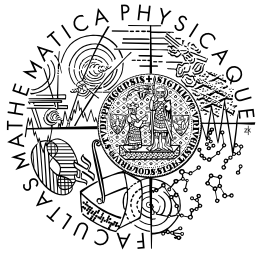

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Miroslav Cimerman

Software RAID for HelenOS

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: Mgr. Vojtěch Horký, Ph.D.
Study programme: Computer Science

Prague 2025

I declare that I carried out this bachelor thesis on my own, and only with the cited sources, literature and other professional sources. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

To my beloved grandfather,
whose wisdom guides and inspires me every day.

This thesis is also dedicated to my brother and parents, with immense gratitude for their love, support and always being there for me.

I am also deeply grateful to my grandmother for providing a peaceful and comfortable study environment, for her love, kindness and generosity.

To the rest of my family – thank you for your belief in me.

I truly appreciate my dear friends – Jardo, Michael, Emanuel, Matúš, Samuel, Aurel and Šimon, for their support, humor, help, adventures and late night discussions.

Special thanks to my supervisor, Mgr. Vojtěch Horký, Ph.D., for suggesting this topic, which sparked a passion in me for storage systems, and for the trust he placed in me to shape this project according to my own interests and curiosity. His invaluable time invested, consultations, and advice provided whenever needed, greatly contributed to the success of this work.

I also want to thank Jiří Svoboda for his time, advice, feedback and organizing enjoyable HelenOS meetings.

Title: Software RAID for HelenOS

Author: Miroslav Cimerman

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Vojtěch Horký, Ph.D., Department of Distributed and Dependable Systems

Abstract: Redundant Array of Independent Disks (RAID) is a technique used to expose multiple physical storage devices as a single virtual disk that has better performance properties and provides fault-tolerance. The goal of this bachelor thesis is to implement software RAID for the HelenOS microkernel operating system with support for RAID metadata formats of other operating systems, such as OpenBSD, FreeBSD, or Linux. The thesis contains an analysis of software RAID challenges and presents a RAID implementation with support for RAID levels 0, 1, 4 and 5, written from scratch with emphasis on I/O performance. We evaluated the performance of our RAID implementation on up to 33 enterprise-class hard drives and showed that its performance is comparable to RAID implementations in mature monolithic kernel-based operating systems.

Keywords: RAID, software RAID, HelenOS, block device, storage, I/O

Název práce: Softwarový RAID pro HelenOS

Autor: Miroslav Cimerman

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Vojtěch Horký, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: Redundant Array of Independent Disks (RAID) je technika používaná k prezentování více fyzických disků jako jednoho virtuálního, který má lepší výkonostní vlastnosti a poskytuje odolnost proti selháním jednotlivých disků. Cílem této bakalářské práce je implementovat softwarový RAID pro mikrokernelový operační systém HelenOS s podporou metadat RAID implementací operačních systémů, jako jsou OpenBSD, FreeBSD nebo Linux. Práce obsahuje analýzu problémů spojených se softwarovým RAIDem a prezentuje vlastní implementaci s podporou úrovní RAID 0, 1, 4 a 5. Tato implementace byla vytvořena od základů, s důrazem na I/O výkon. Výkonnostní testy, provedené až na 33 enterprise-class discích, ukazují, že dosažený výkon je srovnatelný s RAID implementacemi ve vyspělých operačních systémech s monolitickým jádrem.

Klíčová slova: RAID, software RAID, HelenOS, blokové zařízení, úložiště, I/O

Contents

Introduction

In many computer systems, data storage is critical. Some systems require data to be reliably stored, other systems require highly performant storage stack, and some require both. In order to reliably and efficiently store data, a technique called *RAID*, short for *Redundant Array of Independent Disks*, (in the past called *Redundant Array of Inexpensive Disks* [1]), is used.

RAID can be implemented in hardware or software, but in both instances, RAID exposes multiple drives as a single device with better properties compared to those of a single drive. These enhanced properties are achieved by using mirroring, striping, and storing redundant information about the data, such as its parity.

Goals

The goal of this thesis is to extend HelenOS [2], a microkernel-based operating system that supports various file systems and block devices, with a software RAID implementation, pushing the fault-tolerant nature of HelenOS due to the microkernel design, further to its storage stack.

Besides the obvious goal of providing a service that exposes multiple devices as one using standard RAID techniques, the other main goals are as follows.

- Support RAID metadata from other software RAID implementations, such as OpenBSD's `softraid(4)`, FreeBSD's GEOM RAID classes, and Linux's MD RAID.
- Analyze the impact of the RAID implementation on other HelenOS components.
- Evaluate the performance impact of the RAID implementation on system performance.

Chapter organization

The following is a brief outline of the text.

- Chapter **1** includes an introduction to RAID levels and parts of HelenOS relevant for block device development.
- Chapter **2** includes an analysis of software RAID challenges, evaluation of key RAID implementation features, and a description of integration of a RAID implementation into HelenOS. This chapter also includes the analysis of metadata properties and the specific foreign (external) metadata formats that shall be supported.
- Chapter **3** includes the design and implementation details. This chapter servers as the implementation documentation.
- Chapter **4** includes the description of unit tests designed to test RAID volumes, foreign metadata tests, and I/O performance benchmarking of our RAID implementation.

1 Context

Even though RAID levels and their properties are generally known, in this chapter we will show a short overview, also for the sake of terminology unification, as it greatly varies between different papers, books, implementations and appliances.

We will also show an overview of HelenOS parts relevant for the development of block devices and its storage stack.

1.1 RAID levels overview

RAID terminology used throughout the text will be gradually explained in this section. The most basic terms are as follows.

- *Volume* is the virtual device provided by the RAID abstraction.
- *Extent* is any device that is part of a volume.
- N will denote the number of extents that a volume consists of.
- C_{extent} will denote the capacity of a single extent.

We are going to make claims and present level properties based on few assumptions, which include the following.

- We assume that the underlying system has the capability to process requests made to independent extents in parallel.
- We assume that the I/O completion time is linear with respect to the I/O size.[1]
- We assume that no concurrent and other in-flight I/O requests are made to the volume or its extents at the time of describing performance properties if not stated otherwise. This also includes that all extents are equally ready to process incoming requests.
- In some cases, we also assume that read and write I/O have the same latency.

There are more RAID levels than those discussed in this chapter, but we will focus only on the most commonly used ones, which are also going to be implemented.

1.1.1 Striping

Although not providing any redundancy, striping is also known as RAID level 0.

[1] This is generally not true, e.g., 4 KiB write I/O can have the same latency as 32 KiB write I/O, with possible reasons being call overhead or DMA. But for simplicity, let us assume that the completion time is linear with respect to the size of the I/O.

Mapping

In this level, *logical block addresses*[2] are grouped on the same extent by *strips* and laid out across extents forming a *stripe*. Example mapping with a strip size of 2 blocks in Figure 1.1.

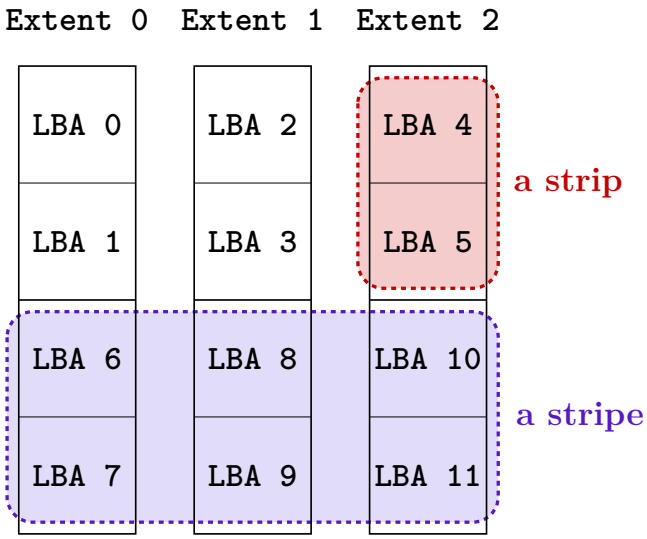

Figure 1.1 LBA mapping of 3-extent RAID 0 volume with a strip size of 2 blocks

Fault-tolerance

Because this level does not provide redundancy, failure of any of the extents makes the volume unusable, thus making it zero fault tolerant.

Capacity

All blocks are used for user data, therefore the capacity of this level is $N \cdot C_{extent}$.

Performance

Performance depends on strip size and can vary between different workloads, but I/O spanning multiple strips can theoretically achieve a speed-up of factor N for both reads and writes.

To illustrate the possible speed-up, for example, suppose that we have a 2-extent RAID 0 volume with a strip size of 32 KiB. When 64 KiB sequential I/O comes to the beginning of a stripe, it is split into 2 internal 32 KiB I/Os, which can be sent to both extents in parallel. When 32 KiB I/O takes half the time to complete as 64 KiB I/O, the total time it takes to complete the original 64 KiB I/O is also halved.

[2] *LBA*s for short, are used by a linear addressing scheme to address data blocks in storage devices [3]

1.1.2 Mirroring

Also known as RAID level 1, this level stores a copy of each block across all extents. Some specifications such as SNIA's[3] *Common RAID Disk Data Format Specification* [4], specify RAID level 1 volume to have exactly 2 (*RAID-1 Simple Mirroring*), or 3 extents (*RAID-1 Multi Mirroring*), but we generalize the concept of mirroring to arbitrary $N \geq 2$, as do other software RAID implementations. We will use the term N-way mirror to depict a mirroring volume consisting of N extents.

Mapping

Block addresses are linear per extent and are the same across all extents. All extents have the same LBA in the same row. Mapping in Figure 1.2

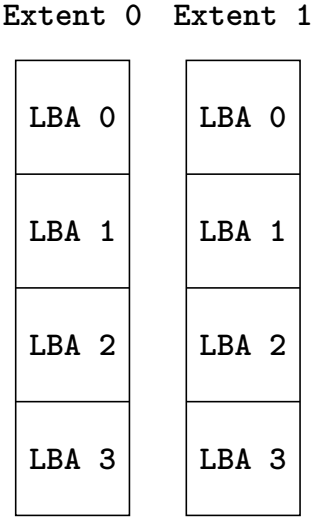

Figure 1.2 LBA mapping of 2-way RAID 1 volume

Fault-tolerance

N-way mirror stores N copies of each block on distinct extents, allowing up to $N - 1$ extents to fail.

Capacity

The capacity overhead is 100%, meaning that we effectively use just the size of a single extent (C_{extent}).

Performance

Unlike in RAID 0, the performance of reads and writes vastly differs because they are handled differently.

To keep the mirror invariant, if a write I/O comes, the volume must replicate that request for each extent. For a user write I/O to be completed, it takes all

[3]Storage Networking Industry Association

internal write I/Os to be completed. The total time it takes to complete the user I/O is the time it takes the slowest extent to complete its internal I/O, if we do not count the RAID overhead[4]. Suppose that we have homogeneous extents with the same characteristics and properties, then the time it takes a user write to complete is equal to that of a single extent.

Read performance can be theoretically the same as in a RAID 0 volume, but that depends on the implementation. For example, `gmirror(8)` in FreeBSD allows different read techniques to be configured and used. Such as splitting - doing the same thing RAID 0 does, or selecting the extent to read from based on load, or in a round-robin fashion.

1.1.3 Parity

There are numerous schemes to store data parity. They differ in the granularity of the data on which they compute the parity. RAID levels 2 and 3 use bit-level striping [5] and byte-level striping [6], respectively. These levels are also mentioned in the original RAID paper [1]. However, none of these 2 RAID levels is implemented inside OpenBSD's `softraid(4)` or in the Linux kernel's `md(4)`. FreeBSD has a RAID 3 implementation accessible by `graid3(8)`. However, we will not focus on these bit- and byte-striped levels. Instead, we will focus on more widely used block-level striped RAID levels.

Block-level striped, parity-storing RAID levels can be further categorized based on the parity block positions and number of parity blocks stored per single set of data blocks.

There are parity schemes and levels such as RAID 6, using cyclic groups [7], NetApps's [5] nonstandard RAID-DP [8], ZFS's RAIDZ2 tolerating 2, or even 3 (RAIDZ3 [9]) failed extents. But due to the complexity of the levels and schemes mentioned above, we will focus only on single-parity RAID levels, which include RAID 4 and RAID 5.

To emphasize the parity, parity-storing RAID volume configurations are denoted as $N = D + P$, where D is the number of data extents and P is the number of parity extents. In our case, we are always going to use $N = D + 1$ extents, as we focus only on schemes that store a single parity. The smallest configuration is $2 + 1$, thus making the smallest $N = 3$.

Parity computation

Parity is computed as a *XOR* (bit-wise eXclusive-OR, denoted by $\oplus$) of data in the same row. In any stripe, the parity block is computed as the XOR of all the data strips. We can write the parity in the stripe x as $P_x = D_{x,0} \oplus \cdots \oplus D_{x,N-1}$, where $D_{x,y}$ denotes the data strip in the stripe x, on the extent number y.

Mapping

RAID 4 and RAID 5 use striping as RAID 0 does, but in addition they store parity blocks. There are multiple *layouts* in which these parity blocks can be

[4]This is vaguely used term, but in this context it means the splitting or the duplication of the I/O itself.

[5]NetApp, Inc.

organized, and we will use the layout terms from SNIA's Common RAID Disk Data Format Specification, where layouts are referred to as *RAID Level Qualifiers* [4].

RAID 4 stores parity on a dedicated, so called parity extent. The dedicated extent can be the first or the last extent, *Non-Rotating Parity 0*, and *Non-Rotating Parity N*, respectively. For simpler visualization of data and parity strips, we use a single-block strip size.

An example mapping of a $2+1$ RAID 4 volume, with a strip size of one block, and parity stored on the last extent, can be found in Figure 1.3.

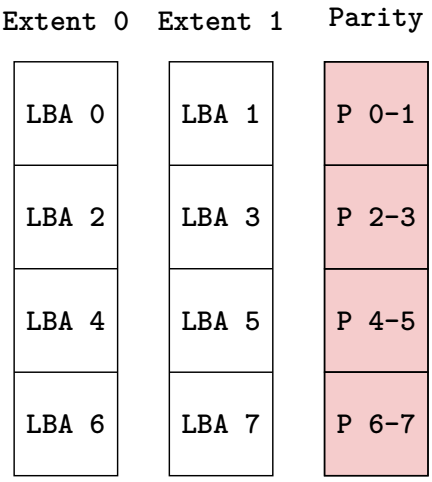

Figure 1.3 RAID 4 volume with parity stored on the last extent

RAID 5 distributes parity across extents. There are numerous layouts in which parity can be distributed.

- Diagonally from left to right:
  - *Rotating Parity 0 with Data Restart*, also known as *right-asymmetric* layout
- Diagonally from right to left:
  - *Rotating Parity N with Data Restart*, also known as *left-asymmetric* layout
  - *Rotating Parity N with Data Continuation*, also known as *left-symmetric* layout

Where Data Restart means that the first data strip of a stripe is placed on the first extent not occupied by a parity strip. Data continuation means that the first data strip of a stripe starts below the parity strip of the previous stripe.

Example mapping of $2+1$ RAID 5 volume with strip size of one block, with Rotating Parity N with Data Restart layout can be found in Figure 1.4, and with Rotating Parity N with Data Continuation layout in Figure 1.5.

Performance implications of dedicated and distributed parity are discussed in a later subsection.

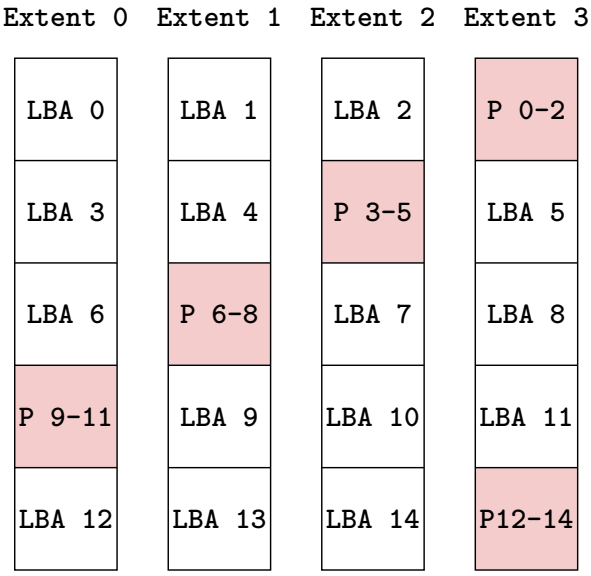

Figure 1.4 RAID 5 Rotating Parity N with Data Restart layout

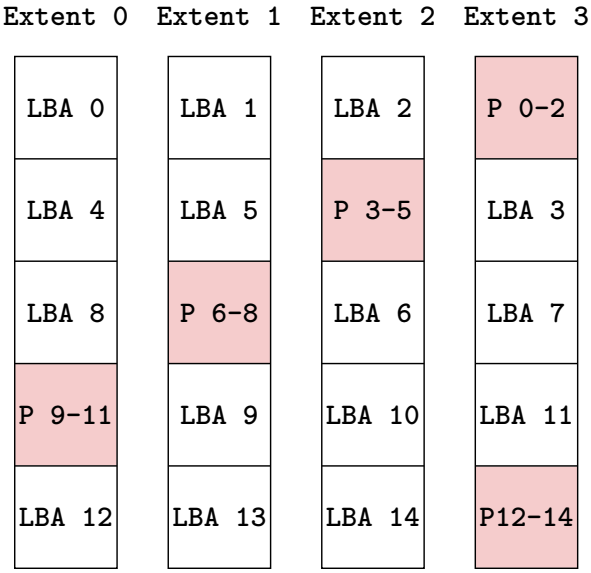

Figure 1.5 RAID 5 with Rotating Parity N with Data Continuation layout

Fault-tolerance

A $D + P$ volume can tolerate exactly P failed extents. When the number of unusable extents is less than or equal to P, the volume is still usable, and we say that it is in a *degraded* state.

In RAID 4 and RAID 5 we only have one parity strip per stripe and therefore when a volume is in a degraded state, in any specific stripe we can either have a failed parity strip or one failed data strip.

Suppose that we have parities of the data in each stripe of a $2 + 1$ volume, $P_x = D_{x,e1} \oplus D_{x,e2}$. And we want to read the data strip $D_{15,e1}$, but one of the extents is unusable.

- Parity strip is unusable. In this scenario, we can retrieve the data as we would in *optimal* state.

- Extent $e2$ is unusable. In this scenario, we can also retrieve the data in a normal way, since extent $e1$ is operating.
- Extent $e1$ is unusable. In this scenario, we must reconstruct the wanted data block from the other extent and the parity. Due to the fact that XOR is self-inverse and commutative, we can write $D_{15,e1} = D_{15,e2} \oplus P_{15}$, which gives us reconstructed data on extent $e1$.

Capacity

From Figure 1.3, which shows a RAID 4 volume, it is easy to see why a RAID 4 volume has the capacity of $(N-1) \cdot C_{extent}$. The same applies to a RAID 5 volume.

Performance

To assess the performance of RAID 4 and 5, a more in-depth breakdown of possible I/O requests is needed. As there are numerous configurations and volume states, various I/O patterns, types, and the amount of concurrent I/O, in this section we will only hint at what performance can theoretically be expected from certain types of I/O. This brief analysis is not exhaustive.

Read I/O

Reads are processed in the same way as in RAID 0. When a read spans multiple extents, $(N-1)$-fold speed-up is theoretically achievable in RAID 4, and RAID 5 with Data Restart layouts. Data Continuation layout generally allows more concurrent read I/O processing, as more extents can be utilized in parallel. Suppose a read request on `LBA 0, 1, 2, 3`. From Figure 1.4 and Figure 1.5 it is easy to see why such I/O could be theoretically faster in a volume with Data Continuation layout.

As was already mentioned, in the parity strip-degraded state, read I/O is handled the same way as in optimal state, as no data has to be reconstructed. However, if one of the data strips to which the I/O is mapped is unusable, that data has to be reconstructed from all other data strips and the parity strip in the affected row[6]. This need of reconstructing data does not theoretically impose any additional I/O latency (except the XOR computation), as the internal reconstruct requests can be done in parallel.

Write I/O

Each access modifying the data must keep the parity invariant, therefore the parity must be recomputed for each write.

Let us assume just small write I/O and large sequential write I/O.

We are going to use terminology from the *RAIDframe* reseach project[10] to distinguish between different types of write requests and the appropriate operations the volume has to execute for them.

[6]We refer to data strips and parity strip because it is independent of the RAID level and layout used.

- Sequential writes. When I/O that completely spans all strips in a stripe is called a *full-stripe write*, also known as a *large write*. When such write comes to the volume, parity can be computed from memory, and all the new data strips and parity strip can be written in parallel. Such writes also do not pose extra latency for request completion, as they are almost identical to non-redundant RAID 0 striped writes, except they additionally need to compute and write the parity. With today's powerful processors, parity computation is negligible compared with I/O to external devices. For an example volume in Figure 1.4, full-stripe write would be one writing data to `LBA 6, 7` and `8`. Full-stripe writes are the most efficient writes that RAID 4 and RAID 5 volumes can encounter.

- Small and *partial-stripe* writes. First, for Figure 1.4 suppose that a partial-stripe write comes to block addresses `0` and `1`. To keep the parity invariant, we must also retrieve data from `LBA 2` and then compute the final parity as XOR of the in-memory blocks targeted for `LBA 0, 1` and the retrieved block from address `2`. This doubles the latency because in the first phase we first need to retrieve all data blocks to have the correct parity, and just then we can write to the data strips and also write the parity in the second phase. Why the reads that complete the row (in our example read to `LBA 2`) must be finished first and writers can start after their completion is discussed in later chapters. This type of write operation is known as *reconstruct-write*, because we must reconstruct the data for parity update in a similar fashion to reconstructing data in the degraded read case.

  For the next example case suppose we have $7 + 1$ volume with single-block strip size, where in the first stripe the parity is stored on the last extent, so the first stripe starts with data blocks that have block addresses from 0 to 6 and then the parity block. Suppose that a write I/O came to block address 0. Using the technique mentioned above, 8 internal I/Os would be generated in total:

  - 6 reads for other data blocks
  - 1 write for block address 0
  - 1 write for the parity update

  There is another method for more efficient parity computation and update. To update the parity we just need to read old data that are about to be rewritten, and the old parity. The new parity is then computed as the new data, old data and the old parity. This works because in any row:

$$P = D_0 \oplus D_1 \oplus D_2 \oplus D_3 \oplus D_4 \oplus D_5 \oplus D_6$$

  After writing new data $D_{0,new}$ and the parity update, this shall hold:

$$P_{new} = D_{0,new} \oplus D_1 \oplus D_2 \oplus D_3 \oplus D_4 \oplus D_5 \oplus D_6$$

  After combining both equations together we get:

$$P_{new} \oplus P = (D_{0,new} \oplus D_1 \oplus \cdots \oplus D_6) \oplus (D_0 \oplus D_1 \oplus \cdots \oplus D_6)$$

Which we can simplify thanks to the XOR properties into:

$$\begin{aligned} P_{new} \oplus P &= D_{0,new} \oplus D_0 \\ P_{new} &= D_{0,new} \oplus D_0 \oplus P \end{aligned}$$

That is the operation described above. This operation is known as *subtract-write* or *read-modify-write*. Subtract-writes also have twice the latency of full-stripe write, same as in the reconstruct-write case. The advantage is that in our example, this operation generates only 4 internal I/Os in total:

- 1 read for old data
- 1 read for old parity
- 1 write for new data
- 1 write for new parity

This is especially useful when multiple concurrent I/O is on the volume, and extents on which un-touched data blocks reside are not affected by these read-modify-writes.

To know when it is more beneficial to use reconstruct-writes and when to use subtract-writes, this simple condition can be used. If the write I/O touches less than half of data strips in a stripe, subtract-write is better to use, and when between half and all data strips are touched, reconstruct-write is better to use.[7] [10]

Last thing to note is that small concurrent writes on a RAID 4 volume can potentially take big performance hit due to the parity not being distributed. For the sample volume in Figure 1.3, suppose two concurrent read-modify-writes, one to block address `1` and the other to block address `4`. Both I/Os stress the same parity extent.

The last thing that is important to show is how writes operate in a degraded-state volume. With parity-degraded stripe, writes going to data blocks are just redundantly processed, in the same way as in RAID 0. With data strip-degraded stripe, two cases can arise. The first is when we write to a valid data strip and another data strip is unusable. In this case, we must use the read-modify-write operation, as there is no way to read the data from the unusable strip. The second is the opposite, that being a write to an unusable data strip. In this case we must use reconstruct-write (because

[7] Author's hypothesis: This more coarse-grained condition based on number of strips touched and not the % of data written in a stripe which is in favor of reconstruct-writes does not affect spinning disk performance that much, because when an I/O affects only a part of one of the strips, after writing the data, read a instantly follows, therefore there might be no seek penalty. Condition taken based on the % of a stripe written might be more favorable for solid-state drives. But for big enough strip sizes, hard-disk drives might also be able to profit from this.

there is no efficient way of reading the old data), with the difference that the write to the unusable strip is a *no-op*[8].

1.1.4 Combined RAID levels

RAID levels can be naturally stacked on top of each other. This can be useful to partly get the best of both levels involved. It is common to stripe mirrors, which means that there is RAID 0 on top of two RAID 1 mirrors, or the other way around, mirror on top of two non-redundantly striped volumes, these are referred to as RAID 10, and RAID 01, respectively, also known as RAID 1+0, and RAID 0+1, respectively. There is also RAID 50 (RAID 5+0), which stripes across multiple RAID 5 volumes.

However, we will not focus on these combined RAID levels.

1.2 HelenOS

HelenOS [2] follows a microkernel-based design, meaning that the kernel handles only the most essential facilities needed for core system functionalities, namely memory management, scheduling, and *Inter-Process Communication* (*IPC*). Everything else runs inside the userspace. One of the advantages of this design is that the kernel size is tiny compared to monolithic kernels. This allows the kernel to be inspected for bugs more easily and even be subjected to formal verification [11], giving systems a trusted computing base. Other system properties that a microkernel design contributes to, include flexibility, security, reliability, and fault tolerance.

Every running instance of a user program is called a *task*. Tasks can be traditional user programs such as a shell, or a so called *servers*, that provide various services such as file systems, block device drivers, network drivers, or audio drivers.

1.2.1 Scheduling and synchronization

Each task consists of one or more *threads*, that are independently scheduled by the kernel in a preemptive manor. Every thread running in userspace is executed by the means of a lightweight execution unit called a *fibril*. The kernel schedules only threads and is completely unaware of fibrils.

Fibrils are cooperatively scheduled by the standard library and will run until completion or when one of the following occurs.

- They explicitly yield the processor to another fibril.
- They wait for an IPC reply that has not arrived yet.
- They request an IPC operation which results in the underlying thread being blocked.
- The underlying thread is preempted by the kernel.

[8] *No-op* means no-operation. In this context, it means that no write is actually issued to the unusable strip.

To exploit potential concurrency and utilize parallelism safely and effectively, synchronization of the execution units is necessary.

To synchronize kernel threads, the kernel implements various synchronization primitives. An active synchronization primitive, a *spinlock*. And passive synchronization primitives, which include *wait queues*, *semaphores*, *mutexes*, *read/write locks* and *conditional variables*.

Since all userspace threads are executed by the means of fibrils, there is a need to synchronize those as well. The standard library provides traditional synchronization primitives, such as the already mentioned mutexes, semaphores, conditional variables, and read/write locks, which can be used to synchronize fibrils inside userspace tasks. The standard library implements them via *fast mutexes*, known as *futexes* [12]. To note, read/write locks do not enable writer starvation.

1.2.2 IPC

All tasks run in a separate address space, which means that one task cannot access another task's address space, and there is a need for inter-task communication[9].

HelenOS implements an asynchronous IPC based on a *phone* and an *answerbox* principle. Each task has an answerbox and initially one open phone. The initial phone is connected to the *Naming service*. The Naming service is a system task that registers services and connects clients to already registered services. One task can only communicate with another task with which it has an open phone. When a task's fibril has a phone already connected to another task's answerbox, it can make a *call*. The caller fibril can wait for an answer, do some work or make another call. The call is put into the callee task's incoming call queue, and one of its fibrils eventually processes the call, and either answers it or forwards it to another answerbox. Eventually, the call is answered, and the answer stored inside the answerbox of the original caller task.

The IPC mechanism provides the following features.

- Short calls, consisting of one argument for the method number and five arguments of payload.
- Answers, consisting of one argument for the return code and five arguments of payload.
- Sending large data to another task.
- Receiving large data from another task.
- Interrupt notifications for userspace device drivers.

Other feature allowing to share memory to or from another task is provided as well, but is irrelevant for our use.

At the lowest level, each call is a data structure that accommodates six native arguments. The first argument of the six is interpreted as a method number for requests and as a return code for answers. Method is either a system method

[9]Even though HelenOS does not have processes, the inter-task communication mechanism is referred to as IPC.

or a protocol-defined method. System method numbers range from 0 to 1023, protocol-defined method numbers start at 1024. In the case of system methods, the payload arguments has a predefined meaning and is interpreted by the kernel. In the case of protocol-defined methods, the payload arguments are defined by the protocol (service) in use.

Even though a user space task can use the low-level IPC mechanisms and system methods directly, it is not necessary as easy to use asynchronous framework is provided. The framework calls all the necessary low-level IPC system methods.

As our main focus is on the storage stack, we are going to be dealing with a lot of large data transfers made between tasks. The current maximum IPC data transfer size is 64 KiB[10]. When task A wants to do large data transfer via IPC with task B, independently of data direction, a form of "handshake" must complete between both tasks. In the first phase, task A sends task B a message with the type of request that A wants to make (data direction and size of the data to be transferred). In the second phase, B receives the request, and in the final phase B can accept and proceed with the request or reject it. When such a request is accepted, memory copying mechanism almost identical to traditional `memcpy()`[11] is used, to copy memory to/from userspace.

Information about IPC and synchronization was largely put together from *IPC for Dummies* [13] and *HelenOS 0.2.0 design documentation* [14] online resources, where a more detailed description of the mentioned mechanisms can be found.

1.2.3 Storage stack

There are various file systems supported in HelenOS, which users can use to access or modify data on storage media. The following file systems are supported.

- ext4
- FAT
- exFAT
- The Compact Disc File System (CDFS)
- Universal Disk Format (UDF)
- The Minix file system

HelenOS also supports a memory-based file system, *tmpfs*, but this file system cannot be used for data storage, as all contents are lost with each system reboot. Another special file system is the *Location Service File System*, in which every file represents a service registered with the Location Service. The Location Service allows servers that provide a service and register it with human-friendly name and make it visible in the file tree. Then the registered service can be easily specified or accessed with user tools.

[10] `<abi/ipc/ipc.h>:DATA_XFER_LIMIT`

[11] amd64 `memcpy_{to,from}_uspace` functions

VFS

To unify and make work with multiple file systems easier, HelenOS implements *VFS*, the Virtual File System, which serves as a frontend to the actual file system implementations. It abstracts the specific operations and data structures used by different file systems and provides a single interface. Contrary to traditional POSIX system calls used to manipulate files, HelenOS uses own `vfs_open()`, `vfs_read()`, `vfs_write()`, and other library functions, to provide similar functionality. Nonetheless, HelenOS also provides a POSIX compatibility layer, which provides the traditional `open()`, `read()`, `write()`, ..., and wraps them to use the HelenOS VFS interface. Since HelenOS is a microkernel, VFS is a standalone server that provides the mentioned file manipulation with its library interface. The VFS is split into *VFS frontend*, and *VFS backend*. The frontend accepts client requests. Since VFS uses its own representation of files known as VFS nodes, some requests can be completed by the VFS frotend itself, like `lseek()`, which just changes the current position pointer that VFS stores in its internal structure. When the VFS receives a request that cannot be fulfilled by the frontend, VFS part that is referred to as backend is used. The backend is used to communicate with the file system server that owns the corresponding file, and the user requests are forwarded to that file system server. File system servers implement *libfs* operations, that represent the actual mapping of the abstract VFS operations to the specific file system operations. [15]

Block layer library

Any request a file system receives that reads or manipulates file data or on-disk metadata, the underlying device on which the file system is mounted on, must be accessed. Like in the case of most operating systems, HelenOS provides a way to interact with *block devices*. Block devices are devices that can be accesses or modified only with request sizes equal to the multiples of their (logical) *block size*[12]. Storage block devices, such as hard disk drives or SD cards, have a block size of 512 bytes, while for solid-state drives, a block size of 4096 bytes (4 KiB) is more common[13]. Block device operations are abstracted and used in a similar way to VFS and libfs. Each block device is registered as a service that implements a `struct bd_ops` interface, additionally typed `bd_ops_t` that includes the following.

- `get_block_size()`, used for getting the device's block size.
- `get_num_blocks()`, used for getting the device size in blocks.
- `read_blocks()`, used for reading blocks into destination buffer.
- `write_blocks()`, used for writing blocks from memory onto the device.
- `sync_cache()`, used for synchronizing the cache, if the device supports it.

Clients can issue requests to block device services with the block device client interface included in the `libdevice`, a device library that includes code to work

[12]Also known as *sector size*.
[13]Although 512 B block size SSDs and 4 KiB block size HDDs exist.

with various devices. However, HelenOS also provides a library called `libblock`, which wraps the block device client interface into one that is more comfortable to use and features additional functionality. The `libblock` interface can be used in two ways.

- Direct I/O. This includes `block_read_direct()` and `block_write_direct()` functions, which are basically just wrappers for the block device server client interface, `read_blocks()` and `write_blocks()`, respectively. This can be seen as a similar behavior `read()` and `write()` syscalls exhibit on Linux if the raw block device was opened with the `O_DIRECT` flag.
- Cache utilizing I/O. Functions `block_get()` and `block_put()` can be used to work with blocks materialized in the memory in order to increase the performance and possibly lower request latency. This way of accessing block devices is predominantly used by file systems.

Block devices

Today, it is typical for block devices to be split into multiple logical partitions. This is done for multiple reasons, either to just logically separate space and use the partitions for different purposes, such as one partition usually dedicated to system files, and a second partition dedicated to multimedia files. Multiple partitions are also used for storing root file systems of different operating systems, when one is booting multiple operating systems from a single device. Another reason is to increase system security, as multiple partitions allow for a more fine-grained application of file system mount options, and possibly also limit the possibility of an application taking up all available space.

HelenOS has a *Virtual Block Device* (*vbd*) server, that takes care of partitioning. It supports the MBR and GPT partitioning schemes. Users can modify the partition labels with the `fdisk` utility. When creating a new partition, a new block device service representing the partition is created and registered with the Location Service under a `partition` category[14]. What the vbd server essentially does is offset and limit the I/O requests to the logical block address range of the logical partition to which the I/O belongs.

Another virtual block device that HelenOS supports is `file_bd`, a *file-backed* block device, which allows a file to be accessed as a block device. This can be useful for mounting a disk image.

The system would not be very useful without providing some block device drivers for real physical devices. Fortunately, HelenOS provides several block device drivers to interact with storage devices.

- **AHCI** driver
- **ISA-based IDE** driver
- **PCI-based IDE** driver

[14]The Location Service allows to specify categories, for example `disk`, `partition`, `keyboard`, `mouse`, `nic`, or `audio-pcm` categories.

