TECHNISCHE UNIVERSITÄT DRESDEN
Institute of Systems Architecture
Chair of Operating Systems

Bachelor Thesis

Extension of an accelerator-friendly in-memory file system for persistent storage

---

Submitted by:	Sebastian Reimers
Date of submission:	12.11.2018
Date of birth:	31.03.1997
Course of studies:	Informatik
Professor:	Prof. Dr. Hermann Härtig
Supervisor:	Dr.-Ing. Carsten Weinhold M.Sc Nils Asmussen

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die von mir am heutigen Tag dem Prüfungsausschuss der Fakultät Informatik eingereichte Arbeit zum Thema

Extension of an accelerator-friendly in-memory file system for persistent storage

vollkommen selbstständig verfasst und keine anderen außer den angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

12.11.2018

_____

Sebastian Reimers

Abstract

In the recent years architectures became more and more heterogeneous to reduce power consumption and increase performance through special-purpose hardware. However, traditional operating systems (OSs) treat the majority of special-purpose hardware like accelerators as devices which are second class citizens. This prohibits accelerators to access OS services, such as file systems, network stacks, or pipes. M^3 (**M**icrokernel-based syste**m** for heterogeneous **m**anycores) treats abundantly available and arbitrarily typed compute units as first class citizens. To take advantage of accelerators now having access to OS services, an appropriate accelerator-friendly interface is required. M^3 currently uses an in-memory file system, specifically designed with accelerators in mind. In this thesis, said file system is extended to support persistent storage while keeping the accelerator-friendly properties.

Contents

List of Figures

Chapter 1

Introduction

Manycore architectures became mainstream around 2003 due to the physical restrictions of single core performance. Dennard scaling [DGR+74] broke down as the power density[1] became inconsistent. The reason for this was that clock frequencies could not be raised as drastically anymore with transistors becoming even smaller. Using multiple cores software has to adapt in order to benefit from parallelism. However, single core performance remained important, this time because of software limitations. Due to data dependencies inside an algorithm, it cannot be split in infinitely small fractions which can be run in parallel. As seen in Amdahl's Law, $Speedup = (r_s + \frac{r_n}{n})^{-1}$ the longest, independent sequential part[2] (r_s) limits the speedup, when executing the parallel part (r_p) in n parallel threads. In the end, single core performance remained to be the critical part, especially for software that cannot be parallelized efficiently.

As a result, architectures became more and more heterogeneous, as special-purpose hardware offers lower power consumption or higher performance for its associated task. This makes heterogeneity lucrative for every system, whether it needs to be especially energy efficient, or whether it executes performance-critical software. A common example are GPUs for highly parallel computations for matrices used for graphic or AI acceleration. General-purpose cores can offload such tasks to the GPU, which results in a high performance increase. Accelerators can be constructed for nearly every problem, e.g. cryptography, signal processing, and video processing [JS08, LCL+15]. In comparison to a general-purpose CPU have accelerators improved performance for their assigned task, although there is a limited number of tasks they can perform altogether.

Whilst the number of accelerators which accompany the general-purpose cores increases, operating systems have to adapt to use them in an efficient way. M^3 was designed to utilize the vastly different accelerators to their full potential. An appropriate in-memory file system, called m3fs, was implemented as a critical part of any OS as well.

The goal and the desired practical improvement, which will be discussed in this thesis, is to allow persistent storage and preserve its good performance. As an IDE driver is present, the missing parts to enable this feature are an interface for the driver and a buffer cache. In M^3 accelerators should run autonomously and use OS services on their own, therefore the services have to be designed appropriately. One of the challenges faced during the built is adapting the buffer cache in a way, which allows direct access to large amounts of data for

[1] Power-usage per area

[2] This could be the whole program, if it cannot be parallelized at all.

accelerators, so they do not have to message m3fs frequently. This contradicts the typical block-based cache design. Another issue that arose was that writing and reading from or to the storage device will take longer the more data is handled. Since it is one of the main goals to handle large amounts of data, it has to be ensured that m3fs remains usable whilst a disk request is running. Hence, disk requests must be asynchronous, and any issues caused by this must be eliminated. After providing the necessary background knowledge concerning both M^3 and the buffer cache, key aspects of the implementation will be discussed, and as a final consideration the built prototype shall be evaluated.

Chapter 2

Technical Background

This Chapter provides the necessary explanations for understanding this thesis.
Firstly the M^3 operating system will be introduced as the target platform. In Section 2.2 the file system m3fs used by M^3 will be depicted, as it is the system which needs to be extended. Afterwards the cache will be introduced as the feature that is to be replaced by the buffer cache mentioned in Section 2.4. Finally the provided IDE driver is mentioned.

