption USB	0.2W
failure rate HDD	4.88%
failure rate SSD	0.75%
failure rate USB	25.92%
Power on Hours	15,000h
Power on Hours (backup drive)	1,500h

Figure 6.16: Assumptions regarding the hardware prices, energy consumption and prices, failure rates and usage time

Result

Most of the results (Table 6.17, Figure 6.18) are as expected. Normally you would expect RAID5 to be cheaper than RAID1 because its effective capacity is n-1 instead of n/2 (n = number of drives). But for our example the worse cost/size relation of the smaller drives used for RAID5 overcompensates its more efficient

scheme	initial cost	replacement cost	energy cost	total costs
HDD	50.00	2.44	18.00	70.44
RAID1	100.00	4.88	36.00	140.88
RAID5	105.00	5.12	54.00	164.12
Backup	100.00	2.69	19.80	122.49
Flashlog (USB)	125.00	9.17	20.40	154.57
Flashlog (SSD)	250.00	3.82	24.30	278.12

Figure 6.17: Costs in €

space usage. Also the replacement and energy costs are higher due to the number of needed drives. The Flashlog scheme with SSD cannot compete with the other schemes cost-wise due to the high costs for the SSD. But in other scenarios with more *Power on Hours* (PoH) the Flashlog schemes can reduce the difference to the RAID schemes due to lower replacement and energy costs. As mentioned before the high failure rate of the USB flash drives is mainly caused by their USB interface and other types of flash devices like *Compact Flash* (CF) cards are specified with much lower failure rates. Using such a device instead of the USB flash device reduces the replacement costs to one third.

We calculated the costs only for the desktop scenario, because the RAID variants are impractical for mobile usage, so that the only remaining schemes are Flashlog and Backup. Their difference cost-wise is only the USB flash drive and its energy and replacement costs.

6.5 Discussion

The Flashlog scheme's performance is comparable with the performance of a single HDD. The flash memory optimizations for the log device reduces the performance penalty of USB flash devices so that in some cases (random write with small / medium block sizes) the actual faster HDD is the limiting factor.

As the qualitative analysis has shown the Flashlog scheme provides additional safety in all cases. Using the log device provides as much safety as a RAID1 in the case that a device fails. In all other scenarios Flashlog offers at least the same protection as the Backup scheme. Its only disadvantage in comparison to the RAID schemes is its availability. If the main device fails Flashlog cannot keep running, but RAID schemes can. The quantitative analysis shows that Flashlog can be even safer than RAID1 because flash memory have much lower failure rates due to the lack of moving parts which are often the cause of drive failures. And you should not forget, if you lose two disks in a RAID5 everything is lost, if you lose the main and log device in a Flashlog array you still have the backup