- **USB mass storage** driver
- **PC floppy disk** driver
- **virtio-blk** driver

These device drivers utilize the *Device Driver Framework* (*DDF*), which coordinates the enumeration of peripheral buses, such as the PCI or USB, discovers devices and automatically starts and attaches drivers to them. DDF consists of the *Device Manager* (*devman*) server, that manages the device tree and exports driver services to clients via registering them with the Location Service, to the appropriate category. In this case the category `disk`.

All services exported by devman are registered under the `devices` namespace, mapped to `/loc/devices` directory via locfs.

For a more detailed description of the Device Driver Framework and general device drivers can be found at [16].

An example of I/O data flow of a `vfs_write()` operation to a file on an ext4 file system mounted to a SATA drive partition, and raw direct I/O is shown in Figure 1.6.

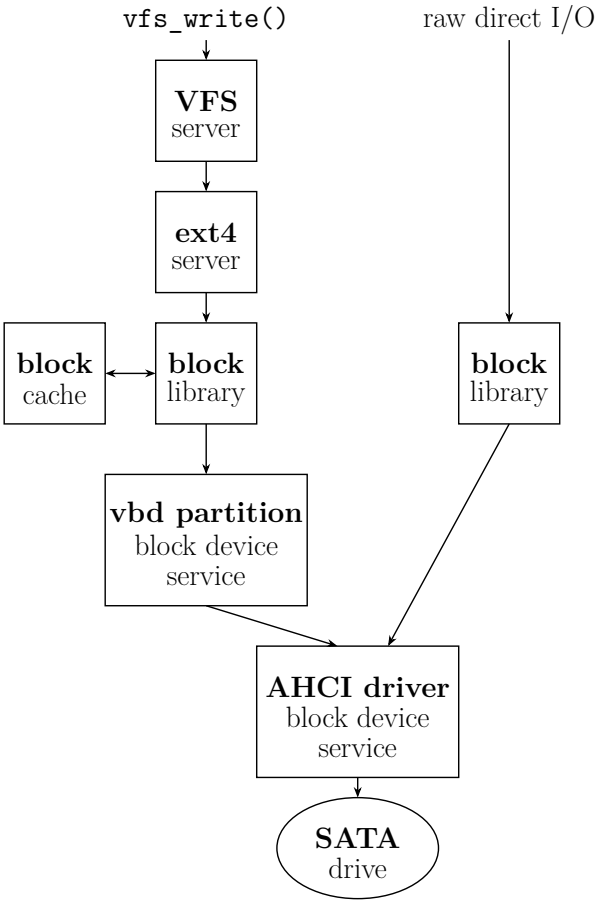

Figure 1.6 Example of different types of I/O to a partitioned SATA drive

1.2.4 Plain C unit testing framework

Plain C unit testing framework (PCUT) is a simple unit testing framework created by Vojtěch Horký [17]. It is integrated into HelenOS, where it is used for unit testing servers, libraries and applications. We will use PCUT for functional testing of the designed RAID implementation. [18]

2 Analysis

In the beginning of this chapter, we will look at software RAID challenges and then analyze different features of various RAID systems to evaluate the requirements for the proposed HelenOS RAID implementation, called *HelenRAID*.

In a later part, we will show how the RAID implementation will fit into and affect the HelenOS storage stack and possibly other components.

And finally in the last section, support of RAID volumes created on other operating systems is going to be discussed.

2.1 Software and hardware RAID

As the title hints, RAID can be implemented within hardware components, such as storage controllers, making the operating system unaware of the RAID abstraction. To increase performance, storage controllers can utilize specialized hardware to offload parity computations, or include extra on-board RAM to serve as a cache. However, these specialized hardware components and storage controllers impose additional cost to storage systems, therefore, this can be one of the reasons why storage solutions might prefer RAID implemented in software. Software RAID is often implemented as an operating system's device driver, which reduces complexity and hardware cost and improves observability from the operating system. [19]

RAID can also be implemented inside in system's BIOS, known as *BIOS RAID*, *hardware-assisted software RAID*, or *"fake RAID"*, because operating system drivers are required for it to function, and it does not use a dedicated processor and memory. In this instance, the chipset's firmware only manages the RAID configuration, where in some cases it can assist booting from RAID volumes. The RAID part of Intel Rapid Storage Technology is an example of BIOS RAID [20].

2.1.1 Advantages and disadvantages

Each form of RAID has its own advantages and disadvantages. For example, hardware RAID has the previously mentioned higher cost, complexity, and limited observability.

Another disadvantage of hardware RAID is that most hardware RAID solutions are proprietary, whereas software RAID is often implemented as a part of an open source operating system, which allows for finer performance tuning, as open source software RAID implementations can be modified for specific workload and system needs.

Hardware RAID advantages

Dedicated hardware to handle RAID operations can have many performance advantages, for example, RAID storage controllers can be utilized specifically to address the RAID 5 level's *small-write problem*.[1] This includes the subtract-writes/read-

[1]We refer to both RAID 4 and RAID 5 as RAID 5, if not explicitly stated otherwise.

modify-writes, which compared to RAID 0, degrade the write performance by a factor of 4. Numerous techniques can be used to address this, one of which takes advantage of a cached RAID controller. Write data are cached inside the controller's memory, delaying the write to the actual extents, possibly saving redundant extent I/O that overwrites data. [10] [21]

Software RAID challenges

Booting from volumes can become an issue with software RAID. To boot from software RAID volumes, the bootloader must support the RAID configuration and specific implementation metadata. Due to the complexity of RAID 5 and other more complex or striped configurations, operating systems usually only support booting from a mirror, which does not require striping or other I/O transformations.

But by far the biggest disadvantage of software RAID is the *consistency problem*, also known as the *write hole*.

2.1.2 The consistency problem

In a scenario of sudden power loss, there exists a vulnerability time window in which an inconsistency can arise. This can happen with RAID levels where an invariant between the extents must be kept, e.g., RAID 1 and RAID 5. Suppose that we write to a 2-way mirrored volume. It is possible that during the write, a power loss occurs, and only one of the internal duplicated I/Os completes. After system reboot and volume assembly, we can see that there is an inconsistency between the 2 mirrors, but we have no way of knowing which is the up-to-date one, therefore we cannot safely correct the inconsistency, without possibly corrupting the "correct" data.

To deal with this problem, synchronous writes can be sent to a log, and at next assembly after a power failure, the log can be used to replay all the writes that where not successfully marked as committed.[2] The synchronous writes are a big performance bottleneck, therefore techniques such as parity logging for RAID 5 volumes were researched [22]. Even though there are generally log-structured file systems used on top of software RAID volumes, they maintain the consistency of file system data structures and do not address the consistency problem at the RAID level. [23]

In ZFS, a file system with integrated software RAID, the *copy-on-write* (CoW) mechanism is leveraged to deal with this problem. Data block pointers are atomically updated, not allowing any inconsistencies to arise, and in a case of a power loss or a system crash happening before the atomic update, no original data is overwritten. In addition to covering the write hole, ZFS also uses checksums to protect data from other types of silent data corruption. [24]

[2]Hardware RAID utilizing fast non-volatile memory for caching can use the cache as a write log. Or it can include a battery backup that powers the controller until all queued writes complete.

2.2 Key features evaluation

The number of features that a RAID implementation can support is huge. Therefore, we will only analyze the features that are deemed the most important and essential for basic RAID volume management and effective I/O processing.

2.2.1 RAID volume management

Traditional RAID volume management operations include.

- Volume **creation**.
- Volume **assembly**. This feature is often automatic, or is automated by saving a persistent volume configuration.

  For assembly to be possible, volume *metadata* must be stored on extents. The metadata must include information enabling the system to pair extents and volumes together, and possibly other details about the volume configuration, such as the level, strip size, and the number of extents.
- Volume **disassembly**.
- **Monitoring** volume's **state**.
- Degraded volume **rebuild**. This allows volumes with one or more failed or unusable extents to be restored to an *optimal* state.
- **Hotspare** extents. Some implementations allow online standby extents (*hotspare extents*) to be specified, allowing automatic volume rebuild when a failure is encountered.[3]

  Techniques for using utilizing the idle hotspare extent for RAID 5 volumes exist to increase the performance, but this is out of scope for this thesis. [25]

Less common features not widely supported on different implementations might include.

- **Growing**, **reshaping** or **restriping** a volume. This can include enlarging a mirror, extending a RAID 5 with another extent or changing the volume's RAID level or its layout. Possibly a simple level change can be RAID 0 into RAID 4 where just the parity is computed.

The Linux software RAID implementation, Multiple Device driver `md(4)` [26] and management utility `mdadm(8)` [27] allow various RAID level and layout reshapes. The driver also supports bitmap write-intent logging for reducing resynchronization time after an unclean shutdown, or temporarily allowing an extent to be removed from the array without having to resynchronize all its blocks. To mitigate the write hole, `md(4)` also supports an additional device to be used as

[3] Although the term hotspare in this scenario might resemble what is called a *warm spare* in other areas of computer systems, we will use the term hotspare as it is common to refer to it as such in the RAID terminology.

a journal, or partial parity logs stored in the metadata. Another notable feature is its monitoring tool's ability to send emails when volume state changes to notify system administrators.

Due to the inherent complexity of the advanced features and such as reshaping, or the mentioned features supported by the Linux kernel's Multiple Devices driver, their support is out of scope for HelenRAID implementation as part of this thesis.

2.2.2 Rebuild

As already mentioned, a rebuild operation is used to restore volumes to the optimal state. In the case of mirrored volumes, this just involves straightforward copying of data blocks from valid extents to the rebuilt one. In the case of RAID 5 volumes, blocks in each row must be read and the contents XORed to produce the blocks that will be stored to the rebuilt extent. And obviously non-redundant striped volumes, i.e. RAID 0, will not support the rebuild operation, because it does not have a degraded state.

The time it takes to finish the rebuild process, i.e. the *rebuild time*, plays a big role in the total reliability properties of RAID volumes. In the original RAID paper [1], a simple formula was derived to model the *mean time to failure* ($MTTF$) of RAID volumes. They generalized the formula to include multiple independent groups of drives, i.e. multiple volumes, but we will use even simpler formula that calculates the mean time to failure of just a single volume, i.e. only a single group of drives.

$$MTTF_{VOLUME} = \frac{(MTTF_{EXTENT})^2}{N \cdot (N-1) \cdot MTTR}$$

Here $MTTF_{VOLUME}$ is the mean time it takes for a second failure to occur during a rebuild operation, rendering the volume unusable. Note that this calculation of mean time to failure of RAID volume models scenarios where a hotspare extent is always available, and the number of tolerated unusable extents is 1, i.e. 2-way RAID 1, RAID 4 and RAID 5 volumes. To calculate $MTTF_{VOLUME}$ of volumes that allow 2 or more faulty extents, a more complicated formula must be used [28]; $MTTF_{EXTENT}$ is the MTTF of a single extent; N is the number of all extents, also including the parity extent; $MTTR$ is the mean time it takes for the volume rebuild operation to finish.

Another thing to note is that this model assumes that extent failure rates are identical, independent, exponentially distributed random variables. To model more realistic scenarios, multiple elements, such as storage controllers or cooling, where failures are possible to happen, should be taken into account with the reliability modeling. More detailed information about this and further resources on this topic can be found in Section *1.2.2.6.7 - Reliability Modeling* in the RAIDframe research project resource [10].

It is quite common for RAID implementations to store rebuild progress, so that volume can be disassembled or the system restarted in the middle of a rebuild, and upon volume re-assembly the rebuild can resume from the position it was interrupted, removing the need of rewriting the whole extent block address range.

2.2.3 Scrub

Scrubbing is a similar process to a rebuild, but the difference is that it is usually done on volumes in optimal state to check for inconsistencies that may arise due to either system crashes or silent data corruption due to subtle physical drive failures or buggy firmware [29]. In the case of mirrored volumes, data blocks are simply checked for equality, and in parity keeping volumes the parity is recomputed from the data blocks and then checked for equality with the stored parity. When a mismatch is found, it can be either only logged or repaired if the RAID implementation is certain which extent has the "correct" data.

Scrubbing is also used to repair inconsistencies that come from read errors. When a read error is encountered on some extent, its data are from other extents and rewritten back to the extent's block addresses that returned a read error.

However, for the following reasons, the scrub feature shall be omitted from the scope of HelenRAID as part of this thesis.

- The write hole. Because non-journaled RAID volumes are vulnerable to the write hole, and also the inability to distinguish which data is "correct" in an instance of silent data corruption.
- We are going to use a simplified fault model, described in the following section, which does not allow read errors to be regarded as temporary failures.

Due to these reasons, we would be unable to do anything about the found mismatches other than logging them.

2.2.4 Error detection and fault model

Devices can return an error for attempted read or write to a certain block addresses while the rest of the device is operating properly. To simplify error handling and actual error detection, we are going to treat all device errors as unrecoverable.

However, to show an example of possibly leveraging this fact to improve reliability, we show how it is done in the Linux kernel's Multiple Devices driver. It stores a so called *Bad Block List*, which stores block addresses to which requests got an error reported by the device. A bad block will be stored in the bad block list if a block cannot be read and cannot be repaired by trying to write reconstructed data to it, or if a write fails. A failed attempt to store the bad block results in the device being marked as faulty. Reads to known bad blocks return an error immediately, and writes are let through. When later write to that bad block succeeds, it will be removed from the bad block list. This, for example, allows a single-tolerant volume to sustain unusable blocks on different extents if the blocks are generally not at the same extent block addresses. [26]

2.2.5 Re-assembly consistency

Although we cannot protect against the consistency problem that arises during a system crash or a power loss without a journal, we can protect against inconsistencies that come from partial volume assemblies. Each extent state must be consistent with the most up-to-date state of the volume.

Suppose that we have a 2-way mirror. We disassemble it, and re-assemble it with only one extent, write some data, and again disassemble it. After re-assembling with both extents, both extents supposedly store identical data, but this is not true in our scenario. To catch these partial re-assemblies, a counter that is incremented after the volume becomes *dirty*, can be saved in the metadata. In the assembly process, the counter is checked across all extents, and the most up-to-date one is selected as the "best" one. All extents that have an equal counter are marked as valid. Otherwise, data inconsistency has probably arisen.

2.3 HelenRAID

It is quite straightforward to determine where a RAID implementation will fit into the HelenOS storage stack. After all, there is already a server that creates virtual block devices. The `vbd` server exposes partitions existing on a device as multiple block device services, which represent the partitions. We want to do something similar in some sense, just in the "opposite" way. We want to expose multiple block device services (extents) as a single block device service representing the volume. The volume is going to be just another block device service implementing `bd_ops_t` block interface, described in Chapter 1, Subsection 1.2.3.

For an illustration of a possible volume consisting of four physical devices, see Figure 2.1.

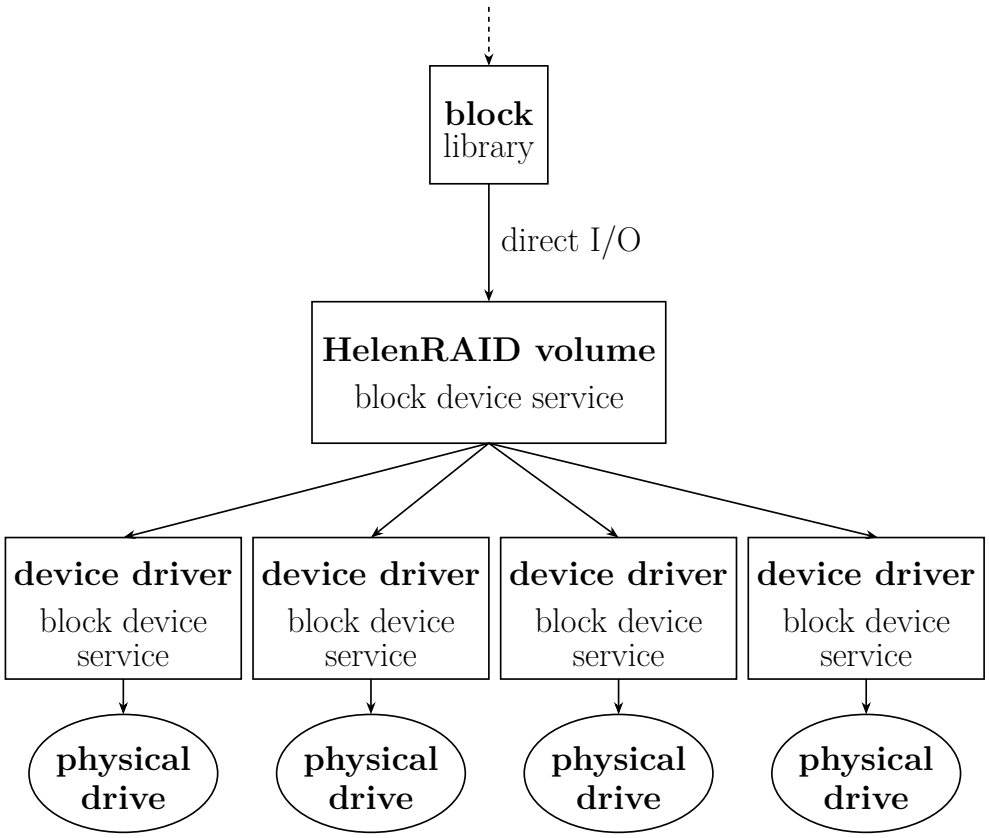

Figure 2.1 HelenRAID volume service handling physical devices

2.3.1 User interface

In addition to providing volumes as block device services, the RAID implementation will have to provide a different interface for the user to manage and monitor volumes. For this, custom IPC methods will have to be implemented. One side of the IPC methods will be used by the client (user) and the RAID implementation (server) will implement the client methods internally. It is common for servers to have different interfaces. An example of such a server is the vbd, which provides partitions as block device services and also provides an interface for the user to create or modify partition tables. Additionally, it is also common to have a user-friendly application that provides the client interface. HelenRAID shall also have such an application, a control utility, that provides a user-friendly interface for managing and monitoring volumes.

2.3.2 RAID levels support

HelenRAID implementation as part of this thesis shall include support for commonly used RAID levels, RAID 0, RAID 1, RAID 4 and RAID 5. Other commonly used RAID levels such as RAID 6, as hinted in Chapter 1, require additional substantial effort to analyze and implement efficiently. Nonetheless, HelenRAID implementation shall allow easy addition of support for other levels in the future.

2.3.3 Concurrent I/O requests

To gain the speedup promised by the RAID levels, we need a way to effectively and concurrently utilize the underlying devices. Another important thing that affects performance is how concurrent incoming I/O is handled, especially in an invariant keeping volumes.

Extent I/O concurrency

Other operating systems, namely their kernels, such as Linux, OpenBSD's kernel, and FreeBSD's kernel, use a common mechanism in their block I/O layer. When user invokes a read or write operation via syscalls, the upper kernel storage layer, namely the file system, if using one, or the VFS/vnode layer itself when raw devices are used, creates a `struct bio` in Linux and FreeBSD, `struct buf` in OpenBSD. It forwards the struct that I/O request to the block layer, which can transform the block I/O requests accordingly, for example, duplicate it in case of RAID 1. The requests are then asynchronously submitted or dispatched to device queues. After an interrupt signals I/O completion, a callback function is invoked by the driver, and the sleeping user thread in the original syscall is woken up and returned to userspace.

HelenOS does not have such block layer allowing to dispatch multiple I/O requests asynchronously by a single thread (fibril). This is due to the fact that all block devices and block device drivers expose only the synchronous interface defined by `bd_ops_t`.

We came up with the idea of using a custom fibril pool for submitting I/O requests in batches/groups and then waiting for I/O requests in specific groups to

finish. This will allow synchronous I/O requests to be concurrently processed in a similar way to the asynchronous model used by the mentioned kernels.

Another option to achieve a quite similar mechanism is to rewrite the block device client interface to take advantage of the asynchronous nature of HelenOS IPC. The fibril that serves incoming I/O would do asynchronous IPC calls to the block device server, and then wait for their completion. However, for the following reasons we find the custom fibril pool to be a better mechanism to achieve effective I/O request submission to extents.

- Multiple fibrils will potentially allow more concurrent execution, because even though the request itself is asynchronous, there might be nontrivial overhead of making the actual syscall to IPC.
- As fibrils are scheduled cooperatively and the scheduler is not aware of them, they should not overload the scheduler.
- We would still need to keep track of what asynchronous requests we are waiting for, and wait for them. Therefore, a similar mechanism to one that the custom fibril pool shall provide would have to be implemented with this option as well.

Incoming I/O concurrency

In general, simple block devices, such as partitions or raw disks, do not need to limit the number of incoming concurrent I/O requests. This is because the upper layer, usually the file system, or the user himself is responsible for synchronizing concurrent operations and dealing with possible race conditions on block addresses.

In case of volumes that hold some form of invariant, such as RAID 1 or RAID 5, volumes must enforce some form of limiting for concurrent I/O.

For RAID 1, for protecting the mirror invariant, limiting write requests is required. This is because interleaved parallel write requests to the same block addresses could result in inconsistencies between the mirrored extents. This is due to a potential race between internal I/O requests regenerated by the user I/O. There is no need to limit reads, because synchronizing the I/O to be thread safe and eliminating races on the same block addresses is still a responsibility of the upper layer.

In RAID 5, limiting writes is required for the same reason as for RAID 1. The difference being that here, the parity consistency is being protected. But opposed to RAID 1, reads must be mutually exclusive with writes when a volume is in a degraded state. This is because when a read I/O requests data from an unusable extent, the data must be reconstructed from parity. And while the data is being reconstructed, parallel write to any extent on the affected stripe will update the parity, therefore, a potential race might occur.

However, completely limiting concurrent I/O on storage devices that have multiple hardware queues or generally support parallel I/O, such as modern solid-state drives, would greatly reduce their potential performance. So, instead of completely locking the whole block address range, HelenRAID shall lock/synchronize only block address ranges affected by active I/O requests.

2.3.4 Metadata

As was already mentioned, re-assembly of volumes after system restart requires metadata to be stored on the extents. From information contained in the metadata, it has to be possible to pair extents from the same volume together. To achieve this, saving a volume identifier and extent's position (index) in the volume is sufficient.

To not interpret random data as metadata field values, the value identifying a present superblock is often put at the beginning, called the *magic*.

Besides the obvious volume properties such as its level, layout and number of extents, other useful fields include.

- Metadata version. This value is useful for allowing future metadata updates or layout changes in the superblock.
- Strip size. Because different volumes might have different strip sizes, this is required to interpret data blocks correctly.
- The size of the smallest extent. This value is required if we want to support extents which do not have equal sizes. In such a case, the minimum of their sizes must be taken. Based on this *truncated* size, volume capacity is calculated and hotspares tested if they are big enough.
- Number of usable blocks provided by the volume. Even though this can be recalculated from the level, number of extents and the truncated extent size, it will be useful to have this stored.
- Incremental on-disk counter. This counter will guarantee re-assembly consistency.
- Rebuild position. As mentioned, it is nice to remember the last position of the rebuild, if it gets interrupted. This will allow the rebuild process to be resumed.

It is common for metadata to also include the offset where user data begin from the start of extent. But this information is redundant if the metadata superblock is at the end of the device and the user data starts at the beginning. The following section discusses possible advantages and disadvantages that different metadata superblock positions have.

Metadata positioning

Different specifications, implementations and their versions, store the metadata superblock at different positions on the extents. The SNIA DDF [4] for example stores the metadata in the front and also a copy at the back as a backup, for the scenario where the first one gets corrupted. However, all software RAID implementations that we are going to analyze and support later store the metadata either at the front or at the back. For simplicity, HelenRAID metadata shall also be stored in a single location.

Metadata stored relative to the end of the device, for example, the last block, have a problem that arises when the underlying extent is not a classic block device.

That means that when the block device is a disk image provided by a hypervisor or a pooled logical block device that can grow in size, it is apparent that after the size change, the metadata will be searched for at the newly added fresh blocks.

However, metadata stored at the end has numerous advantages.

- With metadata stored at the end, mirrored extents can be used as typical disks, allowing them to be used outside RAID. It could also help with booting from mirrors without the bootloader having to understand RAID or its metadata. However, booting from RAID volumes was not analyzed as part of this thesis and is out of scope for this text.

  For mirrors, this could become a problem if an existing file system there is mounted on the extent itself.

- If there was a way to know that the last block on a block devices was not used, a mirror could be easily created by adding additional disk.

We decided to store metadata at the end of the extents. It shall be trivial to make changes to the metadata structure and its position thanks to HelenRAID's metadata handling interface. The interface requirements are discussed in the following section.

The overview of metadata fields can be seen in Table 2.1.

Field	**Description**
magic	Present metadata superblock identifier.
version	Metadata version identifier.
UUID	Unique volume identifier.
extent no.	Number of extents the volume consists of.
level	RAID level specifier.
layout	RAID layout specifier for parity volumes.
strip size	Strip size of a RAID 0 or RAID 5 volume.
extent index	Extent position in the volume.
data block no.	Number of user usable blocks.
truncated block no.	Smallest extent size.
block size	Block size of the volume.
counter	Incremental counter for re-assembly consistency.
rebuild position	Block address specifying the last rebuild position.

Table 2.1 Metadata fields overview

2.4 Foreign metadata support

One of the goals we have set is to support external metadata that are foreign to HelenRAID's native metadata format. This will allow system migration without the need to copy volume contents to another. One reason for accessing volumes on HelenOS as opposed to other systems could be the preference of HelenOS over other operating systems due to properties provided by its micorkernel design, or possibly even because of performance reasons.

We have picked the following software RAID implementations provided in OpenBSD, FreeBSD, and the Linux kernel.

- OpenBSD's `softraid(4)` [30].
- FreeBSD's software RAID implementations provided by `gstripe(8)` [31] (RAID 0) and `gmirror(8)` [32] (RAID 1).

  FreeBSD also has `graid(8)`, which supports numerous BIOS RAID metadata formats. It has also support for RAID 5, but only in read-only mode [33]. Due to the BIOS RAID formats complexity, and missing write support for RAID 5, we will not support `graid(8)`.

  FreeBSD has `gvinum(8)` that provides RAID 5 functionality, but in the upcoming release 15.0 and later it will not be available, as its support is already deprecated [34] and was already removed in the CURRENT branch [35].
- Linux Multiple Device software RAID `md(4)` [26] provided by `mdadm(8)` [27].

These software RAID implementations were selected primarily because of the author's familiarity with those operating systems and their source code. We believe that it will be enough to demonstrate HelenRAID's ability to work with foreign RAID metadata.

Only the metadata of the mentioned RAID implementations will be analyzed. There is no need to analyze the internals of other RAID implementations, because they are heavily dependent on the kernel and its block I/O layer. It is also not a goal of the thesis to employ all possible RAID features, tricks, techniques, and optimizations that might be implemented and supported in other operating systems. Due to time constraints, hotspare extents will not be dealt with.

2.4.1 Format agnostic metadata handling

To handle multiple different metadata formats, there is a need for a metadata-agnostic interface. The RAID implementation shall use this interface to invoke format-specific implementations for different purposes. In this section, we discuss the operations that the interface will include. It shall allow volumes that use metadata formats mapable to this interface to be utilized, given that the implementation for the format is provided.

The following operations will be needed.

- **Probing extents**. This is the most important function that metadata implementations will have to provide. It will search extents for metadata format superblocks and validate them based on magic. If the magic is valid, whole superblock is going to be decoded.
- **Compare UUIDs of volumes** across extents, to pair them together into volumes they belong to.
- **Volume initialization from metadata**. In this step the structure representing the volume will be filled with needed information such as what is the volumes level, number of extents, strip size, and so on.

- **Metadata population** from the newly created volume. The opposite of the above.
- **Metadata saving** to extents. This shall also include rebuild position saving.
- Re-assembly consistency **counter increment**.

In the following sections, we will analyze foreign metadata formats of the mentioned RAID implementations. We will not show how volumes are created in each operating system as it is irrelevant in this part of the text. How foreign volumes can be accessed and their support tested in HelenOS will be shown in a later chapter. We will discuss how the mapping to our metadata interface shall look for each format.

2.4.2 softraid (OpenBSD)

OpenBSD's softraid emulates a SCSI Host Bus Adapter (HBA), that provides RAID and other I/O related services such as disk encryption. It supports traditional RAID 0, RAID 1 and RAID 5, and additionally also volumes from concatenated extents (CONCAT) and disk encryption.[4] Optionally, an encrypted mirror can also be created. [30]

In softraid terminology, an extent is called a *chunk*. Volumes can only be created out of chunks that are partitions of type `RAID`. Concrete steps for how the volumes are created is irrelevant in this section, however, the details can be found in the `softraid(4)` manual page [30].

Metadata layout

Metadata starts at 8 KiB from the partition beginning and has 32 KiB reserved. However, its actual size depends on the number of chunks it consists of because there is also information about each chunk.

All multibyte values are stored in little endian.

The first part of metadata is `struct sr_metadata`[5] which is followed by `struct sr_meta_chunk`[6] for each chunk. After all chunk metadata, there can possibly be optional metadata headers, `struct sr_meta_opt_hdr`[7], but these are standardly not used. The actual metadata layout can be seen in Figure 2.2.

In the following points we comment on notable metadata fields, first on `struct sr_metadata` members and then `struct sr_meta_chunk`.

- Volume flags. This field is used for volume specific flags, currently only `BIOC_SCNOAUTOASSEMBLE` and `BIOC_SCBOOTABLE` are utilized by softraid, specifying if the volume should not be assembled during automatic configuration and specifying that it is bootable, respectively.

[4] In softraid code there is also an implementation for RAID 6, but its support is not enabled in the base system.

[5] http://bxr.su/OpenBSD/sys/dev/softraidvar.h#125

[6] http://bxr.su/OpenBSD/sys/dev/softraidvar.h#163

[7] http://bxr.su/OpenBSD/sys/dev/softraidvar.h#191

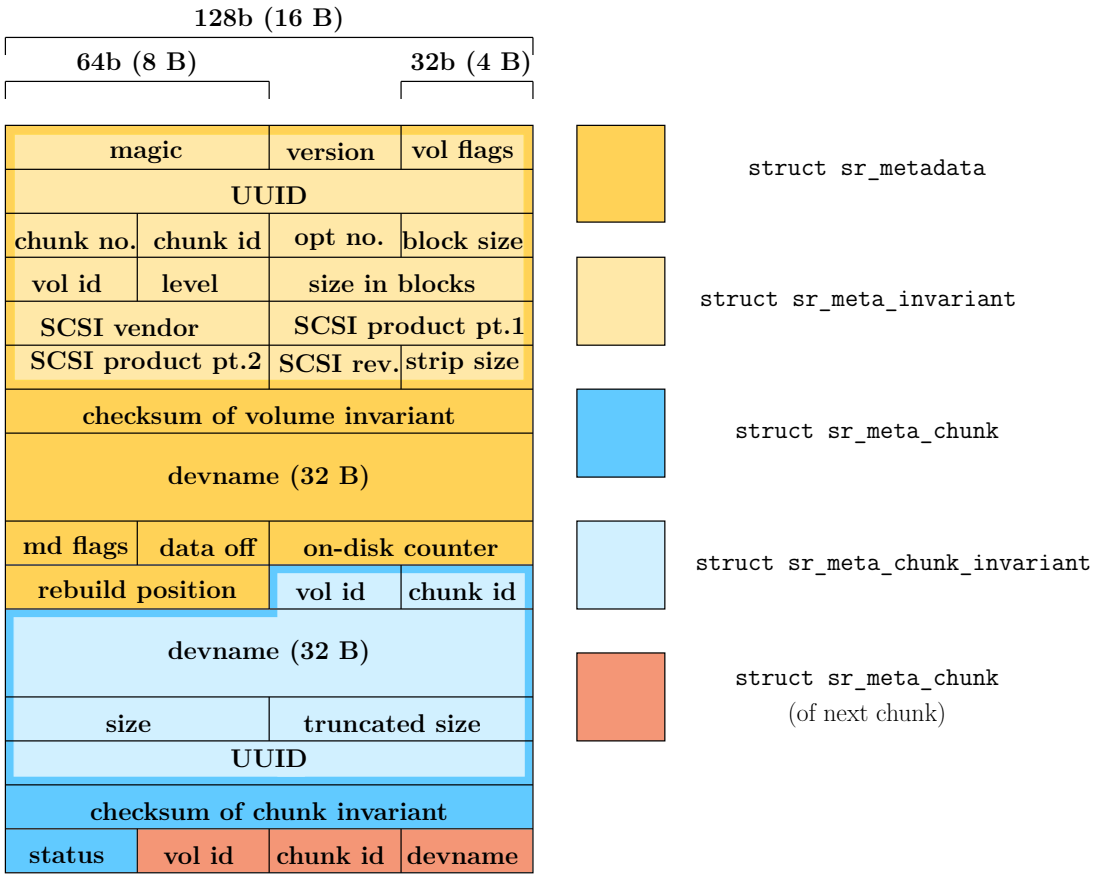

Figure 2.2 softraid metadata layout

- Volume UUID. This is the unique volume identifier that can used to distinguish if different softraid chunks belong to the same volume.
- Chunk identifier. This is the identifier of the chunk on which this metadata was read. Its used as a volume chunk index.
- Number of optional metadata elements. This is not commonly used.
- Volume identifier. Used internally, for us, it is irrelevant.
- Size in blocks. This is the number of blocks usable by the user.
- SCSI specific fields, *vendor*, *product* and *revision*. These are here because softraid emulates an HBA. We can ignore these.
- Metadata flags. This can be either 0 or `SR_META_DIRTY` (1). This is set when the metadata was written to since assembly, and is set back to 0 on volume shutdown.
- Data offset (actual field name is `ssd_data_blkno`). This value is the block address where user data starts.
- On-disk counter. This is the counter that is incremented each time metadata is saved to disk.
- Rebuild position. This marks the position of the last rebuilt block.

`struct sr_meta_chunk`

- Volume identifier to which the chunk belongs. We can ignore this because we already have volume UUID and have both this `struct sr_meta_chunk` and `struct sr_metadata` describing the whole volume together.
- Chunk identifier. For us, this is redundant, as we have the chunk ID of the chunk we are reading from in the `struct sr_metadata`.
- Device name. This is the device name of the chunk used by OpenBSD.
- Size of partition in blocks. This is the number of blocks the chunk has.
- Coerced (truncated) size in blocks. This is the actual number of blocks that is utilized.
- Status. This field is set with status values used by `struct bioc_disk`[8] – *Online*, *Offline*, *Failed*, *Rebuild*, *Hot spare*, *Unused*, *Scrubbing* and *Invalid*.

Mapping to HelenRAID metadata handling interface

- Probing. With knowledge of metadata position and size, probing extents is straightforward. The metadata blocks will need to be read, then the block decoded to CPU byte-order so that the metadata can be understood and manipulated.
- Comparing volumes UUIDs is straightforward.
- Volume initialization from metadata. This should be straightforward as well, only the on-disk counter and rebuild operation have to be accounted for.
- Metadata population. This will not be supported, because of fields intended for internal usage by OpenBSD, and the need to create RAID type OpenBSD partitions in the first place.
- Metadata saving. This is similar to what volume initialization will have to handle, specifically the rebuild operation and marking the correct extent as rebuild. Following the encoding to little endian, computing the checksums and writing to extents.
- Re-assembly consistency counter increment – On-disk counter field increment.

