University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Piotr Sarna
Student no. 321169

Bypassing the limit of RAM capacity in distributed file system LizardFS

Master's thesis
in COMPUTER SCIENCE

Supervisor:
dr Janina Mincer-Daszkiewicz
Institute of Informatics

June 2016

Supervisor's statement

Hereby I confirm that the presented thesis was prepared under my supervision and that it fulfils the requirements for the degree of Master of Computer Science.

Date Supervisor's signature

Author's statement

Hereby I declare that the presented thesis was prepared by me and none of its contents was obtained by means that are against the law. The thesis has never before been a subject of any procedure of obtaining an aca- demic degree. Moreover, I declare that the present version of the thesis is identical to the attached electronic version.

Date Author's signature

Abstract

Distributed file systems with a centralized metadata server are constrained by many boundaries, one of which is limited RAM capacity. This particular issue can be potentially bypassed with the help of fast solid-state disks. My research will be based upon creating a library of alternative memory allocation methods, which would utilize underlying drive (e.g. SSD, ZRAM) for data storage, using RAM only as cache. The next step would be to implement the library as primary memory allocator in LizardFS — an open-source, distributed file system. Results could then be verified and benchmarked on real-life installations.

Keywords

distributed, file system, storage, memory, optimization, SSD, ZRAM, LizardFS, library, C++

Thesis domain (Socrates-Erasmus subject area codes)

11.3 Computer Science

Subject classification

Software and its engineering
- Software organization and properties
  - Contextual software domains
    - Operating systems
      - Memory management
        - Allocation / deallocation strategies

Information systems
- Information storage systems
  - Storage architectures
    - Distributed storage

Tytuł pracy w języku polskim

Obejście limitu pamięci RAM w rozproszonym systemie plików LizardFS

Contents

Introduction

Global need of maintaining very large amounts of data is still increasing. The storage sector is currently enjoying the benefits of having many independent distributed file systems designed to store petabytes of data safely and efficiently. Unfortunately, these file systems have their own limitations — CPU and RAM usage, power consumption, network latencies, keeping consistency of distributed data.

One of the interesting architectures is a "file system with centralized metadata server". Its huge advantage is not having to worry about metadata consistency, which is a non-trivial problem to cope with. On the other hand, memory and CPU usage is centralized as well, which leads to very high hardware requirements for large installations.

Some systems (e.g. LizardFS [13]) implement a policy of keeping all metadata in random-access memory. By that means, as long as there is enough memory available, metadata operations are performed quickly, without any need to use disk I/O. It is worth noting, that metadata operations are very frequent on a POSIX-compliant file system. The most excessively used one is a *lookup* call, which is used to extract information about a file (more specifically, an inode) from its path. Ensuring that this operation is fast should be the primary goal of a distributed file system developer.

This performance-oriented policy has one big flaw, which is extremely high memory consumption. An attempt to reduce RAM constraint without rewriting the whole application is the main topic of this thesis.

Princeton University researchers, Anirudh Badam and Vivek S. Pai presented an interesting idea on how to move the burden of dynamically allocated memory from RAM to solid-state hard drives [1, 3]. The project is called SSDAlloc and its goal is to intercept the allocation process and keep the data in designated SSD block device, using RAM only as temporary cache for frequently used pages.

Solid-state drives are not the only reasonable way to store allocated data, though. Another noteworthy idea is to take advantage of compressed RAM devices, e.g. ZRAM module [15] available in Linux kernel.

The thesis contains documented implementation of a memory management library — *fsalloc*, designed to overload traditional *malloc/new* calls with ones that keep allocated data in the database. The library was injected into distributed file system LizardFS in order to compare its performance and memory usage with and without *fsalloc*.

The thesis consists of 7 chapters:

1. Background — introduction to open-source distributed file system LizardFS and its memory management.
2. Motivations — introduction to SSDAlloc project and its summary.
3. Design and implementation — implementation details of *fsalloc* — memory management library.
4. Choice of storage — review of storage back-ends that could be used by *fsalloc*.
5. Integration with LizardFS — steps performed to introduce *fsalloc* as a memory manager for frequently used structures in LizardFS system.
6. Results — performance results of experiments carried out on production-ready installations.
7. Conclusions — comments regarding performance results and future development opportunities.

Electronic version of this thesis, implementation of *fsalloc* library and detailed performance results are available on the attached CD.

Chapter 1

Background

Allocation library is not a fully standalone product — it is used along with an application that benefits from its routines. In this particular case the *application* is a distributed file system cluster with high resource consumption — *fsalloc* is to be integrated with LizardFS and tested in terms of both correctness and performance.

In order to properly understand the design decisions, approaches and other details, overall description of LizardFS — background for the whole project — is presented in this chapter.

1.1. Introduction to LizardFS

LizardFS [13] is an open-source, distributed file system. Project's goal is to deliver fault-tolerant, highly available, scalable and POSIX-compliant alternative to other solutions. Source code is freely available and can be found at `https://github.com/lizardfs/lizardfs`.

1.1.1. Architecture

Conceptually, LizardFS can be divided into 4 parts:

1. metadata server, which can be either of:
   - master,
   - shadow,
   - metalogger,
2. chunkserver,
3. mount,
4. client-side extra features, namely:
   - CGI web server,
   - console administration tool.

Master is a centralized metadata management process. It holds all metadata present in the system in RAM, which makes it a primary target for memory usage optimization. Clients gather all needed information about files from master before performing any reads/writes. Chunkservers are managed from master server only — decisions on performing a replication between chunkservers or getting rid of certain chunks are commissioned directly from master.

Shadow is an online copy of master server, ready to become promoted to master once a failure occurs. Shadows are used in high-availability solutions and keep the same memory structure as masters, so they are the target of RAM optimization as well.

Metalogger is an offline copy of system's metadata and is not memory-consuming, so it will not be discussed further in this thesis.

Chunkserver is responsible for:

- keeping chunks of files on internal storage,
- ensuring that all chunks hold valid information (via checksum calculation),
- sending and receiving file data from clients.

All files in the system are divided into chunks that are up to 64 MiB in size. Chunks are then distributed over chunkservers in order to keep a safe number of copies and follow the rules of georeplication. Each file holds information on how many copies should be kept and on which groups of chunkservers. This information as a whole is called a *goal*.

Mount represents the most important client-side feature. It enables users to create a mount-point and use LizardFS transparently as a regular file system.

On Linux, LizardFS is available through FUSE (Filesystem in Userspace) mechanism, a widely known kernel driver which allows custom file systems to be developed entirely in userspace.

Additionally, LizardFS-specific functionalities are provided via a set of external binaries. Examples:

- setting a goal (replication policy for file's data),
- creating a snapshot (copy-on-write duplicate of a file),
- managing user quotas.

In order to access files, client contacts master for vital metadata information, and, if need be, requests data directly from chunkservers. Figures A.1, A.2, A.3 included in appendix A show how data and metadata flows between servers on I/O operations.

Client-side extra features are created for convenience of LizardFS administrators. Various system statistics can be viewed via web browser thanks to CGI server. Maintenance information can also be extracted directly from a command-line tool.

1.1.2. Resource consumption

Components of LizardFS have different needs in terms of resources offered by the hardware.

Master server is both RAM and CPU sensitive. It occupies large amounts of memory, since all system metadata is stored there. Processor time is spent on serving requests, recalculating various statistics and deciding whether chunks present in the system need to be replicated/checked/deleted/moved. It is highly recommended to reserve a dedicated physical machine for master server to run on, if possible.

Chunkserver is constrained by I/O throughput. It extensively performs read/write operations on storage. It is discouraged to share disks between chunkservers and other processes.

Mount and other client-side features are not considered resource-demanding.

In this thesis, attention will be focused on reducing master server's dependence on RAM capacity, bearing in mind, that it must not be done at the expense of very high CPU overhead.

1.1.3. Memory of the master server

Metadata kept in master server is internally held as directory tree. Information about each file is kept in its inode, represented by *fsnode* structure, while paths (with semantics similar to dentries) are stored in *fsedge* structure.

The *fsnode* contains various details of every file, especially:

- inode id,
- creation time, access time, modification time,
- ids of owner and group,
- goal,
- parent fsedge,
- child fsedges (if file is a directory).

Structure's overall size can be rounded to 128 B.

Size of the *fsedge* structure is hard to predict, since its main component is file name, which is variable in size.

One more structure that highly affects memory usage is *struct chunk*, which holds metadata of each data chunk. Mentioned metadata consists of chunk's location, its safety policies and number of replicas present in the system.

All three structures are kept in hash tables. They create the most heavy pressure on memory usage, since their number increases linearly as more files appear in the installation.

In order to visualize importance of these structures, let us see in table 1.1 how much RAM an installation would need to store *fsnodes* alone and all file-related structures.

№ of files	*fsnode* overhead	all structures overhead
1	128 B	500B
1 000	128 KB	500 KB
1 000 000	128 MB	500 MB
100 000 000	12 GB	50 GB
1 000 000 000	128 GB	500 GB

Table 1.1: Average memory overhead of the master server

Overall RAM usage for LizardFS is 500 B per average file. The extra usage is caused by several factors, for example:

- data layer — i.e. chunks belonging to a file,

- internal helper structures of LizardFS (caches, temporary buffers),
- additional file properties — extended attributes (*xattr*), access control lists (*ACL*).

It must be clarified, that statistics above apply to an average file on an average installation. It is absolutely possible to drain a significant amount of system resources with only few files. For instance, the assumed *average file* has 1 name, which is about 16 characters long. Extended attributes and access control lists are considered rare.

A good way to observe memory usage of a process is to create and browse its heap profile. Heap profile is a visual representation of used memory with various statistics — the most important one being percentage of heap used per function call. Profiling a heap requires gathering internal allocation data of a process and is usually connected with overriding library calls — *malloc*, *mmap*, *brk*, etc. As it happens, Google perf tools package [9] provides an efficient and handy heap profiler along with its flagship product — tcmalloc allocation library [19]. Results of a profiling session executed on running LizardFS master server are presented in appendix A.4. For clarification, most resource-consuming symbols are explained below.

_ZN6fsnodeC4Eh is a symbol for fsnode class constructor. It was responsible for claiming the biggest part of the heap, so it is clearly the best candidate for optimization.

get16bit is a macro used for extracting 2-bytes values from binary files and network. In this case, it was used for loading edges (i.e. file names) from disk.

chunk_malloc is used for allocating memory for chunk structure.

fs_loadnode is a function that handles reading fsnode definitions from binary files and deserializing them into objects that reside in RAM.

fs_loadedge behaves as above, but it handles file names instead of fsnode definitions.

chunk_new is a constructor of chunk structure.

From description above it can be safely concluded, that removing overhead exerted by fsnode, fsedge and chunk structures from LizardFS master server's memory will result in bypassing its current RAM boundaries.

1.1.4. Existent memory optimizations

LizardFS implements several useful mechanisms for memory saving:

Size-aware types are used — *int8_t*, *int16_t*, *uint32_t*, etc. instead of regular *int*, *long*, *unsigned long long*. This issue becomes vital when quantity of objects present in the system reaches millions.
Example in listing 1.1 shows a casual definition of structure A and its memory-aware version — *struct compactA*. While the first one can occupy, depending on the architecture, 32 bytes, its compact variant occupies exactly 4 bytes. Beyond size-aware types, specifying bitfields

(*uint8_t is_valid : 1*) can pack values smaller than 1 byte together in order to save even more space.

Listing 1.1: Example of size-aware typed structure

```
struct A {
        long long id;
        int order;
        int flags;
        int is_valid;
        int has_acl;
        int has_xattr;
        int is_redundant;
}

struct compact_A {
        uint16_t id;
        uint8_t order;
        uint8_t flags : 4;
        uint8_t is_valid : 1;
        uint8_t has_acl : 1;
        uint8_t has_xattr : 1;
        uint8_t is_redundant : 1;
}
```

Fields in structures are ordered in a way that minimizes padding. If compact structure shown in 1.1 would have been ordered like in listing 1.2, it would have occupied 6 bytes, because, due to padding rules in C/C++, compiler would insert at least 1 empty byte between *order* and *id* fields, in order to ensure that 2 byte *id* field is properly aligned. Additionally, a trailing 1 byte field would be added, because the whole structure is required to be aligned as strictly as its most strict member. It is also noteworthy, that bitfields specified inside the structure must be listed consecutively or they would not work as desired. Example struct presented in listing 1.3 would occupy at least 8 bytes.

Listing 1.2: Example of field ordering in a structure

```
struct wrong_compactA {
        uint8_t order;
        // char transparent_padding_after_order[1];
        uint16_t id;
        uint8_t flags : 4;
        uint8_t is_valid : 1;
        uint8_t has_acl : 1;
        uint8_t has_xattr : 1;
        uint8_t is_redundant : 1;
        // char transparent_trailing_padding[1];
}
```

Listing 1.3: Example of padding

```
struct wrong_compactA {
        uint8_t flags : 4;
        // bitfield supplemented to 8 bits
        uint8_t order;
        uint8_t is_valid : 1;
        // bitfield supplemented to 8 bits
        uint16_t id;
        uint8_t has_acl : 1;
        uint8_t has_xattr : 1;
        uint8_t is_redundant : 1;
        // bitfield supplemented to 8 bits
}
```

Additional properties that are relatively rare, are often kept in hash tables instead of being inlined into every object. This is a very important design decision. If some properties of a class apply to the minority of its objects, keeping a pointer to nearly always empty list of these properties in every instance is a waste of space. What can be done instead, is to store these properties in an external dictionary-like container, e.g. hash map. Then, with a reasonably low CPU overhead for edge cases, total size of each structure could be reduced.