2.1 Microkernel-based System for Heterogeneous Manycores

The target platform for this thesis is the *microkernel-based system for heterogeneous manycores* or M^3, currently in development at the Chair of Operating systems at TU Dresden. It lets applications run on bare metal, on heterogeneous compute units, each connected to a network on a chip over a special piece of hardware. Since M^3 follows a microkernel design, OS services such as the file system are separate applications as well. Applications can communicate over the network via a message-based protocol. Specifically system calls are implemented as messages to the kernel, avoiding context switching entirely.

2.1.1 Capabilities

Capabilities are M^3s means of access control. They are pairs of references to kernel objects and permissions regarding these objects. The kernel stores a table of capabilities per VPE as well as a tree of all operations that transfer them. There are four possible operations implemented as syscalls. Even though VPEs can communicate directly, the kernel must be involved because it handles all capabilities:

- *Obtain*, requests a capability from a different VPE,
- *Delegate*, gives another VPE access to a capability. This capability must be owned by oneself,
- *Derive*, only applies to memory capabilities. It creates a new capability, which gives access to a subset of the memory granted by the original capability,
- *Revoke*, reverses ALL delegates or derives of a capability. This is done by recursively walking over the tree kept by the Kernel.

2.1.2 The DTU

Whilst traditional operating systems treat dedicated hardware such as accelerators as second class citizens, i.e. as devices, M^3 treats everything as first class citizens [AVN+16]. It does so by introducing a *Data Transfer Unit* (DTU). The DTU is a special hardware component, over which all cores and accelerators are connected to a *Network on a Chip* (NoC). The combination of core/accelerator, scratchpad memory and a DTU is referred to as *Processing Elements* (PE). The DTU is the only way for a PE to communicate with any other PE or to access external resources. Specifically it is the only way to send syscalls to the kernel and access global memory.

2.1.3 Processing Elements

Similar to NIX [BEF+12], M^3 differentiates between Kernel PEs and Application PEs. However, it is allowed that not every PE is able to run the kernel.
The Kernel PE is a dedicated PE on which the kernel runs. It is privileged in terms of communication, meaning any communication over its DTU is allowed. The kernel PE is capable of restricting privileges of Application PEs by remotely configuring their DTU, assigning applications to PEs and managing the main memory, services as well as capabilities.
On an Application PE only its assigned applications are run, but only one at a time. Its communication is restricted and it cannot connect to other PEs by itself, thus enforcing isolation between applications.

2.1.4 Inter-PE Communication

Should an application PE wish to connect itself with another PE, it has to invoke the kernel via a syscall because only the kernel PE is allowed to configure the DTUs. The Kernel evaluates if a connection is allowed and then configures the DTU of the participating PEs remotely, if it is. Once a direct connection is established, it can be used without involving the kernel.

To establish a connection, each DTU has a finite set of *Endpoints* (EP) which can be configured by the kernel. There are three types of Endpoints:

- Send Endpoints to send messages to other PEs
- Receive Endpoints to receive messages from other PEs
- Memory Endpoints to get direct access to PE external memory.

As a software abstraction M^3 offers *Virtual Processing Elements*(VPE) and *Gates*. VPEs must be bound to exactly one PE at all times and represent one activity of an application. Several VPEs can share one PE via Time Multiplexing in case not enough PEs are available.
Gates represent connections and are bound to exactly one EP at any point in time. Again similar to PEs, endpoints can be multiplexed amongst multiple gates, so the VPE can have more connections than EPs. Three types of Gates exist corresponding to the three types of EPs. For each Gate a Capability has to be obtained from the kernel through a syscall.

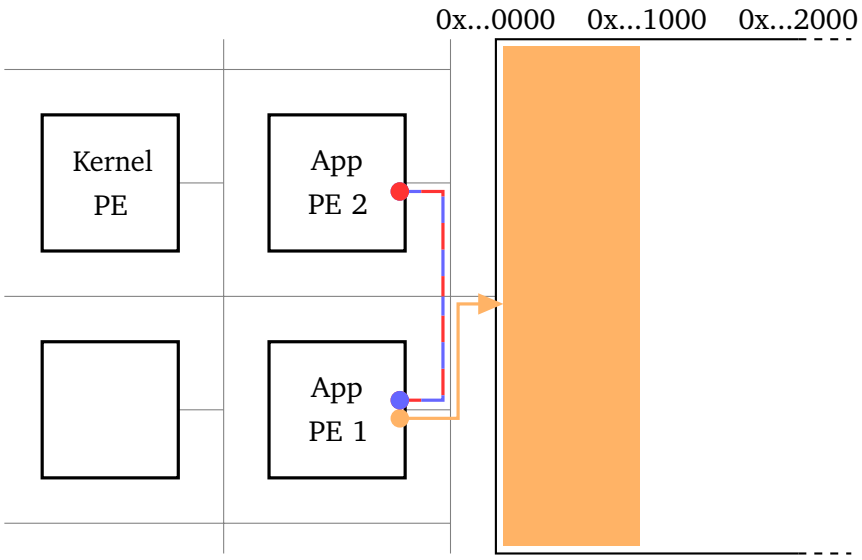

Figure 2.1: Example setup with three active PEs, one kernel and two applications. Application 1 has a Memory Gate over which it can access 4 KiB of global memory, as well as a Send Gate to PE 2 whose EP resides a Receive Gate. Note that default connections, e.g. to the kernel are not depicted.