2.4.3 GEOM Stripe (FreeBSD)

RAID 0 provided by `gstripe(8)` [31], same as all I/O transforming devices in FreeBSD, is implemented via GEOM (`geom(4)`). GEOM is a modular framework that provides an infrastructure for different *classes* to be implemented. Classes implement and define how incoming I/O is treated. An instance of a class is called a *geom*. The virtual device provided by a geom is called a *provider*, and the underlying devices utilized by the provider are called *consumers*. In our case, the provider is a striped volume and consumers are its extents. [36]

[8] http://bxr.su/OpenBSD/sys/dev/biovar.h#84

Metadata layout

The metadata is stored on the last block of an extent, and is 1 block in size. It is described by `struct g_stripe_metadata`[9]. Since RAID 0 cannot be in a degraded state and no rebuild is possible, the metadata describing the volume are very minimal. GEOM Stripe metadata fields can be found in Table 2.2. All values are stored in little endian.

Field	Size in bytes	Description
magic	16	Magic value.
version	4	Version number.
name	16	Stripe name.
id	4	Unique ID.
no	2	Disk number.
all	2	Number of all disks.
stripesize	4	Stripe size.
provider	16	Hardcoded provider.
provsize	8	Provider's size.

Table 2.2 GEOM Stripe metadata fields

Comments on notable fields.

- Stripe name. This is the name of the virtual device, set by the user.
- Unique ID. This will be used to pair extents together.
- Disk number. The index of the extent in the volume.
- Hardcoded provider. `gstripe(8)` allows consumer device names to be hardcoded inside the metadata, so the striped volume can only be assembled when the device names also correspond. By default, this field is empty.

  The field name is called provider, because from the internal view of the GEOM Stripe class, the extents are actually providers. Only the view of the whole volume, "from outside", sees the extents as consumers, and the provider is the actual striped volume.
- Provider's size. This is the size of the extent in bytes. Because no truncated value is provided, to get it we have to iterate over all extent metadata to find the smallest provider size.

Mapping to HelenRAID metadata handling interface

- Probing. Straightforward, we need to read the last block and decode the metadata to cpu byte order.
- Comparing volumes UUIDs is straightforward.

[9] http://bxr.su/FreeBSD/sys/geom/stripe/g_stripe.h#71

- Volume initialization from metadata. No special handling shall be required other than getting the truncated block number.
- Metadata population. This is possible due to no mandatory fields used internally. However, since the native metadata will be preferred this might be omitted.
- Metadata saving. No metadata updates are needed, therefore this is a no-op.
- Re-assembly consistency counter increment – not needed and not present.

2.4.4 GEOM Mirror (FreeBSD)

RAID 1 provided by `gmirror(8)` [32], same as `gstripe(8)`, is implemented via the GEOM I/O transformation framework.

Metadata layout

Same as GEOM Stripe, GEOM Mirror's metadata are located at the last block of extents, and are 1 block in size. GEOM Mirror metadata fields are represented by `struct g_mirror_metadata`[10], and can be found in Table 2.3. All values are stored in little endian.

Field	Size in bytes	Description
magic	16	Magic value.
version	4	Version number.
name	16	Mirror name.
mid	4	Mirror unique ID.
did	4	Disk unique ID.
all	1	Number of disks in mirror.
genid	4	Generation ID.
syncid	4	Synchronization ID.
priority	1	Disk priority.
slice	4	Slice size.
balance	1	Balance type.
mediasize	8	Size of the smallest disk in mirror.
sectorsize	4	Sector size.
sync_offset	8	Synchronized offset.
mflags	8	Additional mirror flags.
dflags	8	Additional disk flags.
provider	16	Hardcoded provider.
provsize	8	Provider's size.
hash	16	MD5 hash.

Table 2.3 GEOM Mirror metadata fields

Comments on notable fields.

[10] http://bxr.su/FreeBSD/sys/geom/mirror/g_mirror.h#240

- Mirror unique ID. This value will be used to pair extents into volumes.
- Synchronization ID. This is the on-disk counter incremented with each metadata update.
- Disk priority. Slice size. Balance type. These fields are used to specify what algorithm to use for read I/O. It supports selecting the extent based on load, priority, or in a round-robin fashion, or splitting the I/O with respect to slice size. These values can be safely ignored.
- Synchronized offset. If this is not equal to zero, this means that a rebuild is in progress on this extent.
- Additional mirror flags. Can specify if stale components (out-of-date extents) should be automatically synchronized, and if the mirror should be synchronized as a whole after a power failure or a system crash. These can be ignored.
- Additional disk flags. Notable flags that are set here specify that the metadata on the disk is dirty, or if the disk is being rebuilt, or is inactive.

Mapping to HelenRAID metadata handling interface

- Probing. Straightforward, we need to read the last block and decode the metadata to cpu byte order.
- Comparing volumes UUIDs is straightforward.
- Volume initialization from metadata. In the GEOM Mirror case, rebuild and out-of-date extents need to be handled.
- Metadata population. This is theoretically possible to support, however as mentioned in the GEOM Stripe section, for creation HelenRAID native metadata are going to be preferred.
- Metadata saving. Encoding back to little endian from CPU byte order and taking rebuild into account should be sufficient. And finally, computing the MD5 hash.
- Re-assembly consistency counter increment – synchronization ID increment.

2.4.5 MD RAID (Linux)

Linux's Multiple Device driver supports RAID 0, RAID 1, RAID 4, RAID 5, RAID 6, RAID 10 and additionally so called Linear level, which is a volume out of concatenated extents (same as softraid's CONCAT). We are going to support only the first 4 listed levels.

Since MD supports various advanced features such as volume reshaping or write-intent bitmaps, bad block logs, we are going to show only fields that are relevant for HelenRAID. MD also had various metadata versions, each with different layout, and additionally numerous minor versions, differing in the superblock location. For these reasons, we will focus only on the newest version of the MD metadata (1.2 at the time of writing).

Metadata layout

MD superblock version `1.2` is 4 KiB from the beginning of an extent. The size of the actual superblock varies depending on the number of extents. The structure representing the superblock, `struct mdp_superblock_1`[11], is 256 bytes itself, and additionally at the end, there is 2 byte value for each extent representing its role in the volume. Since 1 KiB allows 384 extents, we will limit ourselves to 1 KiB superblock size. Other data in addition to the actual volume information can be stored behind the mentioned structure, like the write-intent bitmap, but we will not focus on that. The relevant fields can be found in Table 2.4. All values are stored in little endian.

Field	Size in bytes	Description
magic	4	Magic value.
feature_map	4	Bit field set with supported features.
set_uuid	16	Volume UUID.
set_name	32	Volume name provided by the user.
level	4	Volume level.
layout	4	Layout for RAID 5.
size	8	Truncated number of extent blocks.
chunksize	4	The strip size.
raid_disks	4	The number of extents.
data_offset	8	Block address where data starts.
dev_number	4	Permanent identifier of this extent.
events	8	Incremented with superblock each update.
resync_offset	8	Data before this address are in sync.
sb_csum	4	Checksum.
max_dev	4	The size of dev_roles array to consider.
dev_roles	2·max_dev	Roles of the devices in the volume.

Table 2.4 Relevant fields of MD superblock version 1.2

Fields that need comments.

- Feature map. Bits set here indicate what features does the volume support. This also includes if a rebuild is in progress on a device.
- Chunksize – strip size in 512 B blocks.
- Permanent identifier of an extent is an index to the dev_roles array, where the actual extent index in the volume can be found.

Mapping to HelenRAID metadata handling interface

- Probing. Straightforward, we need to read the 1 KiB at offset 4 KiB and decode the metadata.
- Comparing volumes UUIDs is straightforward.

[11] include/uapi/linux/raid/md_p.h::`struct mdp_superblock_1` @ github.com

- Volume initialization from metadata. For simplicity we will entirely ignore any extents that have any bit set in the feature_map.
- Metadata population. For the same reasons as in the case of softraid, this will not be supported.
- Metadata saving. Since we will only work with valid extents, encoding and calculating the checksum should be sufficient.
- Re-assembly consistency counter increment – events field increment.

3 Design and implementation

In this chapter, we show how was HelenRAID implemented, the design ideas and techniques used to handle volume management and RAID I/O transformation operations.

This chapter itself is regarded as the implementation documentation.

3.1 Implementation overview

HelenRAID was written completely from scratch. Its implementation is around 12,000 (twelve-thousand) lines of code[1]. No HelenOS code is modified other than build scripts to include HelenRAID directories.

The project is currently in the process of being reviewed, so that it can be merged into HelenOS. A Pull Request on GitHub [37] has been opened to allow developers to review and comment on the code. Although there has not yet been any public code review, positive feedback on HelenRAID was received from an active HelenOS developer via private communication.

3.1.1 Implementation structure

The whole implementation consists of a server, a client interface, and an application that uses the client interface to communicate with the server. The top-level project structure with file and directory descriptions can be found in Table 3.1. The directory/file paths specified in the table are relative to the HelenOS source root.

File/Directory path	Description
uspace/srv/bd/hr/	Server implementation.
uspace/lib/device/src/hr.c	Implementations of client IPC methods.
uspace/lib/device/include/hr.h	Common client–server types and constants.
uspace/app/hrctl/	Volume management and monitoring utility.

Table 3.1 HelenRAID project files and directories

3.1.2 Volume management commands

User commands and management operations supported by the RAID implementation are briefly described here.

- **Volume creation**. Volumes can be created by specifying properties and extents on the command line, or from configuration files for comfortable management.
- **Volume assembly**. Manual and automatic assembly of volumes is supported.

[1] This number does not include unit tests and I/O benchmarking tool code described in later chapter.

- **Volume disassembly**. This stops and unregisters the service representing the volume.
- **Extent failing**. This is useful for simulating extent failure scenarios.
- **Hotspare addition**. Hotspare extents are allowed to be specified, so that in a case of an extent failure, rebuild operation can be started instantly.
- **Volume monitoring**. This includes getting volume information and volume state.

3.2 Architecture

As discussed in Chapter 2, Section 2.3, the main part of HelenRAID is the server. The server, named *hr*, provides volumes as block device services, that can be used as any other block device. To communicate with the server, there are client IPC methods defined in `lib/device`. For convenient volume management, a control and management utility *hrctl* is provided. It utilizes client IPC methods to manage and monitor volumes provided by the server.

Diagram showing a user command/operation sent to the server can be seen in Figure 3.1.

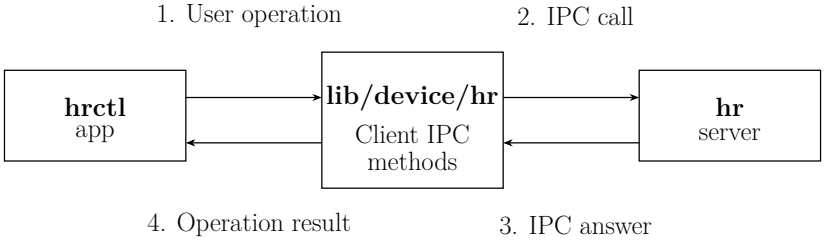

Figure 3.1 HelenRAID client app and server diagram

3.2.1 Client IPC interface

In the following points, we describe the client IPC methods that are used to send commands to the server. All volume management and monitoring operations are defined by these IPC methods, there is no other way to send the server commands or instructions. They are implemented in `libdevice`, and their source code can be found at *uspace/lib/device/src/hr.c*.

- `hr_sess_init()` – Initialize the server session. `hr` server's *service id* is obtained, and a session with its IPC interface is created.
- `hr_sess_destroy(sess)` – Destroy the server session. This method hangs up the IPC phone. Used after communication with the server is over.
- `hr_create(cfg)` – Create a single volume specified by `hr_config_t` structure describing the volume. This structure includes volume name, extent service IDs, number of extents, RAID level, and additional volume flags.

The additional volume flags can include `HR_VOL_FLAG_NOOP_META` and `HR_VOL_FLAG_READ_ONLY`, specifying if no-op dummy metadata should be used, or the volume made read-only, respectively.

- `hr_assemble(cfg)` – Assemble volume(s). This makes the server try to assemble volumes from the extents specified in a `hr_config_t` structure. All usable volumes formed from the specified extents in the configuration structure are going to be assembled.
- `hr_auto_assemble()` – Automatically assemble volumes. This instructs the server to perform automatic volume discovery. Automatic assembly from the server's viewpoint is described in a later section.
- `hr_stop(vol)` – Disassemble a volume. This tells the server to disassemble and unregister a single volume.
- `hr_stop_all()` – Disassemble all active volumes.
- `hr_fail_extent(vol, ext)` – Mark an extent as failed. Tells the server to mark the specified extent in a volume as failed. The server currently also erases the superblock on the extent.
- `hr_add_hotspare(vol, dev)` – Add a hotspare to a volume.
- `hr_get_vol_states()` – Get brief state of all volumes.
- `hr_get_vol_info(vol)` – Get detailed info about a volume.

3.2.2 Client application

To make it convenient for users to interact with the server, a management application – `hrctl` has been implemented. In essence, the application translates command line options into the IPC methods described in the previous subsection. Its implementation can be found at *uspace/app/hrctl/hrctl.c*. User documentation describing the usage of the hrctl utility can be found in Appendix C.

One thing worth mentioning, what this application does except parsing command line options, is its ability to read configuration files in Structured Information Format (SIF) which is used in HelenOS by numerous applications for this puprose. SIF is similar to an XML document that just contains tags with attributes[2]. The following is an example of a SIF configuration file that defines a 3-way mirrored volume *hr0*. For further details on the configuration files, please refer to Section C.1 of Appendix C.

```
<sif>
<hrconfig>
    <volume devname="hr0" level="1">
        <extent devname="devices/\hw\sys\00:01.1\c0d0"></extent>
        <extent devname="devices/\hw\sys\00:01.1\c0d1"></extent>
        <extent devname="devices/\hw\sys\00:01.1\c1d1"></extent>
    </volume>
```

[2]API to work with SIF can be found at*uspace/lib/sif/src/sif.c*

```
</hrconfig>
</sif>
```

3.2.3 Server

The main part is the server. Because it contains many files, an overview of each file and directory can be found in Table 3.2.

Other than foreign metadata format headers in metadata/foreign/, all files were written by the author. A more detailed description specifying what files were written by the author and which were copied from the respective project sources is stated in the section describing the implementation of foreign metadata support.

File/Directory	Description
var.h	Internal types.
hr.c	Server main with server IPC methods.
util.{c,h}	Utility and range lock code.
fge.{c,h}	Fibril Group Executor.
io.{c,h}	I/O workers.
raid0.c	RAID 0 (striping) implementation.
raid1.c	RAID 1 (mirroring) implementation.
raid5.c	RAID 4, 5 (parity) implementation.
parity_stripe.{c,h}	RAID 4, 5 stripe I/O execution.
superblock.{c,h}	Format agnostic metadata interface.
metadata/native.h	Native metadata superblock.
metadata/native.c	Native metadata mapping.
metadata/noop.c	No-op type metadata mapping.
metadata/foreign/geom/	GEOM Stripe and Mirror support.
metadata/foreign/md/	Linux MD RAID support.
metadata/foreign/softraid/	OpenBSD softraid support.

Table 3.2 HelenRAID server structure (uspace/srv/bd/hr/)

The rest of this chapter is dedicated to describing the internals of the server.

3.3 Structures and primitives

In this section, we first describe the primitives designed to achieve desired functionality, such as range locking and I/O batch execution. Then we describe how volumes are represented, their lifetime, and how the server stores them.

3.3.1 Range locks

As mentioned in Chapter 2, Section 2.3.3, we need a locking primitive that will allow concurrent user I/O requests to volumes while protecting their invariants. The so-called *range locks* that allow ranges to be locked were implemented.

They are represented by the type `hr_range_lock_t` defined in *util.h*. A range lock is acquired by specifying a block address and a block count to the acquire

function. After the acquire function returns, the calling fibril owns the range $[block\ address, block\ address + count - 1]$. After the critical section, the lock is released by the release function. The function declarations are as follows.

```
hr_range_lock_t *hr_range_lock_acquire(vol, uint64_t ba, uint64_t cnt);
void hr_range_lock_release(hr_range_lock_t *rl);
```

Note that the type of volume argument was left out, because it will be described later. The acquire function needs a back-pointer to the volume structure, as the volume keeps a list of all active range locks. The core part of the range lock is a mutex, which is used as a fibril queue[3]. The acquire function allocates the range lock structure and finds already locked ranges that overlap the requested range. If there is an overlapping range, the fibril attempts to acquire the overlapping range. If there is no overlapping range, the caller's range lock is added to the list. There is a reference counting mechanism that frees the allocated memory depending on the reference counts. The implementation can be found in *util.c*.

An example of concurrent range lock acquisition by different fibrils is shown in Figure 3.2. Tx denotes a point in time x, when the described actions occurred. Note that the range locks **C** and **D** in the beginning wait for **A**, because **B** is not yet in the range lock list that is checked for overlaps.

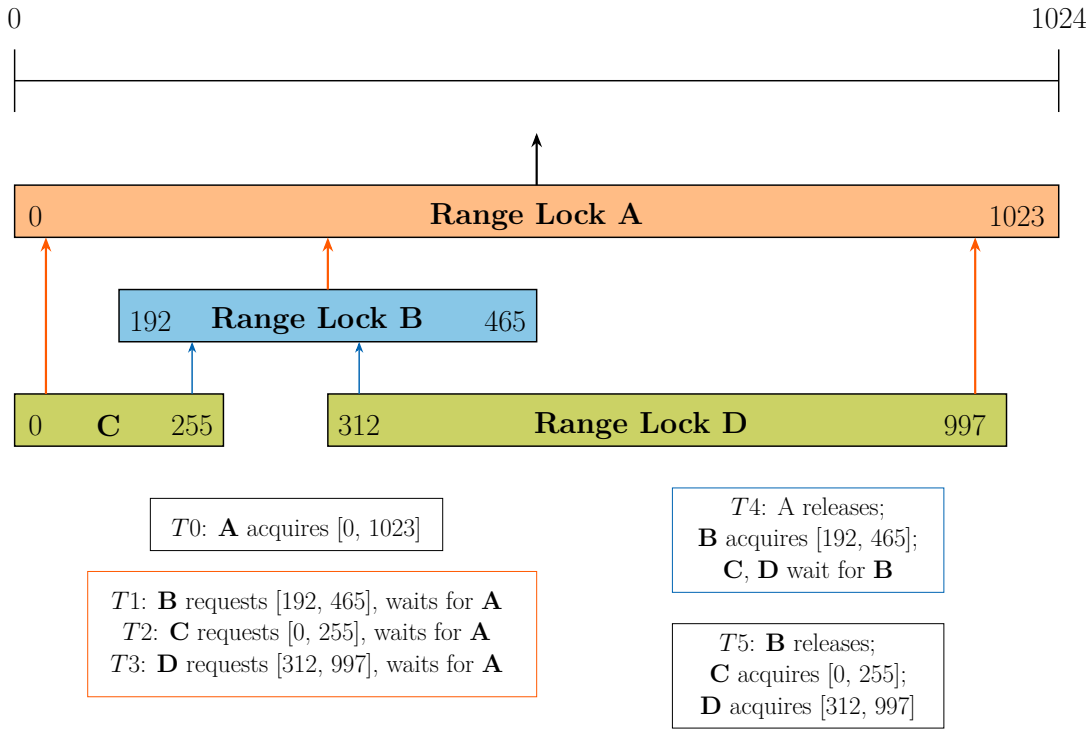

Figure 3.2 Range lock queuing

[3]When a fibril calls lock() on a locked mutex, it puts itself on the mutex's waiting list and sleeps. When the mutex is unlocked, the first fibril in the waiting list is popped out and given the mutex ownership. After this, the new mutex owner fibril is woken up.

3.3.2 Fibril Group Executor

As discussed in Chapter 2, Section 2.3.3, we came up with the idea of using a fibril pool to send I/O requests to the extents, possibly allowing parallel I/O processing.

The fibril pool is utilized by batching workers into groups. This allows waiting for a created set of workers that are part of a group. All worker fibrils must be part of a group. After a worker finishes, its resulting error code (*errno*) can be aggregated among the group workers to indicate that the error was returned by one of the workers.

The pool is created with the pool create function and destroyed with the pool destroy function. The declarations of the create and destroy function are as follows. The pool is represented by the `hr_fpool_t` type. All fibril pool types and function implementations can be found in *fge.{c,h}*.

```
hr_fpool_t *hr_fpool_create(fibril_cnt, queue_sz, worker_data_size);
void hr_fpool_destroy(hr_fpool_t *pool);
```

The pool create function takes the number of fibrils that should be created, the size of the queue that is used for queuing worker jobs, and finally the size of data a single worker needs. The size of single worker's data argument is there for possible pre-allocation of worker data space. This was done to possibly increase performance and, more importantly, to decrease the chance of the user getting `ENOMEM` errno indicating that not enough memory is available when submitting worker jobs. However, to simplify the implementation details and types, we are going to leave out the memory pre-allocation part.

To submit worker jobs, a group must be created. The group creation function takes a pool pointer and the number of workers representing the upper bound of workers that can be submitted. After the group is created, the worker data allocation function must be called to obtain a pointer to the worker data memory that must be initialized for the worker. The worker data memory can, for example, be a structure representing an I/O request including the block address, block count, data pointer, and the target block device. After the worker data is initialized, the worker job can be submitted. After all submitted worker jobs, it is then necessary to wait for the group to finish executing. The group functions' declarations are as follows.

```
hr_fgroup_t *hr_fgroup_create(hr_fpool_t *pool, worker_count);
void *hr_fgroup_alloc(hr_fgroup_t *group);
void hr_fgroup_submit(hr_fgroup_t *group, worker_fcn, worker_arg);
errno_t hr_fgroup_wait(hr_fgroup_t *group, size_t *ok, size_t *fail);
```

An illustration of fibril pool operation can be found in Figure 3.3. The queue in the fibril pool is a circular buffer that is used to submit worker jobs. Elements of the queue are of type `fge_fibril_data_t`, a triple containing a worker function pointer, argument for the function, and group back-pointer. Fibril workers wait for the queue to be populated. After there is something pushed to the queue, and a free fibril worker is available, the worker pops the triple out of the queue and starts executing the provided function with the provided argument. Each worker in the fibril pool at the end of the execution of the job notifies the group and

increments the group "ok counter" if the job returned `EOK` errno, otherwise the "fail counter" is incremented. It can be useful for the group owner to know these counts. Additionally, the group can propagate a specific final errno to the wait function. This is useful for RAID 5. The group wait function waits on a condition variable until all worker jobs submitted via the group finish.

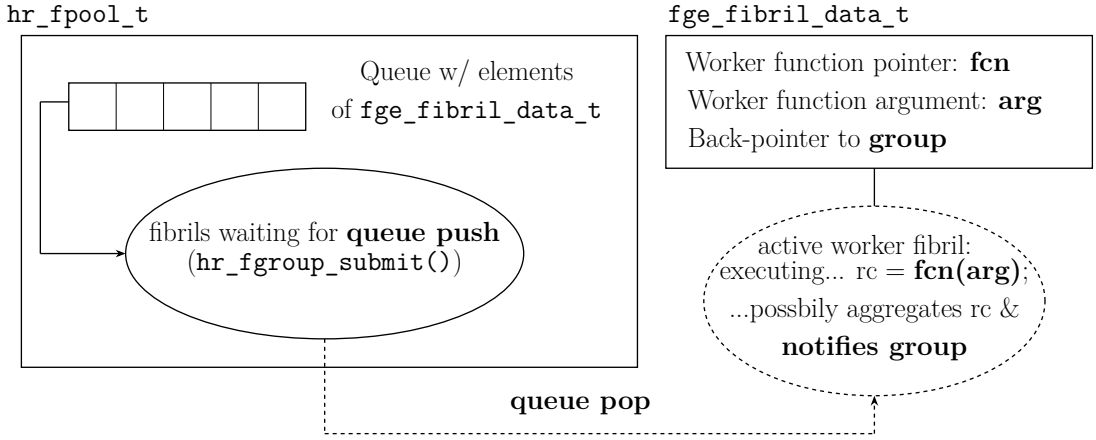

Figure 3.3 Fibril pool internals

3.3.3 ENOMEM safety

It is important for internal RAID I/O requests to fail only when the actual extent to which the I/O request was sent to fails. Because of how HelenOS IPC is currently implemented, it is possible for an IPC call that wants to write or read data from a block device service to fail because there is not enough memory available. This issue was discussed with other HelenOS developers during an online HelenOS developer meeting. It was concluded that the `ENOMEM` errno returned from in our case libblock I/O functions does not need to be specially handled because if we get an `ENOMEM` errno, the whole system will be in an unusable state anyway. Additionally, a new IPC implementation, whose design is currently in progress, shall not have this limitation.

However, we did not want the extents to be marked as invalid only because there was not enough resources available. Suppose a write operation to 2-way mirrored volume. The internal write I/O to one extent succeeds, while the second I/O request gets an `ENOMEM` errno. The I/O either has to be restarted, or the extent invalidated. To overcome this limitation, block I/O operation wrappers that repeat the requests until non-`ENOMEM` errno is returned, were implemented. The following is an example of `block_write_direct()` wrapper function body. The wrapper never returns the `ENOMEM` errno.

```
while ((rc = block_write_direct(service_id, ba, cnt, data)) == ENOMEM)
        fibril_usleep(MSEC2USEC(250)); /* sleep 250ms */
return rc;
```

Complex I/O paths, such as in RAID 5's case, need not only I/O requests to be `ENOMEM` safe, memory sometimes also needs to be allocated in the middle of I/O request execution. For example, for acquiring range locks, or allocating new

fibril pool groups and corresponding worker storage. In other operating systems, such as FreeBSD or OpenBSD, the kernel memory allocator, `malloc(9)`, can be given additional flag `M_WAITOK` that puts the calling process to sleep when there is not enough memory available. Then it waits until enough resources are released by other processes. Without touching HelenOS' libc malloc implementation and creating new malloc interface, the only way to get the desired behavior is to use the same principle as the `ENOMEM` safe libblock wrappers. The following is a malloc wrapper that never returns `NULL`.

```
void *hr_malloc_waitok(size_t size)
{
        void *ret;
        while ((ret = malloc(size)) == NULL)
                fibril_usleep(MSEC2USEC(250)); /* sleep 250ms */

        return ret;
}
```

This malloc wrapper is used to create new range locks in range lock acquisition, for creating fibril worker groups, and in specific RAID 5 I/O workers that are going to be described later.

3.3.4 Volumes

The actual volume is represented by a single structure of type `hr_volume_t`. Its definition can be found in *var.h*. However, to list and describe all the structure members, we first need to introduce some members in their own sections.

Level defined volume operations

Each level has its own set of functions that are specific for some level. These level functions are implementations of the `hr_ops_t` interface. The interface is as follows.

```
typedef struct {
        errno_t (*create)(hr_volume_t *vol);
        errno_t (*init)(hr_volume_t *vol);
        void (*vol_state_eval)(hr_volume_t *vol);
        void (*ext_state_cb)(hr_volume_t *vol, size_t extent, errno_t);
} hr_ops_t;
```

- **Volume state evaluation** function evaluates the volume state based on the states of its extents. These state evaluation functions are effectively an evaluation of a finite state automaton's state and in later sections each level's state evaluation will be discussed.
- **Extent state callback** is called when an I/O request to an extent fails. The arguments specify the extent index and the errno that the I/O returned.

- **Initialization** function is mainly used to calculate usable blocks. It is called only once in volume's lifetime, as later this information is gathered from saved on-extent metadata.
- **Create** function is called for each volume before its upbringing and registering the block device service. Each level's create functions fill the volumes block device operations with the level specific ones defined in *raid{0,1,5}.c*. And calls the volume state evaluation function. After the volume state is evaluated, the `EOK` error code is returned if the volume is usable, otherwise invalid value is indicated and `EINVAL` error code is returned.

Now we can continue to describe the volume structure.

Volume structure

The `hr_volume_t` structure includes the following.

- `hr_ops_t` interface.
- `bd_srvs_t` block service setup which includes already mentioned block device operations `bd_ops_t` (read, write, sync, get block size, get block number). This member is filled in the level specific volume creation function.
- Service ID representing the volume block device service.
- Range lock list. This list contains all currently active range locks. It is protected by a mutex.
- Fibril pool.
- Extent number. The number of extents the volume consists of.
- Block size.
- Truncated block number. This is the block number of the smallest extent.
- Data block number. This block number specifies the number of blocks available to the user.
- Data offset. This specifies the block where data starts.
- Strip size. This number specifies the strip size in number of bytes.
- The RAID level.
- The RAID level layout. Used only by parity levels.
- Device name. This is the device name set by the user at volume creation.
- Volume state. The current state of the volume. The volume state is protected by a read-write lock.

- An array of type `hr_extent_t` representing the extents. The type is a tuple containing service ID of the extent block device service and the extent state.

  The extent service IDs are protected by a read-write lock. The extent states are protected by another read-write lock, that also protects the whole volume state. The need this will be explained later.

- Number of available hotspares.

- An array of `hr_extent_t` representing the hotspares. The array is protected by a mutex.

- Atomic boolean indicating that the volume state might have changed. Its importance will be explained later. Because 1 byte atomic compare exchange operation is not provided by gcc builtins for arm32, ppc32 and mips32 which HelenOS supports, this operation was additionally implemented to enable HelenOS compilation for all its supported architectures.

- Atomic boolean set by the first write to the volume. This triggers the metadata counter to be incremented. This ensures volume re-assembly consistency, while allowing the volumes to be only read while not incrementing the metadata counter.

- Rebuild position. This is the starting block address of currently rebuilt blocks. Only relevant during the rebuild process. Because this is a 64-bit atomic variable, atomic 8 byte load and stores where additionally implemented for the arm32, ppc32, and mip32 architectures to enable HelenRAID compilation for all architectures.

- Open and close counter. We want to count number of volume users so that used volumes cannot be disassembled.

- Additional volume flags. These flags are used for specifying if the created or assembled volume extent is read-only, or if No-op metadata type shall be used.

- Format agnostic metadata interface pointer to the specific implementations of the used metadata format. This will be described in a later section.

- In-memory stored metadata. This removes the need of reading and decoding the superblock before writing it.

The basic structure members are filled or initialized by a utility function `hr_create_vol_struct()` defined in *util.c*.

Volume and extent states

An extent state is represented by the type `hr_ext_state_t` and can be one of the states described in Table 3.3.

The volumes can be in one of the states described in Table 3.4. The volume state is represented by the `hr_vol_state_t` type.

State	Description
Online	Up-to date and working.
Missing	Extent was not provided.
Invalid	Working, but inconsistent with the rest.
Failed	Extent unusable.
Rebuild	The extent is being rebuilt.
Hotspare	The extent is marked as hotspare.

Table 3.3 Extent states

State	Description
Optimal	All extents are in the Online state.
Degraded	One or more extents is not in the Online state. Volume usable.
Rebuild	A rebuild operation is currently in progress.
Faulty	The volume is in an unusable state.

Table 3.4 Volume states

3.4 Server IPC call handling

In this section, we describe how the server handles IPC calls targeted at the server itself and block device IPC calls targeted at volumes.

When the server is started, it is registered under the name *hr* with the location service, and it is assigned a service ID. For servers to be able to handle their own IPC methods, an async fallback port handler must be set. The port handler is invoked when an IPC call arrives for the registered server. Services provided by the server, in our case the block device services representing the volumes, are registered on behalf of the server. This means that all IPC calls for either the server of any of the volumes are handled by the port handler set after the server's registration. The `hr_call_handler()` function is used as a port handler. It distinguishes between IPC calls targeted for the server, and calls targeted at volumes by testing the target service ID for equality against the service ID of the server.

If the service ID is equal to *hr*'s service ID, the IPC call is handled by `hr_ctl_conn()`, a switch case that calls the correct server function corresponding to the client IPC method used.

If the service ID is not equal to *hr*'s service ID, a list of volume structures that represent active volumes is traversed. During the list traversal, the target service ID is compared with all volume service IDs. If a volume with corresponding service ID is found, the call is forwarded to block device service IPC handler[4] along with the volume's `bd_srvs_t` server handle described in Subsection 3.3.4.

The server's main function, server-side IPC method implementations, and the volume structure list are defined in *hr.c*.

[4] `bd_conn()`, defined in *uspace/lib/device/src/bd_srv.c*

Volume creation

The server's active volume list is populated with volume structures created in two ways.

- By actually creating a new volume with the volume create client library function `hr_create()` described in Subsection 3.2.1.
- By assembling existing volumes from metadata. Assembly can be either manual (`hr_assemble()`) or automatic (`hr_auto_assemble()`). Difference between these client methods is described in Subsection 3.2.1.

In both cases, the path from the creation of the volume structure to the final volume registration follows a similar pattern. However, in this section, we will only show the process of creating and registering a newly created volume. Both the assembly and the automatic assembly will be described in a later section. The maximum number of extents is hardcoded to `HR_MAX_EXTENTS` defined in *uspace/lib/device/include/hr.h*.

The following is a description of the steps made by `hr_create_srv()` defined in *hr.c*.

1. `hr_create_vol_struct()` is called. It allocates memory for the volume structure, assigns `hr_ops_t` the specified level's operations, initializes locks, atomic variables, and the range lock list.
2. As mentioned in Subsection 3.2.1, the create IPC method sends a `hr_config_t` structure containing extent service IDs to the server. These extents are initialized via libblock and the truncated block number is calculated. This is done via `hr_init_extents_from_cfg()` defined in *util.c*.
3. Level specific `hr_ops_t` init function is called. It calculates the level specific block counts with respect to the truncated block number and the number of extents.
4. Metadata population. The format-agnostic metadata handling interface, as discussed in Chapter 2, Section 2.4, needs to provide metadata population function, so that the metadata can be later saved to the extents to enable assembly. The implementation of the metadata handling interface is discussed in a later section.
5. Level specific `hr_ops_t` create function is called.
6. Metadata is saved to the extents.
7. The volume is registered as a new service with `hr_register_volume()` defined in *util.c*. The volume is also assigned its service ID. Currently, volumes are registered under the new Location Service category *raid*.

Stopping (disassembling) volumes

The server counterparts of stopping volume client interface functions described in Subsection 3.2.1, are `hr_stop_srv()` for stopping a single volume and `hr_stop_all_srv()` for stopping all volumes, respectively. These functions internally call the function `hr_remove_volume()` defined in *util.c*. The volume removal function checks the number of currently opened block service connections to the volume. The volume's `bd_ops_t` open and close implementations are called when someone does `bd_open()` or `bd_close()`. These are called when initializing and finalizing libblock on a block device. If the number of open connections is greater than zero, the removal function returns with `EBUSY` error code, else it removes the volume structure from the server's volume list, finalizes libblock for active extents and frees the structure. Finally, the volume's service ID is unregistered with the Location service, and it returns with the `EOK` error code if the unregister operation was successful.

The single volume removal client function gets answered the return value that the server got from the removal function. The all-volumes removal server function calls the removal utility function for each volume and tries to stop all volumes even if there was an error in removal for previous iterations. The client either gets `EOK` signaling no volume was busy, or `EBUSY` if at least one of the removals returned `EBUSY`.

Marking an extent as failed

The server counterpart implementation of the client function for failing extents is `hr_fail_extent_srv()`. The volume is found in the volume list and if the extent's state is not Missing or Failed, its state is set to Failed, libblock for its service ID is finalized, and the volume state is evaluated. Additionally, if the metadata format used supports superblock erasure, then the block is erased as well. This will be shown in the metadata-agnostic interface design and implementation.