Smart data structures can lower unneeded memory consumption as well. Standard containers provided by STL (standard template library) often provide more functionalities than necessary at the cost of bigger size. Instead of using popular *std::vector* (24 bytes in size for x86/64 architecture), it is often better to directly allocate a memory region for object storage — reducing RAM overhead to address size (8 bytes on 64 bit architectures) per instance.

One of notable examples of a smart data structure is *flat set*, or its more specific version — *flat map*. Traditionally, ordered sets are implemented on the basis of balanced trees. Flat ordered set, on the other hand, stores all its data in a continuous array. Naturally, choice of underneath data structure affects the pessimistic time complexity of set operations, listed in table 1.1.

Operation	tree-based set	flat set
find	$O(\log n)$	$O(\log n)$
insert	$O(\log n)$	$O(n)$
remove	$O(\log n)$	$O(n)$

Figure 1.1: Complexity for selected dictionary operations

Linear complexity of insert/remove operations on a flat set is a result of potential need to shift nearly all array elements, in order to ensure it is still continuous after the process. Finding an element can still be achieved in $O(\log n)$ time by applying binary search.

There is one big advantage of using a flat structure, though. Specifically, it is a lack of need to use pointers. A perhaps surprising fact is that in some cases, switching to a flat structure could bring profits to both CPU and RAM load. First of all, continuous arrays, by definition, provide high locality for memory access. Heap allocation based tree nodes may not have this kind of feature. Secondly, space overhead per single element can be much smaller for flat structures. A real life example is shown in listing 1.4.

Listing 1.4: Example of set elements

```
// sizeof(struct A) = 1
struct A {
        char value;
}

// sizeof(struct tree_node) = 24
struct tree_node {
        struct A value;
        struct tree_node *lhs;
        struct tree_node *rhs;
}
```

In this simple code snippet, a structure occupies only 1 byte of space, while its tree node can occupy up to 24 bytes on 64 bit architecture — 16 because of two addresses, 1 for structure itself and (wasted) 7 bytes of padding to ensure proper alignment. It results in twenty-four-fold overhead for storing an ordered set of characters.

External allocation libraries like *tcmalloc* [19], *jemalloc* [11] can be applied to an existing project. Most of them are based on the idea of per thread memory pools, in some ways similar to slab allocator present in the Linux kernel. LizardFS offers an option to compile file system binaries already linked to *tcmalloc* library, turned off by default. Some allocation libraries can be preloaded in runtime, which can be achieved on Linux by setting *LD_PRELOAD* environment variable to appropriate path before executing a program. All available allocation libraries have their strong supporters and strong opponents, but the only way to see if it is applicable for a project is to apply and test it thoroughly in terms of performance.

All methods above share one big disadvantage — they only slightly delay the moment when master server runs out of memory.

One way of dealing with high memory usage is to move responsibility for storing metadata to a database residing on persistent storage. It can be achieved by rewriting fair amount of

code in order to adapt the whole project to the new logic. The biggest disadvantage of this solution is time — spent first on performing all the changes, and then on full regression tests, since majority of system's code was rewritten. An interesting alternative method is presented in next chapters of this thesis.

Chapter 2

Inspiration

The source of inspiration to create a new memory management library is a work of researchers from Princeton University — SSDAlloc project [1, 3].

2.1. Introduction to SSDAlloc

SSDAlloc was designed to transparently extend random-access memory with slower, but more capacious solid-state drives. The interesting aspect was to minimize the amount of code that should be rewritten in order for the user application to work with new allocation system. SSD drives have irrefutable advantage of being cheaper than RAM. With 1 terabyte storage available for about $500, random-access memory could not be a competition. What is more, modern SSD's are statistically only 4 times slower than RAM, which makes them perfectly usable even in highly responsive applications.

2.1.1. Motivation

SSDAlloc targeted the popular need of performing a significant change to a big project without a necessity to rewrite large amounts of code. Simplified table 2.1 shows how SSDAlloc addresses this problem. Full chart can be found in publicly available documents regarding SSDAlloc project [2].

Solution	high performance	persistence	programming ease
Total application rewrite	✓	✓	✗
Swapping to SSD	✗	✗	✓
SSDAlloc	✓	✓	✓

Table 2.1: Simplified comparison of SSDAlloc and its alternatives

2.1.2. Keywords

- OPP — object-per-page
  policy to store only 1 object per 1 system page, even if it is significantly smaller than the page size
- RAM Object Cache
  frequently used allocated objects stored in random-access memory

- Page Buffer
  a first-in first-out style queue of pages currently mapped into RAM

2.1.3. Architecture

SSDAlloc works on three distinctive levels: application, runtime and storage.

Application level
The main goal of application level design was to be as transparent to the user as possible. It was vital to ensure that memory allocated via SSDAlloc was similar to the one obtained by *malloc/new*. Particularly, virtual addresses of SSD-allocated pages should be immutable.

The requirements were successfully achieved by implementing a simple user-space memory manager.

Runtime level
Internally, SSDAlloc uses 2 important data structures: RAM Object Cache and Page Buffer.

RAM Object Cache is used for storing frequently used objects and uses LRU algorithm for enqueuing them. It also distinguishes clean and dirty objects. By a general rule, cached object is clean until its data is modified in any manner. Thanks to that mechanism, only dirty objects need to be written back to storage, which can dramatically decrease the I/O load. This feature is especially useful when working with solid-state drives, because each unneeded write wears them down.

Page Buffer is a container of active mapped pages. These pages are actually used by an application to access allocated objects. Simple FIFO algorithm proven to be enough to ensure sufficient performance. Practical experiments have shown that Page Buffer capable of storing 8192 pages did not create a bottleneck either.

Page manager is responsible for serving memory allocation requests in user-space. Its simplified workflow is the following.

1. Map a virtual page.
2. Protect a page from reading/writing.
3. Accessing memory will trigger a segmentation fault signal handler.
4. Run a custom handler.
5. Load needed data into memory in the handler.
6. Once page is not needed, free the page frame.

Mapping an anonymous page will reserve a virtual address, without using any physical memory yet. Only after page is filled with data from solid-state drive, it actually occupies physical space. When page is no longer needed, Linux kernel can be advised to treat a page as *not needed*. Linux behaviour on getting such advice is to immediately remove the page from cache, so it no longer consumes resources.

There are potential problems to be considered with user-space memory management and they will be addressed in subsection 3.4.1. Fortunately, many implementations of operating systems based on UNIX (including Linux) are capable of dealing with these problems.

Storage level
Storage layer consists of 2 parts: address translator and log-structured object store.

Address translator is a mechanism of mapping virtual memory addresses to data location on solid-state drive.

Log-structured object store is responsible for storing data efficiently on SSD block device.

2.2. Conclusions

Overall architecture of SSDAlloc is very inspirational. User-space memory management proved to be very useful in creating an allocation library. Runtime system keeping pages cached and distinction between clean and dirty ones proved to be helpful as well. Another big advantage is a simple, natural division of layers: application, runtime and storage.

As a result, *fsalloc*'s high-level design is in many ways similar to SSDAlloc.

Chapter 3

Design and implementation

In order to evaluate the potential in allocation strategies presented by SSDAlloc, *fsalloc* — an open source allocation library — was implemented. The purpose of creating *fsalloc* was to provide means of moving the responsibility of storing objects from RAM to other types of storage, such as solid-state drives, hard drives and compressed memory. Emphasis was placed on minimizing the invasiveness of the library, which can be measured by lines of code needed to be rewritten in target project in order to make it compatible with *fsalloc*.

3.1. Design

The library can be seen as three closely cooperating layers: application, runtime and storage.

Application layer is responsible for providing an interface to end-user. In case of *fsalloc*, end-user can be regarded as a programmer willing to use alternative memory allocation in his/her project. Specifically, *fsalloc*'s application layer is a set of C/C++ functions and it is thoroughly described in subsection 3.5.1.

Runtime layer represents a core part of *fsalloc* library. It is responsible for providing a way of overriding memory access routines, maintaining cache of most frequently used objects and gathering statistics. Its details are presented in subsection 3.5.2.

Storage layer provides means of keeping data in persistent and efficient way. In *fsalloc*, all objects that need to be written back from cache are processed by storage layer. Detailed description can be found in subsection 3.5.3.

3.2. Choice of programming language

One of the decisions to be made before implementing a library is the one regarding used programming language. Language chosen for the implementation of *fsalloc* is C++, with a strong emphasis on the fact, that it could be easily ported to C.

Reasons why C/C++ are preferred languages for creating such library are as follows:

1. *fsalloc* is defined as a library for C/C++ programming,
2. C is fast,
3. C offers direct interface for system calls, which is a core functionality for this project,

4. with C, creating a complex, architecture-specific signal handler is easy,

5. with C, registering a signal handler is easy,

6. C allows direct operations on memory,

7. C++ is excellently compatible with C,

8. C++ has templates, which makes generic programming type-aware,

9. C++ has standard template library with efficient implementations of frequently used data structures and algorithms.

No other languages were seriously considered as candidates for *fsalloc* implementation.

3.3. Used libraries and portability

fsalloc does not depend on many external libraries. Although some interesting data structures can be found in Boost [7], *fsalloc*'s requirements were simple enough to avoid creating additional external dependencies. There are 2 aspects of portability that must be mentioned, though.

3.3.1. Linux-specific headers

Memory management is both system and architecture-specific. As such, it uses many Linux-specific header files, which makes it not easily portable to other platforms.

sys/mman.h is needed for architecture-specific *mprotect* flags and structures used by signal handlers. On x86/64, which was chosen as a representative architecture for *fsalloc* implementation, it is possible to extract useful pieces of information from data passed to segmentation fault signal handler.

Function signature passed to sigaction call used for registering a signal handler has a third parameter, which is architecture-dependent. Code snippet 3.1 shows how to find out if memory access that triggered *SIGSEGV* was a read or write attempt on *x86/64* processors. Without that information, it would not be possible to distinguish dirty memory regions (i.e. with different contents in cache than on persistent storage) from clean ones. Hence, all cache entries, even the ones that were never accessed, would need to be synchronized with storage — which is a severe performance issue, especially if the storage happens to be a magnetic disk.

Listing 3.1: Checking access type in segmentation fault handler

```
int get_mprotect_flags(void *ctx) {
        int flags;
        ucontext_t *context = (ucontext_t *)ctx;

        if (context->uc_mcontext.gregs[REG_ERR] & 0x2) {
                flags = PROT_READ | PROT_WRITE;
        } else {
                flags = PROT_READ;
        }

        return flags;
}

void handler(int signum, siginfo_t *info, void *ctx) {
        ...
        int flags = get_mprotect_flags(ctx);
        ...
}
```

unistd.h contains POSIX interface for system calls. Since implementation is definitely Linux-specific, portability of the library is considered only within POSIX standards.

3.3.2. BerkeleyDB

fsalloc enjoys the benefits of using open-source database library — Berkeley DB [6]. It is used as a primary storage back end in the project. Details are thoroughly described in section 3.5.2.

3.4. User-space memory management

The core part of *fsalloc* library is handling the allocation process. One way of interfering with low level memory management is to write kernel-space code. Page allocation routines can, however, be pulled to user space with the help of signal handlers.

3.4.1. Details

Implementation of user-space memory manager is based on a schema described in section 2.1.3. Steps below illustrate how to achieve custom memory access routines. Each line has a corresponding description.

```
1. sigaction(SIGSEGV, &sa, &default_sigsegv);
2. addr = mmap(NULL, size, PROT_NONE,
                MAP_ANONYMOUS | MAP_PRIVATE, -1, 0);
3. x = addr;
4. mprotect(addr, size, PROT_READ | PROT_WRITE);
5. memcpy(addr, tmp, size);
```

```
6. mprotect(addr, size, PROT_READ);
7. madvise(addr, size, MADV_DONTNEED);
```

1. Register a custom signal handler for segmentation fault.
   The handler should check if a fault was caused by a regular error or by the protection set by *fsalloc*. It should only proceed with pages that are a part of *fsalloc* pool.

2. Map an anonymous page, obtaining its virtual address.
   Anonymous mapping is very similar to a *malloc* call. It returns a contiguous memory area consisting of at least 1 whole page. Mapped page does not occupy space until something is actually written to it.

3. Access the address, which triggers custom SIGSEGV handler.
   On x86/64 architecture it is possible to distinguish if a fault was triggered by a read or write request. It allows a huge performance optimization based on writing back only dirty (modified) pages to storage, evicting the clean ones without any I/O operations.

4. Unprotect a page, so it could be written to.
   The page needs to be unprotected or it would cause another segmentation fault, which leads to a livelock.

5. Copy data from storage to the page.
   The exact procedure of extracting data from storage will be discussed in 3.5.2.

6. (optional) Reprotect page if it was accessed only for reading.
   This is another architecture-dependent feature, fortunately available in x86/64. Page accessed only for reading remains clean, so it should be still protected from write attempts. That way, pages will be marked dirty if and only if a write occurs.

7. Remove page from system cache.
   A *madvise* call will trigger Linux kernel's aggressive policy of removing the page from cache right away, which is very helpful for working with anonymous mappings — it stops occupying physical space as soon as the *madvise* call ends.

There are two potential problems to be considered with user-space memory management, already mentioned in subsection 2.1.3. Firstly, only a small subset of functions can be safely used in signal handlers. These functions need to be reentrant, which according to the standard, is to meet the following conditions.