2.2 m3fs

As an integral part of any operating system, file systems must be designed not only to fit the underlying OS, but also with the typical applications run by the OS in mind. Because M^3 aims to utilize accelerators as first class citizens, the used file system needs to consider them as well. Therefore, three assumptions can be made:

- In M^3 global memory is accessed via memory gates as a contiguous sequence of bytes.
- M^3 strives to let PEs work autonomously and aims to reduce the overhead caused by syscalls.
- Accelerators make use of extensive files to allow them to work autonomously for longer.

With these key points in mind *m3fs* was designed. M3fs is set up like typical UNIX file systems: Data is stored in fixed sized blocks and files consist of several *Extents*. Extents, like in many modern file systems [MCBD07], are pairs of a start block number and a length, thus defining a contiguous range of blocks. Each file system image starts with a superblock, which contains the necessary data to use the rest of the file system:

- The block size,
- The location of inode/block bitmaps and the number of blocks they consist of,
- The location of the first inode block, the number of inode blocks and the number of inodes per block,
- The location of the first data block,
- The number of extents per block,
- A checksum.

Every file is described by an inode, which stores the usual information such as the file size, link count and the last date when the file was accessed, as well as a list of extents describing where the actual data resides. Up to three extents can be stored directly inside the inode, more can be stored in extra blocks, referenced by an indirect block. Several indirect blocks can be referenced themselves by an double indirect block, if needed.

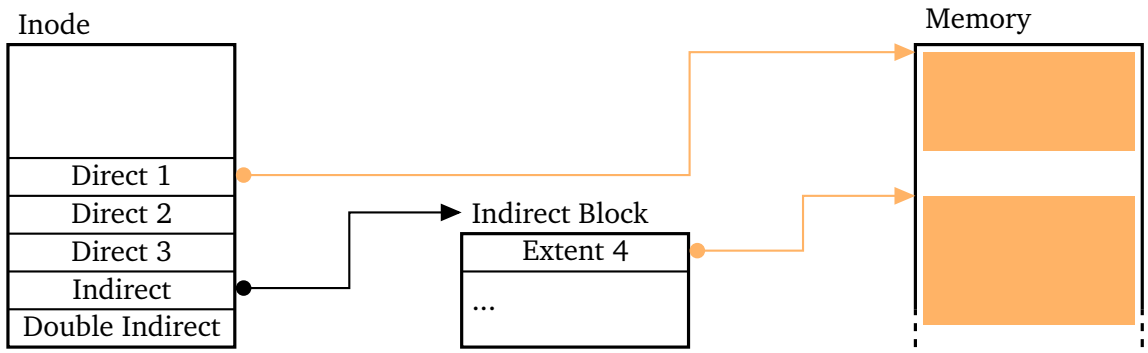

Figure 2.2: An inode with at least four extents, since there is an indirect block. Different extents of one file will never be contiguous, because then they would be merged into one larger extent.

M3fs registers a service at the kernel. If an application wants to use a service, it makes a syscall. The kernel then creates a pair of sessions, one for the service (Server Session) and one for the application (Client Session), and stores the responding capabilities in a table. A part of this client session is a send gate, configured to pass messages to the server sessions receive gate. The application can directly send requests to the server and may receive a response once the request is handled.
For m3fs, two types of server sessions are provided: A meta session for metadata and a file session for the actual data. This is necessary because, similar to the Google File system [GGL03], m3fs handles these two types of data differently.
Metadata such as bitmaps or inodes reside in exactly one block each and are only used by m3fs, if the client requests certain modifications. Therefore, the needed blocks are loaded into local memory because the data is accessed via a pointer to the block and a local offset.
However, the actual data of the file is modified directly by the application through a memory gate. The capability to access the main memory where this data resides needs to be obtained from m3fs first. M3fs has access to the whole file system inside the main memory and derives the appropriate parts, should a client request data. Because a capability can only be created for a continuous range of bytes, the natural limit for it is one extent, which means that at least for each extent in a file the client has to send a request to m3fs. Since the goal is to let the client work autonomously for a long time and operations like seek will become more expensive the more extents a file has, evidently the creation of small extents should be avoided. Therefore, a large number of blocks will be allocated, if a new extent will be appended, e.g. during a write operation, and the commit operation removes all unused blocks at the end of an extent.
The typical POSIX-like file operations are provided and implemented in their corresponding

server session, i.e. the file session covers *read, write, seek, fstat*, while the meta session implements *stat, mkdir, rmdir, link, unlink*.