Hotspare addition

The user hotspare addition method server counterpart is `hr_add_hotspare_srv()`. The volume level and metadata type are checked to determine whether they support hotspare addition and then the function `hr_util_add_hotspare()` defined in *util.c* is called. If the number of current hotspares is less than `HR_MAX_HOTSPARES`, the device is checked if it is large enough, and placed into the hotspare `hr_extent_t` array. After that, the volume state is evaluated. The maximum number of hotspares is limited by `HR_MAX_HOTSPARES` defined in *uspace/lib/device/include/hr.h*.

Getting volume states

The server-side implementation of getting brief state of all volumes is represented by the `hr_get_vol_states_srv()` function. For each volume a pair of the volume's service ID and its state is sent to the client. The pairs are of type `hr_pair_vol_state_t`.

Getting detailed volume information

The server-side implementation of getting detailed volume information is represented by the `hr_get_vol_info_srv()` function. The server fills the type `hr_vol_info_t` defined in *uspace/lib/device/include/hr.h* from the volume and sends it to the client.

3.5 RAID 0

Non-redundant striping is the simplest RAID technique that is implemented in HelenRAID. That is because no redundancy is needed to be handled, no rebuild operation, and the I/O transformations are the same for reads and writes. The RAID 0 implementation is in *raid0.c*.

3.5.1 RAID 0 state and management operations

In this subsection, we describe how `hr_ops_t` are implemented for RAID 0.

RAID 0 state evaluation

The volume state depends on the states of the extents. The state itself and state transitions during runtime can be represented by a form of state automaton. The state evaluation of RAID 0 can be found in Figure 3.4. The ellipses represent the volume state. The text in **boldface** is the name of the state of the volume, and the expression under the name of the state describes the conditions that must be met in order to be in that state. The starting state is based on these conditions. $Online_{ext} = N$ means that the number of extents in the Online state is equal to N. When a volume is already in some state, the state can change based on the number of extents in a specific state, described by the arrow comment. $Online_{ext} - 1$ means the state of one extent changed from Online to something else, generally to Failed or Missing state. The absence of arrows coming out of an ellipse means that once the volume is in that state, its state cannot be changed.

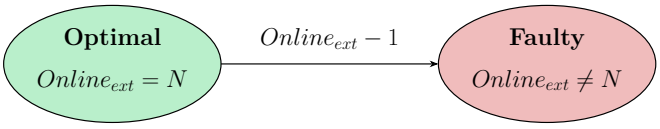

Figure 3.4 RAID 0 state evaluation

The volume state evaluation function pointer in `hr_ops_t` is set to the function `hr_raid0_vol_state_eval()` that changes the volume state depending on the number of extents in the Online state.

RAID 0 extent state callback

The extent state callback is a simple function that is called when an error code other than `EOK` (signaling no error) is returned by an I/O request sent to an extent. It changes the extent state based on that error code. In this instance, it changes the state to Missing when *ENOENT* signaling no entry is returned, and to Failed

when anything else was returned. Additionally, when an extent state is changed to either Missing or Failed, the volume state is changed to Faulty.

During volume and extent state updates, the volume's state read-write lock is write-locked.

The extent state callback is represented by the `hr_raid0_ext_state_cb()` function.

RAID 0 volume initialization

As mentioned above, in Subsection 3.3.4, the initialization function is called once in volume's lifetime. It calculates the number of blocks that the exported volume service is going to provide. For RAID 0 it is

$$C_{vol} = (C_{ext} - M_{blocks}) \cdot N$$

where C_{vol} is the usable number of volume blocks provided to the user, C_{ext} is the size of the smallest extent in blocks, M_{blocks} denotes the number of blocks used by the metadata and N is the number of extents.

This equation does not take unused space into account, but as mentioned in Chapter 2, the native metadata format is preferred and it does not waste any blocks.

The volume data offset is set along with the strip size in this function as well. The strip size is set to the maximum IPC transfer size divided by the number of extents. If the division result is not a power of 2, it is rounded down to the closest power of 2.

All levels have almost identical init functions differing in the block number calculation.

RAID 0 volume creation

The RAID 0 volume creation function evaluates the state and, depending on the result, it either returns an error code indicating unusable state or proceeds to set the volume's block device service operations (`bd_ops_t`) to RAID 0 specific ones.

3.5.2 RAID 0 I/O

In this subsection, we will show how logical block addresses are mapped to extent block addresses and how are RAID 0 workers dispatched.

I/O mapping

To map I/O requests to extents and offsets within the extents, we must calculate the starting extent and the row to which the block address corresponds to extent-wise.

To get the needed parameters, we need to calculate the strip number and the offset inside the strip, because strips are generally larger than one block.

Where LBA is the requested block address, $stripsize$ is the strip size in blocks, $strip_{offset}$ is the request offset with respect to $strip$, $extent$ is the index of the extent on which $strip$ resides and $Stripe$ is the corresponding stripe number.

With this information I/O requests to extents can be dispatched.

Parameter	Expression
strip	$\frac{LBA}{stripsize}$
$strip_{offset}$	$LBA \mod stripsize$
extent	$strip \mod N$
Stripe	$\frac{strip}{N}$

Table 3.5 RAID 0 I/O mapping parameters

I/O worker dispatching

RAID 0 uses identical I/O workers for reads and writes, the difference being only in the I/O function used, `block_read_direct()` and `block_write_direct()` for reads and writes, respectively. The worker used is `hr_io_worker()` defined in *io.c* and it uses the `hr_io_t` type defined in *io.h* as its storage. The storage structure contains needed parameters such as the block address, the size of the request, the data buffer pointer, the I/O type (read/write), the extent index, and the volume structure pointer. Then it calls the corresponding libblock functions. If an error was indicated, the volume's extent callback function is called.

From the initial strip number and the logical request length, we can calculate the number of strips that the logical I/O request spans. Each strip touched will be assigned one fibril worker. A fibril group is first created and then workers are dispatched as follows. The extent LBA (we will denote it by PBA) corresponding to the logical LBA is calculated as $PBA = Stripe \cdot stripsize + strip_{offset}$. The PBA is additionally increased by the data starting offset of the volume. The size of the request is calculated as $stripsize - strip_{offset}$. The worker fibril's storage is filled with PBA as the block address, the size of the request, the data buffer pointer, the I/O type (read/write), the extent index and pointer to the volume structure. It is then dispatched via `hr_fgroup_submit()`. This is repeated until the whole I/O range is not covered in a round-robin fashion with respect to extents. When the end of a stripe is reached, the *Stripe* variable is incremented and the *extent* index is set to 0. With each I/O dispatched, the data buffer pointer is incremented with respect to the dispatched I/O size.

After all workers are dispatched, the group is waited on, and the number of failed I/O requests is counted. If the failed count is greater than zero, we know that the request was not completely processed and that the volume state must be Faulty. In that scenario, we return the `EIO` error code indicating that an I/O error occurred. Otherwise, we return `EOK`.

Note that no range locking is done. This is because no invariant is kept, and synchronizing data accesses to the same block addresses is a responsibility of the upper layer.

3.6 RAID 1

RAID 1 implementation is bit more complex than non-redundant striping because it is fault-tolerant and supports the rebuild process. Additionally, the implementation supports multiple read strategies that decide how a read I/O is handled internally. The re-assembly consistency must also be handled because the volume can be used

even in scenarios where not all the extents are present. RAID 1 implementation is in *raid1.c*.

3.6.1 RAID 1 state and management operations

In this subsection, we describe how `hr_ops_t` are implemented for RAID 1.

RAID 1 state evaluation

RAID 1's state evaluation automaton can be found in Figure 3.5. Its elements have the same meaning as in RAID 0's case. The only additional element is that more arrows representing the same operation are coming out of some states. In such a case, all target conditions are checked, and based on the condition evaluation the state changes. It is always deterministic.

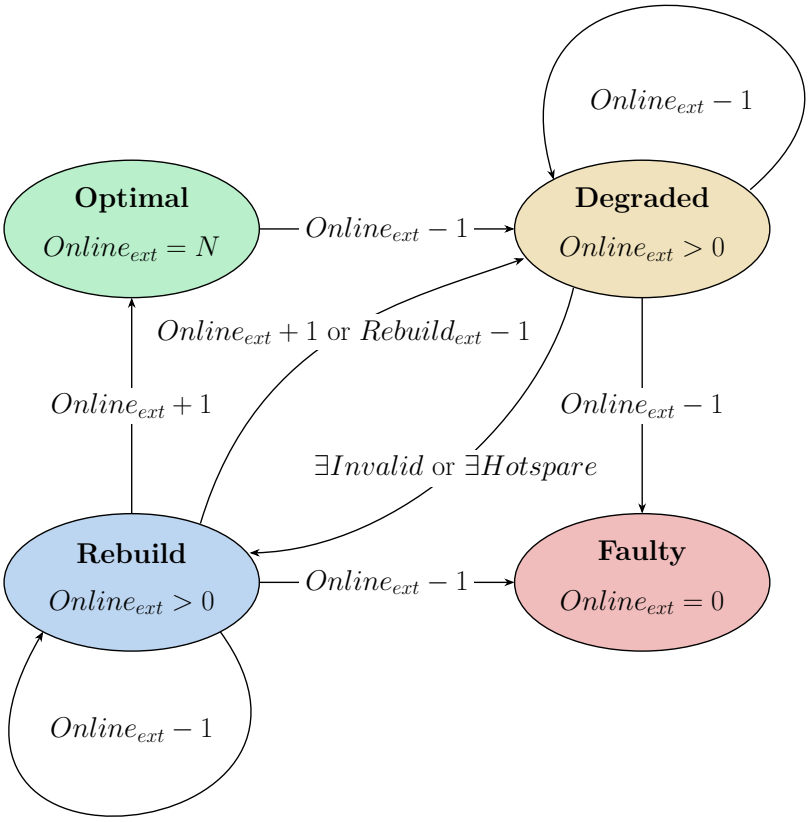

Figure 3.5 RAID 1 state evaluation

The initial state depends on the number of extents in the Online state. If all extents are Online, the starting state is Optimal, else if the number of Online extents is less than N but greater than zero, it is Degraded. Else the Faulty state. The starting state is never Rebuild. The state change is changed to rebuild only under certain conditions. The conditions are as follows.

- The volume must be in the Degraded state.
- There must exist an extent in the Invalid state, or an hotspare extent.
- The rebuild must be allowed by the metadata format used.

Note that there can be only one rebuild operation in progress with this model. While the active rebuild operation is active, more Online extent failures are allowed under the condition that there is at least one Online extent. A rebuild operation can finish in three ways – no Online extent is left, the rebuilt extent fails, or the operation successfully finishes and the number of Online extents increases by one. Note that even if the rebuild operation successfully finishes, the state can change back to the Degraded state.

The implementation closely follows the described automaton, with minor differences due to technical reasons such as the rebuild operation always changing the volume state to Degraded when it finishes, and then the "initial" state is re-evaluated, possibly jumping from Degraded to Optimal. Note that the semantics are the same.

However, the actual `hr_ops_t` volume evaluation function for RAID 1 is a wrapper on top this automaton (`hr_raid1_vol_state_eval()`). This is for the following reasons.

- Because the volume state transitions are not as simple as in RAID 0's case, the volume state must be re-evaluated after every I/O request. To optimize away the need for the automaton evaluation every I/O, an atomic boolean signaling extent state change is utilized. `atomic_compare_exchange` is used, so that multiple fibrils do not waste time evaluating the same state.
- The metadata on-disk counter is incremented and the metadata is saved to the extents every time the state changes, in an effort to keep the re-assembly consistency across the extents.

Note that when any state is inspected or updated, the volume's *extent read-write lock* is at least read-held. This is because of a possible rebuild operation initialization, which can possibly change an extent's service ID.

RAID 1 extent state callback

The extent state callback function for RAID 1 is identical to RAID 0's with the additional setting of the atomic boolean signaling dirty state. It is implemented by the `hr_raid1_ext_state_cb()` function.

RAID 1 volume initialization

The number of blocks in RAID 1's implementation of `hr_ops_t` init function is as follows.

$$C_{vol} = C_{ext} - M_{blocks}$$

RAID 1 volume creation

The RAID 1 volume creation function evaluates the state with internal function `hr_raid1_vol_state_eval_forced()` as opposed to the evaluation function exported by `hr_ops_t` described above, because we do not want to trigger metadata saving on assembly. This allows the volume to be assembled and read from without invalidating Missing extents. When the state is not Faulty after evaluation, the creation function returns `EOK`.

3.6.2 RAID 1 I/O

While RAID 1's I/O requests do not need to be transformed like in RAID 0's case, the I/O code must handle locking and, most importantly, take rebuild operation progress into account.

Writes

Write requests as opposed to read requests must lock their operating block address range with range locks. This is because even though the upper layer is responsible for synchronizing potentially racing accesses, we must protect the mirroring invariant. After the range lock is acquired, we load the current rebuild position, and based on that the position we replicate and dispatch the internal write I/O to extents. The write I/O is dispatched to each extent in the Online state, and if an extent is in the Rebuild state (is being rebuilt), we send the write to it if the starting block address of our I/O is below the current rebuild position. If the rebuild position is bellow after block address, we do not send it to the Rebuild extent, as the write is going to be replicated by the rebuild operation later. During the I/O dispatching, the volume's state lock is read-held to correctly inspect the extents' states. However, for a worker to be able to change an extent's state in case of a failure, the read-held states lock must be unlocked. This, however, creates an opportunity for rebuild to start and possibly change an extent's service ID. To ensure workers have consistent targets, the extent lock must be read-held during the whole I/O.

RAID 1 uses the same workers as RAID 0 does, `hr_io_worker()` defined in *io.c*. After dispatching the workers, we wait for the fibril group to finish executing. After the group finishes execution, the range lock is released and the successful number of worker jobs is checked, and if it is at least one, we return `EOK`, otherwise not a single write was successful, and the volume must be in the Faulty state. In that case, we return `EIO`, indicating I/O error.

The special thing that the first write to a mirror does is increment the metadata counter and save it to extents. This ensures re-assembly consistency.

Reads

Reads in RAID 1 do not alter the level's invariant, therefore there is no need for locking. There are multiple ways in which volume read I/O requests can be internally processed. Regardless of strategy, reads can only be sent to so-called *"good"* extents. An extent is good if it is either in the Online state, or it is in the Rebuild state, and the read I/O block address range ends before the rebuild

position. During the evaluation of an extent's "goodness", the state lock is read-held, and then it is unlocked to allow workers to possibly write-lock the state lock in case of failure. The extent lock is read-held during the whole I/O, as in the write I/O case.

We implemented the following strategies.

- The **first** strategy. Read from the first good extent starting from the extent with index 0.
- The **closest** strategy. Read from the extent whose last seek position is the closest to the user request's starting block address. The `hr_io_worker()` in RAID 1's case additionally atomically updates the extent's last position in the volume's array dedicated to last seek positions. This can be useful for spinning disks.
- The **round-robin** strategy. Good extents are rotated for each read I/O.
- The **split** strategy. The I/O is split between good extents similarly to RAID 0. The need to split very small I/O requests is removed by a set *split threshold*. The I/O size is divided by the number of good extents, and if it is greater than or equal to the set threshold, the I/O is split among the good extents. Otherwise, the first good extent is taken.

The default is the split strategy with 1 KiB split threshold. The strategies are set at the compile time in *var.h*

The strategies were inspired by the possible configurations of read I/O in FreeBSD's `gmirror(8)` [32].

3.6.3 RAID 1 rebuild

A rebuild operation is started in the state evaluation by spawning a fibril to the function `hr_raid1_rebuild()` and is initialized with the function `hr_init_rebuild()` defined in *util.c*. The initialization is successful if there is either a hotspare or an extent in the Invalid state, with the Invalid extents having priority. In the case of rebuilding onto a hotspare, the service ID of the hotspare is put in place of the Failed/Missing extent, and the extent is marked as Rebuild. In the case of a rebuild of an Invalid extent, the only thing changed is the extents state to Rebuild. During the initialization, the extent lock is write-held, as well as the state lock, and the volume's state is changed to Rebuild. There is also a special case of successful rebuild initialization that is done when an extent is in the Rebuild state. This is due to the support of saving rebuild progress, so that the volume can be disassembled or the system restarted without the need of having to restart whole rebuild process from block address 0 again. The progress is saved every `HR_REBUILD_SAVE_BYTES` defined *var.h*. There can be only one rebuild operation at a time, which means that the data can be restored to a single extent.

The rebuild itself process is a simple synchronous chain of the following.

1. Lock the current rebuild range.
2. Updating the volume rebuild position.

3. Sending a basic user I/O read to the volume.
4. Writing the data read in the previous point.
5. Unlock the range.
6. Until not all block addresses are rebuilt, shift the rebuild range, go to Step 1.

The size of the I/O requests is `DATA_XFER_LIMIT`, the maximum transfer size allowed by IPC. Currently, it is 64 KiB. Also note that sending a basic user I/O reads the correct data because of the rebuild position. And when multiple good extents are present, the read request will be possibly processed faster thanks to a more optimal read strategy used then the first strategy.

3.7 RAID 5

RAID 5 implementation is by far the most complex part of the whole HelenRAID implementation. This is due to the support for completely concurrent internal per-stripe I/O. This means that I/O spanning multiple stripes is concurrently executed on stripe level and also inter-stripe level. The implementation is split into multiple files – *raid5.c*, *parity_stripe.{c,h}* and *io.{c,h}*. We do not explicitly mention RAID 4 because the only thing different is the I/O mapping, other mechanisms are identical to RAID 5. To remove the need to mention both RAID 4 and RAID 5, we refer to both of them as RAID 5. When necessary, RAID 4 is explicitly mentioned.

3.7.1 RAID 5 state and management operations

In this subsection, we describe how `hr_ops_t` are implemented for RAID 5.

RAID 5 state evaluation

RAID 5's state evaluation automaton can be found in Figure 3.6.

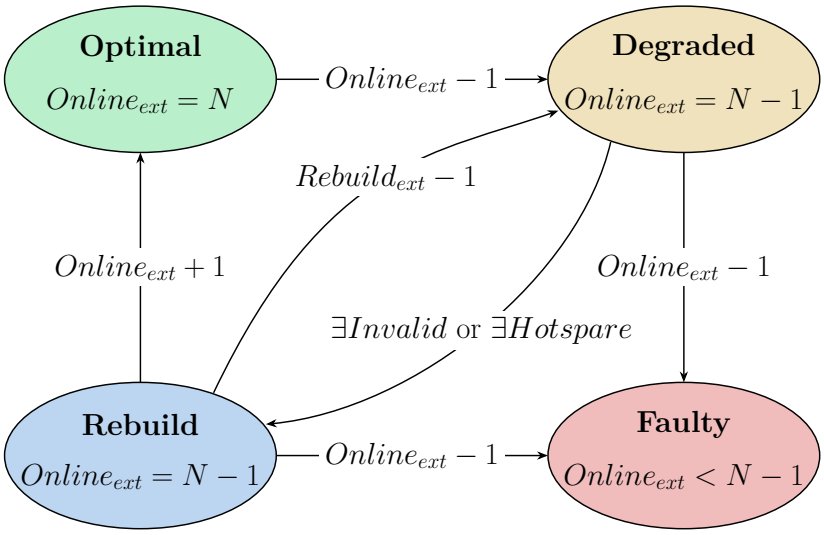

Figure 3.6 RAID 5 state evaluation

The initial state depends on the number of Online extents. If there is N extents in the Online state, the volume state is Optimal. If there is $N-1$ Online extents, the initial state is set to Degraded, and finally if there is less than $N-1$ Online extents, the initial state is set to Faulty. The initial state is never Rebuild. The Rebuild operation is triggered from the Degraded state if the following conditions are met.

- The volume is in the Degraded state.
- There must exist an extent in the Invalid state or an available hotspare.
- The rebuild must be allowed by the metadata format used.

When an Online extent fails during rebuild operation, the volume state is changed to Faulty. If the rebuilt extent fails, the volume state is set back to Degraded. If the rebuild operation successfully finishes, the volume's state is restored to Optimal.

As in the case of RAID 1, the volume state evaluation function is a wrapper on top of this automaton. The wrapper atomically checks if the state has changed, so that there are no CPU resources wasted to redundantly check the automaton. If the state has changed, it additionally increments the re-assembly consistency counter and saves the metadata to extents before evaluating the automaton.

The state evaluation function is represented by `hr_raid5_vol_state_eval()`.

RAID 5 extent state callback

The extent state callback function changes the extent's state depending on the error code, to Missing (`ENOENT`) or Failed ($\neq$`EOK`). Additionally, it sets the atomic boolean signaling dirty state.

The callback function pointer is set to `hr_raid5_ext_state_cb()`.

RAID 5 volume initialization

The init function sets the strip size, the layout, the data offset and calculates the volume's usable number of blocks as follows.

$$C_{vol} = (C_{ext} - M_{blocks}) \cdot (N - 1)$$

The strip size is set to the maximum IPC transfer size divided by $N-1$, and in case when the result is not a power of 2, it is rounded down to the closest power of 2.

RAID 4 uses the last extent for dedicated parity and RAID 5 uses the Rotating Parity N with Data Restart layout (left-asymmetric) by default.

RAID 5 volume creation

The creation function evaluates the state with the evaluation automaton by internal function `hr_raid5_vol_state_eval_forced()`, not with the public state evaluation wrapper. If the state is not Faulty, the creation function proceeds to set the block device service operations to RAID 5 specific ones, and returns `EOK`.

3.7.2 RAID 5 I/O

The processing and execution of RAID 5 I/O is split into multiple phases and steps. This is due to multiple reasons, the main being that to support maximum concurrency, the I/O must be split per-stripe. This is because when parity is rotated, each stripe I/O can take different code paths depending on parity state.

In Chapter 1 was a brief introduction to RAID 5 I/O operations, however to support the described internal I/O concurrently, we must handle numerous situations depending on the I/O span across extents. And, additionally take the unusable extent into account when the volume is in the Degraded or the Rebuild state.

Stripes

We designed a type `hr_stripe_t` to hold all the necessary information needed to dispatch concurrent internal I/O. It is a structure defined in *parity_stripe.h* and includes members described in Table 3.6. More detailed functionality of member's is described in the following text.

Member	Description
I/O type.	Type of the I/O request. (Read/Write)
Strips touched.	Number of strips touched.
Partial strips touched.	Number of partially touched strips.
Subtract hint.	Whether subtract-write (small-write) is preferred.
Per-extent span.	Includes I/O span range for each extent, pointers to data, strip offset and block count.
I/O height ranges.	Total I/O height ranges.
Parity extent.	The index of the parity extent for this stripe.
Parity storage.	Memory for computing parity.
Parity lock.	Mutex protecting the parity storage.
Parity counting info.	Includes the number of total parity commits to be done and parity commits already done.
Parity condvar.	Used for waiting on parity commits.
Fibril group.	Pointer to group executing the stripe.
Abort hint.	For aborting parity commit waits in case of failure.
Done hint.	Set when stripe execution is finished.
Final errno.	Errno representing the result of stripe execution.

Table 3.6 `hr_stripe_t` members

When read/write I/O requests are made to a RAID 5 volume, the I/O is logically split into stripes, and for each stripe relevant parameters are calculated. Reads and writes differ in the calculations made, because writes need a more detailed information about the stripe, such as the number of partial strips touched, in order to decide whether to use the reconstruct or the subtract (small) write operation for the stripe. After the needed initial parameters are calculated, then the *extent span* is filled in for each extent. This follows the same mechanic as RAID 0 I/O, where I/O was dispatched to each strip in a round-robin per-extent manner. The data and parity extents are calculated with respect to the

layout used. Mapping for each layout described in Chapter 1 is implemented, however, to save space we are going to leave out the specific equations. The mapping function for calculating parity extent is `hr_raid5_parity_extent()` and `hr_raid5_data_extent()` for data extent, both defined in *raid5.c*.[5] Opposed to RAID 0, in RAID 5's case only the I/O parameters such as the extent, I/O range for the strip, strip offset, data pointer and block count are saved for each stripe in the extent span structure and no actual I/O is dispatched in this phase. The filled-in information up to this point is invariant and does not change for the rest of request processing. The filling is performed in *raid5.c*.

After this information is filled, the range lock for each stripe is acquired, the extent lock is read-locked, the volume state is checked for unusable extents and a so-called *stripe execution* is initialized.

Stripe execution

Each stripe along with a "bad" extent[6] if there is one, is handed to the stripe execution function `hr_execute_stripe()` that further selects specific function for dispatching stripe I/O. All these execution functions are defined in `parity_stripe.c`. The functions approximate the number of fibrils needed[7] and create a fibril group. Then possibly other operations are done with the extent spans, such as merging them to get the total I/O "height" in the stripe or calculating the span's non-overlapping extensions. We refer to stripe I/O height as the union of ranges across all extent spans. There can be one or two total ranges the I/O spans per-stripe height. The two ranges situation arises when a user I/O starts at a block address mapped to an end of a strip, and then the I/O "overflows" to the next strip, but not so much that it overlaps with respect to rows. The total I/O height is needed to know which part parity will have to be read or written. An example of non-overlapping user I/O is shown in Figure 3.7 that spans 2 strips (S0 and S1) with the I/O height projected to on the parity. If the ranges overlapped, there would be only one total range.

After this, submitting of appropriate I/O workers to the extents is started. The functions are non-blocking with respect to waiting for the dispatched I/O requests, because we want to execute each stripe concurrently. After each stripe execution is dispatched, each stripe is waited for via its fibril group pointer. After all stripes finish execution, the volume state is evaluated. If it is not Faulty, each stripe's final errno is checked. If all stripes finished successfully, the range locks are released and the user I/O request is completed. However, a failure scenario where stripe execution finishes unsuccessfully but the volume is not in the Faulty state exists. In this case, the I/O can still be completed by restarting the stripe execution with specified unusable extent. To describe this possible scenario, we must first introduce internal RAID 5 I/O workers and *parity commits*.

[5]The reader can find the equations in document format in the SNIA's DDF specification [4] which includes the mapping function equations for each layout.

[6]Bad extent is one that is not in the Online or Rebuild state, additionally extents are bad if they are in the Rebuild state and the rebuild position is below the first stripe touched by the I/O in case of writes, or that the I/O ends above the rebuild position in case of reads.

[7]With some types of I/O the exact number of fibrils needed is known at the beginning, while some require just an upper bound to be used, since additional fibrils might be needed during the execution.

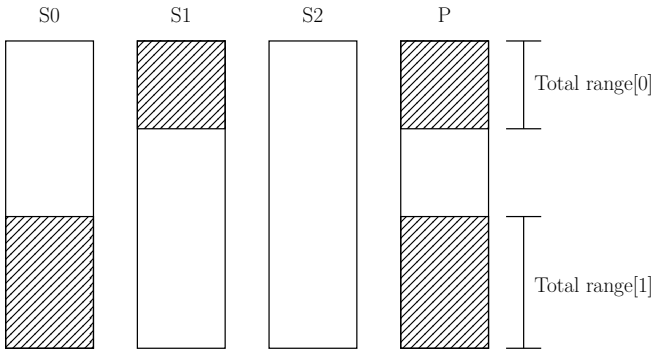

Figure 3.7 Total stripe I/O height ranges

Parity commits

Parity commits are a mechanism that ensures volume consistency in different types of operations, mainly the reconstruct-write operation. However, it is also useful in general for operations whose output depends on the parity, such as degraded reads. The mechanism works as follows. In the beginning a number of parity commits to be done is set[8]. Workers wait on the conditional variable dedicated for this in `hr_stripe_t`, until all parity commits are done. The act of committing is just incrementing parity commits done counter and notifying fibrils waiting on said conditional variable.

To quickly illustrate the need to wait for parity commits, suppose a 3-extent RAID 5 volume with a strip size of 1 block for simplicity. Let parity be written as $P^{old} = D_0^{old} \oplus D_1^{old}$, where D_0^{old} and D_1^{old} data blocks on different extents. Now suppose a reconstruct-write to strip D_0 with data D_0^{new}. The parity after the write should be $P^{new} = D_0^{new} \oplus D_1^{old}$. If the new data were written to D_0 without waiting for the read of D_1^{old} to complete the parity, the following situation could arise. While reading the old data D_1^{old}, a failure happens, which means that we cannot write any parity, so no parity is written. However, the write to D_0 succeeded and now we have D_0^{new} and P^{old} in the stripe. However, D_0^{new} and P^{old} are independent, making the data on D_1 – D_1^{old} unreconstructible. This can be prevented by waiting while also expecting a possible abortion. If the write with D_0^{new} waited for the read of D_1^{old} and the failure occurred, the write would be aborted, so there would be no inconsistencies. Then the write can be restarted with the knowledge that D_1 has Failed, so subtract-write operation would be used.

Workers

Here, we describe all worker types used within processing of both read and write I/O requests. RAID 5 workers use the `hr_io_raid5_t` type for storing necessary information for execution. It is similar to `hr_io_t` used by RAID 0, 1, but additionally `hr_stripe_t` back-pointer and strip offset is included[9]. The worker implementations are in *io.c*.

[8]However, with a help of a boolean hint, this number does not have to be the final number of parity commits needed, because as previously mentioned, some operations do not know the total number of fibrils (workers) needed in the beginning.

[9]Strip offset inclusion is necessary because since the parity is allocated for whole stripe height and the worker's I/O request might be offset from strip's start – the basic worker parameters do not provide location with respect to strip start, which is necessary for committing the data to

Diagrams showing the actions each worker takes can be found in Figure 3.8. The topmost dotted arrow means that the data is passed from memory (usually the pointer to memory containing user input data obtained in the `bd_ops_t` request). **End** denotes the end of worker execution, and arrow pointing to the **End** text denotes that the worker outputs some data (usually into memory pointed to by the data pointer provided by a read operation call). The **XOR & COMMIT** operation takes data blocks as input, and XORs them into the stripe's parity, and increments the number of parity commits done. This is done while holding the parity mutex. The **WAIT** node denotes waiting on a conditional variable until all needed parity commits are done. However, the **WAIT** operation is where possible interruption can happen, which makes the worker end its execution there and not continue past the **WAIT** node.

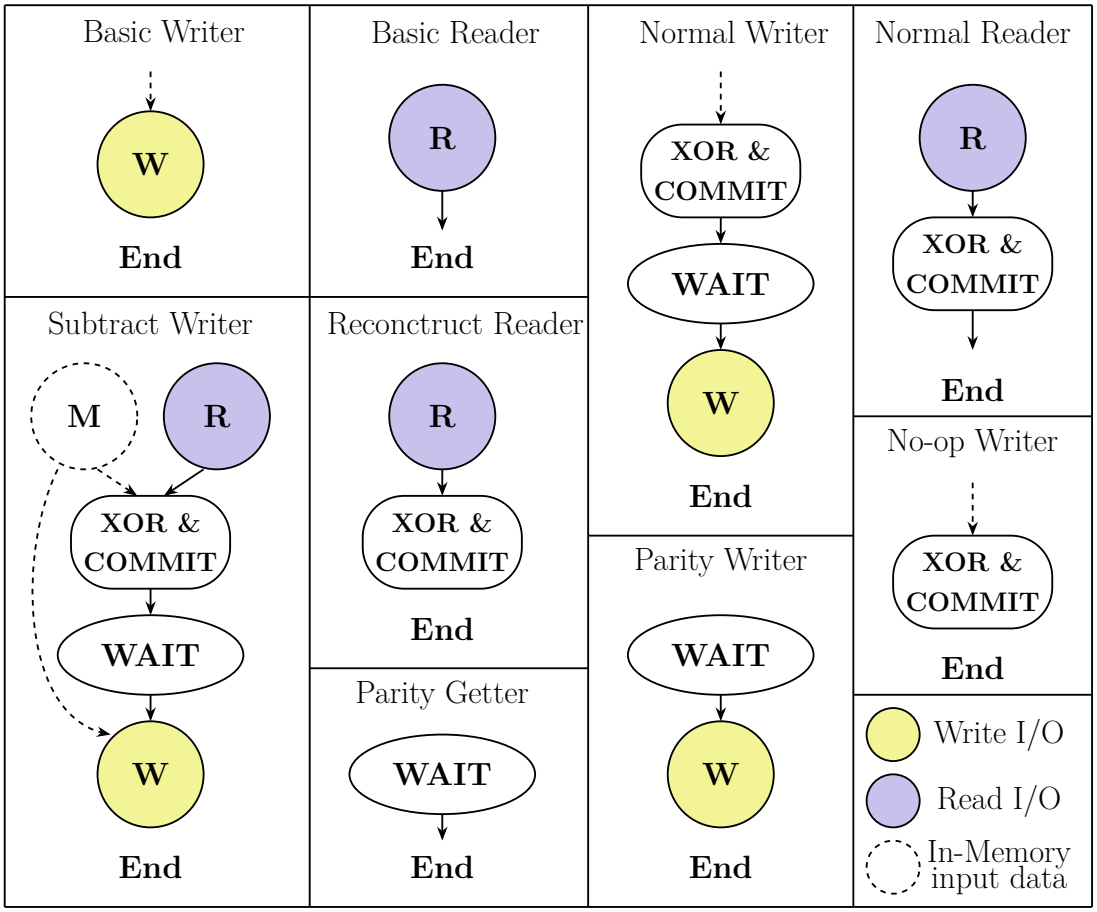

Figure 3.8 RAID 5 workers

When worker's execution is aborted, it returns `EAGAIN` to indicate abortion. To ensure that the `EAGAIN` error code cannot be returned by anything else, the `ENOMEM`-safe libblock operation wrappers regard the `EAGAIN` error code as `EIO`, as the extent would get Failed either way.

- The **Basic Reader** and **Basic Writer** are identical to RAID 0's and RAID 1's I/O worker, they do not touch the parity or handle any stripe-specific information.

right parity memory offset.

- The **Normal Writer** takes the input data, XORs, and commits it; then waits for all commits (it can be aborted), and finally writes the input data to the target extent.
- The **Normal Reader** reads data from the target extent into storage provided by the caller, if the read fails, the abort hint is set and all parity waiters are notified; if it succeeds, it XORs and commits the data read.
- The **Subtract Writer** sends a read to the target extent, and in a case of failure it sets the abort hint and notifies the parity waiters; if the read succeeds, it XORs and commits the input data and the data read; then it waits for other parity commits (can be aborted); finally, it writes the input data to the target extent.
- The **Reconstruct Reader** reads data from the target extent into its own-allocated buffer (note that there is no user-provided buffer); if the read fails, abort hint is set; if it succeeds, it XORs and commits the data read and frees the allocated buffer.
- The **Parity Getter** waits for all parity commits, and then copies the parity to user provided buffer.[10]
- The **Parity Writer** waits for all parity commits and then writes the parity to the target extent. Note that the Parity Writer can be targeted to any extent, not just the parity extent. Besides writing parity while processing user I/O requests, this is also useful for the rebuild operation, where data are reconstructed inside the stripe's parity storage, to be written to the extent being rebuilt.
- The **No-op Writer** just XORs and commits input data blocks to the parity.

Reads

After user read I/O requests are split into `hr_stripe_t` stripes, they are handed to `hr_execute_read_stripe()`. If no parity is involved, Basic Readers are sent to the target extent spans, same as non-redundant reads in RAID 0's case. This is when the volume is in the Optimal state, or when the parity extent is unusable or the unusable data extent is untouched by the I/O.[11] Parity is involved in a case where data are wanted from an unusable extent. We define the I/O range for the bad extent as *bad range* and define the I/O request range on a good extent as *good range*.