Reentrant function

- must hold no static (or global) non-constant data,
- must not return the address to static (or global) non-constant data,
- must work only on the data provided to it by the caller,
- must not rely on locks to singleton resources,
- must not modify its own code (unless executing in its own unique thread storage),
- must not call non-reentrant computer programs or routines.

Though *mprotect* is not on a reentrant-safe list according to POSIX standard, being a system call, meets the required conditions in Linux.

Second issue is to ensure, that signal handler does not use any global/static variables, which is not the case in *fsalloc*. However, if implementation guarantees, that access to these variables is never simultaneous, everything would work fine.

The strict rules of reentrancy might be in conflict with handlers which use static data structures, but, in *fsalloc*, it is assured that operations on such structures will never be interrupted by another handler of the same family. It is the case because *SIGSEGV* is a synchronous signal in Linux — it is guaranteed that its handler will be executed right after memory access fault.

Given these facts, *fsalloc* can be safely treated as reentrant friendly, as long as it is used as allocation library in single-threaded projects or ones that ensure thread safety on their own. Current implementation of LizardFS's master server meets these conditions.

3.4.2. Performance of signal-based management

One might wonder if memory management based on overloading signal handlers is efficient. Having a closer look at implementation details of signal handling in *x86/64* Linux should dispel any doubts. This subsection is based on an article from Linux Journal [4].

Signals are used to notify the process/thread of a certain event. They can be divided into two groups.

Synchronous signals occur as a direct result of instruction stream, e.g. trying to perform an illegal operation. *SIGSEGV*, a result of illegal memory access, is a good example of a synchronous signal.

Asynchronous signals occur independently to the process' instruction stream. A natural example of asynchronous signal is *SIGKILL*, sent to a process in order to force its termination.

Each task structure which represents a process in Linux kernel keeps certain information related to signal handling. The most important ones are:

- table of signal handlers,
- bitmask of blocked signals,
- signal queue, implemented as a double-linked list.

Handling a synchronous signal is a detailed process, simplified example of which (for *x86/64* architecture) is presented below.

1. User attempts to write to a read-only page.
2. Page fault handler (source: kernel/entry.S) is executed.
3. entry.S calls do_page_fault (source: arch/x86/mm/fault.c), which extracts hardware-specific fault parameters and does basic range validation.

4. do_page_fault calls generic mm_fault (source: mm/memory.c), page information is gathered from page table and further checks occur.
5. Signal is forced to be the first one by force_sig(*SIGSEGV*, current) call.
6. Current signal — previously forced *SIGSEGV* — is queued for execution.
7. All user-space registers are saved, user stack is modified to jump back to kernel after handling is finished (since the handing itself will be performed in user space).
8. Kernel returns to user space.
9. Execution jumps to registered signal handler.
10. Control is given back to the kernel by sys_sigret call, previously enqueued for execution on user stack.
11. User-space data is restored.
12. Control goes back to user space.

Despite switching from user to kernel-space and back, described procedure consists of relatively simple operations. Thanks to that, total overhead of handling a segmentation fault signal ranges from 3 to 4 microseconds [4]. For comparison, latency for reading 4KB from solid-state drive can reach over 100 microseconds.

Given these facts, it can be stated that signal handling time overhead is negligible compared to latencies imposed by SSD storage, not to mention magnetic hard drives.

3.5. Implementation details

Implementation details of each layer (application, runtime, storage) are described in this section.

3.5.1. Application layer

fsalloc offers several interfaces for accessing its functionalities. All definitions are included in header file <**fsalloc.h**>.

fsalloc::init()
Initialization function. Initializing *fsalloc* module is required before using its functionalities. Init function takes two parameters:

- path to database
  Full path to a place where database file should be stored. It is advised to pick a place that provides enough memory for storage. If the underneath device is an SSD, it will provide greater performance than casual hard disk drive. A detailed comparison of storage variants is presented in chapter 4.
- size of page cache
  Amount of pages constantly kept in random-access memory. This value is a compromise between performance (high values equals more cache hits) and memory usage (low values consume less resources). Default capacity of page cache should be selected on the basis of practical experiments.

Example:

```
void system_init() {
        ...
        fsalloc::init("/mnt/zram/fsalloc.bdb", 2048U);
        ...
}
```

fsalloc::term()
Termination function. Terminating *fsalloc* module should be done on system exit in order to keep structures consistent and avoid memory leaks. This function does not need any parameters, because it is responsible only for triggering database shutdown.
Example:

```
void system_exit() {
        ...
        fsalloc::term();
        ...
}
```

fsalloc::fsalloc()
Basic function with *malloc*-like interface. It takes one numerical parameter, which is a number of requested bytes. Return value is, as in *malloc* call, of type *void **.
Example:

```
Object *obj;
void *memory = fsalloc::fsalloc(sizeof(Object));
obj = (Object *)memory;
obj.process(data);
```

fsalloc::fsfree()
Basic function with *free*-like interface. It takes one argument, which is a memory address previously allocated by *fsalloc*. It does not return anything, same as *free* call.
Example:

```
Object *tmp = (Object *)fsalloc::fsalloc(sizeof(Object));
tmp.process(data);
fsalloc::fsfree((void *)tmp);
```

fsalloc::fsalloc<T>()
Type-aware version of *fsalloc()* call. It is semantically equivalent to calling *fsalloc::fsalloc* function with size of templated type passed to it and returns a pointer already casted to appropriate type.
Example:

```
using namespace fsalloc;
int *i = fsalloc<int>();
struct entry *e = fsalloc<struct entry>();
```

fsalloc::fsfree<T>()
Type-aware version of *fsfree()* call. It is semantically equivalent to calling *fsalloc::fsfree* function.
Example:

```
using namespace fsalloc;
fsfree<int>(i);
fsfree<struct entry>(e);
```

fsalloc::fsnew()
C++-style variation of *new* statement. It allows custom constructor parameters to be passed to it.
Example:

```
using namespace fsalloc;
int *i = fsnew<i>();
std::string s
    = fsnew<std::string>("custom_constructor_parameter",
                                0, 6);
std::vector<int> x = fsnew<std::vector<int>>(5);
```

fsalloc::fsdelete()
C++-style variation of *delete* statement. Template parameters can be omitted, as they are deducible from arguments.
Example:

```
fsdelete<int>(i);
fsdelete<>(s);
fsdelete<>(x);
```

struct fsalloc::managed
Convenient class designed to be a base for other structures that should use *fsalloc* as their allocation method. *fsalloc::managed* class has overridden *new/delete* operators, which causes structures inheriting from it to use *fsalloc* as allocator.
Example:

```
struct foo : public fsalloc::managed {
        int x;
        int y;
        int z;
        char name[256];
        struct foo *next;
        struct foo *prev;
};
```

3.5.2. Runtime layer

Runtime engine for *fsalloc* consists of the following elements described in this section.

Region cache
A first-in first-out container of addresses. A *memory region* is defined as a contiguous virtual memory block composed of at least 1 page. Maximal capacity of this structure is specified by a parameter passed to initialization function. Once a memory region is allocated and accessed, it is inserted into region cache. It resides in there until other regions are accessed often enough to push it out of cache.

If region is clear, i.e. it was not accessed for writing, it does not need to be written back, as there are no differences between its contents in RAM and in the database.

If region is dirty (accessed for writing at least once), it needs to be written back to the database before eviction, as its state changed.

After successful writeback, given region is freed from memory with *madvise()* system call.

BerkeleyDB driver
Database access point.

BerkeleyDB offers various methods of storing objects.

1. Btree — where data is kept in a sorted, balanced tree structure.
2. Hash — which uses dynamic hash table algorithms
3. Heap — which stores objects in a file and references them only by their offset.
4. Queue — designed for keeping fixed-size records and fast tail inserts.
5. Recno — which stores both fixed and variable-length records.

Heap method was chosen as the best suited for *fsalloc* needs. It works similarly to memory allocation — a key for database entry is returned once the item is inserted. Structure used as key in heap method is of type **DB_HEAP_RID**, which is 8 bytes long (conveniently, just like a regular pointer on 64 bit architectures). Inside, it contains a 32 bit page number and 32 bit offset. Passing this 8 byte value is enough to extract the desired item from the database.

Heap storage works exceptionally well when inserted items have fixed size, which is fortunately a case in most important structures of LizardFS. If sizes do not vary much, internal database fragmentation is at a negligible level.

Database initialization happens during *fsalloc* module start-up. *db::init()* call invokes creating database file and setting up necessary parameters:

- size of item cache, needed for creating BerkeleyDB internal entry caching,
- type of storage (in this case, heap method).

Once an *fsalloc* allocation occurs, a page is mapped from virtual memory. Then, if page wasn't modified, no database access is required. Otherwise, data extracted from page is stored in BerkeleyDB with a call to *db::put()*. Overwriting an existing element is, of course, possible as well.

Pages are extracted from database with a *db::get()* call.

During *fsalloc* termination procedures, *db::term()* invokes closing a database, with a special *"NO SYNCHRONIZATION"* flag. Passing the flag accelerates the process, since data kept in this database is dynamic, so it would be overwritten on another module initialization anyway.

3.5.3. Storage layer

BerkeleyDB stores its database in a file. Choosing a path for this file is a crucial performance and safety oriented decision. Underlying file system should ensure that:

- database file does not get corrupted easily,
- access to the file contents is as swift as possible.

According to the guidelines, any POSIX compliant file system should be sufficient to serve a BerkeleyDB file, but finding a solution that provides high performance and reasonable level of safety is a complex task.

Chapter 4 covers the topic of comparing possible ways of storing allocation database.

Chapter 4

Choice of storage

Where to store data allocated with *fsalloc* library? Answer to this question is not simple and it depends on particular needs of the application. Typical hard drives do not offer high I/O speed, but, on the other hand, they are relatively cheap. SSD drives are clearly a better choice if performance is the main issue, but they cost more and wear faster than HDD. An interesting idea is to take advantage of in-kernel memory compression mechanisms included in Linux.

Table 4.1 shows a brief summary of possible means of database storage, taking into account three important features: speed, price and efficiency in saving memory.

Type of storage	fast	cheap	memory efficient*
HDD	✗	✓	✓
SSD	✓	✗	✓
ZRAM	✓	✓	✗

*meaning that RAM overhead is constant and insignificantly low

Table 4.1: Qualities of chosen storage types

Let us consider advantages and disadvantages of specific types of storage in detail.

4.1. HDD

4.1.1. Introduction

Hard disk drives use rapidly rotating magnetic platters to store and serve data. *IBM 350* is believed to be a first hard drive and was introduced in 1956 as a part of *IBM 350 RAMAC* computer. *IBM 350* was able to store 3.75 MB of data on 50 disks and was reported to have the size of two refrigerators.

Each write operation applied to a hard disk drive triggers magnetizing a ferromagnetic film placed on a platter. Reading from a HDD involves detecting changes in magnetization of a mentioned film. Input/output operations are performed by read-and-write heads, which are responsible for both translating magnetic field to electrical current (reads) and vice-versa (writes). In order to operate on a given memory sector, head must be mechanically moved to the region.

4.1.2. Access time

Due to its nature, which consists of storing data on constantly rotating disks and heads trying to place themselves above certain sectors, hard disk drives most costly factors are *seek time* and *latency*. Block I/O schedulers designed for hard drives apply various sophisticated heuristics to minimize these factors by trying to sort and merge individual read/write requests in order to reduce redundant disk head operations. Nonetheless, even hypothetical clairvoyant algorithm would not be able to eliminate the overhead caused by moving hardware components.

Seek time is the time needed to place an arm of read-and-write head above region where I/O operation should be performed. If two consecutive requests refer to physically distant regions, seek will be correspondingly slower. Due to this fact, sorting individual requests on the basis on their position is essential in I/O schedulers.

Latency is the time needed to rotate the platter to bring the requested sector under the head so it could be read from/written to. It is highly dependent on disk's rotational speed, expressed in *rpm* unit, which stands for *Revolutions per minute*. Example values from drives available on the market are 15 000 rpm, 7 200 rpm, 4 800 rpm.

Data transfer rate is a third access time factor worth mentioning. It determines how fast data can be read from the sector, once arms and platters are appropriately set.

Average values of described factors are presented in table 4.2.

Value	Seek time	Latency	Data transfer rate
low	4 ms	2 ms	200 MB/s
high	5 ms	5 ms	300 MB/s

Table 4.2: HDD features

4.1.3. Prices

Hard drives are considered to be cheap when it comes to price per storage capacity. An average 1 terabyte HDD can be acquired for about $50.

4.1.4. Benchmarks

Table 4.3 shows average speeds of selected operations, gathered from benchmark results from various disks of different vendors and prices [21]. All extracted values were associated with customer-grade disks.

Operation	Speed
read	150 MB/s
write	140 MB/s
mixed read/write	128 MB/s
4K read	1 MB/s
4K write	1 MB/s
4K mixed read/write	0.4 MB/s
	Peak speed
read	190 MB/s
write	180 MB/s
mixed read/write	160 MB/s
4K read	1.5 MB/s
4K write	2 MB/s
4K mixed read/write	0.6 MB/s

Table 4.3: HDD benchmark [21]

4.2. SSD

4.2.1. Introduction

Solid state drives use integrated circuits to provide persistent memory storage. As of present year, most SSD's are based on NAND-based flash memory, which is able to retain data without constant power source. NAND flash architecture was introduced by Toshiba in 1989 and its memory can be accessed similarly to other block devices, like HDD. For specific performance purposes, SSD can also be constructed from RAM, but then it might require a separate power source, e.g. a battery for ensuring data persistence.