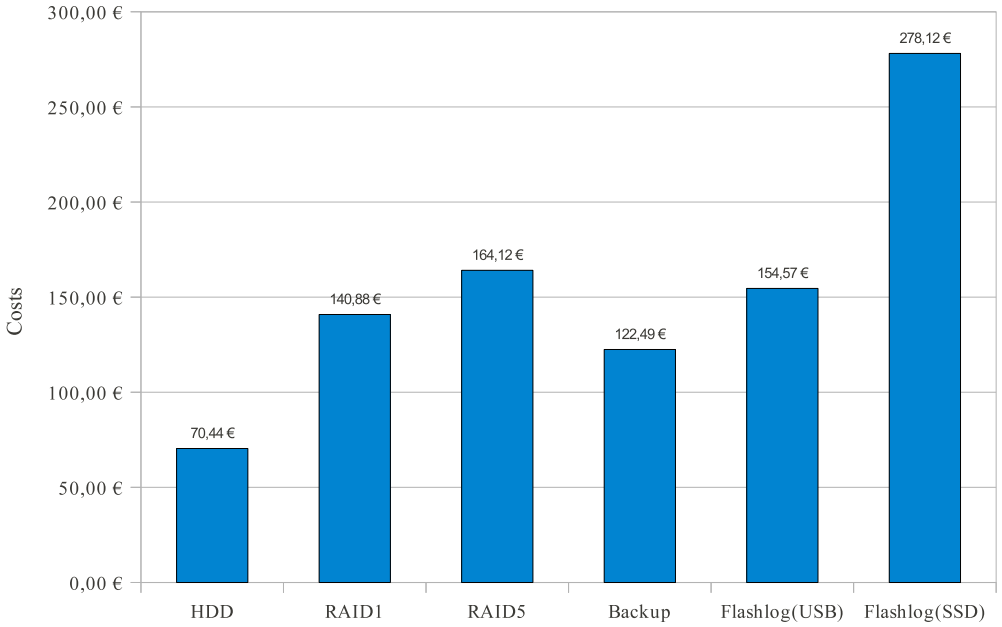

Figure 6.18: total costs of the redudancy schemes

device.

If performance is not so important you must pay only 10% more for the Flashlog scheme with USB flash device than for a RAID1. For smaller arrays it is even cheaper than RAID5 due to the better price per TB of the bigger disks. If you need more speed you can use a high speed flash memory card[4]. In contrast to the Backup or RAID schemes the costs for the Flashlog scheme do not only depend on the size of all data, but also on the amount of written blocks. Another point is the reduced energy consumption and the lower failure rate (assuming that a flash memory card is used instead of a USB device) of flash memory. These costs can add up to more than 50% of the initial hardware costs.

Besides all this there is one important aspect which has not been mentioned before in this chapter. For mobile scenarios one of the most important aspects is the form factor. Standard HDDs are impractical for this scenario, they are too big and too heavy. For this reason RAID arrays are too expensive because you would need a flash memory device with the capacity of your internal HDD. USB flash drives are available with capacities up to 256GB, but they cost more than average notebooks.

[4]Compact Flash cards are available with 90MB/s, CFast cards can even reach 200MB/s for reading and writing. Also there are some USB3.0 flash devices that are faster than common HDDs.

Chapter 7

Conclusion

The goal of this work was a replication scheme, that offers the same or even better safety than the existing schemes. It should have a performance comparable to a single HDD or better and it should not be much more expensive than a RAID1 array. Furthermore it should be usable in mobile scenarios. As explained in 6.5 all these goals have been reached. A reduced power consumption and performance improvements in some rare cases are just additional bonuses.

7.1 Future Work

During the implementation phase our main focus was on safety and data integrity. For the other aspects there is still some work to do.

Currently every error leads to a "shutdown" no matter what kind of error it is. This behaviour has been implemented to prevent any kind of data corruption after an error. In some cases this is stricter than necessary, e.g. if you try to initiate a backup before the backup device is plugged in or switched on. In other cases you have to balance pros and cons, e.g. if the log device fails you can either just stop logging and use the main device alone like a RAID1 does after a device failure or you can stop the array to reduce the probability of data losses. The current policy is good for testing but for the usage in the field a policy with finer granularity is required.

The copy daemons currently use synchronous I/O which leads to a bad performance, although we implemented a simple algorithm to merge multiple I/O requests if possible. But this algorithm can only merge I/O requests with contiguous source and destination sectors. Furthermore blocks that have been modified multiple times are also copied for each time they have been modified. An "intelligent copy" algorithm can analyse the log, remove I/O requests that would be overwritten later, reorder and merge I/O requests to get more continuous blocks.

Asynchronous I/O can be used to interleave multiple requests.

If a filesystem is mounted with the O_DIRECT flag (this is used to bypass the kernel's buffering mechanisms) on a Flashlog array the performance for small blocks is significantly worse than on a single HDD. This is not so important because O_DIRECT is normally used only by big databases or for benchmarking. Anyway this problem can be solved by implementing a small buffer for the I/O thread. This allows the I/O thread to write bigger blocks, even if their destination sectors are not contiguous, because of the log structure.