1. For each good extent, calculate a subrange of the bad range not in the good extent's good range. We call this the *non-extension*. For each such non-extension, Reconstruct Readers are dispatched.
2. For each good range, we calculate its overlap with the bad range. If such overlap exists, a Normal Reader is dispatched.

[10]The Parity Getter worker is necessary for asynchronous execution of stripes.

[11]We say unusable parity extent and unusable data extent, but we mean it only in the context of the current stripe, as parity is rotated in RAID 5.

3. For each part of good ranges non-overlapping with the bad range Basic Readers are dispatched.

4. Finally, a Reconstruct Reader and Parity Getter are dispatched to the parity extent on the bad range.

An example of stripe I/O in which all mentioned workers are used is shown in Figure 3.9.

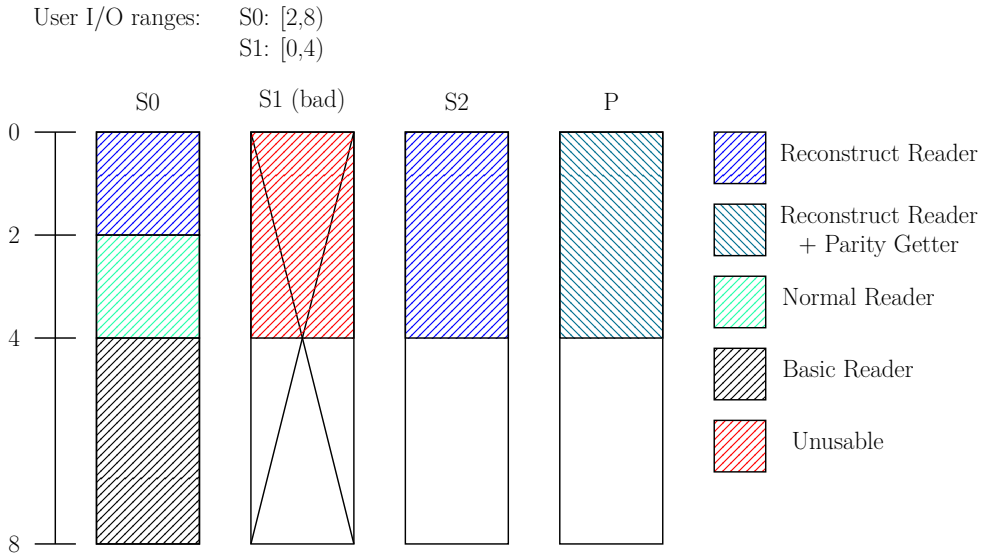

Figure 3.9 RAID 5 degraded read

Optimal reconstruct-writes

The write operation depends on the number of strips inside a stripe touched. As mentioned in Chapter 1, it is more beneficial to use reconstruct-writes when half or more strips are touched. This is decided in the beginning of the write in `hr_raid5_bd_write_blocks()` (*raid5.c*), where the strips touched are calculated. Depending on the result of the mentioned condition, subtract hint is set to either true or false. If there is no bad extent, and the subtract hint is set to false, the `hr_execute_write_stripe_optimal_reconstruct()` function is used, which dispatches workers as follows.

1. Based on the number of strips and partial strips touched, a full-stripe hint is set if the number of strips touched is equal to all $N-1$ (all data strips), and the number of partial strips touched is zero.

2. A union of all extent spans is calculated. This union is the total stripe I/O height.

3. For each extent span, a Normal Writer is dispatched.

4. If it is a full-stripe write, go to 6.

5. For each non-extension of the total stripe I/O height and each extent span, a Reconstruct Reader is dispatched.

6. Finally, a Parity Writer is spawned for the total height. Note that there might be two total ranges as shown in Figure 3.7, so two Parity Writers might be needed to be spawned.

An example of reconstruct-write in which all mentioned workers are used is shown in Figure 3.10.

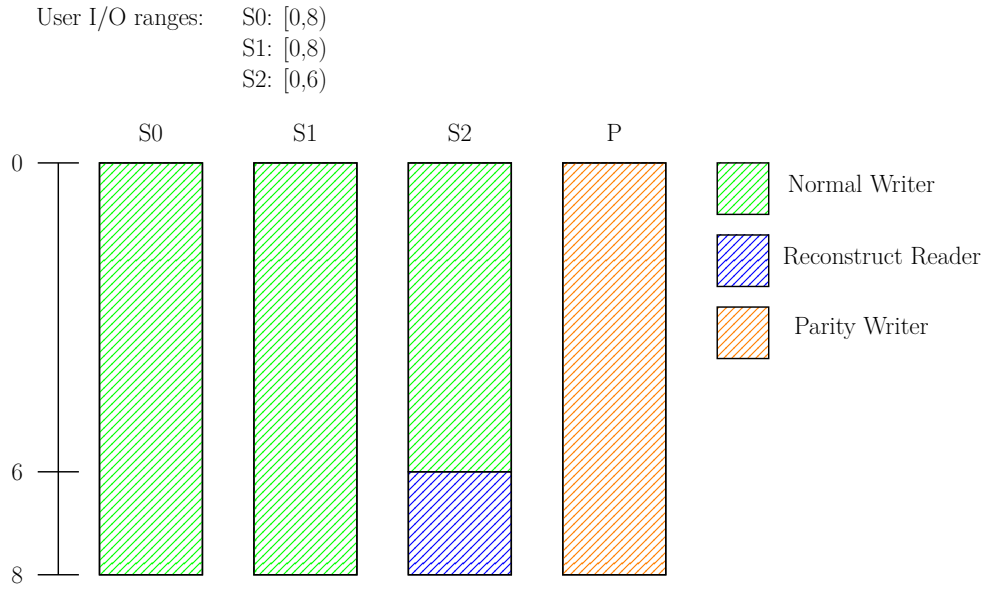

Figure 3.10 RAID 5 reconstruct-write

Optimal subtract-writes (small-writes)

Optimal subtract-write I/O is simple. Worker dispatching is done by the function `hr_execute_write_stripe_subtract()`.

1. A union of all extent spans is calculated. This union is the total stripe I/O height.
2. For each extent span, a Subtract Writer is dispatched to the target extent.
3. A Reconstruct Reader and Parity Writer is dispatched to the parity extent for each total stripe I/O height range.

An example of a subtract-write with two total stripe I/O height ranges is shown in Figure 3.11.

Degraded writes

Degraded writes are handled by `hr_execute_write_stripe_degraded()`, which decides further action. In the case of parity-degraded stripe, a Basic Writer is issued for each extent span. This is like a non-redundant write I/O in RAID 0.

In the case of an untouched unusable extent, a normal subtract-write operation is dispatched. In the case where the I/O requests also write data to the unusable extent, the following workers are dispatched via `hr_execute_write_stripe_degraded_mixed()`[12]. The steps are as follows.

[12]Mixed as in mixed I/O to both good and bad extent

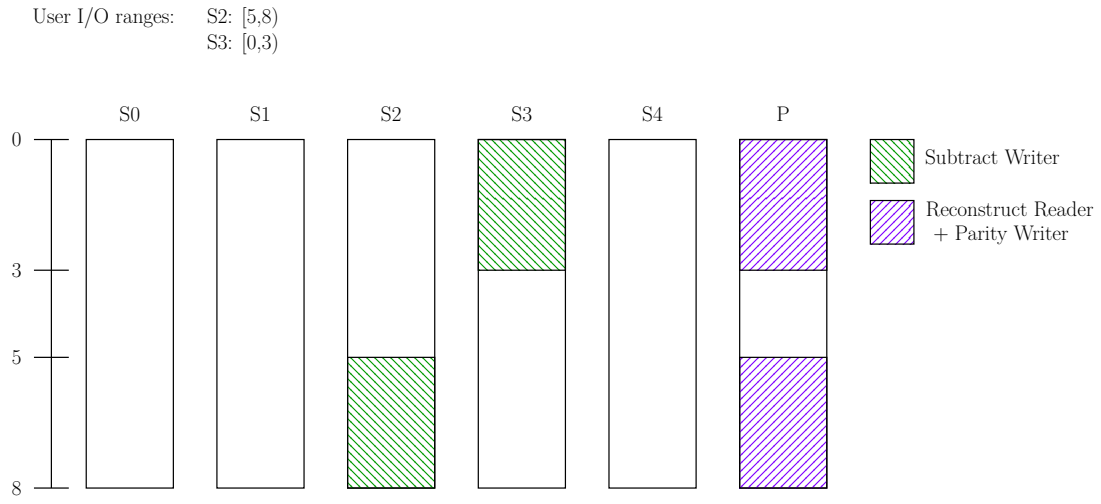

Figure 3.11 RAID 5 subtract-write

1. The total height range(s) is calculated as the union of all extent spans.
2. A No-op Writer is dispatched to the bad range, representing a write the bad extent.
3. For each good (non-parity) extent there is a non-extension of the bad range and the good extent's range calculated. If such a range exists, a Reconstruct Reader is dispatched to the good extent. Such a range exists, especially if the I/O does not touch the good extent at all.
4. An overlapping range is calculated for each of the good extent ranges and the bad range. If such an overlapping range exists (per-good extent), a Normal Writer is dispatched for that calculated range.
5. Another non-extension is calculated for each good extent, but now in the opposite order as in Step 3. This means that a subrange of the good extent's range that is not an extension of the bad range is calculated. If such a range exists, a Subtract Writer is dispatched for this range to the good extent.
6. If there exists a subrange of the total I/O height not in the bad range, a Recontruct Reader is dispatched to this subrange, to the parity extent.
7. Finally, a Parity Writer is dispatched for the total I/O height range(s).

An example of a degraded write scenario that includes all these steps is shown in Figure 3.12.

Failure fallbacks

Note on stripe execution restarts. There is a lot of theory behind what an I/O operation should do in case of extent failures, and different models for recovery such as forward or backward error recovery exist. RAID implementations with forward error recovery have tens-of-thousands lines of code just for failure scenarios, as for each scenario there is a specific code path to be taken. This approach might be better performance-wise, but due to the complexity of having to analyze and

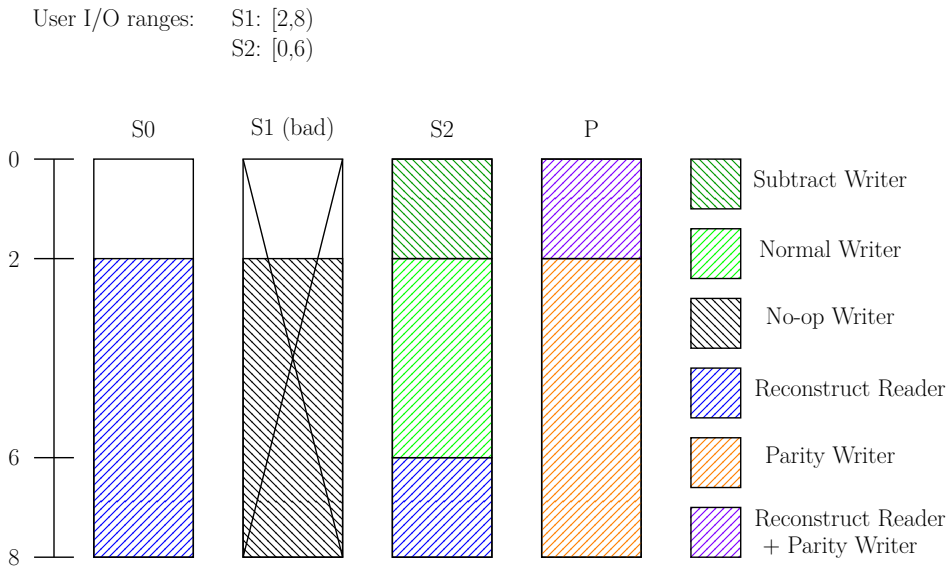

Figure 3.12 RAID 5 degraded write

implement fallback for each scenario, it is unrealistic to implement. Backward error recovery logs all actions that an operation took (including the data read) and potentially reverts everything back. However, this approach has poor performance. A technique proposed and implemented in RAIDframe [10] uses Directed-Acyclic Graphs (DAGs) to model and execute I/O operations. In a sense, HelenRAID uses something similar but greatly simplified, as RAIDframe even counts with not enough memory to allocate resources, but this does not include getting `ENOMEM` from the underlying extents. More detailed explanation and information on this can be found in the RAIDframe research material [10].

3.7.3 RAID 5 rebuild

Rebuild operation in RAID 5 is very similar to that of RAID 1. The rebuild function for RAID 5 is `hr_raid5_rebuild()`. Initialization and saving rebuild progress is the same. The only difference is how the data are reconstructed and that after the rebuild is initialized with `hr_init_rebuild()` a stripe (`hr_stripe_t`) is created for calculating the data as parity. The following operations are repeated until the whole block range is not reconstructed, or until an extent failure occurs.

1. Create a fibril group.
2. Lock the current rebuild range.
3. Update the rebuild position.
4. Dispatch Reconstruct Readers to all extents except the rebuild one.
5. Dispatch a Parity Writer to the rebuilt extent.
6. Wait for the fibril group. If some worker failed, release the range lock and abort.
7. Release the range lock.

8. Reset (not restart) the stripe. This zeros the parity and allows new data to be reconstructed.

9. Shift the rebuild range and go to Step 1.

In the end, the state is evaluated.

3.8 Metadata

In this section, we describe the format-agnostic metadata handling interface, the native format, and notable implementation details about foreign metadata support.

3.8.1 Format agnostic metadata handling interface

As discussed in Chapter 2, Subsection 2.4.1, the interface should have functions for probing, comparing UUIDs of volumes, volume initialization from metadata, metadata population, metadata saving and re-assembly counter handling.

The interface designed is represented by the type `hr_superblock_ops_t`, defined in *superblock.h*.

```
typedef struct hr_superblock_ops {
        errno_t (*probe)(service_id_t dev, void **rmeta_struct);
        errno_t (*init_vol2meta)(hr_volume_t *vol);
        errno_t (*init_meta2vol)(const list_t *devs, hr_volume_t *vol);
        errno_t (*erase_block)(service_id_t dev);
        bool (*compare_uuids)(const void *meta1, const void *meta2);
        void (*inc_counter)(hr_volume_t *vol);
        errno_t (*save)(hr_volume_t *vol, bool use_ext_state_callback);
        errno_t (*save_ext)(hr_volume_t *vol, size_t ext, bool ext_cb);
        const char *(*get_devname)(const void *meta);
        hr_level_t (*get_level)(const void *meta);
        uint64_t (*get_data_offset)(void);
        size_t (*get_size)(void);
        uint8_t (*get_flags)(void);
        void (*dump)(const void *meta);
        hr_metadata_type_t (*get_type)(void);
} hr_superblock_ops_t;
```

Each metadata format implements the above interface. Metadata types are identified by `hr_metadata_type_t` enum defined in *uspace/lib/device/include/hr.h*.

- **Probe** function takes a service ID of a block device and tries to find its own valid superblock on the device. Probing involves reading blocks were the metadata format's superblock is supposed to reside, and then it tries to decode the superblock. The decoding generally begins with validating the magic. If the format's magic is present, the decode function decodes the rest of the superblock, and the structure representing the superblock is pointed to by `*rmeta_struct` argument. This gives the caller of the probe function the superblock structure.

- **Init metadata from volume** (`init_vol2meta()`) is a function that takes a newly created volume and populates new metadata created from the volume parameters. The newly created metadata are stored in the volume's pointer dedicated to in-memory stored metadata.
- **Init volume from metadata** (`init_meta2vol()`) is a function used to populate the volume structure from a list of devices. The list members are of type `struct dev_list_member` defined in *util.h*, which besides the service ID of the device includes probed metadata structure and few hints useful for initializing and finalizing libblock on the device. The list is populated by the assembly mechanism that is described later.
- **Block erasure** is a function used to erase the superblock when an extent is marked as Failed by the user.
- **Compare UUIDs** function compares UUIDs of two metadata structures.
- **Counter increment** function increments the re-assembly counter.
- **Save** is a function that saves the metadata to all extents.
- **Save to extent** is a function that saves metadata to a single extent. This is useful for saving the rebuild position only on the rebuilt extent without stressing the other extents with useless I/O.
- **Get device name** function returns a pointer to the volume's device name.
- **Get level** function returns level that is specified in the metadata structure.
- **Get data offset** returns the offset of the first block where user data starts.
- **Get size** returns the number of blocks that the metadata superblock resides on.
- **Get flags** returns flags enabled by each type implementation for internal use. Currently `HR_METADATA_HOTSPARE_SUPPORT` and `HR_METADATA_ALLOW_REBUILD` are supported. The former allows hotspare extents to be added and the latter allows rebuild process to be initialized.
- **Dump** function prints metadata fields. Useful for debugging.
- **Get type** function returns the type of metadata format.

The following types were implemented.

```
typedef enum {
        HR_METADATA_NATIVE = 0,
        HR_METADATA_GEOM_MIRROR,
        HR_METADATA_GEOM_STRIPE,
        HR_METADATA_SOFTRAID,
        HR_METADATA_MD,
        HR_METADATA_NOOP,
        HR_METADATA_LAST_PLACEHOLDER
} hr_metadata_type_t;
```

The first five represent the native format used by HelenRAID, GEOM Mirror, GEOM Stripe, OpenBSD's softraid, and Linux's MD RAID, respectively. The No-op metadata type does not store any metadata. This can be useful in a situation where one has, for example, two identical devices that can be put inside a mirror and RAID 1 read speeds can be utilized this way. The last type is a helper constant.

Note that there is no function for handling hotspare extents. Due to time constraints and additional complexity, saving/remembering hotspares was not implemented.

3.8.2 Native format

HelenRAID's native metadata format superblock is represented by `hr_metadata_t` defined in *metadata/native.h*. The native metadata fields are shown in Table 3.7.

Field	Size in bytes	Description
magic	16	The magic. It is a string "HelenRAID".
uuid	16	The volume UUID.
version	4	The metadata version, currently 1.
extent_no	4	The number of extents.
level	4	The level. Possible values are 0, 1, 4, 5.
layout	4	The volume layout. Used by RAID 4, 5.
index	4	Extent index.
strip_size	4	Strip size in bytes.
bsize	4	Volume's block size.
data_blkno	8	Usable number of blocks.
truncated_blkno	8	Size of the smallest extent.
counter	8	Re-assembly counter.
rebuild_pos	8	Current rebuild position.
devname	32	Volume's device name.

Table 3.7 HelenRAID's native metadata fields

Notable comments. Each field except the magic, uuid and the device name is stored in little endian. The UUID is randomly generated 16 bytes. The rebuild position is zero for each non-rebuilt extent, and is non-zero for a rebuilt extent.

Implementation

The advantages and disadvantages of superblock locations were discussed in Chapter 2, Subsection 2.3.4. We decided to store the superblock at the last block of an extent. Thanks to the metadata handling interface, it is easy to move it to different locations with little modifications.

Native metadata handling functions are defined in *metadata/native.c* and are straightforward, therefore, we will only briefly show notable parts of the volume initialization from metadata. In the beginning, the most up-to-date extent from the device list is selected. This selection is based on the maximum value of the re-assembly counter. Invariant volume parameters are filled in from this metadata,

and in-memory storage for the metadata is allocated, and the metadata copied there. After that the device list is iterated, the service IDs filled in according to extent indices, and each extent's state set to Online if the its counter is equal to that of the main metadata and the rebuild position is zero. If the counter is equal to that of the main metadata but the rebuild position is non-zero, the extent's state is set to Rebuild. Else, the extent's set is set to Invalid. After this, all extent service IDs of the `hr_extent_t` array are checked, and if there is a service ID equal to zero, the extent's state is set to Missing.

Note that it is possible for the list to have more than N extents that the volume consists of, because the user could potentially rebuild missing extents and then try assembling with all of them ($> N$). This situation is not taken into account because such situations are very unlikely to happen during traditional usage.

3.8.3 Foreign formats

Foreign (external) software RAID metadata formats were analyzed in Chapter 2, Section 2.4. Since the implementation of the metadata handling interface for each format is very similar, we will omit the details and only comment on notable aspects. We will also show what code was written by us and what was copied and modified from other implementations. The initialization of metadata from volume is not supported by any of the foreign metadata interface implementations. This is because as mentioned, with some it is not possible as there are fields for internal use by their native operating system and HelenRAID native metadata is preferred when creating volumes in HelenOS.

softraid

The softraid support is located in the *metadata/foreign/softraid/* directory. The directory contains the following files. *softraidvar.h* – contains metadata types and constants, *softraid.c* – contains helper functions such as `sr_meta_print()` function for dumping the metadata. These files were stripped of irrelevant functions, macros and types and imported from OpenBSD's kernel source files with the same name. [13][14]

Additionally, there were disk states used by `struct bioc_disk` copied from file *biovar.h* into *softraidvar.h*.[15]

Metadata decoding and encoding functions are not directly provided by softraid, so we had to write them. They are located in *metadata/foreign/softraid/hr_softraid.c* with the rest of the interface implementation. The code in this file was written by the author. Besides retyping `u_intXX_t` types to `uintXX_t` no substantial modifications to the copied code were needed. The initialization of volume from metadata is almost identical to that of the native implementation, and other function implementations are straightforward, so we will omit comments.

During the implementation of softraid support, two bugs were discovered in the current OpenBSD softraid implementation.

[13] *OpenBSD/sys/dev/softraidvar.h* at bxr.su
[14] *OpenBSD/sys/dev/softraid.c* at bxr.su
[15] *OpenBSD/sys/dev/biovar.h* at bxr.su

- The first bug was found while implementing the decode function. The checksum in `struct sr_meta_chunk`, that should checksum whole `struct sr_meta_chunk_invariant`, is currently done only on the first 16 bytes and additionally before the coerced (truncated) size is calculated. But since the checksum is never checked, this has effectively zero impact. Nonetheless, a fix was proposed to the *openbsd-tech* mailing list, but it has not caught the attention of any OpenBSD developers yet. [38]

- The second bug is dangerous and easy to trigger. It is in the chunk sorting algorithm that sorts chunks to their correct positions regardless of user-provided chunk order when assembling volumes. To briefly illustrate the dangerous behavior, assume a 3-extent (e_0, e_1, e_2) RAID 5 volume in optimal state. When the system is rebooted without the last extent (chunk), only with e_0, e_1, the assembled volume has the following configuration. Using the original extent indices – e_0, e_2, e_1 (but e_2 is Missing). The second (e_1) and the last (e_2) extent have swapped positions, effectively swapping data and parity strips, or data and data strips depending on the stripe. These mangled chunk positions get written back to disk right after assembly, effectively destroying the volume. A proposed fix and more detailed information can be found in [39]. However, at the time of writing, this has not been fixed yet.

GEOM Stripe and GEOM Mirror

FreeBSD's GEOM Stripe and Mirror metadata support is located in the *metadata/foreign/geom/* directory. The implementation is split to *g_stripe.h* and *hr_g_stripe.c* for GEOM Stripe support and *g_mirror.h* and *hr_g_mirror.c* for GEOM Mirror support, respectively. Files *g_{stripe,mirror}.h* contain constants, metadata superblock types and the encode and decode functions for both gstripe and gmirror. These files were stripped from the FreeBSD's source files with the same names.[16][17]

The files *hr_g_{stripe,mirror}.c* include the interface implementations. These files were written by the author.

For minimal modifications, FreeBSD's `<sys/endian.h>` header had to be imported. It is represented by the *sys_endian.h* file.[18]

Notable modification includes changing the calculation of MD5 checksum to HelenOS's MD5 functions, however, since the implementation is straightforward, and almost identical to the native implementation, we will omit comments.

MD RAID

Linux's MD RAID metadata support is located in the *metadata/foreign/md/* directory. The implementation consists of two files. *md_p.h* – contains constants and the metadata superblock type, and additionally a function to calculate the metadata checksum. The relevant types and constants were copied from Linux

[16] *FreeBSD/sys/geom/stripe/g_stripe.h* at bxr.su
[17] *FreeBSD/sys/geom/mirror/g_mirror.h* at bxr.su
[18] *FreeBSD/sys/sys/endian.h* at bxr.su

source file with the same name.[19] The checksum calculation function was copied from the mdadm utility source file *super1.c*.[20] The other file that the implementation consists of is *hr_md.c*. This file contains the interface implementation and was written by the author.

In addition to basic retyping of unsigned integer types, no major modifications were required, but the encode and decode functions had to be written. The interface implementation itself is very similar to that of native format, so we will omit comments.

Note that the header file for MD RAID is the only one licensed under GPL. HelenOS is licensed under the BSD license, with some components licensed under GPL, such as GRUB. By the time of writing, it has not yet been discussed and decided whether it will be part of the Pull Request to the upstream branch.

3.9 Assembly

Assembly and has the following stages.

- Assembly is initiated in `hr_util_try_assemble()`, where first a list of `struct dev_list_member` elements is filled. The members hold device service IDs, a boolean hint that signals if libblock was already initialized, a metadata pointer and a boolean hint that signals if metadata is present. This initial list is filled either from the configuration structure (`hr_config_t`) provided by client `hr_assemble()` function, or automatically, if it is automatic assembly. The automatic filling is done as follows.

  Session with the vbd server is initialized, and each device's service ID in the category disk known to vbd is retrieved. If there is a partition table present in the device, each partition's service ID is added to the list. If there is no partition table present, the device's service ID is added to the list.

- After the initial list is filled, a single list member is popped from the list. Metadata is tried to be found by `hr_find_metadata()` defined in *superblock.c*. This function iterates through all the metadata types and calls their probe implementations. If no metadata is found, another device is popped from the list again. If metadata was found, a new so called *matching list* is created, and all devices from the initial list are probed, and if they contain valid metadata of the same type and the volume UUID matches, the devices are added to the matching list. Finally, the original device popped from the initial list is added to the matching list, and the matching devices are removed from the initial list.

- The matching list is handed to `hr_util_assemble_from_matching_list()` that creates the volume structure, fills it by calling `init_meta2vol()` implementation of the metadata format, the `hr_ops_t` create function is called, and the volume finally registered.

[19] *linux/include/uapi/linux/raid/md_p.h* at github.com
[20] *mdadm/super1.c* at github.com

Automatic assembly is tried in the server's main function, which means that connected supported volumes are going to be automatically assembled during the server's launch by the system init service.

Volume registering category

As already mentioned, volume services are registered under new *raid* category. The reason why they are not registered under the *disk* category is as follows.

The vbd server has a callback registered that is triggered every time something in the disk category is added or removed. It saves an internal handle for each new block service in the disk category to its internal structures and initializes libblock on it. This calls `bd_open()`. The vbd needs this to work with partition tables and also to serve I/O to the partition services. When a service is unregistered from the disk category, vbd sees this, finalizes the libblock for the service, and removes it from its internal structures. Partitions within partitions are not supported, so by design it does not count with "stacked" block services such as RAID volumes. The first problem is that since we want to be conservative with what volumes are safe to be disassembled by counting the number of opens and closes, because vbd has the service libblock initialized for its whole lifetime, this is simply not possible. Even if we enabled "forced" disassembly that would ignore the number of opens, the following problem would arise. When a volume is disassembled, the libblock is finalized on the extents, the whole volume structure is freed, and the volume's service ID unregistered. After this, vbd would finalized libblock on the volume's service ID by `block_fini()`, which internally calls `bd_sync_cache()`. This would crash the HelenRAID server, because the volume block device operations would call `bd_sync_cache()` on the extent service IDs, which have already been finalized when disassembling. The crash would happen because libblock asserts that every block operation is performed on an initialized block service. Not to mention that the volume pointer still held by the active block server session would be invalid as the memory to which it points has been already freed. This behavior was discovered while testing volume registration to the disk category.

To allow volumes to be registered under the disk category would require a partial vbd rewrite that would initialize libblock only when there is a need to work with partition tables, and then finalizing it after the partition table operations are done. Additionally, it should initialize libblock when someone opens a partition and finalize it when the last partition user closes the partition service. Also note that due to this issue, it is not possible to create partitions on volumes. In case a user wants to have partitioned volumes, has to first partition the extents and then create multiple volumes on the created partitions.

4 Evaluation

In this chapter, we evaluate the HelenRAID implementation. In the first section, the functional aspects of the implementation are tested and evaluated. In the second section, accessing foreign volumes is tested, and the rest of the chapter is dedicated to performance testing and evaluation.

4.1 Unit tests

As mentioned in Chapter 1, Subsection 1.2.4, PCUT was used for the functional tests of HelenRAID.

We implemented 11 scenarios that are tested with different volume configurations, resulting in 38 total unit tests.

- **Volume block number**. The usable number of blocks exposed by the volume service is tested for all levels (RAID 0, 1, 4, 5), once with with the native metadata and once with no-op metadata.
- **Read-only flag**. A volume is created as read-only, testing if writes return `ENOTSUP`, signaling operation not supported.
- **Volume state evaluation**. In the beginning the volume state is asserted to be Optimal, then the maximum number of failures tolerated are simulated by failing that many extents, asserting the state to be Degraded. Finally one more extent is failed, asserting the Faulty state. This is tested for RAID 0, 1, 5.
- **Optimal-Optimal data write checking**. A random data block is written at random offsets in Optimal volume state. Then read requests are sent to those written offsets, and the read data block is checked for equality with the data block written. This is done for RAID 0, 1, 4, 5.
- **Optimal-Degraded data write checking**. Similarly to the previous test, a random data block is written at random offsets in Optimal state, and then an extent failure is simulated, making the volume change its state to Degraded. After this, reads are sent to the written offsets, checking the read block for equality with the written block. This can be done only for fault-tolerant levels, therefore it is done only for RAID 1, 4, 5.
- **Degraded-Degraded data write checking**. Similar to previous 2 test scenarios, with the difference being that the writes done in Degraded state, and reading is done right after the writes, with the volume still in the Degraded state. This is done for RAID 1, 4, 5.
- **Rebuild operation**. A volume is created, the Optimal state is asserted and then extent failures are simulated, making the volume change its state to Degraded. After this a rebuild operation is started by adding extents as hotspares, starting rebuild operations until the volume is back in the Optimal state. This is done for RAID 1, 4, 5.

- **Degraded-Rebuild-Degraded write checking**. A volume is created and extent failure is simulated, making the volume change its state to Degraded. Then a random data block is written at random offsets. A rebuild operation is started on a new extent. After the rebuild operation finishes, another extent failure is simulated on a different extent than the rebuilt one. This makes the volume again change its state to Degraded. Then read requests are made to the written offsets, checking the read block for equality with the written one. This is done for RAID 1, 4, 5.
- **Basic assembly tests**. A volume is created, disassembled, assembled again and the volume information is retrieved. Relevant parts of the volume information are check for equality with volume information before its disassembly. This is done for RAID 0, 1, 4, 5.
- **Non-destructive partial assembly**. A volume is created, disassembled, and then partially assembled. Random read requests are made and then the volume is disassembled again. Then the volume is fully assembled, and the Optimal state is asserted. This tests that the volumes can be partially assembled and read from, without invalidating the Missing extents. This is done for RAID 1, 4, 5.
- **Destructive partial assembly**. A volume is created, disassembled, and then partially assembled. Some write requests are made, and then the volume is disassembled. After this the volume is fully assembled, and the Degraded state is asserted. This tests that the Missing extents are invalidated. This is done for RAID 1, 4, 5.

The tests are in the *uspace/srv/bd/hr/test/* directory. For information on how to run these unit tests, refer to Section D.2 of Appendix D.

4.2 Foreign metadata

All foreign formats were tested during development, however, the tests were focused only on a specific part, such as if the mapping is correct or if some metadata fields are of expected values. However, we did the following test to simulate how users might utilize the volumes. We created volumes on other operating systems with the supported metadata, and created the ext2 file system on them. After this we copied an image file to the volume in the other operating system and then assembled the volume in HelenOS. HelenOS has a TGA image format viewer application that we used to verify that the image was correctly read. We created a 4-extent RAID 0 GEOM Stripe and softraid, 4-way RAID 1 GEOM Mirror, and 4+1 RAID 5 softraid and MD volumes. The image was 4.5 MiB and therefore spanned multiple stripes in the case of striped volumes. Even on MD's 4+1 RAID 5 volume that had 512 KiB strip size. However, something such as file checksum calculation would be more practical and easier to verify mapping correctness, but that would require exposing MD5 or SHA1 algorithms as part of a new application.

Specifics including volume creation commands ran on other operating systems, and instructions on how these can be tested with provided foreign volume disk images can be found in Section D.4 of Appendix D.

The rest of this chapter is dedicated to performance testing.

4.3 Methodology

In this section, we describe the hardware used, the versions of software used, a workload generator, the types of workload tested, and the tool for measuring different performance metrics.

4.3.1 Hardware

The machine used for performance evaluation is a Dell Precision Rack 7190 equipped with 2x Intel Xeon E5-2699 v3 and 8x 32GB LRDIMM memory modules. There were two types of storage devices connected to the machine.

- 48x 600GB 2.5" 10K RPM Seagate Savvio SAS-2 hard disk drives connected through two external Fujitsu Eternus disk shelves, each containing 24 drives. The shelves were connected to the system using two Dell H200e SAS Host Bus Adapters (HBA) via QSFP to SFF-8088 cables.

  Although the disks are over 10 years old, they are fully functional, have somewhat predictable performance[1] and are reliable enough to serve as storage devices for these performance evaluation benchmarks.

- 6x 120G M.2 SATA solid state drives connected to the system's integrated AHCI SATA-III controllers.

  These drives have a number of different manufacturers and have variable performance. However, we will also show benchmarks for these solid-state drives, as it can be interesting to see flash memory vs. spinning disks.

A diagram showing the connection of the storage devices to the test host can be seen in Figure 4.1.

4.3.2 Test system software

Because we were unable to boot HelenOS on bare metal, and additionally because of its lack of support for SAS HBAs, we had to test the RAID implementation in a virtual environment. HelenOS is mainly tested and developed on QEMU, so it was natural to use QEMU for these tests as well.

Alpine Linux was installed on the host system. The relevant software versions are shown in Table 4.1.

[1]The performance of writes is very predictable, but the performance of reads is non-trivial to analyze, as the disks utilize internal cache and prefetch blocks. Around 60% of the disks have 16 MB cache, while the remaining disks have a 64 MB cache because of slightly different disk models – `ST9600204SS` and `ST9600205SS`.