In contrast to hard disks, SSD's do not rely on mechanical elements like platters and moving read/write heads. Aside from flash memory, solid-state drive contains a controller, which is an intermediary between the memory and host computer.
Controller's responsibilities may include:

- bad block mapping — needed to locate blocks that are unusable because they wore down or other combined error occurred,
- I/O caching — used for increased performance when subsequent memory access calls are local,
- garbage collection — needed to get rid of small amounts of freed data in block granularity,
- encryption — optional feature for security purposes,
- ECC — error-correcting codes, used for detecting memory corruptions,
- read scrubbing — repairing corrupted memory with the help of error-correcting codes.

4.2.2. Access time

Many performance bottlenecks of HDD's do not apply to solid-state drives. First of all, start-up time for SSD's is negligible, because virtually no physical components need to be prepared. For comparison, hard disks may need several seconds to reach their spinning speed and prepare the read/write head.

Seek time for an SSD is typically under 100ns, because its underneath memory already supports random access. No mechanical heads need to be moved in order to access the data.

Latency is not a concern for solid-state disk, as its value is similar to seek time and can be measured in nanoseconds.

4.2.3. Prices

A consumer-grade 1 terabyte solid-state drive can be obtained for about $300, which makes it around six times more expensive than a consumer-grade HDD.

4.2.4. Benchmarks

Benchmark table 4.4 corresponds to table 4.3.

Operation	**Speed**
read	480 MB/s
write	400 MB/s
mixed read/write	380 MB/s
4K read	36 MB/s
4K write	80 MB/s
4K mixed read/write	30 MB/s
	Peak speed
read	530 MB/s
write	512 MB/s
mixed read/write	512 MB/s
4K read	40 MB/s
4K write	140 MB/s
4K mixed read/write	50 MB/s

Table 4.4: SSD benchmark [21]

4.3. In-kernel compression

4.3.1. Introduction

In-kernel memory compression became popular as Linux was ported to smaller and smaller devices. These devices usually have small RAM capacity and sometimes they do not possess persistent memory at all. Such configurations make even the smartest swapping mechanisms useless — there is no place available to store the excessive pages. Compressed memory pools changed this unfortunate case. If pages that occupy memory are easily compressible (e.g.

they are filled with zeros, which is surprisingly often a case), they could be stored in these pools and save up to 100% of RAM. Experiments have shown that an average page could be compressed to approximately 50% of its original size. To provide a simple example, creating a memory pool of 256MB could potentially accept 512 MB of data, which could be a life saving amount. Linux developers provided in-kernel memory compression mechanisms for various use cases. Two main modules currently residing in the kernel tree are **zswap** and **zram**.

4.3.2. Implementations

zswap is a mediator between memory access and swap mechanism. A page that candidates for being swapped is checked by **zswap** first, and if it meets the conditions, it is put into compressed memory pool instead of being swapped to disk. Only pages with good compression ratio are allowed into the pool. Zswap is an interesting module, but it was not found useful for *fsalloc* storage back end, since it is designed specifically to intercept native swap mechanism in Linux.

zram is a block device that stores its data compressed in RAM. In order to work, zram module must be enabled during kernel compilation. Fortunately, most popular Linux distributions have this module enabled in their kernel by default.

4.3.3. Usage examples

Step by step guide for enabling **zram** module [12], e.g. as *fsalloc* storage:

- Turning on the module can be done with modprobe utility. Optionally, number of zram devices might be passed (default is 1).

```
$ modprobe zram num_devices=2
```

- Compression algorithm (discussed later) can be set by passing a value to *sysfs*.

```
$ echo lzo > /sys/block/zram0/comp_algorithm
$ echo lz4 > /sys/block/zram1/comp_algorithm
```

- Initialization is done by setting disk size to devices via *sysfs*. Note, that this amount of memory will actually be reserved in RAM.

```
$ echo 512M > /sys/block/zram0/disksize
$ echo 8G > /sys/block/zram1/disksize
```

- **zram** can be treated as a regular block device, so first thing to do is to create a file system on it.

```
$ mkswap /dev/zram0
$ mkfs.xfs -l logdev=/dev/sdh,size=65536b /dev/zram1
```

- After the file system is initialized, **zram** devices can be mounted or registered as swap

```
$ swapon /dev/zram0
$ mount -t xfs /dev/zram1 /mnt/storage
```

- Passing a path of mounted **zram** device to fsalloc initialization function will result in keeping its data still in RAM, but in compressed fashion.

```
fsalloc::init("/mnt/storage/fsalloc.bdb", 8192);
```

- Various **zram** statistics are available at *sysfs*.

```
$ cat /sys/block/zram0/orig_data_size /sys/block/zram0/compr_data_size
$ cat /sys/block/zram1/num_{reads,writes}
$ cat /sys/block/zram1/failed_{reads,writes}
```

4.3.4. Compression algorithms

In-kernel memory compression's performance depends almost exclusively on its underlying algorithm. For various reasons, the most popular algorithms for this specific use case are from the Lempel-Ziv family of dictionary compression — **lzo** and **lz4**. Compared to other algorithms, the mentioned ones are ahead of the rest when it comes to decompression time, mainly at the cost of compression ratio. Decompression time is the key factor because of the nature of random memory access itself — reads are considered to be the most frequent operation, and therefore overhead for these operations should be minimal.

lzo is a real-time data compression library, available both on open-source license and proprietary one. It was created and is maintained by Markus F. X. J. Oberhumer. It is written in C, and, as advertised, provides very high decompression speed.

lz4 is a fast, lossless compression algorithm written in C by Yann Collet, available on open-source license. Similarly to lzo, its top priority is to provide very high decompression speed.

4.3.5. Benchmarks

Tables 4.5, 4.6, 4.7, 4.1 show a comparison of various popular compression algorithms to **lzo** and **lz4** [17]. Tests were executed on a compressed Linux kernel source code in version 3.3 with file size of 466083840 B (445 MB).

Level	gzip	bzip2	lzma	lzma-e	xz	xz-e	lzo	lz4
1	26.8%	20.2%	18.4%	15.5%	18.4%	15.5%	35.6%	36.0%
2	25.5%	18.8%	17.5%	15.1%	17.5%	15.1%	35.6%	35.8%
3	24.7%	18.2%	17.1%	14.8%	17.1%	14.8%	35.6%	35.8%
5	22.0%	17.6%	14.9%	14.6%	14.9%	14.6%	-	35.8%
7	21.5%	17.2%	14.4%	14.3%	14.4%	14.3%	-	24.9%
9	21.4%	16.9%	14.1%	14.0%	14.1%	14.0%	-	24.6%

Table 4.5: Compression ratio. Source: Quick Benchmark: Gzip vs Bzip2 [17].

Level	gzip	bzip2	lzma	lzma-e	xz	xz-e	lzo	lz4
1	8.1s	58.3s	31.7s	4m37s	32.2s	4m40s	1.3s	1.6s
2	8.5s	58.4s	40.7s	4m49s	41.9s	4m53s	1.4s	1.6s
3	9.6s	59.1s	1m2s	4m36s	1m1s	4m39s	1.3s	1.5s
5	14s	1m1s	3m5s	5m	3m6s	4m53s	-	1.5s
7	21s	1m2s	4m14s	5m52s	4m13s	5m57s	-	35s
9	33s	1m3s	4m48s	6m40s	4m51s	6m40s	-	1m5s

Table 4.6: Compression time. Source: Quick Benchmark: Gzip vs Bzip2 [17].

Level	gzip	bzip2	lzma	lzma-e	xz	xz-e	lz4	lz4
1	3.5s	3.4s	6.7s	5.9s	7.2s	6.5s	0.4s	1.5s
2	3s	15.7	6.3s	5.6s	6.8s	6.3s	0.3s	1.4s
3	3.2s	15.9s	6s	5.6s	6.7s	6.2s	0.4s	1.4s
5	3.2s	16s	5.5s	5.4s	6.2s	6s	-	1.5s
7	3s	15s	5.3s	5.3s	5.9s	5.8s	-	1.3s
9	3s	15s	5s	5.1s	5.6s	5.6s	-	1.2s

Table 4.7: Decompression time. Source: Quick Benchmark: Gzip vs Bzip2 [17].

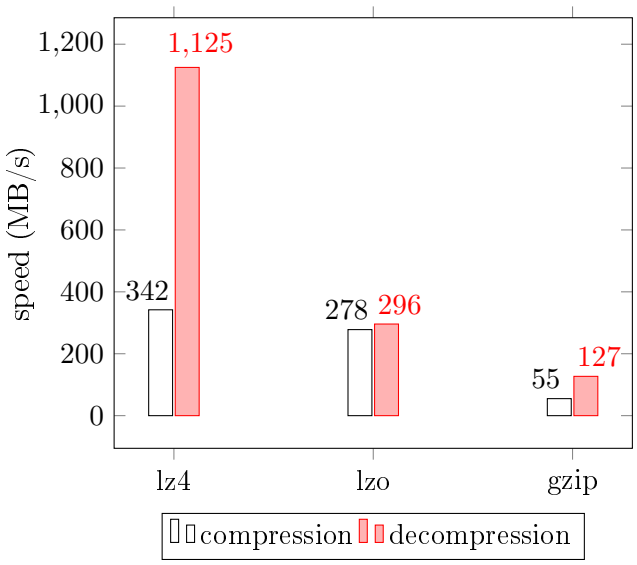

Figure 4.1: Compression/decompression speeds for lzo, lz4 and gzip algorithms

4.4. Comparison

There is no universal best solution among presented mechanisms. The choice should be based on performance requirements, budget and balance between these factors.

4.4.1. Price-oriented choice

Emphasis on low price rules out solutions based on high-end solid-state drives.

HDD There is one big advantage in choosing hard drives for allocation purposes — it is the most popular storage solution on the market. Because of that, creating a *JBOD (just a bunch of disks* configuration is usually free from additional expenses — existing hardware can be utilized.

ZRAM An interesting alternative for low price HDD is to use in-kernel memory compression mechanisms. It outperforms hard drives in access speed by orders of magnitude (see tables in section 4.3.4). There is a major disadvantage of using compression as storage — it only extends available RAM by about a half, but it is not scalable after that threshold. Moreover, it is often not possible to predict the compression ratio of data provided to compression algorithms — in a very pessimistic scenario, it can even reduce the amount of memory available in the system.

4.4.2. Performance-oriented choice

If a primary goal is to ensure high efficiency, using a storage solution based on HDD is a bad choice.

SSD access time is closer to RAM than HDD, which can be seen in section 4.2.4. There are two disadvantages to be taken into account when considering solid-state drives as a storage back end:

- cost of an SSD is higher than a regular HDD
- solid-state drives wear out much faster than HDD, especially when used inappropriately. Frequent writes to the same memory sector make it unusable because of flash memory properties. As a result, one must consider the lifespan of an average solid-state drive before choosing it as a basic storage for memory allocation.

ZRAM — in-kernel memory compression is not only an affordable solution, but also a high performance one. In uses that do not require decreasing memory overhead to a large extent it should be considered a legitimate candidate.

4.4.3. Summary

Charts 4.2-4.5 visualize selected important traits of storage back ends. Advantages and disadvantages of each particular solution is clearly visible on these charts – HDD is slow, SSD is fast, yet expensive and finally, zram is fast and cheap, but does not offer 100% memory save.

Performance

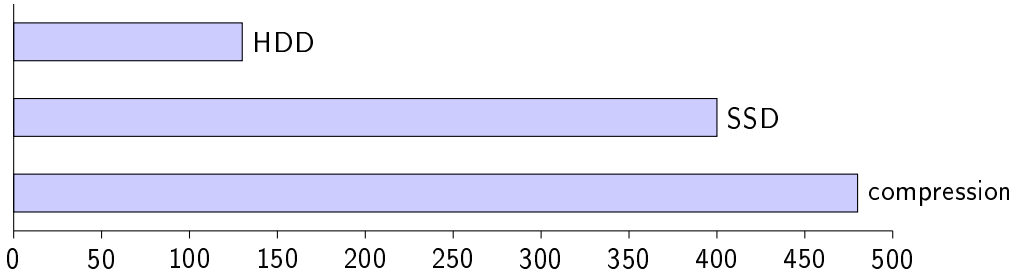

Figure 4.2: Expected read speed for selected storage back ends (MB/s)

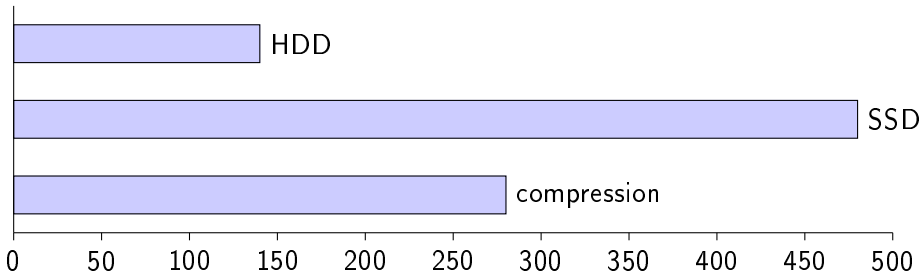

Figure 4.3: Expected write speed for selected storage back ends (MB/s)

Utility

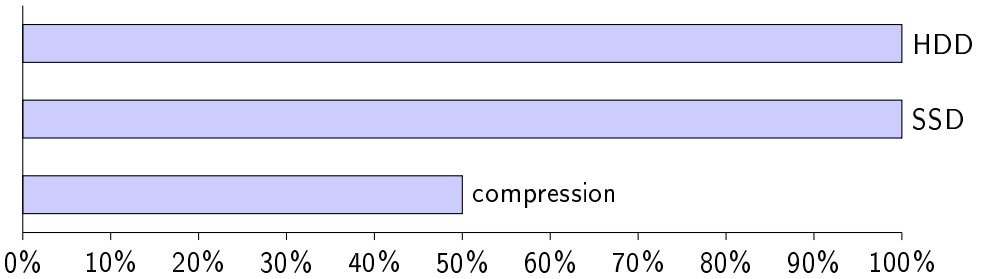

Figure 4.4: Expected reclaimed RAM capacity for selected storage back ends

Cost

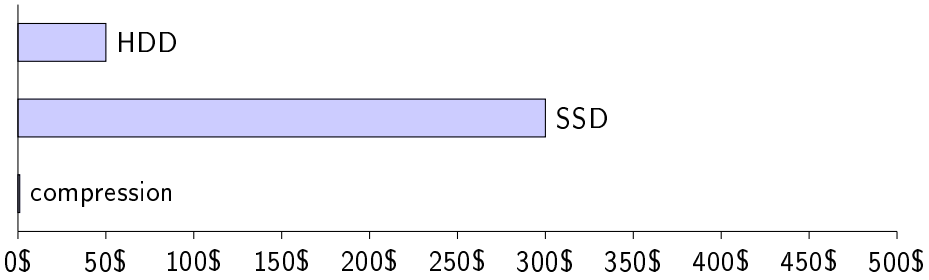

Figure 4.5: Average price per 1 TB for selected storage back ends

Conclusion

Of all back-end storage choices, HDD and in-kernel compression are the ones that could be sufficient for small scale use cases, while imposing little or no expenses. When performance and reliability are the key factors, solid state drives are probably the most fitting choice.