We have defined a disc header region, but currently the disc header is always written to the first sector. This is no big deal on a SSD due to their advanced wear leveling algorithms, but on a simple USB flash device this behaviour wears out the first sector early. Implementing an algorithm similar to the one used for the metadata and data area can solve this. Moreover the disc header is always written synchronously. This blocks the I/O thread until all previously issued write requests has been fulfilled. This is necessary to guarantee the coherence between disc header, metadata and data. A better implementation using write barriers can use asynchronous I/O and increase the I/O throughput.

Currently the asynchronous replication mode is not very useful, because every read forces a flush and resync. Making the log device readable can reduce the number of reads to the main device. It can also improve the performance if the smaller log device is faster than the main device (e.g. log=SSD, main=HDD). Implementing this would be a huge modification. The metadata must be kept in memory to find the data's sectors on the log device without reading the complete log. The problem is the granularity we currently use. We use a metadatum per I/O request. The amount of memory which is needed to keep all metadata in memory is far too much. The solution would be coarsen the granularity, but this requires additional reads from the main device to fill the gaps, because write requests are often smaller than the chunk sizes necessary to reduce the amount of metadata so that they can be kept in memory. Besides we would have to make the log entries modifiable which leads to a reduction of simpler flash memory device's durability if some sectors are modified frequently.

Bibliography

[1] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design tradeoffs for ssd performance. Technical report, Microsoft Research, Silicon Valley, 2008.

[2] Daniel P. Bovet and Marco Cesati. *Understanding the linux kernel*. O'Reilly, 2003.

[3] Milan Broz. Device mapper — (kernel part of lvm2 volume management), April 2008.

[4] btrfs Wiki. btrfs wiki - project ideas.

[5] Jonathan Corbet. FUSE - implementing filesystems in user space. *lwn.net*, January 2004.

[6] Jonathan Corbet. Barriers and journaling filesystems. *lwn.net*, May 2008.

[7] George Di Falco. What is continuous data protection?, July 2007.

[8] Fabian Franz. Dm-relay - safe laptop mode via linux device mapper. Study thesis, System Architecture Group, University of Karlsruhe, Germany, April 20 2010.

[9] Eran Gal and Sivan Toledo. Algorithms and data structures for flash memories. *ACM Journal*, 2005.

[10] Neeta Garimella. Understanding and exploiting snapshot technology for data protection, part 1: Snapshot technology overview. *IBM developerWorks*, April 2006.

[11] Gordon F. Hughes and Joseph F. Murray. Reliability and security of raid storage systems and d2d archives using sata disk drives. *ACM Transactions on Storage*, 1(1):95–107, December 2004.

[12] Tim Jones. Anatomy of the linux file system. *IBM developerWorks*, October 2007.

[13] Poul-Henning Kamp. Geom - disk handling in freebsd 5.x.

[14] Aljoscha Krettek. Version control systems, May 2010.

[15] Vamsi Kundeti and John A. Chandy. Fearless: Flash enabled active replication of low end survivable storage. In *Proceedings of 1st (ACM ASPLOS-09) satellite Workshop on Integrating Solid-state Memory into the Storage Hierarchy (WISH)*, March 2009.

[16] LaCie. Raid technology white paper, oct 2008.

[17] Krzysztof Lichota. Block devices and volume management in linux.

[18] Sun Microsystems. Zfs - eine neue generation.

[19] David A. Patterson, Garth A. Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (raid). Technical Report UCB/CSD-87-391, EECS Department, University of California, Berkeley, Dec 1987.

[20] Aditya Rajgarhia and Ashish Gehani. Performance and extension of user space file systems. In *SAC '10: Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 206–213, New York, NY, USA, 2010. ACM.

[21] Chris Ruemmler and John Wilkes. A trace-driven analysis of disk working set sizes, 1993.

[22] Samsung. Page program addressing for mlc nand application note, November 2009.

[23] Bianca Schroeder and Garth A. Gibson. Disk failures in the real world: What does an mttf of 1,000,000 hours mean to you? In *5th USENIX Conference on File and Storage Technologies*, 2007.

[24] USB Implementers Forum, Inc. Universal serial bus cables and connectorsclass document revision 2.0, August 2007.