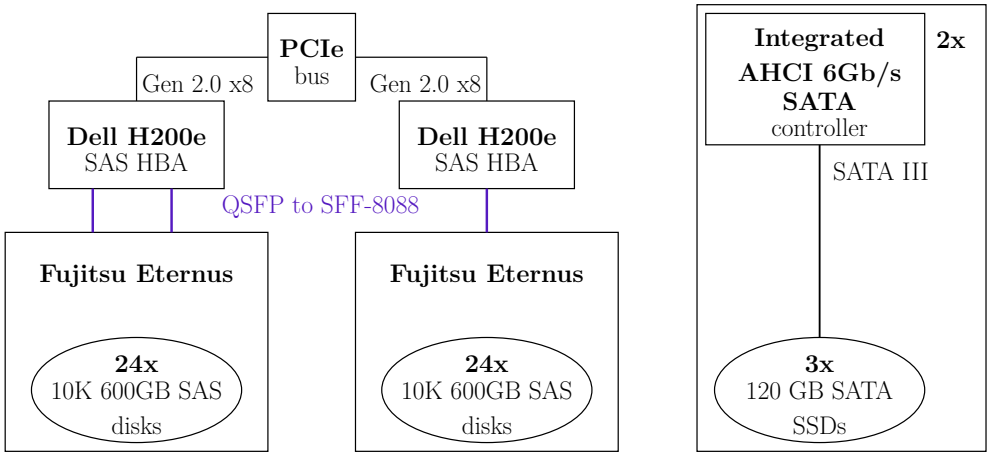

Figure 4.1 Connection of storage devices to the test system

Software	Version
Linux	6.12.32-0-lts
Distribution	Alpine Linux v3.22
mpt3sas driver	48.100.00.00
QEMU	10.0.0

Table 4.1 Host system software versions

QEMU options

The following QEMU options were used to launch HelenOS.

```
qemu-system-x86_64 -enable-kvm -m 131072 -serial stdio \
    -boot d -cdrom ./helenos-image.iso \
    -display none -vnc 0.0.0.0:2000,password=off \
    -virtio-blk-drives (snipped)
```

Note that HelenOS was dedicated just a single core. This is because we noticed a drastic performance decrease when HelenOS was launched with more cores (with `-smp cores`). The same behavior was also observed on a different host with a Zen 2 AMD CPU. Thanks to the Kernel-based Virtual Machine (KVM) using the CPU's virtualization extensions, the HelenOS guest's code execution does not have significant performance degradation.

The storage devices were connected using the VirtIO block interface. This was done for performance reasons and to allow more than four drives to be specified[2]. The attached virtio-blk devices had additionally `cache=none` specified, which should prevent the host from using page cache for I/O coming from the guest to the target device. For the command used to launch HelenOS, please refer to Appendix A.

[2]QEMU connects drives to IDE by default, resulting in maximum of four drives attached.

HelenOS configuration

HelenOS was built with the default configuration for amd64, with the Window system disabled. One batch of tests were additionally run with the Kernel memory model option set to Large, because the frame allocator had an issue when allocating around 2 GiB for virtio-blk DMA mappings. The number of maximum extents was increased to allow up to 33 extents. The size of the fibril pool was increased to 256 for both the number of fibrils and the size of the queue.

virtio-blk driver in the upstream branch issues I/O requests to devices in a very inefficient way. I/O spanning multiple blocks is processed synchronously block by block. This means that a two-block request has twice the latency of a single-block request minus the overhead. This scales linearly with the number of blocks. This is because each virtio-blk's request buffer (there are 32 of them) part for data is allocated only 512 B of memory for DMA. In total, that is 16 KiB per virtio-blk device. However, to benchmark the maximum throughput that HelenOS and HelenRAID is capable of, the virtio-blk driver had to be modified to allow I/O requests spanning multiple blocks to be processed in a single virtio-blk read/write operation. And because the maximum IPC transfer size is 64 KiB, we also increased it to 2 MiB, so that the I/O performance will be possibly competitive even with other operating systems. This resulted in each virtio-blk device consuming 64 MiB (32 ·2 MiB) for the request buffers' memory for DMA. And because we tested with at most 33 devices simultaneously connected, the total memory consumption was around 2 GiB just for the virtio-blk devices. This was just to demonstrate the system's capabilities, because the memory consumption might not be reasonable for some use cases. Linux, for example, allocates the DMA buffers on demand.

We performed all the benchmarks with both the upstream settings and also with the increased transfer size and the modified virtio-blk driver. We will refer to these two system settings as the *upstream configuration* and *2 MiB configuration* for the upstream and modified transfer size and virtio-blk driver, respectively. For modification details, please refer to Section A.1 of Appendix A.

FreeBSD, OpenBSD and Linux

We also benchmarked FreeBSD's gstripe and gmirror, OpenBSD's softraid and Linux's MD for comparison with HelenOS's HelenRAID levels 0, 1 and 5. Striped volumes (RAID 0, 5) were created with the same strip size that HelenRAID's volumes were created with. softraid's volume strip size is hardcoded inside the source code to 64 KiB, therefore we did not modify it, as recompiling and reinstalling the kernel would take additional time. OpenBSD is also at a disadvantage as its maximum transfer size is 64 KiB. Additionally, OpenBSD's kernel has very limited concurrency in its I/O path because of still disabling interrupts to synchronize accesses to some internal data structures. The operating systems were launched with the same QEMU options as HelenOS – with KVM enabled, single core, and 128 GiB of memory. The systems were not modified or tuned in any way. The system versions can be found in Table 4.2.

For shell scripts used to start the tested operating systems, create, and benchmark volumes, refer to Appendix A.

System	Version
FreeBSD	14.3-RELEASE
OpenBSD	7.7
Linux	6.12.36-0-lts (Alpine Linux)

Table 4.2 Versions of compared operating systems

4.3.3 Workload types

We performed I/O benchmarks for random and sequential access patterns. More specifically, we focused on throughput in sequential I/O tests, and on I/O operations per second (IOPS), and latency in random access pattern tests. All tests were performed for reads and writes. The tests were performed using a single reader or writer fibril. Although we also experimented with concurrent I/O requests, the I/O performance was the same as in the single reader/writer scenario. We assume that this is because of limited parallelism of per-task fibrils, as they run by the means of the task's single kernel thread.

We omitted benchmarking file system I/O, because the file systems use block cache, which would require additional analysis and cache modeling. However, after the analysis of the devices' latency profiles, assumptions about file system performance are presented.

Workload generation

We have written a simple I/O generation tool that allows multiple parameters to be specified. We designed a simple binary format that is used for telling I/O workers the block address offsets and the block counts. The **iotest** only saves 64 bit unsigned integers in little endian. The files use the first value for specifying the I/O concurrency (denoted c), the second value represents the count of I/O requests each I/O worker should make (denoted C), and the third value represents the maximum transfer size for which this file was generated.[3] After these three 64 bit values, exactly $c \cdot C$ pairs of (block address, count) follow. The first C pairs are for the first worker, etc.

The following is the tool's usage message.

```
iotest_gen: I/O test file generator.
Usage: iotest_gen -f file -c concurrency -C cnt_per_thread -O max_off
                  -m min_size -M max_size -T max_xfer -A alignment

All options are mandatory:
  -f    Output file
  -c    Level of concurrency
  -C    Count of I/O operations per thread
  -O    Capacity range (offset to (last + 1)-th block in bytes)
  -m    Minimum I/O size (in bytes, must be multiple of 512)
  -M    Maximum I/O size (in bytes, must be multiple of 512)
```

[3]The third value is only a hint for HelenOS, so that we know we are doing the right tests.

```
  -T    Maximum XFER size (in bytes, must be multiple of 512)
  -A    I/O alignment (in bytes, must be multiple of 512, or 0)
```

We generated two types of workloads. Random access pattern workloads containing single-block I/O requests, generated by specifying `-m` and `-M` as 512. Sequential workloads representing a single very large I/O, generated by setting `-m` and `-M` to the maximum transfer size and the I/O count to sufficiently large value[4]. Also note that the sequential workloads were aligned to maximum transfer size to not put RAID 4 and RAID 5 writes in a disadvantage during the sequential these tests, and requests in the random workload where aligned to 4 KiB. For hard drives, the maximum offset was set to 500 GiB, and for solid-state drives to 100 GiB. This is because even though some volume configurations will have over 18 TiB capacity, we wanted to test the same workloads also for single drives for comparison. For each type of workload, multiple sets were generated. We generated all the sets for both the upstream and 2 MiB configurations.

We wrote the tool only for Linux, however we were able to compile it on FreeBSD as well. For the tool's source code and generated workload files used for benchmarking, please refer to Appendix A.

4.3.4 I/O benchmarking tool

We also implemented a tool that executes the generated I/O workloads. It has support for concurrent requests, however as was mentioned, only a single worker was utilized for the tests. The tool first reads the whole workload to memory and then starts executing and measuring I/O requests. It measures throughput, IOPS and average request latency. It has the following interface.

```
iotester: I/O performance testing utility.
Usage: iotester -t type -f file -d dev -T seconds

All options are mandatory:
  -t    I/O type: r, w
  -f    Input file
  -d    Target device
  -T    Seconds after which to interrupt workers
```

We have implemented the tool for HelenOS and also for Unix-like systems, because we also tested and compared software RAID implementations of other operating systems vs. HelenOS and HelenRAID. The tool was tested on FreeBSD, OpenBSD and Linux. The difference is that in HelenOS it saves the results inside a file named after the testfile's name, I/O direction (r/w) and the device name. This is because there was not an easy way to export the results other than to a mounted file system. The tool's source code for HelenOS can be found in *uspace/app/iotester/*, and for the iotester's "Unix implementation", please see Appendix A.

[4]The exact I/O sizes are not important because the I/O tests were given a timeout, and each I/O workload in tests was big enough to be timed out

4.3.5 Testing methodology

All tests were done for reads and writes. For hard drives the volume configurations used 2, 4, 8, 16, 32 extents, because the highest power of 2 of drives available is 32. The power of 2 extent configurations were done because of non-rounded strip size. For solid-state drives, only 2, 4 extent configurations were used. Parity volume configurations used $N + 1$ extents.

The tests featuring 2,4,8 and 16 hard disks were connected to the virtual machines interleaved from the first and second disk shelf. The tests featuring 32 hard disks were performed with the first 17 disks connected from the first shelf and the remaining from the second disk shelf. This was done because during one test on a specific configuration, the random requests were only targeting one shelf due to the 4 KiB alignment of random accesses.

Upstream configuration

To assess single drive performance, we ran 5 sets of random and 5 sets of sequential workloads on three different drives, each test having the timeout set to 10 seconds. To assess volume performance, 3 sets of random workload and 3 sets of sequential workload tests were performed for both reads and writes. The values shown in the plots are an arithmetic mean across all the sets for each workload and I/O direction.

2 MiB configuration

For the 2 MiB configuration, we ran 3 sets of sequential workloads and focused only on throughput, as the latency and IOPS of single-block I/O are the same as in the upstream configuration. The single drive performance was measured on 3 different drives. The values shown in the plots are an arithmetic mean across all the sets for each workload and I/O direction.

Disk latency profiles

The disk latency profiles shown in the following sections were measured by running generated workloads to block address 0 with an increasing number of blocks. For each size, the request was repeated 100 times in succession. This was done only for writes, as the drive's internal cache would significantly influence read performance. This was done for three different hard drives. The values shown in the plots are an arithmetic mean across the three different drives.

Other operating systems

Tests done on other operating systems were performed with only a single set for each type of workload. This might have resulted in a higher variance over the measured data. Also note that among the operating systems tested, only Linux supports `O_DIRECT`. Because FreeBSD and OpenBSD do not support such `open()` extension, they can use page cache.

Including the other operating system benchmarks, roughly 950 tests were performed in total. The test results from which the following plots were compiled

can be found in the thesis attachments. For further information, please refer to Appendix A.

4.4 Upstream configuration

In this section, we compare I/O performance between the RAID levels with unmodified maximum IPC transfer size (64 KiB) and unmodified virtio-blk driver. The test results shown here include only the hard disks. Because the solid-state drives had very variable performance and also the highest number of extents used was 4, we omit the analysis of the SSD performance. The plots compiled from the SSD volume benchmarks can be found in Appendix E.

Disk latency profile

The write latency profile of a disk with the upstream virtio-blk is shown in Figure 4.2. The values show are an average of three different disks. The variance of the measured values on these disks was zero or very close to zero for each measurement. As mentioned earlier, the upstream virtio-blk driver sends multi-block requests to device block by block. These results in the shown latency profile. The latency of each request linearly grows with its size.

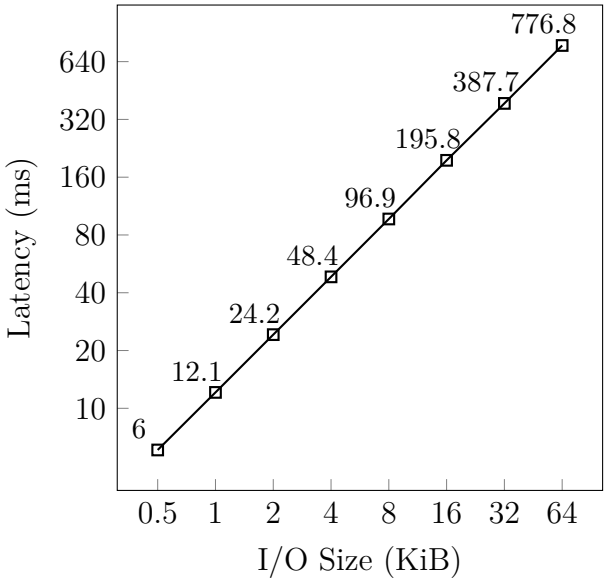

Figure 4.2 Disk write I/O latency vs. I/O size, upstream configuration

We did not measure latency profiles of reads, because the disk cache would interfere with the measurements[5], and it would not give the reasonable information needed to analyze accesses to different block addresses, as this test always used block address 0.

Read I/O

The comparison of read latency and IOPS during random access I/O, and read throughput during sequential I/O across different levels is shown in Figure 4.3.

[5]Writes can be also affected by the cache, however they have very predictable behavior.

The *single* plot values are the arithmetic mean of measurements performed on three different disks.

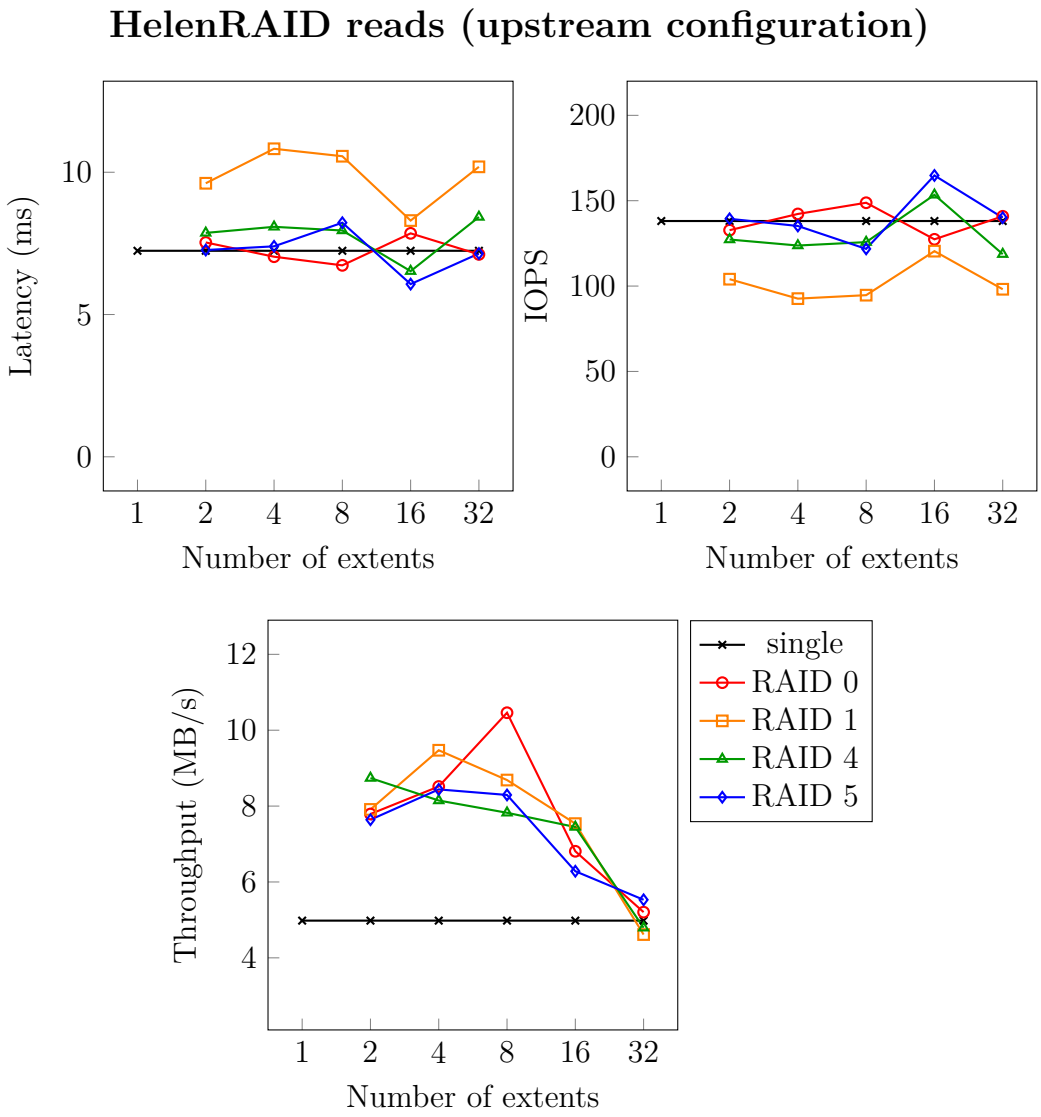

Figure 4.3 HelenRAID level read performance comparison, upstream configuration

The latency and IOPS are roughly the same as for the single disk as expected. The higher read latency of RAID 1 could be because of additional good extent calculation overhead. However, to analyze this, the CPU time spent in the functions would need to be investigated.

The throughput for reads on the upstream configuration is difficult to analyze, as the disks probably prefetch blocks to cache from addresses following the original request. Because of this, there are no significant differences between different levels. The declining throughput as the number of extents approaches 32 could be attributed to multiple elements, such as limited concurrency, or the methodology.

Write I/O

The comparison of write I/O across different levels is shown in Figure 4.4.

The latency, therefore, IOPS, as IOPS are "inverted" latency, have expected values, namely that the RAID 0 has the same latency as the single extent, and

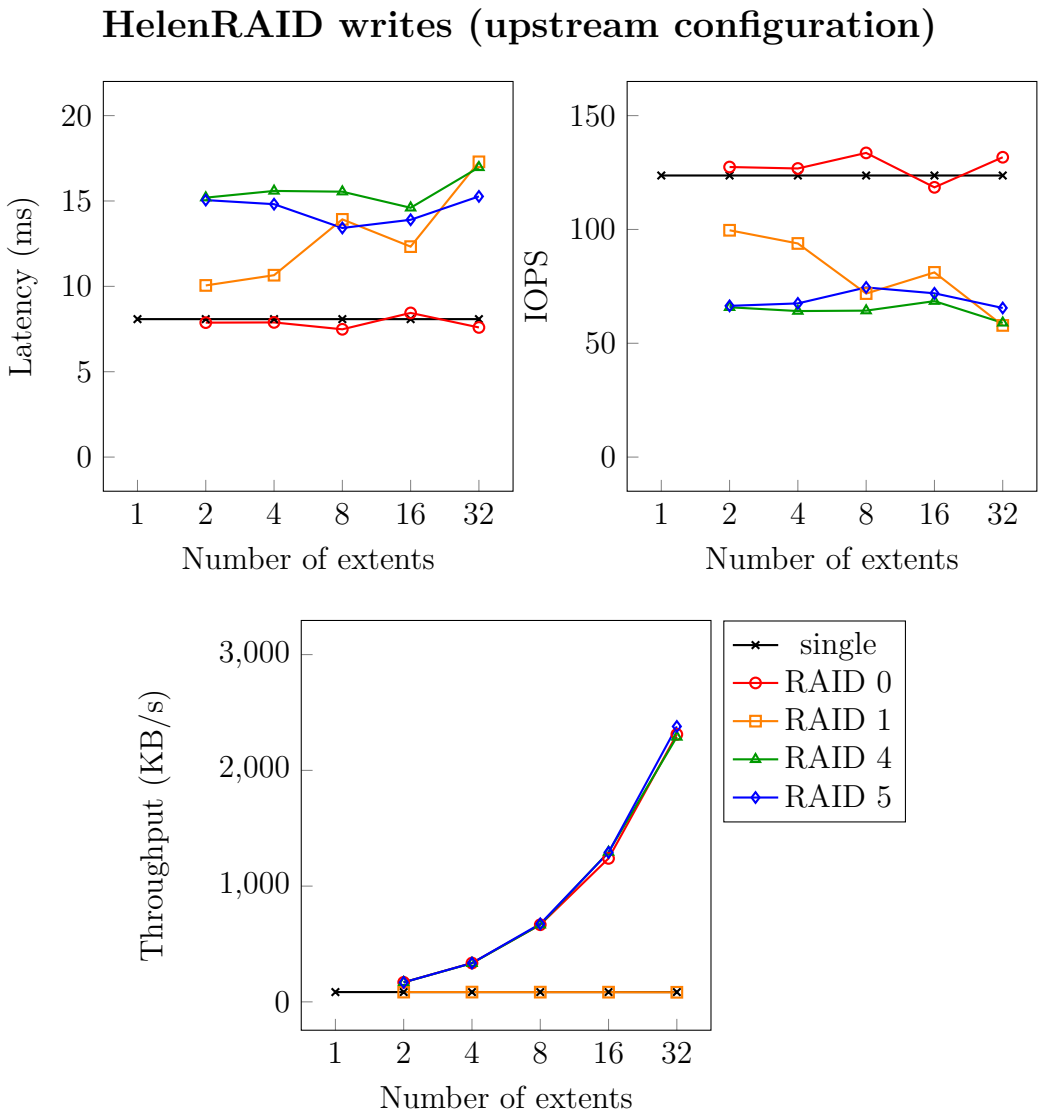

Figure 4.4 HelenRAID level write performance comparison, upstream configuration

that RAID 4, 5 have roughly double the latency of the single extent because of reconstruct-writes and subtract-writes. RAID 1's write latency shall be equal to the minimum of all extents. This seems like the case, as only a single slightly slower disk is needed to increase the latency. And as the number of extents increases, the probability that such a slower extent is present is higher.[6]

Due to the linear increase in the write latency with I/O size, the write throughput perfectly aligns with the theoretical speedup of each level. RAID 1 has the same throughput as the single disk, and the throughput of striped volumes increases linearly with the number of extents. Note that RAID 4, 5 have the same throughput as RAID 0 thanks to full-stripe writes. The throughput plot clearly shows that the time spent doing the parity computation is negligible.

Even though the relative speedups are equal to theoretical performance gains, we conclude that using the upstream configuration would be wasting of under-utilized system resources.

This is also the only case where file systems using block size bigger than 512 B,

[6]Note that to prove this, the write latencies of all disks would have to be measured.

such as ext4 which uses 4 KiB block size by default, would benefit from striping. With the strip size set to $\frac{4096}{N}$, there would be N-fold speedup until the strip size reaches 512 B. After $N = 8$, RAID 0 would have constant speedup, but RAID 4, 5 will not be able to do full-stripe writes, and the performance would degrade due to the subtract or reconstruct write's latency.

4.5 Increased transfer size configuration

In this section, we compare the throughput of different levels using increased maximum IPC transfer size from 64 KiB to 2 MiB and the modified virtio-blk driver. We only show the throughput of sequential workloads, because latency and IOPS would be the same.

Disk latency profile

Write latency profile showing the arithmetic mean of latencies across three different disks with increasing I/O size can be found in Figure 4.5. The variance of the values measured between the three disks was again zero or very close to zero.

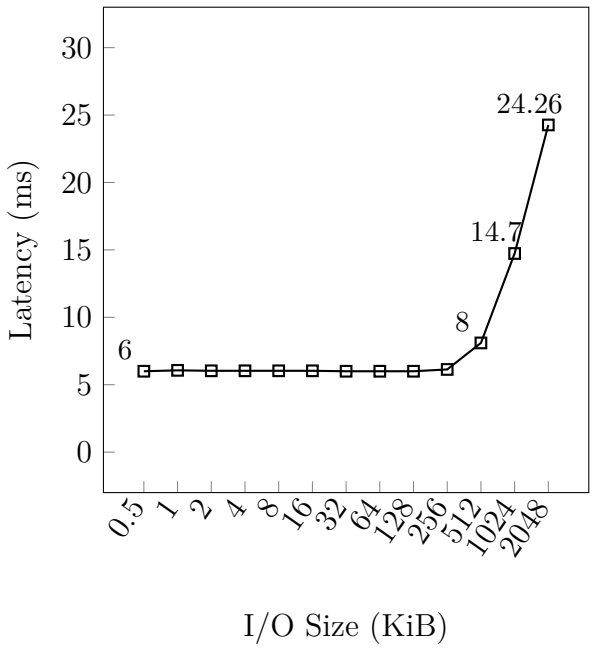

Figure 4.5 Disk write I/O latency vs. I/O size, 2 MiB configuration

The write I/O latency is constant until 512 KiB request size. After 512 KiB, the latency increases, but non-linearly, as opposed to the write latency profile measured with the upstream virtio-blk. Because of this, it is easy to predict that splitting small write I/O to even smaller parts will not give us any speedup.

Read I/O

Read I/O throughput comparison across levels with the 2 MiB configuration is show in Figure 4.6.

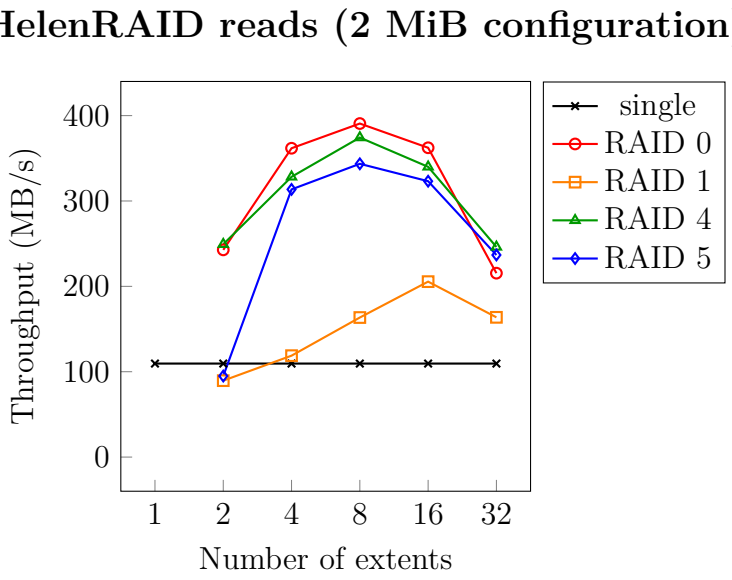

Figure 4.6 HelenRAID level read performance comparison, 2 MiB configuration

The throughput for the striped volumes follows the same pattern, while RAID 1 has lower throughput for the following reason. RAID 1 uses the split strategy, which means that 2 MiB is divided by N, the same size that the I/O is striped in the striped volumes case. However, since the extents in the striped volumes have sequential data "next to each other" in the same row, the next I/O that will come to a specific extent will always start where the previous ended. This is not the case for RAID 1, because the I/O is split across different "rows". This requires each extent to seek to $LBA_{new} = (LBA_{end} + 1) + (N - 1) \cdot S_B$, where LBA_{end} is the last seek position and S_B is the size of each split I/O in blocks. Even with the smallest $N = 2$, each extent for a new read request has to seek 1 MiB forward.

The trend of decreasing read throughput as N increases can also be observed in this configuration. We are probably hitting the limits of HelenOS with the number of fibrils, and therefore as concurrency degrades the I/O throughput decreases. Note that this is just an assumption and that this problem arises mainly with read requests.

Write I/O

Write I/O throughput comparison between levels with the 2 MiB configuration can be found in Figure 4.7.

The throughput of striped volumes increases as N increases, which is expected, however, we can see that these speedups are nowhere near the theoretical predictions. In the $N = 32$ case, requests probably hit the mentioned bottleneck.

We expected RAID 1 writes to have around the same throughput as a single disk, however, this behavior is surely linked to the RAID 1 write latency as observed in Section 4.4.

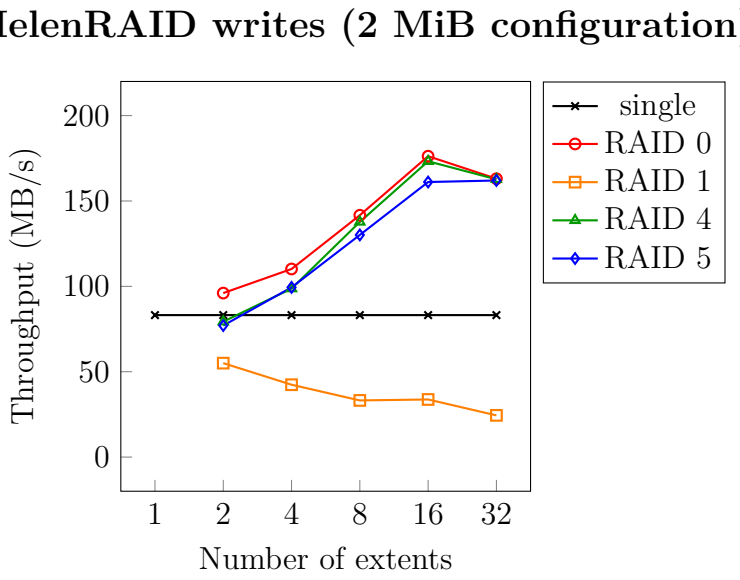

Figure 4.7 HelenRAID level write performance comparison, 2 MiB configuration

4.6 HelenRAID vs. other implementations

In this section, we compare the performance of HelenOS and HelenRAID against other operating systems and their software RAID implementations. We have chosen operating systems and implementations for which we have written support. That means FreeBSD's GEOM Stripe and Mirror, OpenBSD's softraid, and Linux's MD. The only thing specifically configured was setting strip size of striped volumes to match that of HelenRAID, and configuring gmirror to use the split strategy for reads. softraid that had the default strip size used (64 KiB). The volumes were not created with any other explicit settings used, only the defaults. Also note that OpenBSD has a maximum physical transfer size of 64 KiB, so that it is going to be in a disadvantage when benchmarking sequential (2 MiB) I/O. Additionally, OpenBSD uses interrupt disabling in some I/O paths to prevent races on internal structures, thus greatly limiting potential concurrency.

Another important thing to note is that to save time benchmarking and processing results, the latency and IOPS values for HelenRAID were taken from the upstream configuration, and the throughput was taken from the 2 MiB configuration section dedicated to HelenRAID levels. Additionally, the tests performed on other operating systems used different random workload files and the same sequential workload file, but only single set for each test, whereas HelenRAID values are the mean of three different sets, so these tests are not completely fair. However, this does not prevent us from doing meaningful analysis and comments on observed performance properties.

4.6.1 RAID 0

In this subsection we compare RAID 0 performance across different RAID implementations and operating systems.

Read I/O

RAID 0 read performance comparison across implementations and operating systems is shown in Figure 4.8.

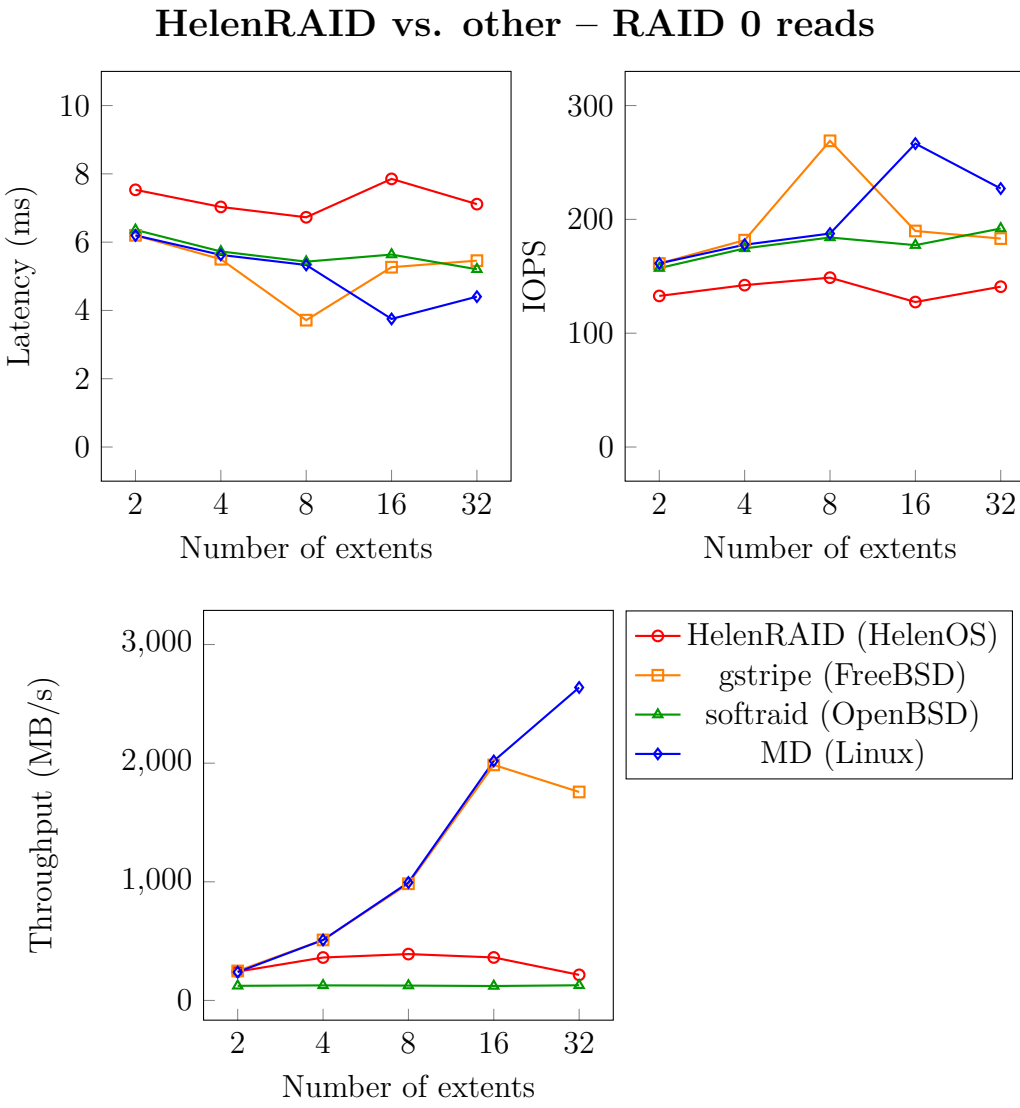

Figure 4.8 RAID 0 read performance comparison, vs. other implementations

Although HelenRAID's latency is higher compared to other implementations, it is not significant, considering that HelenOS uses a microkernel and is not as mature as the other tested systems. The throughput degradation of HelenRAID starting at $N = 4$, can be attributed to the probable HelenOS bottleneck already mentioned that limits the scaling of reads. gstripe stops scaling at $N = 32$, whereas MD continues to scale. softraid's throughput is limited by the 64 KiB strip size, and it does not scale and stays constant across all N because of OpenBSD's mentioned concurrency limitation. Although it is not clear from the plot, softraid has a throughput roughly equal to that of a single disk in all N.

Write I/O

RAID 0 write performance comparison across implementations and operating systems is shown in Figure 4.9.