Chapter 5

Integration with LizardFS

By design, *fsalloc* is intended to be easily applicable to existing projects. Incorporating *fsalloc* into LizardFS required only a few lines of code. Actually, adding *fsalloc* library to automatic compilation system took more script lines than all performed code modifications. For simplicity, only fsnode structure allocation was chosen to be managed by *fsalloc* instead of native routines. The reasoning is that fsnode is responsible for majority of master server's overhead (ref: A.4), so memory reduction should be clearly visible in test results. On the other hand, should *fsalloc* prove to have bad impact on performance even if it was used to handle only a single structure, no further code changes and testing would be necessary.

5.1. Build configuration

LizardFS' auto building system is based on CMake [8] project. Adding *fsalloc* as a module of LizardFS requires preparing a simple configuration file — available on the attached CD.

5.2. Code changes

Required code modifications are so sparse, that they can all be presented and briefly described in this chapter. Changes are presented in *diff unified* format [20].

First of all, appropriate header (figure 5.1) needs to be included in source file. Then, class which represents file system's inode is extended with custom allocation provided by *fsalloc* (figure 5.2). Afterwards, loading of *fsalloc* structures is added to system's initialization routines (figure 5.3). Finally, library unload is added to system's termination routines. These steps are sufficient to integrate *fsalloc* allocation procedures into LizardFS. Essential changes involved adding 3 lines of code and amending only a single one, making it 4 lines in total.

Listing 5.1: Adding header

```
@@ -50,6 +50,7 @@
 #include "common/slogger.h"
 #include "common/tape_copies.h"
 #include "common/tape_copy_state.h"
+#include "fsalloc/fsalloc.h"
 #include "master/checksum.h"
 #include "master/chunks.h"
 #include "master/goal_config_loader.h"
```

Listing 5.2: Changes in fsnode class

```
@@ -182,7 +183,7 @@ struct xattr_inode_entry {
        struct xattr_inode_entry *next;
 };

-class fsnode {
+class fsnode : public fsalloc::managed {
 public:
        std::unique_ptr<ExtendedAcl> extendedAcl;
        std::unique_ptr<AccessControlList> defaultAcl;
```

Listing 5.3: Changes in initialization code

```
@@ -8270,6 +8272,7 @@ LIZARDFS_CREATE_EXCEPTION_CLASS
 char const MetadataStructureReadErrorMsg[]
        = "error reading metadata (structure)";

 void fs_loadall(const std::string& fname,int ignoreflag) {
+    fsalloc::init("/var/lib/lizardfs/fsalloc.bdb", 8192);
     cstream_t fd(fopen(fname.c_str(), "r"));
     std::string fnameWithPath;
     if (fname.front() == '/') {
```

Listing 5.4: Changes in termination code

```
@@ -8227,6 +8228,7 @@ void fs_term(void) {
             gMetadataLockfile->writeMessage("no_metadata: 0\n");
         }
     }
+fsalloc::term();
 }

 void fs_disable_metadata_dump_on_exit() {
```

5.3. Summary

One of the primary objectives set for *fsalloc* was to assure that integrating it with an existing project would be trivial. This goal was definitely achieved — the process of injecting *fsalloc* library into LizardFS was a matter of few technical amendments. This optimistic result makes *fsalloc* a great candidate for performing proof of concept checks of memory optimizations. Overriding native allocation routines with the ones offered by *fsalloc* can be done even without having deep knowledge of project's implementation. Moreover, required changes apply to allocation routines only, which makes them fully reversible — if test results suggest that original allocation procedures provided better performance, a full rollback is always possible.

In conclusion, the following requirements for the integration process have been successfully fulfilled:

- only few code changes were necessary (approx 0.013% of project size[1]),
- code changes were trivial,
- changes were fully reversible.

With *fsalloc* properly injected into LizardFS master server, it is possible to begin the most vital part of this thesis — performance tests.

[1] Assuming that LizardFS consists of around 30 000 lines of code and integration process required 4 lines to be amended.

Chapter 6

Test results

An important part of *fsalloc*'s implementation process involved creating and executing tests. Validation tests were used to identify bugs, regression tests ensured that the number of passed test cases never decreased after applying code modifications. Allocation libraries are implemented for one purpose, though — increasing performance. Therefore, this whole chapter is devoted to presenting results of extensive performance tests of *fsalloc*. Instead of preparing a set of isolated cases for every single operation, performance tests were conducted as a long-term examination of a real application using *fsalloc* routines for memory management.

Naturally, integrating LizardFS with *fsalloc* created a good opportunity to measure the allocation library's performance in practical environment. Using an exact replica of real life production installation of LizardFS cluster allowed to see if *fsalloc* implementation is a step in the right direction — or, conversely, it causes impassable bottlenecks and needs a thorough redesign.

Cluster consisted of a master server, shadow master and 3 chunkservers, all residing on dedicated machines. The installation kept files in various number of copies (from 1 to 3). A small portion of all files contained extended attributes and access control lists.

6.1. Parameters

Test scenarios are intended to show various statistics of master server with different allocation routines. To provide a fuller context, hardware parameters of the machine that hosted master server are listed in table 6.1. Table 6.2 shows more detailed view into LizardFS cluster configuration.

processor	Intel(R) Xeon(R) CPU E3-1226 v3 @ 3.30GHz
memory	4 x 8 GB RAM, DDR3, 1600 MHz
network	1Gbps Ethernet
hard disk drive	1 TB SATA 7200rpm
solid state drive	128 GB SATA

Table 6.1: Hardware parameters

number of files	18630347
RAM usage	8 GiB
shadow servers	1
metaloggers	1
chunkservers	3
clients	4
average file size	256KiB
average name length	32

Table 6.2: LizardFS cluster configuration

6.2. Configurations

The following configurations were tested:

- no fsalloc enabled,
- fsalloc with single stream in-kernel compression,
- fsalloc with multiple stream in-kernel compression,
- fsalloc with HDD,
- fsalloc with SSD.

Performance tests were carried on each configuration without amending any internal LizardFS options. The only variable factor was the storage back end for allocation library.

6.3. Scope

Performance tests were designed to provide information on two factors – memory usage and time spent on performing metadata operations.

Template result sheet for a single configuration contains the following data:

1. time spent on starting master and loading metadata to memory,
2. time spent on a set of metadata operations:
   - creating files,
   - adding extended attributes to files,
   - changing file owners,
   - attaching a replication goal to files,
   - reading file goals,
   - removing files,
3. CPU load on the machine during metadata operations,
4. I/O load on the machine during metadata operations.

6.4. Result sheets

6.4.1. without fsalloc

This result sheet presents the state of LizardFS cluster before supplying it with *fsalloc*. Table 6.3 plays the role of reference point for the rest of test sheets. Additionally, all results are combined into figure 6.9 in order to make the comparison more vivid. Memory and CPU usage charts for this case also include the moment of transition to a configuration with *fsalloc* enabled. First part of the chart presents original master state. Transition to *fsalloc* assisted version is identifiable as a gap in both memory and CPU charts. The break was caused by the need of restarting master server with preloaded allocation routines — substituting a memory manager without restarting a binary would be very troublesome for both programmers and clients.

Operation	# of files	value (s)
Master server restart time	N/A	58.614
Creating files	10 000	8.645
Creating files	100 000	134.737
Setting extended attributes	10 000	8.645
Setting extended attributes	100 000	85.662
Changing owner	10 000	10.187
Changing owner	100 000	101.393
Setting goal	10 000	26.932
Setting goal	100 000	249.210
Reading goal	10 000	26.410
Reading goal	100 000	243.189
Removing files	10 000	7.344
Removing files	100 000	127.701

Table 6.3: Metadata operations with fsalloc disabled

6.4.2. single stream zram

Table 6.4 contains results for the first in-kernel compressed memory configuration. Difference between single and multiple streams can be examined by comparing this result sheet with figure 6.5. Charts 6.1 and 6.2 visualize the moment of reloading master server with new memory allocation routines — around 18:00. A drop in used memory as well as a rise in CPU usage are clearly visible. The most noticeable difference is a nearly twentyfold increase in master server restart duration. This issue is addressed and explained in subsection 6.7.2.

Operation	# of files	value (s)
Master server restart time	N/A	1164.16
Creating files	10 000	9.625
Creating files	100 000	141.627
Setting extended attributes	10 000	8.015
Setting extended attributes	100 000	84.552
Changing owner	10 000	11.120
Changing owner	100 000	104.738
Setting goal	10 000	27.268
Setting goal	100 000	252.440
Reading goal	10 000	28.451
Reading goal	100 000	247.175
Removing files	10 000	8.123
Removing files	100 000	126.135

Table 6.4: Metadata operations with fsalloc enabled with single stream zram

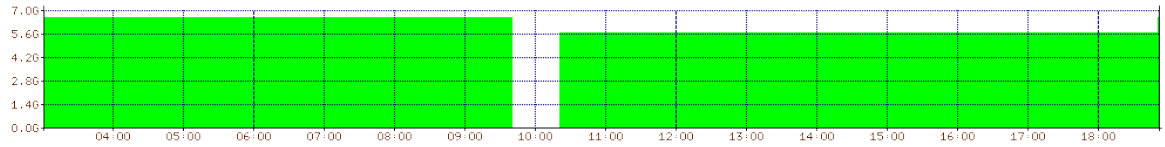

Figure 6.1: Memory chart representing transition
from plain configuration (no *fsalloc*) to 1 stream zram[1]

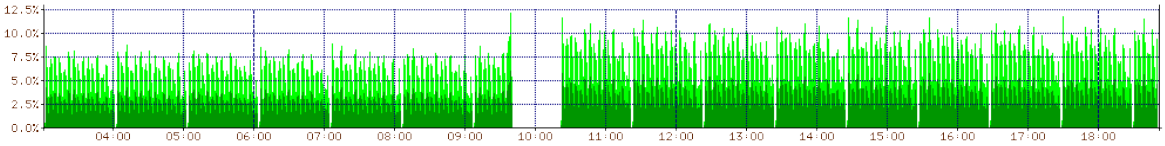

Figure 6.2: CPU chart representing transition
from plain configuration (no *fsalloc*) to 1 stream zram

6.4.3. multiple stream zram

Table 6.3 shows the results after modifying zram configuration to enable more streams to be compressed simultaneously. Charts 6.3 and 6.4 present the moment of switching from single to multiple stream configuration. A straightforward observation is that number of streams changed neither memory usage, nor CPU load. Master server start-up duration remained very high.

[1]Empty space indicates the moment of changing configuration, which requires restarting the master server.

Operation	# of files	value (s)
Master server restart time	N/A	1099.153
Creating files	10 000	8.401
Creating files	100 000	131.190
Setting extended attributes	10 000	8.225
Setting extended attributes	100 000	84.361
Changing owner	10 000	9.011
Changing owner	100 000	102.522
Setting goal	10 000	27.633
Setting goal	100 000	252.350
Reading goal	10 000	26.414
Reading goal	100 000	242.160
Removing files	10 000	6.994
Removing files	100 000	124.500

Table 6.5: Metadata operations with fsalloc enabled with 4 stream zram

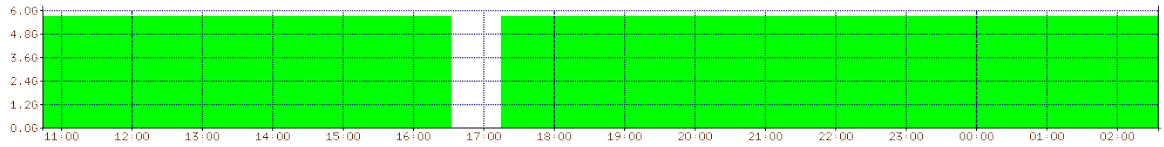

Figure 6.3: Memory chart representing transition
from single stream to 4 stream zram[2]

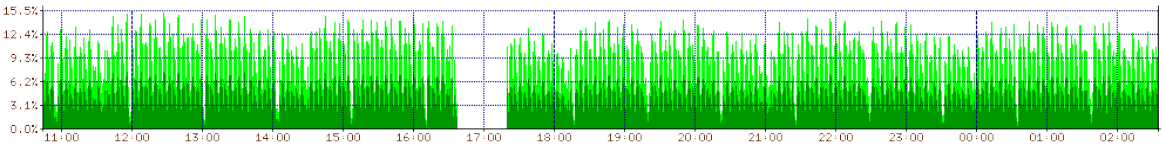

Figure 6.4: CPU chart representing transition
from single stream to 4 stream zram

6.4.4. HDD

Table 6.6 contains results for fsalloc with HDD as back-end storage. It can be observed that results for HDD, presumably the slowest of all chosen storage back ends, do not deviate from neither zram nor standard configurations. As previously, a slight increase in CPU load (figure 6.6) can be observed after restarting master server with *fsalloc* enabled.

[2]Empty space indicates the moment of changing configuration, which requires restarting the master server.

Operation	# of files	value (s)
Master server restart time	N/A	1182.090
Creating files	10 000	8.747
Creating files	100 000	141.260
Setting extended attributes	10 000	8.110
Setting extended attributes	100 000	83.442
Changing owner	10 000	9.187
Changing owner	100 000	101.000
Setting goal	10 000	29.003
Setting goal	100 000	248.225
Reading goal	10 000	27.217
Reading goal	100 000	242.155
Removing files	10 000	7.384
Removing files	100 000	128.770

Table 6.6: Metadata operations with fsalloc enabled with HDD

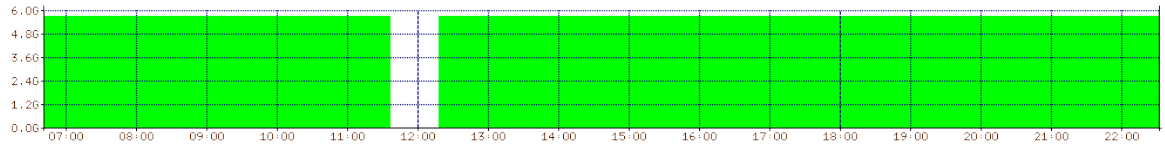

Figure 6.5: Memory chart representing transition
from zram configuration to HDD[3]

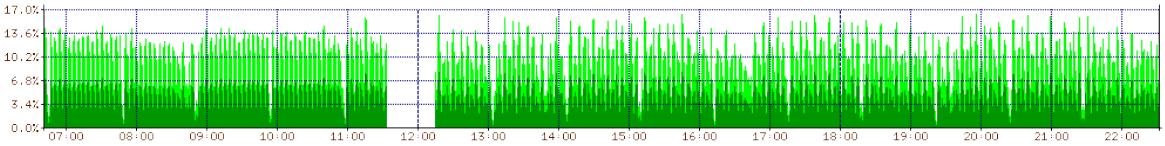

Figure 6.6: CPU chart representing transition
from zram configuration to HDD

6.4.5. SSD

Table 6.7 contains results for fsalloc with SSD as back-end storage. Surprisingly, the most costly storage back end does not seem to be worthy the money. Result sheet shows that using a solid-state drive did not influence performance metadata operations compared to slower and cheaper solutions, i.e. hard drives. As before, CPU overhead (figure 6.8) is observable, but not severely high.

[3]Empty space indicates the moment of changing configuration, which requires restarting the master server.

Operation	# of files	value (s)
Master server restart time	N/A	1081.198
Creating files	10 000	8.347
Creating files	100 000	132.234
Setting extended attributes	10 000	8.356
Setting extended attributes	100 000	87.747
Changing owner	10 000	9.986
Changing owner	100 000	101.474
Setting goal	10 000	26.264
Setting goal	100 000	248.85
Reading goal	10 000	27.623
Reading goal	100 000	244.623
Removing files	10 000	7.679
Removing files	100 000	128.01

Table 6.7: Metadata operations with fsalloc enabled with SSD

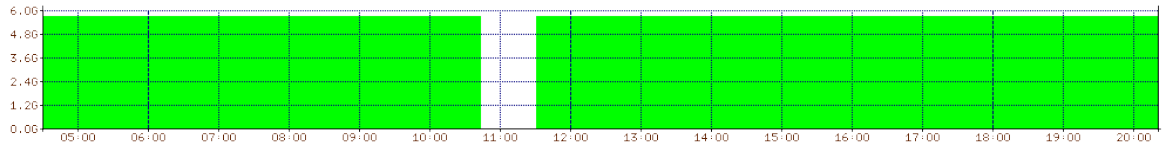

Figure 6.7: Memory chart representing transition from HDD configuration to SSD[4]

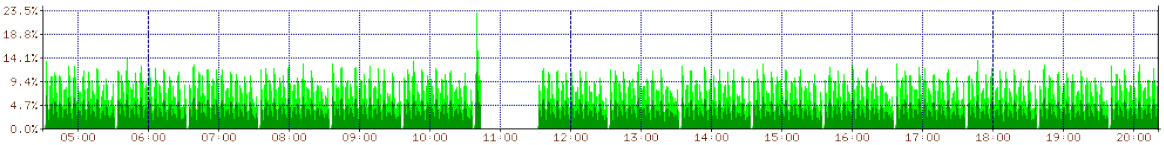

Figure 6.8: CPU chart representing transition from HDD configuration to SSD[5]

6.5. Combined results

Figure 6.9 shows a combined plot of performance tests for each configuration. It is clear that elapsed time of performing a chosen metadata operation multiple times does not strongly depend on any *fsalloc* configuration. On the contrary, it seems that none of the values is different from their average by more than 5%.

[4]Empty space indicates the moment of changing configuration, which requires restarting the master server.

[5]Peak value of nearly 25% (at 10:45) is an anomaly caused by external factors and should not be taken into account in comparison of HDD and SSD configurations.

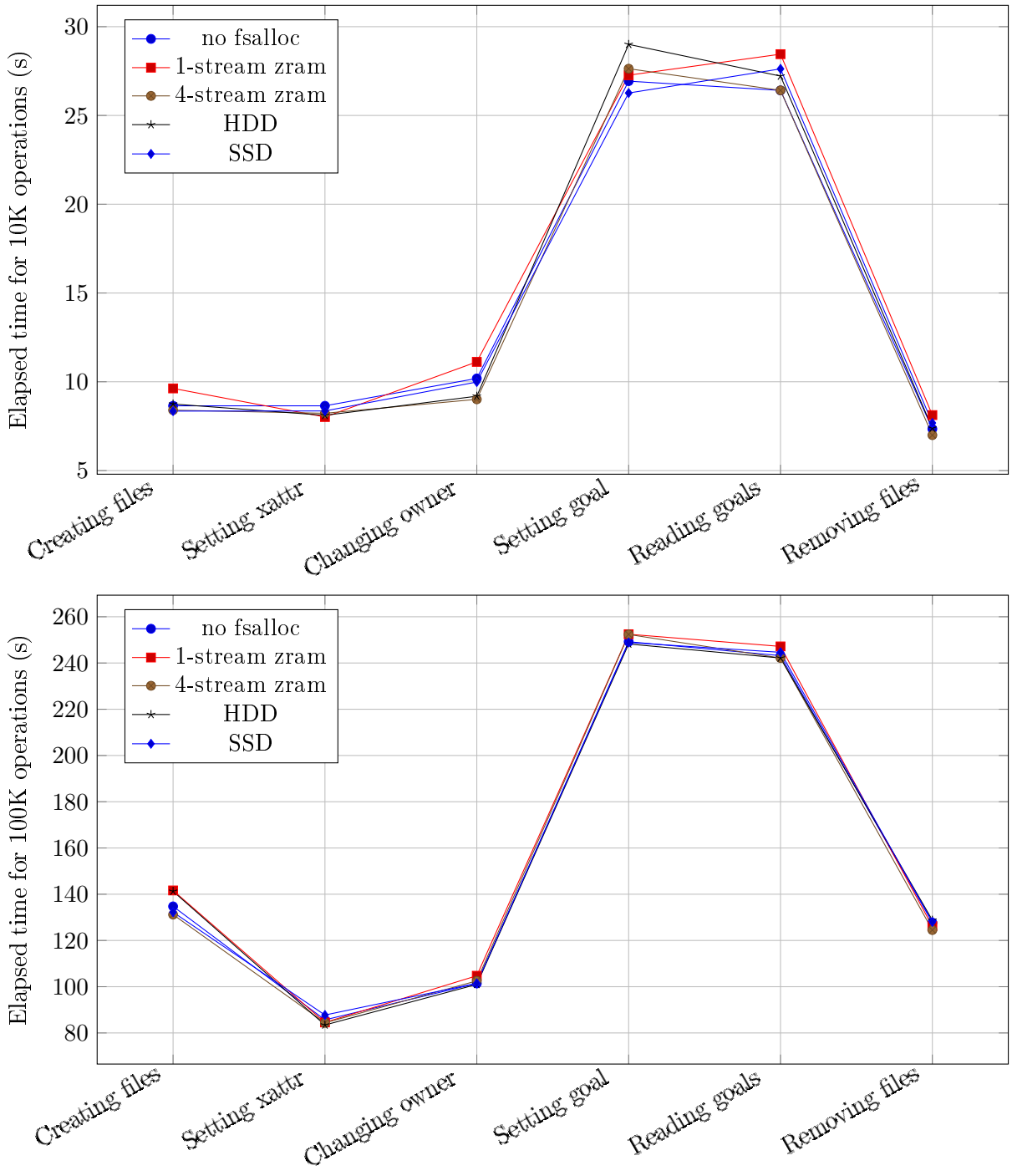

Figure 6.9: Combined times of metadata operations for all tested configurations. Low diversity of values corresponding to different configurations indicates that back end storage is not a major performance factor for small metadata operations.

Compression streams	1	
Number of reads	269	
Number of writes	7910336	
Memory used (total)	710250496 B	(710 MiB)
Compressed data size	682413960 B	(682 MiB)
Original data size	2062983168 B	(2 GiB)
Reclaimed memory	66.9%	

Table 6.8: Statistics for single-stream zram configuration

Compression streams	4	
Number of reads	447	
Number of writes	4963835	
Memory used (total)	710184960B	(710 MiB)
Compressed data size	682405652B	(682 MiB)
Original data size	2063036416B	(2 GiB)
Reclaimed memory	66.9%	

Table 6.9: Statistics for 4-stream zram configuration

6.6. In-kernel compressed memory

Two configurations of **zram** block device, based on the maximal number of compression streams, were prepared for testing. Tables 6.8, 6.9 present statistics provided by **zram** module.

Statistics were gathered during LizardFS' master server start. The booting procedures involve reading all metadata from hard drive to memory. Thus, an imbalance between number of reads and writes is noticeable.

An important conclusion is that file system's metadata proved to be a great fit for the chosen compression algorithm. All structures were packed to one third of their original size, which is significantly more than 50% — an average result for this class of algorithms. The reason behind this could be that mentioned structures are similar in size and often contain values close to one another in the sense of Hamming distance[6]. Consecutive index numbers or pointers to the same structures are examples of such congruent values.

As shown, number of compression streams does not influence the compression ratio. On the other hand, more compression streams led to slightly better performance results, which can be observed in section 6.4.

6.7. Profiling

6.7.1. Results

Overriding allocation procedures carries the risk of creating a bottleneck in a programming project. Fortunately, tools designed for detecting potential bottlenecks exist and they are broadly known as profilers. Profiler chosen to be used in this experiment is *perf* [16], very powerful, yet straightforward utility. It is officially supported by Linux kernel, which makes it a perfect candidate for profiling a project based on system calls and interrupts. Listing 6.10

[6]Hamming distance between bit arrays A and B of length n is the number of positions on which A and B have different bit values, i.e. $\sum_1^n [A[i] \neq B[i]]$.

contains a report of most frequent calls performed by LizardFS' master with *fsalloc* enabled. Each row in figure 6.10 consists of the following data:

- average percentage of processor time spent on a given call,
- name of the process responsible for the call,
- a bit informing if call originated from user-space ([.]) or kernel-space ([*k*]),
- source of the call's symbol (LizardFS symbol table, kernel or external library),
- call's symbol.

Supplementary profiling output is shown in figure 6.11. It presents a map of calls that used most of all resources. The graph clearly indicates, that mprotect call was directly responsible for over 32% of CPU overhead. Indirectly, the culprit was LizardFS master server itself — all calls whose identifier starts with **fs_** are part of metadata handling routines.

Figures 6.10 and 6.11 show only an abstract of profiling results. Full graphs and all perf [16] outputs are available on the attached CD.

6.7.2. Conclusions

The most active call, which used approximately 10% of processor time was a cryptic *_M_insert_aux* function. It is actually an internal method of vector and deque classes from Standard Template Library for C++. A deque (double-ended queue) is used as a container for cached memory regions in *fsalloc* — hence, the most costly operation turned out to be updating the cache of active regions. It is clear that using a less complex data structure would reduce the cost of this most expensive operation. A static circular buffer implemented with C-style array provides all the functionalities for first-in first-out cache, while much more flexible double-ended queue generates more overhead than necessary for a simple use case. However, despite being first on most expensive calls list, *_M_insert_aux* is not an actual bottleneck. For reference, *perf* output of the same experiment performed on improved implementation of fsalloc with better suited data structures can be found in figure A.5. The real, less obvious bottleneck is described below.

Second place belongs to page fault handler, which demanded nearly 7% of processor time. Restarting a server triggers loading all metadata from disk to RAM, which, in case of *fsalloc*, causes a lot of cache invalidation. The next 3 costly operations relate to page faults as well — each mapped memory area needs to be inserted into a red-black interval tree, pages need to be zeroed and cleared from translation lookaside buffers, etc.