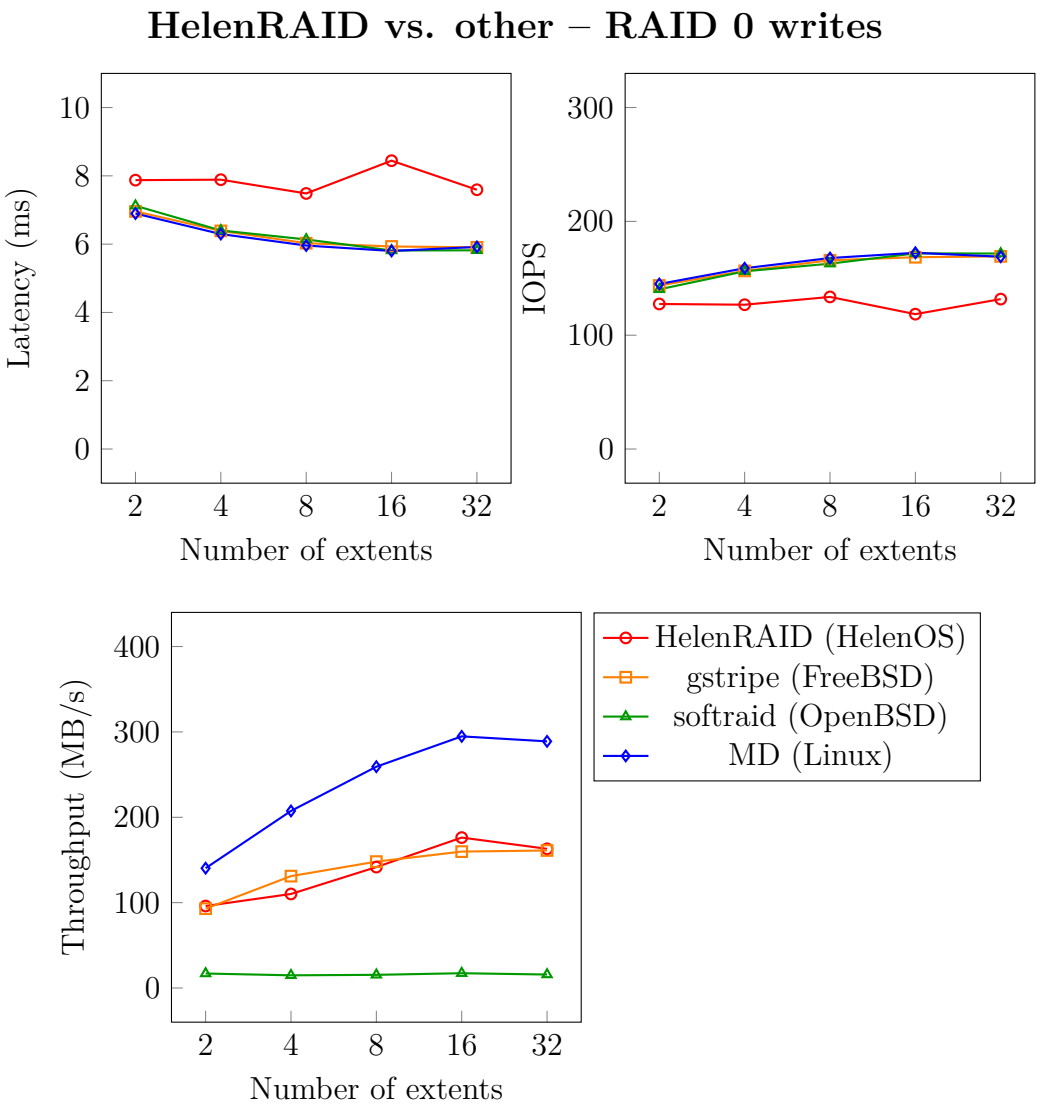

Figure 4.9 RAID 0 write performance comparison, vs. other implementations

HelenRAID's higher latency and lower IOPS can be attributed to multiple elements such as the already mentioned kernel design, the maturity of other operating systems, or the fact that these values are compiled from different workload sets.

The throughput of HelenRAID's RAID 0 writes is very comparable to the throughput of gstripe.

4.6.2 RAID 1

In this subsection we compare RAID 1 performance across different RAID implementations and operating systems.

Read I/O

RAID 1 write performance comparison across implementations and operating systems is shown in Figure 4.10.

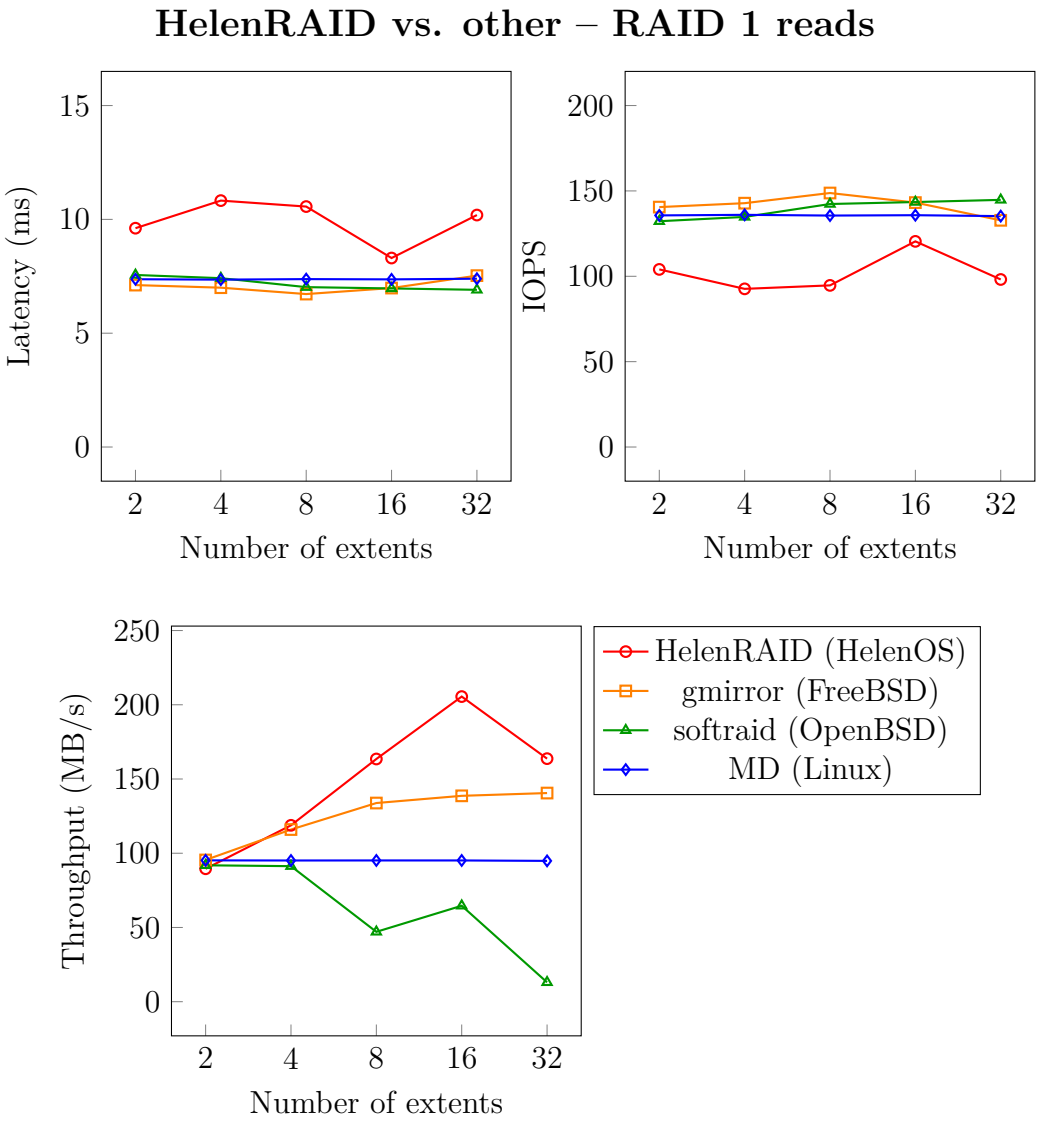

Figure 4.10 RAID 1 read performance comparison, vs. other implementations

Notable observations on throughput include that it seems like MD is using just a single disk in its default configuration because the plot is constant. softraid seems to split I/O, but hits its concurrency limitations, forcing each internal I/O to be processed synchronously. gmirror is gaining slight speedups with increasing N, while HelenRAID has an almost constant speedup until $N = 16$. The drop from $N = 16$ to $N = 32$ can be observed in all HelenRAID plots and its because of the already mentioned probable HelenOS bottlenecks.

Write I/O

RAID 0 write performance comparison across implementations and operating systems is shown in Figure 4.11.

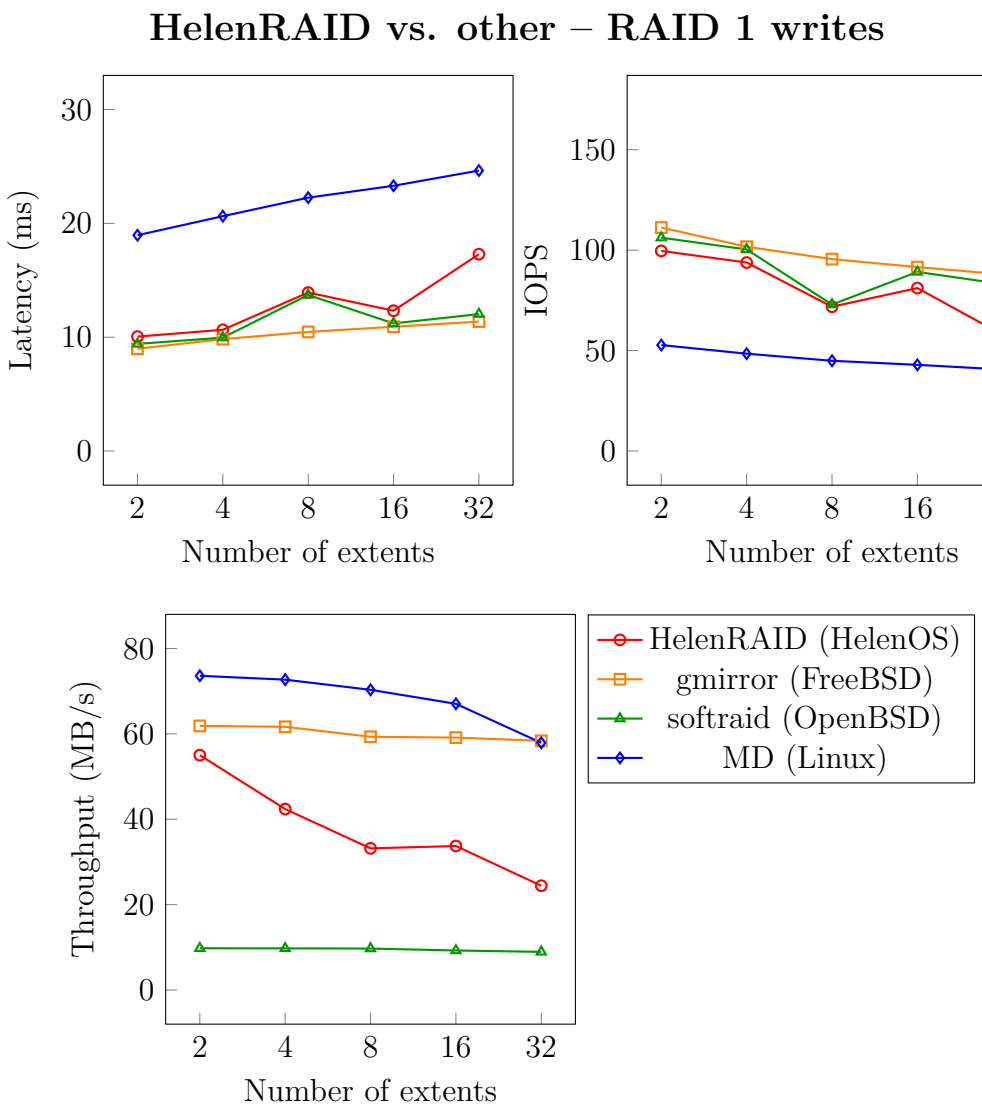

Figure 4.11 RAID 1 write performance comparison, vs. other implementations

HelenRAID's latency is very similar to that of gmirror and softraid, almost 10ms lower than MD for each N. This could be attributed to MD using a write-intent bitmap, but because we have not studied the internals of MD RAID implementation, this is just an assumption.

As N increases, HelenRAID's throughput drastically drops, while MD and gmirror follow a similar trend, but the degradation is not so significant.

4.6.3 RAID 5

In this section, we show performance comparisons of RAID 5 between different operating systems and their RAID implementations.

Read I/O

RAID 5 read performance comparison across implementations and operating systems is shown in Figure 4.12.

The read measurements are very similar to RAID 0, and therefore we omit further analysis and comments.

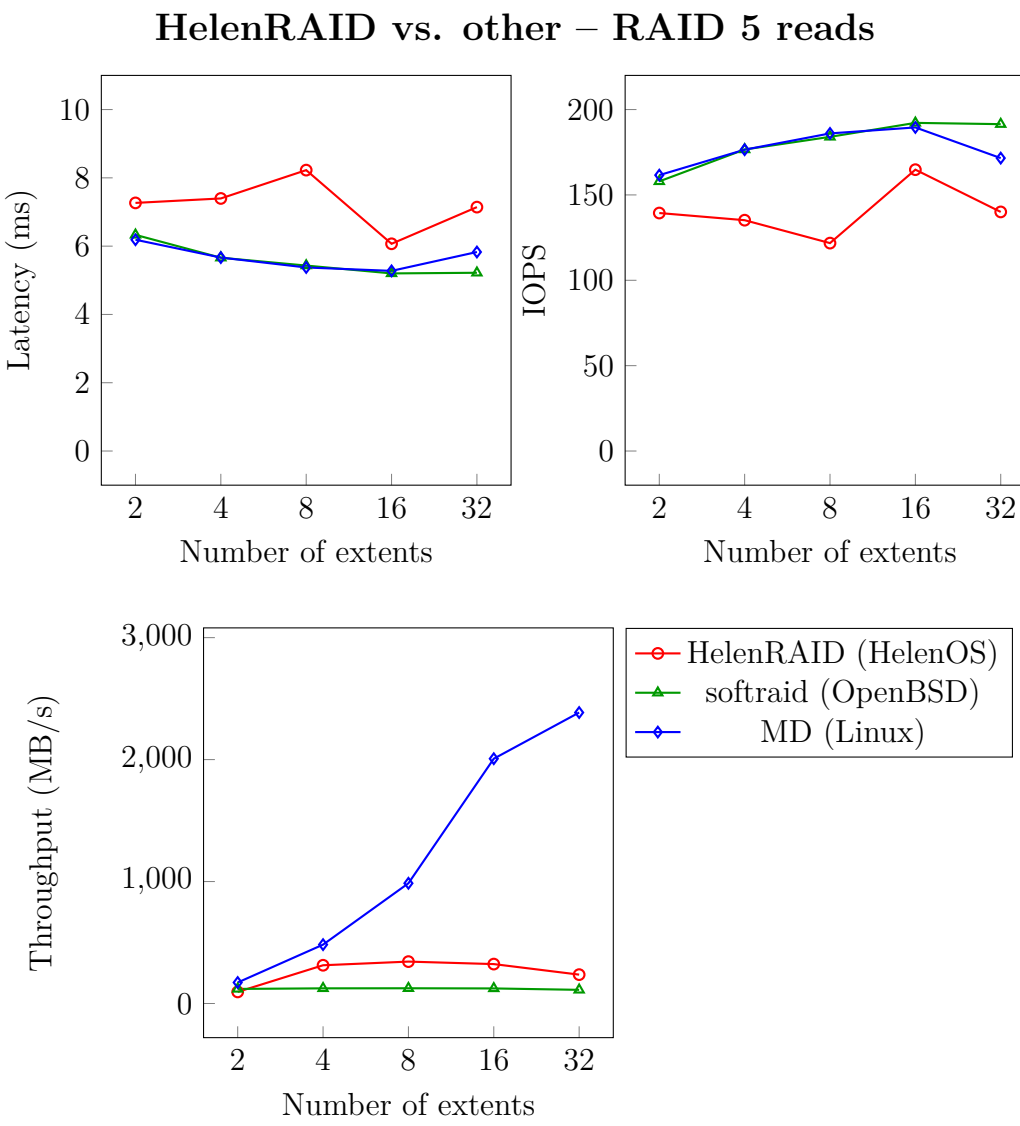

Figure 4.12 RAID 5 read performance comparison, vs. other implementations

Write I/O

RAID 5 write performance comparison across implementations and operating systems is shown in Figure 4.13.

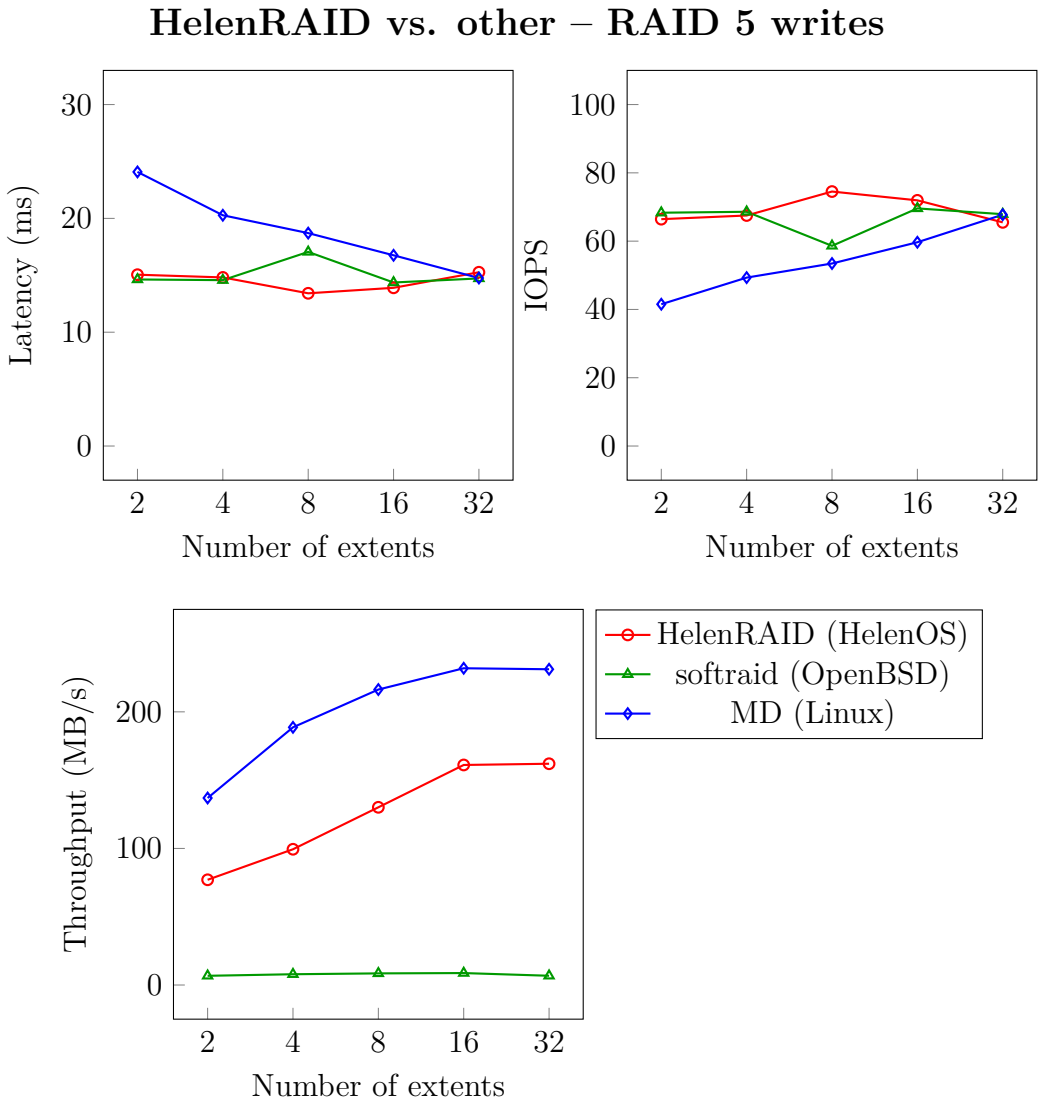

Figure 4.13 RAID 5 write performance comparison, vs. other implementations

As in other tests, the throughput of MD scales better as N increases, but HelenRAID follows a similar curve.

4.6.4 RAID 5 mixed size I/O (512 B – 2 MiB)

In this subsection, we show the comparison of RAID 5's performance on mixed I/O, with I/O size ranging from 512 B to 2 MiB to random offsets, not aligned to any size. We compare HelenRAID (HelenOS) with MD (Linux). We mainly wanted to test our designed parity striping mechanism, and see how it compares to Linux's MD RAID 5 implementation. We tested three different sets generated with the same parameters.

Read I/O

The comparison of mixed size read I/O can be found in Figure 4.14.

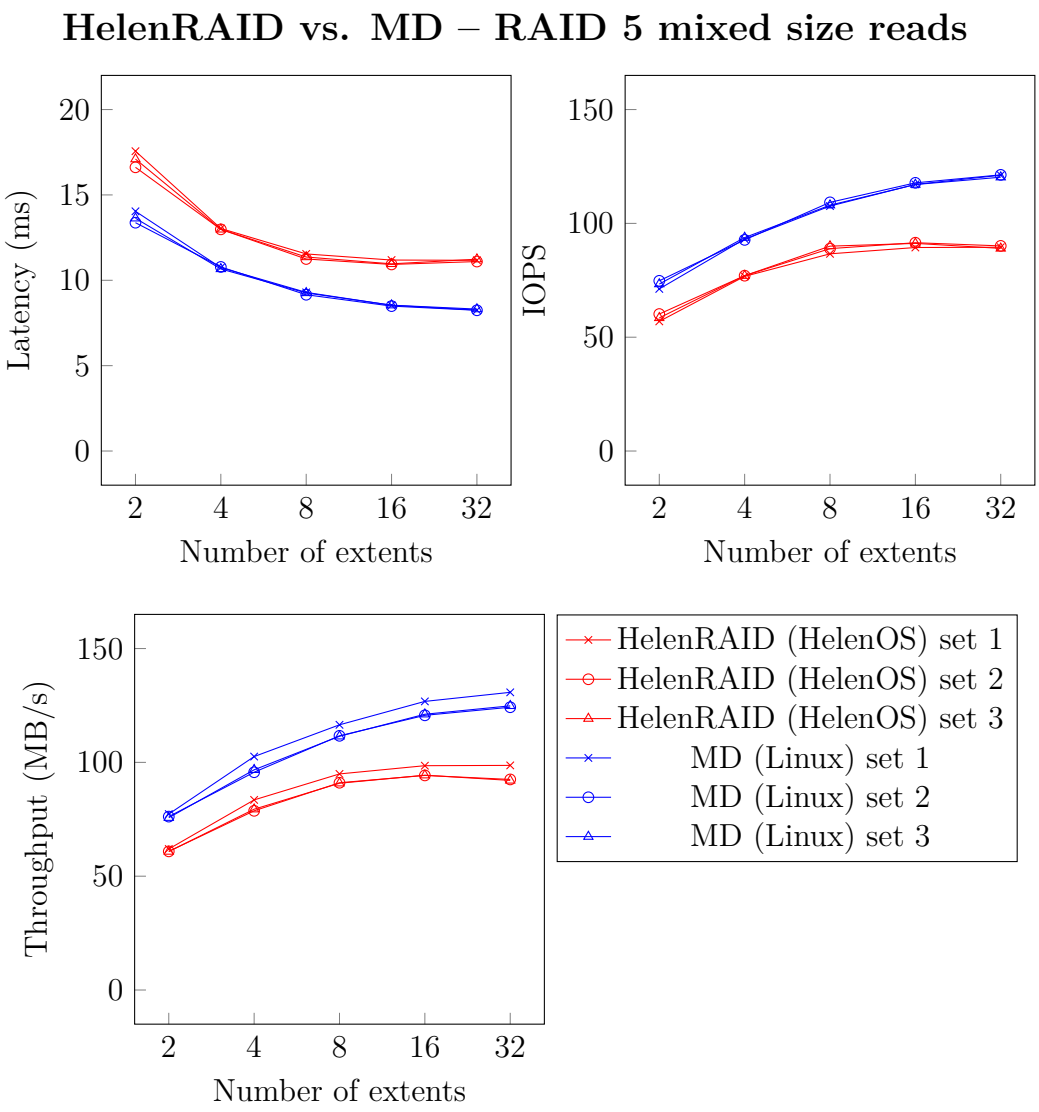

Figure 4.14 RAID 5 read performance comparison vs. MD, 512 B – 2 MiB I/O

Reads in optimal state RAID 5 volumes are essentially the same as in RAID 0, and given that HelenRAID has higher read I/O latency in general, as seen in previous read benchmarks, this is expected.

Write I/O

The comparison of mixed size write I/O can be found in Figure 4.15.

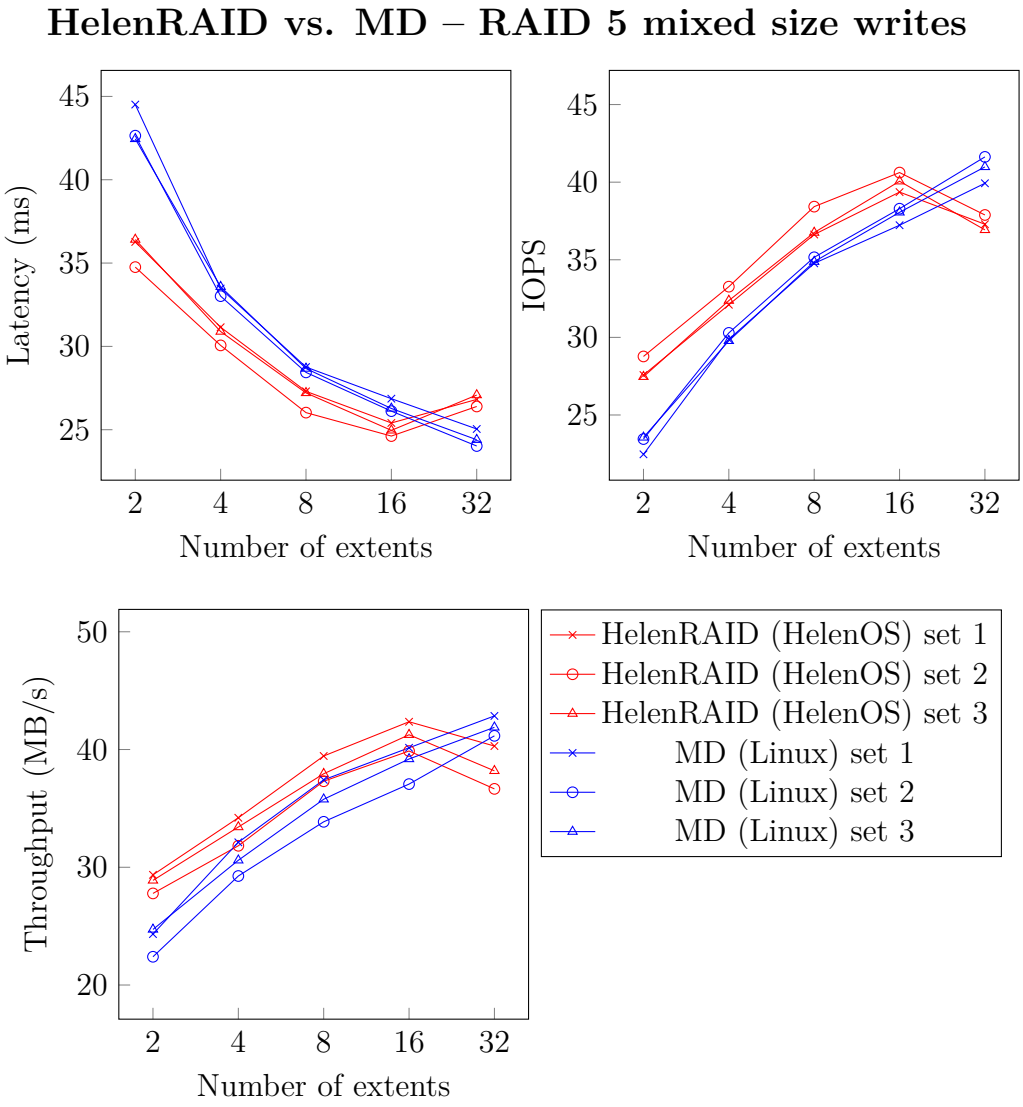

Figure 4.15 RAID 5 write performance comparison vs. MD, 512 B – 2 MiB I/O

HelenRAID is more performant than MD for each parameter measured in each tested set up to $N = 16$. In the case of $N = 32$, HelenRAID's performance degrades due to HelenOS's bottleneck, which can be observed in most of the previous tests. Namely, the latency difference is quite significant, even if we account for MD's write-intent bitmap (roughly 6-7 ms for a write as seen in RAID 0 benchmarks), and some additional latency for seeks, but this greatly depends on Linux's `mq-deadline` I/O scheduler used by default for spinning disks, and also the disk's internal I/O scheduler. As N increases, the latency gap is slightly closing. This can be attributed to Linux's better scalability with more disks.

Overall, given HelenOS's microkernel design and Linux's maturity, the results show that the design and implementation of HelenRAID's RAID 5 is reasonably competitive with MD's RAID 5 implementation.

Conclusion

We have successfully designed and implemented RAID for the HelenOS operating system, called *HelenRAID*. HelenRAID fully supports traditional RAID levels, specifically RAID 0 (non-redundant striping), RAID 1 (mirroring), RAID 4 (striping with dedicated parity) and RAID 5 (striping with distributed parity). Common basic volume management operations, such as volume creation, automatic assembly, and basic monitoring of volume state, are supported. Fault-tolerant RAID level (1, 4, 5) volumes support automatic rebuild process initialization when an active device fails, given a hotspare device is provided. Although the thesis goals did not explicitly specify what the implementation should support and be capable of, we believe that HelenRAID fulfills practical requirements for basic traditional use cases of RAID volumes in storage systems.

In addition to the main goal of designing and implementing RAID, another goal was to support RAID metadata formats from other software RAID implementations. Using the format-agnostic metadata handling interface designed, we implemented support for OpenBSD's `softraid(4)`, FreeBSD GEOM's `gstripe(8)` and `gmirror(8)`, and Linux's `md(4)` volumes provided by `mdadm(8)`. The support of foreign volumes includes the support of an ongoing rebuild operation.

HelenRAID was designed and implemented with considerable attention to I/O performance. We evaluated HelenRAID's performance using enterprise-class hard drives, with volume configurations reaching up to 33 disks. We showed that HelenRAID's performance is comparable to software RAID implementations in mature monolithic kernel-based operating systems, namely Linux, FreeBSD, and OpenBSD. In multiple scenarios and configurations, HelenRAID performed even better than the other operating systems, which is a great achievement given HelenOS's microkernel design and HelenOS being mainly just a research project.

To the best of our knowledge, HelenRAID is the first native RAID implementation for a microkernel-based operating system. Because it is written entirely from scratch and effectively with zero modifications done to HelenOS, it can be used to implement RAID in other microkernel-based operating systems.

Future work – improvements and extensions

HelenRAID is currently in the process of being reviewed so that it can be merged into the upstream HelenOS branch [37]. Further modifications such as creating a new library dedicated just to HelenRAID – *libraid*, in which also the unit tests are going to be integrated, were discussed with an active HelenOS developer, and will follow. And while HelenRAID's implementation is quite complete, we would like to additionally add metadata saving to hotspare extents.

Thanks to the format-agnostic metadata interface, it is easy to extend HelenRAID with the support of other RAID metadata formats. Support for other RAID levels such as RAID 6 could be added, but HelenRAID can be used as a base for I/O transformation applications other than RAID, such as disk encryption, special cache or journaling devices implemented as RAID levels.

Other work could include support for booting from RAID volumes.

Bibliography

1. Patterson, David A.; Gibson, Garth; Katz, Randy H. A case for redundant arrays of inexpensive disks (RAID). *SIGMOD Rec.* 1988, vol. 17, no. 3, pp. 109–116. ISSN 0163-5808. Available from DOI: 10.1145/971701.50214.
2. *HelenOS homepage* [online]. [visited on 2025-06-18]. Available from: https://www.helenos.org/.
3. *Logical block addressing* [online]. [visited on 2025-06-20]. Available from: https://en.wikipedia.org/wiki/Logical_block_addressing.
4. *Common RAID Disk Data Format Specification* [online]. [visited on 2025-06-18]. Available from: https://www.snia.org/sites/default/files/SNIA_DDF_Technical_Position_v2.0.pdf.
5. *RAID 2* [online]. [visited on 2025-06-19]. Available from: https://en.wikipedia.org/wiki/Standard_RAID_levels#RAID_2.
6. *RAID 3* [online]. [visited on 2025-06-19]. Available from: https://en.wikipedia.org/wiki/Standard_RAID_levels#RAID_3.
7. Jackson, Robert; Rumynin, Dmitriy; Zaboronski, O. An approach to RAID-6 based on cyclic groups of a prime order. *arXiv preprint cs/0611109*. 2006.
8. White, Jay; Lueth, Chris; Bell, Jonathan. *RAID-DP: NetApp Implementation of DoubleParity RAID for Data Protection* [online]. [visited on 2025-06-19]. Available from: https://www.netapp.com/pdf.html?item=/media/19939-tr-3298.pdf.
9. *OpenZFS RAIDZ documentation* [online]. [visited on 2025-06-19]. Available from: https://openzfs.github.io/openzfs-docs/Basic%20Concepts/RAIDZ.html.
10. Courtright II, William V.; Gibson, Garth; Holland, Mark; Reilly, LeAnn Neal; Zelenka, Jim. *RAIDframe: A Rapid Prototyping Tool for RAID Systems*. 1996. Available also from: https://www.pdl.cmu.edu/RAIDframe/raidframebook.pdf. Version 1.0.
11. Klein, Gerwin; Elphinstone, Kevin; Heiser, Gernot; Andronick, June; Cock, David; Derrin, Philip; Elkaduwe, Dhammika; Engelhardt, Kai; Kolanski, Rafal; Norrish, Michael; Sewell, Thomas; Tuch, Harvey; Winwood, Simon. seL4: formal verification of an OS kernel. In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. Big Sky, Montana, USA: Association for Computing Machinery, 2009, pp. 207–220. SOSP '09. ISBN 9781605587523. Available from DOI: 10.1145/1629575.1629596.
12. Franke, Hubertus; Russell, Rusty; Kirkwood, Matthew. Fuss, futexes and furwocks: Fast userlevel locking in linux. In: *AUUG Conference Proceedings*. AUUG, Inc, 2002, vol. 85, pp. 479–495.
13. *IPC for Dummies* [online]. [visited on 2025-06-21]. Available from: https://www.helenos.org/wiki/IPC.

14. *HelenOS 0.2.0 design documentation* [online]. [visited on 2025-06-21]. Available from: https://www.helenos.org/doc/design.pdf.

15. *Implementation and design of the file system layer* [online]. [visited on 2025-06-22]. Available from: https://www.helenos.org/wiki/FSDesign.

16. *Writing Device Drivers for HelenOS* [online]. [visited on 2025-06-22]. Available from: https://www.helenos.org/wiki/DeviceDrivers.

17. Horký, Vojtěch. *Plain C unit testing framework* [online]. [visited on 2025-07-06]. Available from: https://github.com/vhotspur/pcut/wiki.

18. *Userspace unit testing with PCUT* [online]. [visited on 2025-07-06]. Available from: https://www.helenos.org/index.fcgi/wiki/PCUT.

19. Gregg, Brendan. *Systems Performance: Enterprise and the Cloud*. 2nd. Addison-Wesley Professional, 2020. isbn 9780136820154.