The fact that calls to BerkeleyDB interface occupy low positions in profiling output is promising. It suggests that chosen database engine is indeed very efficient. Restarting LizardFS' master server is a write-oriented operation when it comes to memory, and yet database back end proved to fulfill its duties without creating a noticeable overhead.

```
# Inserting a memory region into fsalloc cache.
11.24%  mfsmaster  mfsmaster             [.] _M_insert_aux
# Serving a low level page_fault operation.
6.77%  mfsmaster  [kernel.kallsyms]     [k] page_fault
# Inserting mapped memory region into internal kernel structure.
5.74%  mfsmaster  [kernel.kallsyms]     [k] anon_vma_interval_tree_ins
# Overhead generated by perf.
3.06%  mfsmaster  [kernel.kallsyms]     [k] perf_event_aux_ctx
# Zeroing the memory of anonymously mapped page (optimized).
2.62%  mfsmaster  [kernel.kallsyms]     [k] clear_page_c_e
# Invalidating translation lookaside buffers.
2.56%  mfsmaster  [kernel.kallsyms]     [k] flush_tlb_mm_range
# Copying memory.
2.46%  mfsmaster  libc-2.19.so          [.] __memcpy_sse2_unaligned
# Locating memory area containing given address (segfault handling).
2.32%  mfsmaster  [kernel.kallsyms]     [k] find_vma
# Expanding the size of fsalloc cache.
1.93%  mfsmaster  mfsmaster             [.] _M_default_append
# Adjusting memory area size and underneath tree structure.
1.90%  mfsmaster  [kernel.kallsyms]     [k] vma_adjust
# Result of mprotect syscall.
1.75%  mfsmaster  [kernel.kallsyms]     [k] change_protection
# Slab allocation.
1.59%  mfsmaster  [kernel.kallsyms]     [k] kmem_cache_alloc
# Overhead for mutual exclusion.
1.50%  mfsmaster  [kernel.kallsyms]     [k] _raw_spin_lock
# Unmapping a memory region.
1.29%  mfsmaster  [kernel.kallsyms]     [k] unmap_single_vma
# Overhead created by BerkeleyDB.
1.29%  mfsmaster  libdb-5.3.so          [.]
```

Figure 6.10: Profiling results

An unpleasant conclusion is that for structures that appear in millions, per-object granularity creates a bottleneck in *fsalloc* library when all (or nearly all) objects need to be accessed. When cache is systematically invalidated with new entries, overhead created by continuous page faults start to dominate processor usage. This is evidenced by the fact that restarting master server with *fsalloc* enabled took nearly 20 minutes compared to 1 minute with *fsalloc*-free configuration. Results produced by perf 6.10 support this argument as well — processor was heavily occupied with context switching and page-level memory management.

On the other hand, if memory-wide operations are sparse (e.g. occur once a week), then it is possible that such increased processing time is acceptable. When locality of accessed regions is usually high, then cache meets its obligations and additional cycles spent on handling page faults are not perceptible for users.

Results presented in section 6.4 indicate, that master server restart puts a lot of pressure on *fsalloc*, which considerably extends the period of time needed for the action to finish. These

results are backed by the output of profiling — processor was forced to spend a lot of time processing page faults and revalidating heavily loaded cache.

One potential optimization idea arises from profiling results — moving the cache querying logic to kernel space would eliminate the need to switch modes so frequently. If active memory regions were kept in the kernel, only cache misses would need to be served by user space, where the Berkeley database interface is available. Cache hits could be served entirely during page fault handling. While having no noticeable impact on memory-wide operations like server restart (because they generate a lot of cache misses), casual operations with high memory locality would most likely save CPU cycles on unexecuted mode switches.

```
# mprotect
    32.71%     0.54%  mfsmaster  libc-2.19.so
[.] __GI___mprotect
     |
     --- __GI___mprotect
         |
         |--98.22%-- 0x2
         |
         |--1.04%-- handler(int, siginfo_t*, void*)
         |        __restore_rt
         |        |-23.94%-- fs_loadedge(_IO_FILE*, int)
         |        |        fs_loadedges(_IO_FILE*, int)
         |        |        fs_load(_IO_FILE*, int, unsigned char)
         |        |        fs_loadall(std::string const&, int)
         |        |        fs_loadall()
         |        |        fs_init(bool)
         |        |        initialize(run_tab*)
         |        |        main
         |        |        __libc_start_main
         |        |
         |        |-13.47%-- fs_loadall(std::string const&, int)
         |        |        fs_loadall()
         |        |        fs_init(bool)
         |        |        initialize(run_tab*)
         |        |        main
         |        |        __libc_start_main
         |        |
         |        |-9.18%-- fs_load(_IO_FILE*, int, unsigned char)
         |        |        fs_loadall(std::string const&, int)
         |        |        fs_loadall()
         |        |        fs_init(bool)
         |        |        initialize(run_tab*)
         |        |        main
         |        |        __libc_start_main
```

Figure 6.11: Profiling results — usage graph

Chapter 7

Conclusions

Implementing *fsalloc* and measuring its performance on a real life file system cluster resulted in many valuable lessons learned. Some of assumptions have proven correct, other, unfortunately, wrong. Nonetheless, a whole study certainly showed that solid-state drives and memory compression are very useful in the field of RAM usage optimizations. Test results have shown that memory reclaimed via using disks or compression did not cause performance drops in several cases (e.g. moderately small, local operations). Thanks to reflections found in SSDAlloc-related documents [1, 2, 3], a simple user-space memory management could be implemented and tested. Differences between using HDD, SSD and compression as a writeback target could be examined and compared to one another.

7.1. Evaluation

Test results revealed various strong and weak points of *fsalloc* library. As a reminder, presented solution was rated in terms of performance and programming ease.

Performance of *fsalloc* proved to be acceptable for performing minor metadata operations on LizardFS cluster, but installation-wide actions (e.g. master server restart) triggered a clear bottleneck — *fsalloc* overhead was responsible for causing the operation to be orders of magnitude slower than with native allocation routines. It should be noted, however, that restarting a master server is a maintenance operation — it is executed in exceptional circumstances only (server failure, software upgrade). As such, it might be acceptable to reduce memory usage at the expense of slowing down operations which happen very rarely.

Hopefully, root causes of mentioned bottleneck were identified. First of all, *fsalloc* treats each stored object separately. If fetching each object potentially requires performing a system call (mprotect) and handling a signal (SIGSEGV), and there are millions of objects to serve at the same time, accumulated overhead is very high, even though a single operation needs microseconds to succeed. Two natural solutions come to mind:

1. implementing memory management in kernel-space instead of user-space decreases the amount of required mode switches,

2. batching similar objects decreases granularity — which is the major cause of high overhead for installation-wide metadata operations.

Both above methods have disadvantages. Implementing memory management as a kernel module reduces portability and forces users to either recompile their kernel or use precompiled

unofficial kernel build before using alternative allocation, which rules out programming ease — the second important trait of *fsalloc*. Batching, in turn, requires more thinking and precision when it comes to integrating *fsalloc* with other projects. Forcing users to provide their own, thoughtful heuristics significantly reduces programming ease as well.

Another minor improvement would be to port *fsalloc* to C and use as simple data structures as possible. Even if allocation/deallocation is to remain a bottleneck for installation-wide operations, it should definitely be minimized.

Programming ease was achieved without a doubt. Simple user-space memory management created on the basis of SSDAlloc [1, 2, 3] and generic C++ interface makes *fsalloc* highly integrable. As shown in chapter 5, overriding native allocation procedures with the ones provided by *fsalloc* was a minor technical issue — neither deep knowledge of LizardFS nor exceptional programming skills were required to complete the task.

7.2. Perspectives

Performance tests showed that examined version of *fsalloc* was constrained by two factors: overly complex data structures and frequent memory-related system calls. Natural continuation of *fsalloc* project should consider dealing with both issues. The first, less serious one is already dealt with. Second iteration of *fsalloc* implementation (available on the attached CD) replaced misused standard data structures with their simplified versions:
deque → array based ring buffer, unordered_map → array based hashmap.
The real bottleneck is now clearly exposed (see figure A.5). Taking this into account, the actual goal should be to research and implement heuristics of bundling and batching multiple requests to avoid the overhead caused by frequent system calls, namely *mmap*.

7.3. Epilogue

Implementing an alternative allocation library and testing it on real-life installation of LizardFS was a very educational experience. With the help of performance tests and profiling, bottlenecks and false assumptions were detected and verified. Hopefully, this research will be useful for people responsible for designing memory management with particular attention to saving space.

Implemented *fsalloc* library is published under open-source compliant license and available at `https://github.com/psarna/fsalloc` and on the attached CD.

Appendix A

LizardFS graphs

This appendix contains visualizations of LizardFS cluster mentioned in chapter 1 and extended performance results related to chapter 6.

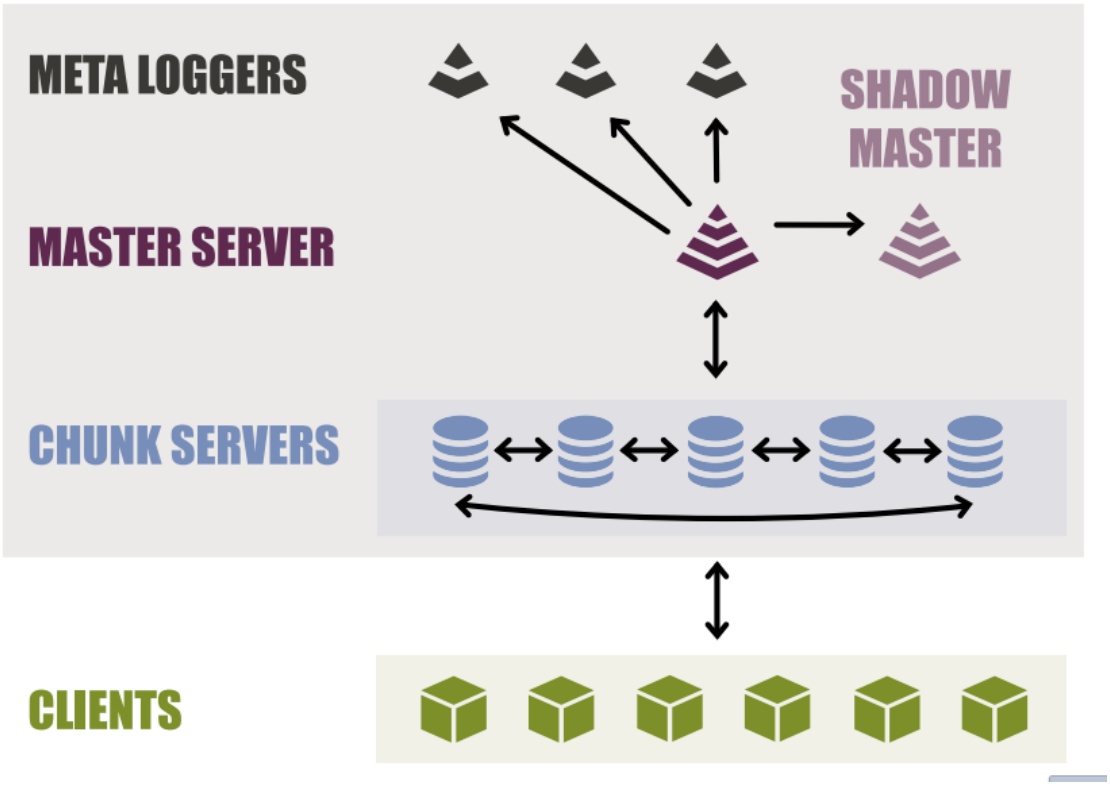

Figure A.1: LizardFS overview
Source: workshop materials of Skytechnology Sp. z o.o.

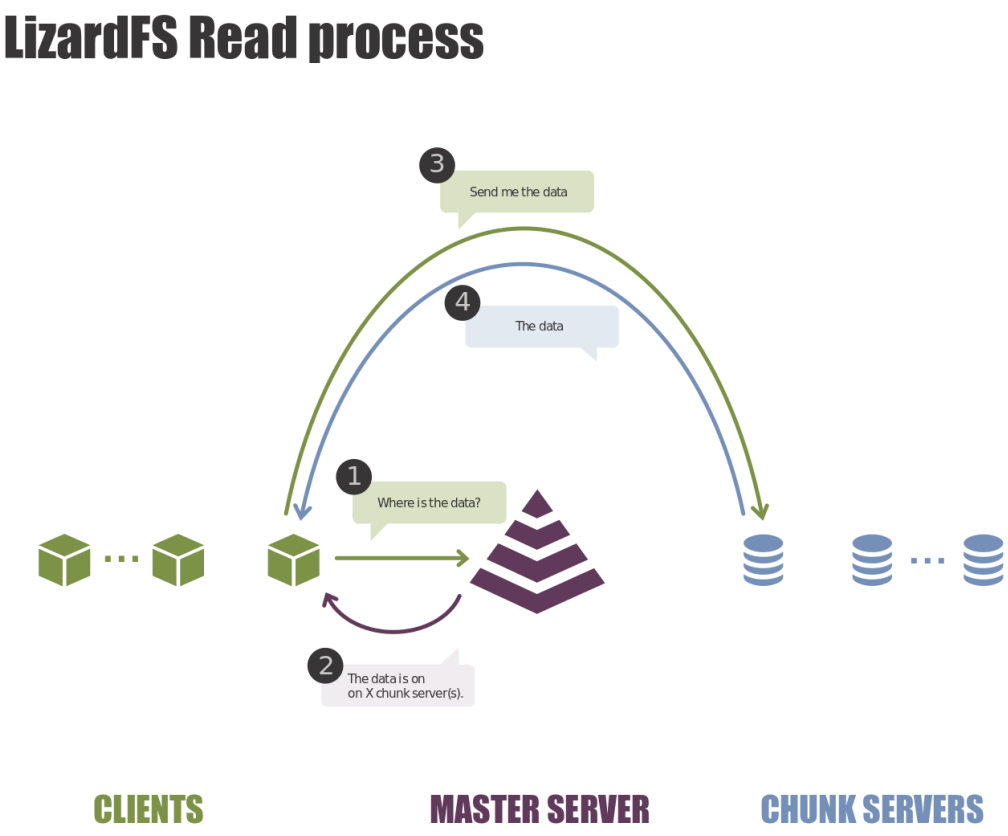

Figure A.2: Read flow of LizardFS
Source: workshop materials of Skytechnology Sp. z o.o.