20. *Intel Rapid Storage Technology* [online]. [visited on 2025-06-24]. Available from: https://en.wikipedia.org/wiki/Intel_Rapid_Storage_Technology.

21. Menon, Jai; Cortney, Jim. The architecture of a fault-tolerant cached RAID controller. *SIGARCH Comput. Archit. News*. 1993, vol. 21, no. 2, pp. 76–87. issn 0163-5964. Available from doi: 10.1145/173682.165144.

22. Stodolsky, Daniel; Gibson, Garth; Holland, Mark. Parity logging overcoming the small write problem in redundant disk arrays. *SIGARCH Comput. Archit. News*. 1993, vol. 21, no. 2, pp. 64–75. issn 0163-5964. Available from doi: 10.1145/173682.165143.

23. Denehy, Timothy E.; Arpaci-Dusseau, Andrea C.; Arpaci-Dusseau, Remzi H. Journal-guided resynchronization for software RAID. In: *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies - Volume 4*. San Francisco, CA: USENIX Association, 2005, p. 7. FAST'05.

24. Zhang, Yupu; Rajimwale, Abhishek; Arpaci-Dusseau, Andrea C.; Arpaci-Dusseau, Remzi H. End-to-end data integrity for file systems: a ZFS case study. In: *Proceedings of the 8th USENIX Conference on File and Storage Technologies*. San Jose, California: USENIX Association, 2010, p. 3. FAST'10.

25. Menon, Jai; Mattson, Dick. Comparison of sparing alternatives for disk arrays. *SIGARCH Comput. Archit. News*. 1992, vol. 20, no. 2, pp. 318–329. issn 0163-5964. Available from doi: 10.1145/146628.140392.

26. *md(4) — Multiple Device driver, Linux manual page* [online]. [visited on 2025-06-25]. Available from: https://man7.org/linux/man-pages/man4/md.4.html.

27. *mdadm(8) — manage MD devices, Linux manual page* [online]. [visited on 2025-06-25]. Available from: https://man7.org/linux/man-pages/man8/mdadm.8.html.

28. Greenan, Kevin M.; Plank, James S.; Wylie, Jay J. Mean time to meaningless: MTTDL, Markov models, and storage system reliability. In: *Proceedings of the 2nd USENIX Conference on Hot Topics in Storage and File Systems*. Boston, MA: USENIX Association, 2010, p. 5. HotStorage'10.

29. Bairavasundaram, Lakshmi N.; Arpaci-Dusseau, Andrea C.; Arpaci-Dusseau, Remzi H.; Goodson, Garth R.; Schroeder, Bianca. An analysis of data corruption in the storage stack. *ACM Trans. Storage*. 2008, vol. 4, no. 3. ISSN 1553-3077. Available from DOI: 10.1145/1416944.1416947.

30. *softraid(4) — software RAID, OpenBSD manual page* [online]. [visited on 2025-06-27]. Available from: https://man.openbsd.org/softraid.4.

31. *gstripe(8) — control utility for striped devices, FreeBSD manual page* [online]. [visited on 2025-06-27]. Available from: https://man.freebsd.org/cgi/man.cgi?query=gstripe&apropos=0&sektion=8&manpath=FreeBSD+14.3-RELEASE&arch=default&format=html.

32. *gmirror(8) — control utility for mirrored devices, FreeBSD manual page* [online]. [visited on 2025-06-27]. Available from: https://man.freebsd.org/cgi/man.cgi?query=gmirror&apropos=0&sektion=8&manpath=FreeBSD+14.3-RELEASE&arch=default&format=html.

33. *graid(8) — control utility for software RAID devices, FreeBSD manual page* [online]. [visited on 2025-06-27]. Available from: https://man.freebsd.org/cgi/man.cgi?query=gvinum&apropos=0&sektion=8&manpath=FreeBSD+14.3-RELEASE&arch=default&format=html#DEPRECATION_NOTICE.

34. *gvinum(8) — Logical VolumeManager control program, FreeBSD manual page* [online]. [visited on 2025-06-27]. Available from: https://man.freebsd.org/cgi/man.cgi?query=gvinum&apropos=0&sektion=8&manpath=FreeBSD+14.3-RELEASE&arch=default&format=html#DEPRECATION_NOTICE.

35. *gvinum: Remove kernel support, FreeBSD commit* [online]. [visited on 2025-06-27]. Available from: https://cgit.freebsd.org/src/commit/sys/geom?id=f87bb5967670914f2f6d9ab4c732ab083a61b4c8.

36. *geom(4) — modular disk I/O request transformation framework, FreeBSD manual page* [online]. [visited on 2025-06-27]. Available from: https://man.freebsd.org/cgi/man.cgi?query=geom&apropos=0&sektion=4&manpath=FreeBSD+14.3-RELEASE&arch=default&format=html.

37. Cimerman, Miroslav. *HelenRAID Pull Request* [online]. 2025. [visited on 2025-07-01]. Available from: https://github.com/HelenOS/helenos/pull/256. Pull request #256 in HelenOS/helenos.

38. Cimerman, Miroslav. *softraid: incorrect chunk metadata checksum* [online]. 2025-04-22. [visited on 2025-07-02]. Available from: https://marc.info/?l=openbsd-tech&m=174535579711235&w=2. Email to the openbsd-tech mailing list.

39. Cimerman, Miroslav. *softraid: dangerous assembly, fix chunk sorting* [online]. 2025-06-30. [visited on 2025-07-02]. Available from: https://marc.info/?l=openbsd-tech&m=175129909212725&w=2. Email to the openbsd-tech mailing list.

List of Figures

List of Tables

A Attachments

The thesis attachments include the following.

- **helenos/**

  This directory contains HelenOS source code that includes HelenRAID, RAID unit tests, iotester, and sample configuration files for testing volumes.

- **helenos-images/**

  This directory contains pre-built images of the included HelenOS source code. Only amd64 images are provided. If you wish to make modifications to the source code or build own images, refer to Appendix B.

  For instructions on how to use HelenOS images and test HelenRAID, refer to Appendix D.

- **foreign-metadata-test/**

  This directory contains steps and volume management commands used to create and fill volumes with a picture for each metadata format. The steps are written in markdown. The directory also contains raw disk images with foreign volumes that can be tested in HelenOS. Please refer to Section D.4 of Appendix D for further details.

- **benchmarks/**

  This directory contains the iotester and iotest generator programs written for Unix systems, under `iotester-unix/`;

  under `workloads/` there are the generated files used for benchmarking, along with parameters used for their generation;

  under `run-scripts/` there are shell scripts used to launch other operating systems as well as the script used to create and run volume benchmarks. All scripts require bash;

  under `results/` are all the measurements from which the performance plots were compiled.

A.1 HelenOS source code

The attached source code (**helenos/**) can be also downloaded with the following command.

```
1 git clone -b helenraid-thesis \
2     https://github.com/mcimerman/helenos.git
```

The included HelenOS source code (**helenos/**) does not contain the increased maximum IPC transfer size and the modified virtio-blk driver. HelenOS tree containing the mentioned modifications can be fetched using the following command.

```
git clone -b helenraid-thesis-2m-cfg \
    https://github.com/mcimerman/helenos.git
```

Or alternatively, it is possible to just switch to the `helenraid-thesis-2m-cfg` branch if the git repository is already fetched.

```
git checkout origin/helenraid-thesis-2m-cfg
```

The specific modifications of the maximum IPC transfer size and the virtio-blk driver can be found in the following commit references. `0b2290168` and `ebde36ee91`.

Comparison of the HelenOS master branch at the time of thesis submission, and the branch with the attached source code can be found at this GitHub.com reference: mcimerman/helenos/compare/helenraid-thesis-master-checkpoint...mcimerman:helenos:helenraid-thesis.

Notice

We find it important to note that the current state of HelenOS's master branch on which the attached HelenRAID tree is integrated on top of is broken in the following way. Approximately every 10–15-th boot, the devman server deadlocks, which later crashes multiple drivers including `pci-ide`, making the boot stuck in the kernel console. Sometimes the boot process is stuck even if the devman deadlock is not detected. Also note that HelenRAID is launched after devman, therefore HelenRAID is not causing this issue. The only known solution is to kill QEMU and retry the boot.

A.2 Precompiled images

There are three precompiled HelenOS images for amd64 (x86-64) from the attached source code in the **helenos-images/** directory.

- **helenos_amd64.iso** – provides 1920x1080 graphical environment.
- **helenos_amd64_1440x900.iso** – provides 1440x900 graphical environment.
- **helenos_amd64_no_windows.iso** – provides 1280x960 terminals with no graphical environment. The terminals can be switched with `ALT + F1`, `F2`, and so on.

B Building HelenOS

HelenOS has its own compilation target, therefore a cross-compiler must be installed. HelenOS uses modified GNU binutils and gcc that have to be built from source.

B.1 Prerequisites

We have listed packages needed to build the toolchain and HelenOS itself in Ubuntu Server 25.04 and Alpine Linux v3.22. For other distributions, install equivalent packages as appropriate. If some issues arise, the reader can refer to the official HelenOS compilation from source wiki article – www.helenos.org/wiki/UsersGuide/CompilingFromSource. Note that this article is a bit outdated, and sufficient information needed to successfully build HelenOS is included in this appendix.

Ubuntu Server 25.04

```
apt install build-essential wget texinfo flex \
    bison dialog xorriso sed make coreutils libelf-dev \
    ninja-build meson python3 python-is-python3 unzip git
```

Alpine Linux v3.22

```
apk add gcc g++ git wget texinfo flex bison dialog \
    xorriso sed make coreutils ninja meson unzip \
    python3 tar sharutils patch
```

B.2 Compilation

Skip the following step if the attached HelenOS source is used.

```
git clone -b helenraid-thesis \
    https://github.com/mcimerman/helenos.git
```

Build the helenos-target compiler toolchain (binutils and gcc). The toolchain is installed to `$HOME/.local/share/HelenOS/` by default.

```
cd helenos/tools/
./toolchain.sh amd64
```

Build HelenOS.

```
cd ../../
mkdir helenos-build
cd helenos-build/
../helenos/configure.sh amd64
ninja config # optional
ninja image.iso
```

When modifications are done to the source code, it is necessary to recompile HelenOS and generate a new image with the `ninja image.iso` command.

- After the image (`image.iso`) is built, HelenOS can be launched in QEMU.

  For volume management documentation, refer to Appendix C.

  For instructions on how to use the built image, and the specific examples showing how HelenRAID can be used, how to run RAID unit tests, and how to assemble foreign volumes from prepared disk images, refer to Appendix D.

C HelenRAID usage

This is the documentation for using HelenRAID's management utility – **hrctl**.

C.1 Volume creation

Volumes are created with the `--create` option in two ways.

By specifying the RAID level and devices on the command line as follows.

```
hrctl --create <volume_name> --level <level> DEVICE...
```

Where `<volume_name>` must be shorter than 32 characters, and `<level>` must be one for the following.

- **stripe**, **striping**, or **0**. These level identifiers specify a non-redundant striped volume (RAID 0). The minimum number of devices needed is 2.
- **mirror**, **mirroring**, or **1**. These level identifiers specify a mirrored volume (RAID 1). The minimum number of devices needed is 2.
- **parity_dedicated**, or **4**. These level identifiers specify a volume that saves data parity to single device (RAID 4). The minimum number of devices needed is 3.
- **parity**, **parity_distributed**, or **5**. These level identifiers specify a volume that saves and distributes data parity among all devices (RAID 5). The minimum number of devices needed is 3.

Or by passing a SIF configuration file via `--create -f` options as follows.

```
hrctl --create -f configuration_file.sif
```

Where the configuration file must use of the following format.

```
<sif>
<hrconfig>
        <volume devname="vol_name_0" level="L">
                <extent devname="device0"></extent>
                <extent devname="device1"></extent>
                ...
                <extent devname="deviceN"></extent>
        </volume>

        ...

        <volume devname="vol_name_N" level="L">
                ...
```

```
        </volume>
</hrconfig>
</sif>
```

Specific volume configurations are represented by the `<volume>` tag, and must be enclosed inside the `<hrconfig></hrconfig>` tags. They must specify the `devname` and `level` attributes. The device name attribute must be shorter than 32 characters. The level attribute `L` must be one of 0, 1, 4, or 5. There can be multiple volumes specified. Note that the whole configuration must be additionally wrapped inside the `<sif></sif>` tags.

The `--create` and `--level` options have also short counterparts, `-c` and `-l`, respectively.

Additional volume flags

The `--create` option can also be given additional volume flags that specify if no metadata should be saved and/or that the volume created should be read-only. The options are `--no-meta` and `--read-only`, respectively.

They can be used as follows.

```
1 hrctl --create [--no-meta] [--read-only] OPTION...
```

Where `OPTION` can be any of the options presented above.

This can be useful, for example, when one has two devices with identical data and would like to have better read performance. A mirror can be created with both additional flags specified, possibly doubling read I/O throughput.

After creating a volume, it can be used as any other block device.

C.2 Volume monitoring

To get a list of active volumes and their state, the `--state` option can be used as follows.

```
1 hrctl --state
```

To get a detailed information about a specific volume such as its metadata format, RAID level used, and other parameters, volume names must be additionally specified to the `--state` option as follows.

```
1 hrctl --state VOLUME...
```

Multiple volumes can be specified.

The `--state` option can be substituted by its short counterpart `-s`.

C.3 Volume disassembly

Active volumes can be disassembled/stopped with the following command.

```
1 hrctl --disassemble [VOLUME]
```

Where `VOLUME` is an optional argument specifying which volume shall be disassembled. If no volume is specified, all active volumes are tried to be stopped.

If there is an open block service session to a volume, the command fails signaling that a volume is busy with the `EBUSY` error. If no volume is specified and there is more active volumes, a single busy volume will not prevent the other volumes from being stopped. In such a case, to find out which volumes were not disassembled, list all active volumes with the `--state` option.

It is advised to disassemble all active volumes before system shutdown.

C.4 Volume assembly

Volumes are automatically assembled upon boot, but because it is possible to disassemble volumes, the assembly operation can be initiated manually with `--assemble`. Devices from which the assembly process should try to bring up volumes can be specified either manually or automatically. The manual device specification can be done as follows.

```
1 hrctl --assemble [--read-only] DEVICE...
```

Devices can be also specified "semi-manually" by providing a SIF volume configuration file as follows.

```
1 hrctl --assemble [--read-only] -f config.sif
```

Completely automatic volume detection can be triggered by not specifying any arguments to the `--assemble` option as follows.

```
1 hrctl --assemble
```

The `--assemble` option can be substituted by its short version `-a`.

C.5 Volume modification

Volumes can be modified in two ways. By specifying a hotspare device for fault-tolerant volumes and simulating extent failures.

Hotspare addition

Hotspare addition is currently supported only for volumes using native Helen-RAID metadata. If the hotspare device is provided before a failure of an active extent happens, in such scenario when the failure happens, the rebuild process is automatically started to restore the volumes state. Hotspares can be added even in the case when the volume is already in a degraded state.

Hotspare devices are provided to a volume as follows.

```
1 hrctl --modify VOLUME --hotspare DEVICE
```

Where `DEVICE` is the device that should serve as a hotspare for the `VOLUME` volume. Note that `DEVICE` must at least as large as the smallest device ever used by the volume.

The `--modify` option has a short version `-m` and `--hotspare` has a short version `-h`.

Extent failing

Extent failures can be simulated by the following command.

```
1 hrctl --modify VOLUME --fail <index>
```

Where `<index>` is index of an extent to be failed in the `VOLUME` volume.

The `--fail` option has a short version `-f`.

Note, that when when native metadata format is used, the simulated failure additionally **deletes RAID metadata superblock**, making the extent not eligible for re-assembly. This is not the case for volumes that use foreign metadata formats.

D HelenRAID testing

This appendix is dedicated to how HelenOS can be started, and HelenRAID tested in numerous ways.

D.1 Launching HelenOS

To launch HelenOS, QEMU must be available, and for ideal performance, KVM support must be present. We show only the steps and commands for the amd64 (x86-64) architecture.

The following command can be used to launch HelenOS with 4 GiB of memory.

```
qemu-system-x86_64 -enable-kvm -m 4096 \
    -serial stdio -display gtk \
    -boot d -cdrom path/to/image.iso
```

- The `-serial stdio` option makes the kernel console be printed in the host's terminal where QEMU was started. This is useful to see HelenRAID's and other system logs.
- The `-display gtk` option tells QEMU to open a GTK window with HelenOS. Unfortunately, HelenOS does not support headless or "nographic" way of being used. Alternatively, instead of `-display gtk` a VNC session provided by QEMU can be used by specifying `-display none -vnc localhost:PORT,password=off`, where `PORT` is a number specifying the port that QEMU shall listen on for VNC client connections. Note that QEMU offsets the port number specified by 5900, so if 1000 is specified, QEMU's VNC server will listen on 6900.[1]

  When the GTK option is used, press `CTRL + ALT + G` to exit the QEMU-focused window.

Connecting drives

HelenOS supports file-backed block devices which can be created inside HelenOS, so there is no need to connect devices or disk images to QEMU. However, if one wishes to connect physical devices or at least file images representing disks, the following options have to be added to the QEMU launch options.

```
-drive file=FILE0,format=raw,media=disk \
-drive file=FILE1,format=raw,media=disk \
-drive file=FILE2,format=raw,media=disk
```

[1] In the VNC option passed to QEMU there can be `0.0.0.0` address specified in place of `localhost`, which will make the server listen on all interfaces, making it possible to connect to QEMU from outside, if a remote machine is wished to be used.

This connects the provided FILE[0-2] as pci-ide block devices inside HelenOS.

Test disk images can be created with the following commands.

```
qemu-img create testdisk1.img 10M
qemu-img create testdisk2.img 10M
qemu-img create testdisk3.img 10M
```

A script provided in the attachments at *helenos-images/run.sh* can be used to comfortably attach drives to HelenOS as follows.

```
cd helenos-images/
./run.sh path-to-file1 path-to-file2 path-to-file3
```

Note that the script has a hardcoded image path, by default it launches the provided `helenos_amd64.iso`.

Note that there can be at most 3 disks connected like this, because of IDE limitations. If one wishes to connect more than 3 drives, the attached devices must be specified as virtio-blk devices. Note that it might not be possible, as we were unable to connect devices with virtio-blk on our Zen 2 AMD CPU laptop. But virtio-blk is not required to properly test HelenRAID's functionality. However, if you would like to try virtio-blk, please refer to the shell script located in the attachments at *benchmarks/run-scripts/hr/helenos-qemu-start.sh* or this blog post: https://blogs.oracle.com/linux/post/how-to-emulate-block-devices-with-qemu.

D.2 Unit tests

Note that all following commands are supposed to be run inside HelenOS's shell if not stated otherwise. Path completion provided by pressing `TAB` can be utilized for faster file system navigation.

To launch the unit tests, run the following commands.

```
batch /hr_testdata/hr-unit-tests-create-disks.bdsh
/test/test-srv_bd_hr
```

The first command creates file-backed block devices used by the unit tests and will fail if it is run a second time. For repeated launch of the unit tests, use the second command.

Note that the `ENOTSUP` error code printed by one of the tests is expected, as the read-only volume flag is tested.

D.3 Manual test proposals

In this section, we propose different ways that a user could manually test HelenRAID. For usage of HelenRAID's management utility, please refer to Appendix C.

Creating file-backed block devices

If you wish to use attached devices to QEMU or the block devices already created for unit tests, you can skip this.

File-back block devices can be created by the following command.

```
mkfile -s 10M /tmp/file1
/srv/bd/file_bd -b 512 /tmp/file1 disk1
```

Identifying block devices

Basic block devices will appear in the *disk* location category. To list services registered under the disk category, use the following command.

```
loc show-cat disk
```

The `loc` command without options can be used to list all categories.

Device `devices/\hw\sys\00:01.1\c1d0` represents the HelenOS disk image. Devices attached to QEMU will appear as follows.

```
devices/\hw\sys\00:01.1\c0d0 <- first drive
devices/\hw\sys\00:01.1\c0d1 <- second drive
devices/\hw\sys\00:01.1\c1d1 <- third drive
```

File-backed block devices are visible under the name they were created with.

Basic operations

Start with creating a volume. Devices can be manually specified, or prepared SIF configuration files can be used as follows. For creating a volume from the file-backed block devices created for unit tests, run.

```
hrctl -c -f /hr_testdata/sample_hrconfig_file_bd.sif
```

For creating volumes from attached devices, the following configuration file can be used.

```
hrctl -c -f /hr_testdata/sample_hrconfig_pci-ide.sif
```

To list active volumes, run.

```
hrctl -s
```

To get detailed information about a volume, run the following command, if the sample configuration file for file-backed devices was used.

```
hrctl -s hr0
```

Configuration files can be editted with the `edit` program provided in HelenOS. After modifications are done, press `CTRL + S` to save them and `CTRL + Q` to quit.

To create a new volume, disassembling will probably be required, as either the devices are already used or a volume with the same name exists. To do so, run.

```
hrctl -d
```

After disassembling, automatic assembly can be tried.

```
hrctl -d
hrctl -a
```

Partial assembly can be tried.

```
hrctl -c mirror -l 1 disk1 disk2 disk3
hrctl -d
hrctl -a disk1
hrctl -s mirror
```

If a volume was created on attached disks, HelenOS can be rebooted and automatic assembly upon boot tested. Note that because of an issue described in Section A.1 of Appendix A, the boot process can be stuck, and QEMU must be killed in such a case. When KVM is enabled, the boot process should not be longer than 20–30 seconds, although it depends on the CPU.

Block address mappings

Block address mappings can be tested with a provided `bdwrite` utility that writes cyclic blocks (`'A'` – `'Z'` filled) to block devices. Note that this tool is not perfect and sometimes starts from `'A'` earlier when a high number of blocks is specified. However, it is good enough to verify how and if each level correctly maps volume block addresses to the extents.

The following command can be used to write 10 blocks from offset 0 to some volume.

```
bdwrite example_raid4_volume -o 0 -c 10
```

Then, the `blkdump` program can be used to check if blocks were written to the extents. Block 0 is read if no options are specified. To read specific blocks, these options can be used.

```
blkdump --offset 0 --count 1 disk1
blkdump --offset 0 --count 1 disk2
blkdump --offset 0 --count 1 disk3
```

If the attached block devices are raw disk images, a nice way to inspect them on the host is to use the `hexdump` program.

```
hexdump -C disk_image.img       # in shell on the host
```

File systems

File systems can be created with `mk*` programs. An ext4 (ext2) file system can be created and used, for example, as follows.

```
mkext4 volume_name
mkdir /mnt
mount ext4fs /mnt volume_name
cd /mnt
mkdir a b c
edit
cd /
umount /mnt
```

Partitions are not possible to be created on top of volumes due to issues discussed in Section 3.9 of Chapter 3.

Rebuild

The restoration of volume state can be tested, for example, as follows. We assume `disk4` is clean.

```
hrctl -c hr0 -l 4 disk1 disk2 disk3
bdwride hr0 -o 0 -c 1000
blkdump disk1
hrctl -s hr0                    # state is OPTIMAL
hrctl -m hr0 --fail 0
hrctl -s hr0                    # state is DEGRADED
blkdump disk4
hrctl -m hr0 --hotspare disk4
hrctl -s hr0                    # if quick enough state is REBUILD
hrctl -s hr0                    # after some time state is OPTIMAL
blkdump disk4
```

The `disk4` can be added as hotspare even before failure simulation. That way, the rebuild will be triggered by the simulated failure, and not by the hotspare addition.

Debug info

Although there should already be logs of at least volume creation, assembly, and deactivation, in the kernel console, the log level can be increased by the following command.

```
logset hr debug
```

To reset it back to default, substitute `debug` with `note`.

D.4 Foreign metadata

In the attached directory **foreign-metadata-test/**, steps used to create and populate volumes on OpenBSD, FreeBSD, and Linux can be found. The steps are written in markdown format.

We also prepared sample disk images containing ext2 file systems and the picture we used for testing. They can be found in their appropriate directory at the mentioned attachment. There are softraid and MD RAID 5 images, gstripe images for RAID 0 and gmirror for RAID 1. Also note that the tests were "officially" on real disks, and the steps there are for those real disk configurations. After the appropriate disk images are attached to QEMU and HelenOS is started, the foreign volume can be mounted, and the picture residing there can be viewed by the following commands. Note that HelenOS must be compiled with the Window system enabled. The Window system is enabled by default in the provided `helenos_amd64.iso` image. RAID 1, 5 volumes can be only partially assembled.

```
hrctl -s       # get the volume's name
mkdir /mnt
mount ext4fs /mnt volume-name
cd /mnt
viewer img.tga
cd /
umount /mnt
hrctl -d
```

We also placed the exact image in the `/hr_testdata/img.tga` path in HelenOS, so "equality" can be tested.

```
viewer /hr_testdata/img.tga
```

E Solid-state drives benchmarks

This appendix is used as a place to store not-analyzed I/O performance benchmarks done on solid-state drives. They were not analyzed due to the high variance in performance of the drives used, which prevented the results from having statistical significance. For a detailed performance evaluation on enterprise-class hard drives, see Chapter 4.

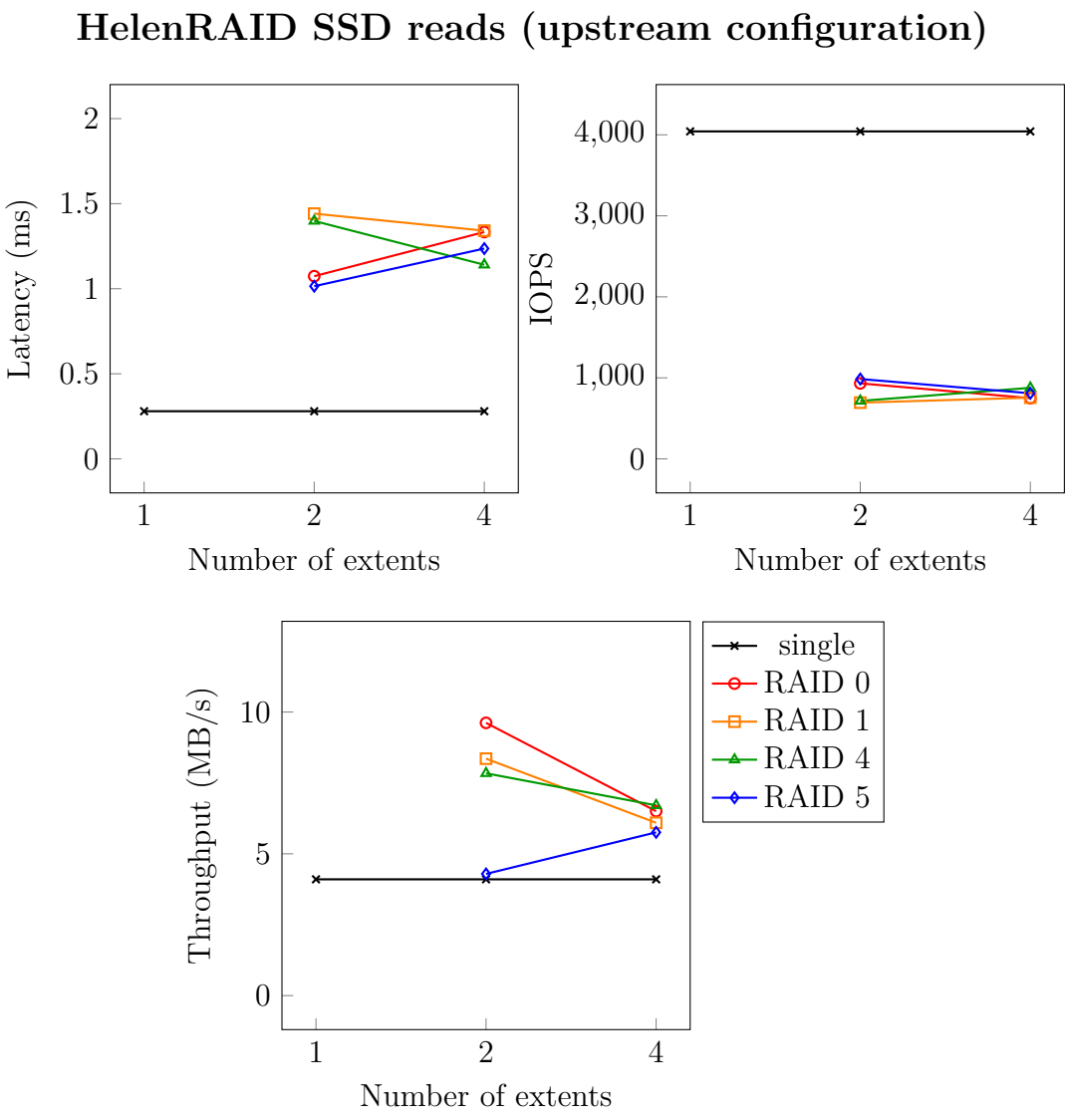

Figure E.1 HelenRAID level read performance comparison, SSDs, upstream configuration

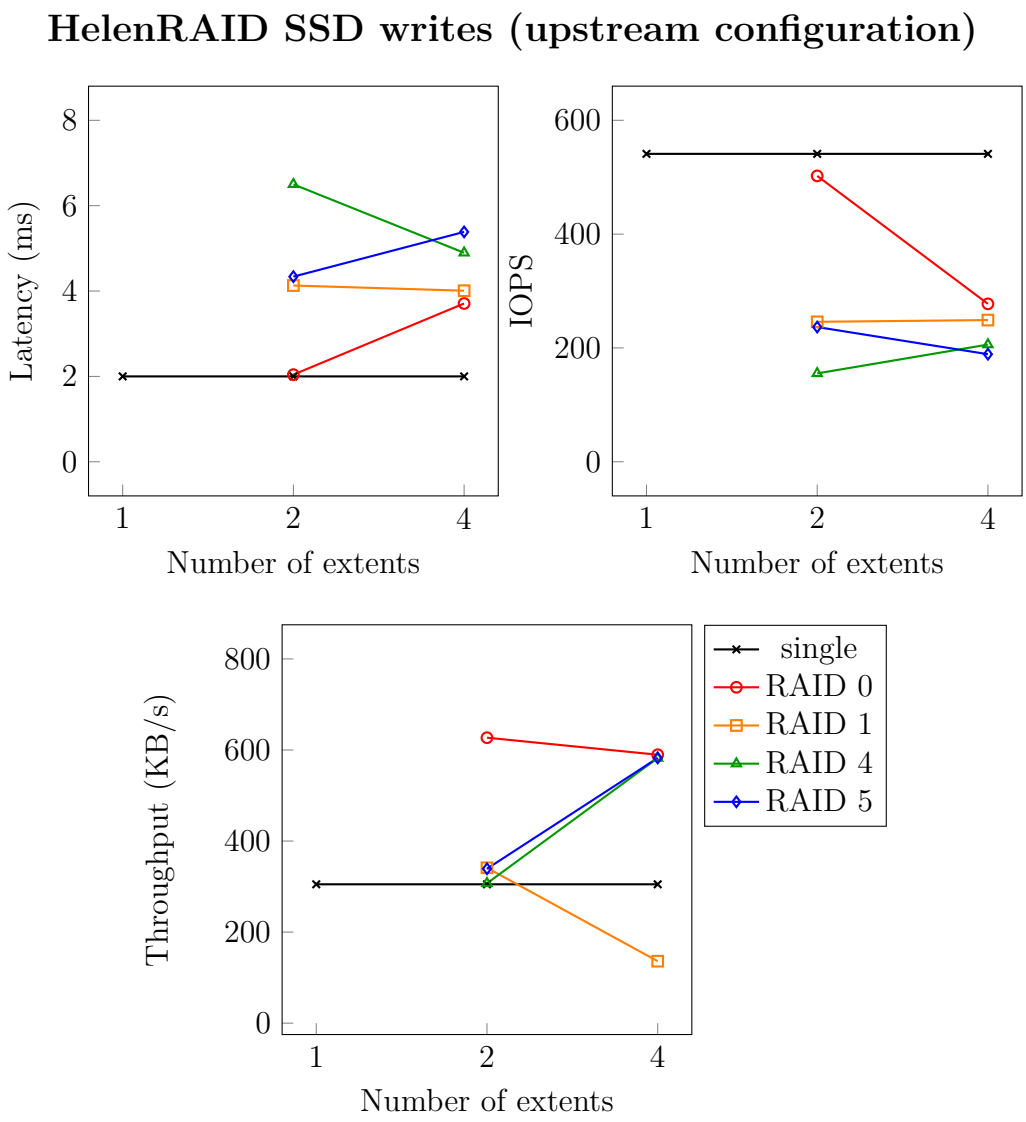

Figure E.2 HelenRAID level write performance comparison, SSDs, upstream configuration

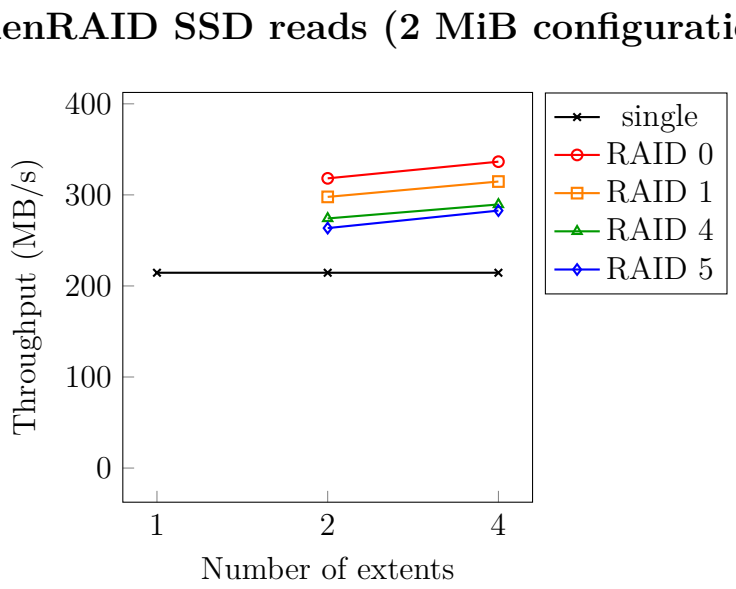

Figure E.3 HelenRAID level read performance comparison, SSDs, 2 MiB configuration

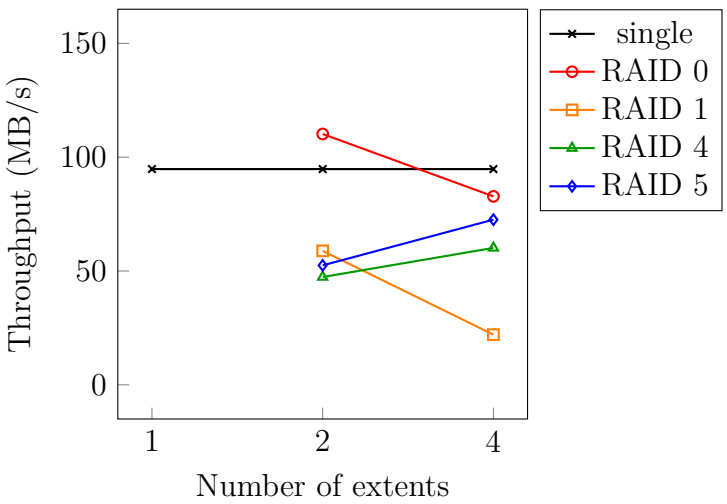

Figure E.4 HelenRAID level write performance comparison, SSDs, 2 MiB configuration