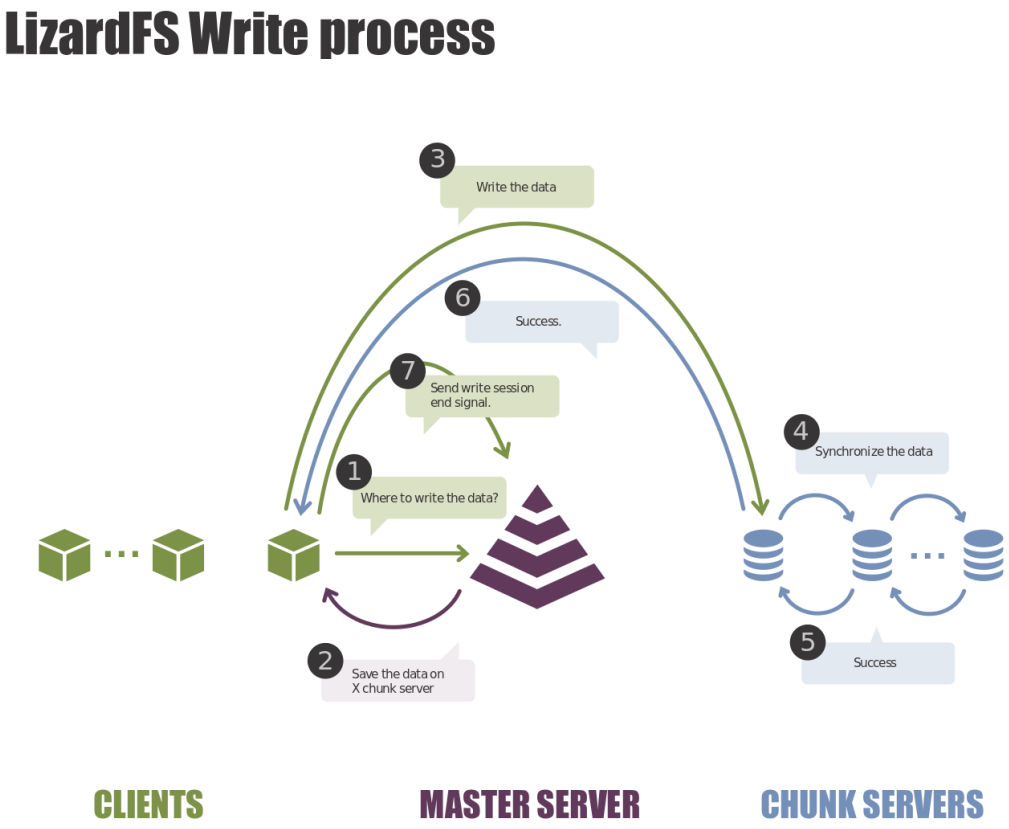

Figure A.3: Write flow of LizardFS
Source: workshop materials of Skytechnology Sp. z o.o.

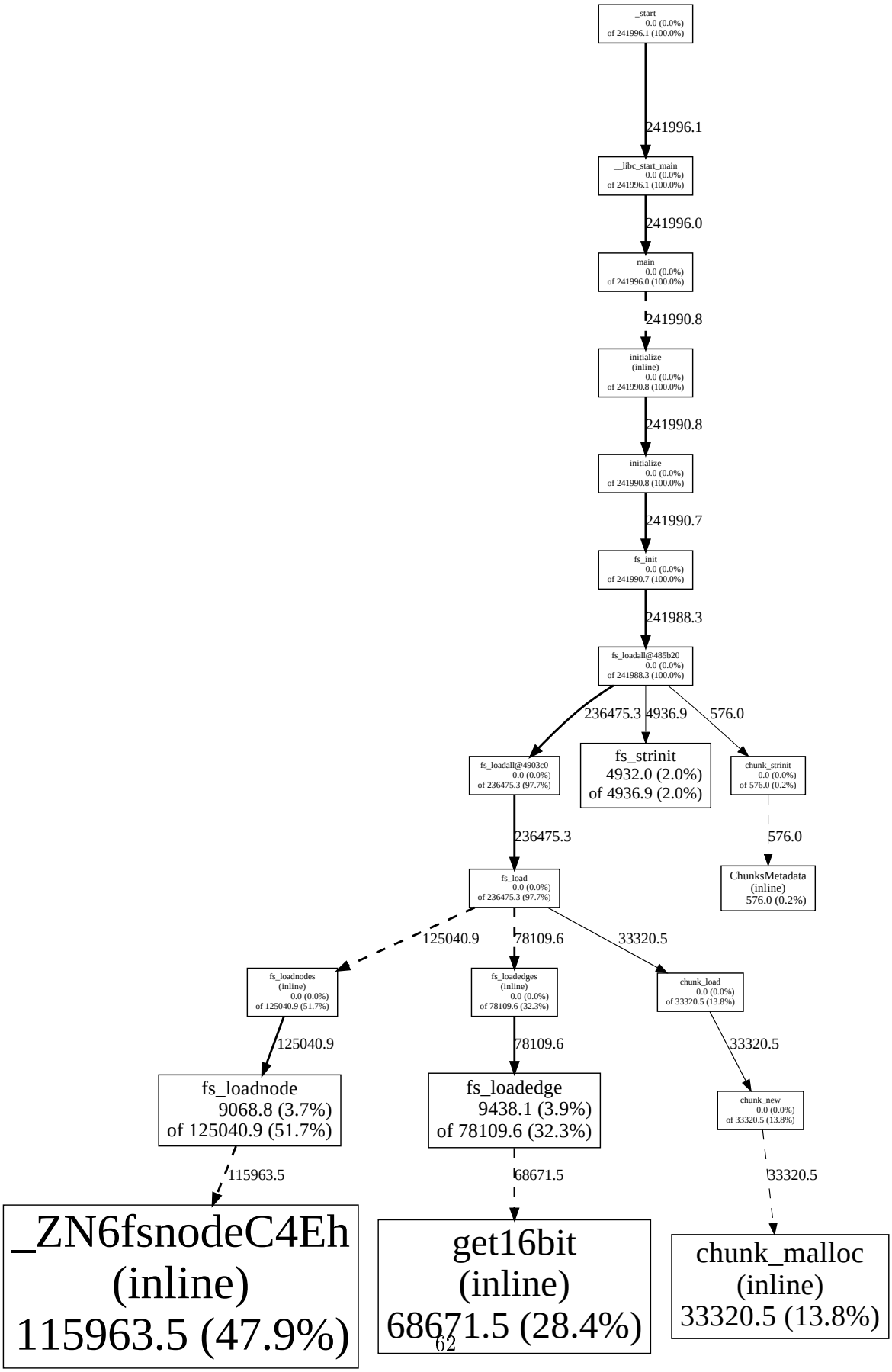

Figure A.4: Heap profiling results

```
8.89%  [kernel]          [k] anon_vma_interval_tree_insert
5.09%  [kernel]          [k] find_vma
4.47%  [kernel]          [k] page_fault
3.66%  [kernel]          [k] anon_vma_interval_tree_remove
3.60%  [kernel]          [k] copy_user_enhanced_fast_string
2.40%  mfsmaster         [.] chunk_checksum(chunk const*)
2.27%  [kernel]          [k] vma_adjust
2.26%  mfsmaster         [.] ChunkGoalCounters::fileCount() const
1.96%  mfsmaster         [.] fsedges_checksum(fsedge const*)
1.66%  [kernel]          [k] unmap_single_vma
1.53%  [kernel]          [k] clear_page_c_e
1.49%  mfsmaster         [.] handler(int, siginfo_t*, void*)
1.46%  [kernel]          [k] vma_compute_subtree_gap
1.42%  [kernel]          [k] perf_event_aux_ctx
1.42%  libc-2.19.so      [.] __memcpy_sse2_unaligned
1.39%  [kernel]          [k] flush_tlb_mm_range
1.35%  libdb-5.3.so      [.] __memp_alloc
1.32%  [kernel]          [k] vma_merge
1.17%  [kernel]          [k] __rb_insert_augmented
1.12%  [kernel]          [k] kmem_cache_alloc
1.12%  [kernel]          [k] find_get_entry
1.10%  [kernel]          [k] __rb_erase_color
0.93%  [kernel]          [k] change_protection
0.91%  [kernel]          [k] kmem_cache_free
0.90%  [kernel]          [k] up_write
0.88%  [kernel]          [k] __mem_cgroup_uncharge_common
0.82%  [kernel]          [k] perf_event_aux
0.81%  [kernel]          [k] save_xstate_sig
0.77%  [kernel]          [k] _raw_spin_lock
0.77%  [kernel]          [k] down_write
0.74%  [kernel]          [k] mprotect_fixup
0.73%  [kernel]          [k] mem_cgroup_charge_anon
0.73%  [kernel]          [k] vmacache_find
0.69%  [kernel]          [k] zap_page_range
0.69%  [kernel]          [k] vma_rb_erase
0.69%  libdb-5.3.so      [.] __memp_fget
0.66%  [kernel]          [k] get_page_from_freelist
0.65%  [kernel]          [k] system_call
0.62%  [kernel]          [k] release_pages
0.61%  [kernel]          [k] anon_vma_clone
0.59%  [kernel]          [k] _raw_spin_lock_irqsave
0.57%  [kernel]          [k] system_call_after_swapgs
0.53%  [kernel]          [k] perf_event_mmap
0.48%  [kernel]          [k] __restore_xstate_sig
```

Figure A.5: Perf results of master server running with new implementation of fsalloc based on better suited data structures: circular buffer and hash table

Appendix B

Contents of the attached CD

- /**thesis** — electronic version of this thesis
- /**fsalloc**
  - /**fsalloc**/**LICENSE** — full text of *fsalloc*'s license
  - /**fsalloc**/**src** — source code of *fsalloc* library
- /**results**
  - /**results**/**performance** — performance tests results
  - /**results**/**profiling** — profiling results

Bibliography

[1] Badam, Anirudh: Bridging the Memory-Storage Gap, Princeton University, 2012

[2] Badam, Anirudh: Easing the Hybrid RAM/SSD Memory Management with SSDAlloc, Princeton University, 2010

[3] Badam, Anirudh and Pai, Vivek S.: SSDAlloc: Hybrid SSD/RAM Memory Management Made Easy, Princeton University, 2011

[4] Bar, Moshe: The Linux Signals Handling Model, Linux Journal, 2000

[5] Beltran, Vicenç and Torres, Jordi and Ayguadé, Eduard: Improving Disk Bandwidth-bound Applications Through Main Memory Compression, Dealing with Applications, systems and architecture, 2007

[6] Berkeley DB Documentation - Oracle, http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/documentation/index.html

[7] Boost Library documentation, http://www.boost.org/doc/libs/

[8] CMake project, https://cmake.org/

[9] Google perf tools, http://goog-perftools.sourceforge.net/

[10] Hu, Jian and Jiang, Hong and Manden, Prakash: Understanding Performance Anomalies of SSDs and Their Impact in Enterprise Application Environment, ACM SIGMETRICS Performance Evaluation Review, 2012

[11] jemalloc documentation, http://www.canonware.com/jemalloc/

[12] Linux kernel documentation, Compressed RAM based block devices, https://www.kernel.org/doc/Documentation/blockdev/zram.txt

[13] LizardFS documentation, https://lizardfs.com/documentation/

[14] Love, Robert: Linux Kernel Development, Pearson Education Inc, 2010

[15] Magenheimer, Dan: In-kernel memory compression, Article for lwn.net, April 2013

[16] Perf Wiki, https://perf.wiki.kernel.org/

[17] Quick Benchmark: Gzip vs Bzip2 vs LZMA vs XZ vs LZ4 vs LZO, http://catchchallenger.first-world.info/wiki/Quick_Benchmark:_Gzip_vs_Bzip2_vs_LZMA_vs_XZ_vs_LZ4_vs_LZO

[18] Samih, Ahmad and Wang, Ren and Maciocco, Christian and Tai, Tsung-Yuan and Duan, Ronghui and Duan, Jiangang and Solihin, Yan: Evaluating Dynamics and Bottlenecks of Memory Collaboration in Cluster Systems, 2012

[19] TCMalloc documentation, http://goog-perftools.sourceforge.net/doc/tcmalloc.html

[20] Unified Format, diff utils, http://www.gnu.org/s/diffutils/manual/html_node/Unified-Format.html

[21] UserBenchmark, http://hdd.userbenchmark.com/, http://ssd.userbenchmark.com/

[22] Yong, Zhang: Kernel Korner: Meddling with Memory, Linux Journal, January 2001