2.3 The Cache

Since the meta session copies the metadata into local memory, m3fs needs a cache. As the local memory is much smaller than global memory, it is impossible to load the whole meta-data into it at once. Should the cache be full but more data must be loaded, another entry has to be evicted. One way to determine what has to be evicted is the popular LRU algorithm. It keeps track of which cache entry is the **l**east **r**ecently **u**sed one through timestamps or a list, and evicts it, if needed. The most important functions of the cache are:

- `get_block()` searches if a block is cached and returns a pointer to it. If the block is not found, the least recently used block (with the smallest timestamp) will be evicted and the block is loaded from main memory. It also sets the block headers timestamp and the dirty bit, if needed.
- `mark_dirty()` simply sets the dirty bit for a given block number.
- `write_back()` flushes a single block.
- `flush()` flushes the whole cache.

The cache has a limited size of 16 blocks and for each block a descriptive block head exists. It contains the block number, a dirty bit and a timestamp. Both the blocks and the block heads are stored in arrays. A block shares its index with the block head referencing it. Therefore it is unnecessary to store the location of the block inside the head as well. Should a block be accessed, its timestamp will be set to the caches timestamp, which is simply an increasing unsigned integer. If a block needs to be evicted, the cache iterates through the array of heads and remembers the one with lowest timestamp, i.e. the least recently used one. While searching for a block number, the cache needs to go through this array as well, foreshadowing a problem for the later implemented, much larger buffer cache.

2.4 The Buffer Cache

Almost every system uses a storage device for persistent storage, whether it is an actual magnetic disk, a SSD or similar. These devices appear at the bottom of the memory hierarchy, having the largest capacity but also the largest access time. With an access time of several milliseconds it is unfeasible to load from the disk for every request to the file system. One solution to this issue would be to load data block wise into main memory the first time it is accessed. Every subsequent operation will then perform on main memory without the need of another disk request. However, main memory tends to be smaller than a persistent memory, therefore a structure is needed to determine which blocks will be evicted and written back, in case a new block is requested but there is no capacity for it. This structure is called a *Buffer Cache*, block cache, or page cache and it is implemented in most operating systems.
To determine whether a block is already inside the cache, a hash table with linked lists

as collision resolution, and as a replacement algorithm LRU through a linked list is most commonly used. Since requesting blocks should be rather infrequent, it is feasible to ignore the overhead from the linked list [TB15]. Each block is described by a Buffer Head which is a simple structure holding any information the file system needs to operate on its referenced block. It typically contains at least the following members:

- Several state flags, such as *dirty*, *locked* and *invalid*.
- A device number,
- The start block number and the size,
- A pointer to the actual loaded data,
- A link count,
- Several pointer to other buffer heads, so it can be inserted into the hash table and the LRU list.

Upon buffer initialization memory is reserved, which functions as a pool where resources can be drawn from. A fixed number of buffer heads will be created, all marked invalid because no blocks are loaded from the disk yet. Should a block from the buffer cache be requested, there two possible scenarios: Its representing buffer head is inside the hash table, or it is not. If option one is true, access to the data is given via the pointer, and it is moved to the end of the LRU list. If not, the block must be loaded from the disk. To accomplish this, a piece of memory gets taken from the pool and assigned to a free buffer head. In case the pool has not enough resources to satisfy the request, the first block in the LRU list is evicted. The data will be written back, if the dirty bit is set, and the memory gets freed, giving it back into the pool. The new buffer head will mark the block as locked because it already points to a piece of memory, but it will not contain the right data until the disk request loading the data from the disk is finished.

2.5 The IDE Driver

Device drivers are pieces of software which control their dedicated device and offer an interface to the applications using them. Drivers are typically part of the kernel because they need to access some registers, at least the ones of their own device [TB15]. This is not the case for M^3 as it is a microkernel-based system, where drivers are independent applications, thus run on their own PE. This allows any other application to directly use the driver, if the kernel allows a connection.
It currently uses a modified version of Escape OSs [Asm] IDE driver which only supports ATA/ATAPI devices in PIO mode. However, it lacks an interface to other PEs, therefore it needs to be extended by a request handler to process requests from from other PEs, mainly the one m3fs runs on.
As in any standard ATA device, the sector size is 512 Byte, which is the minimal amount of bytes the controller can read. For simplicity, it can be assumed that there is only one device with only one partition, which contains nothing but the file system image. Handling partitions

would be part of the driver and having multiple disks would not change how the buffer cache operates.

Chapter 3

Design

Since the traditional design of the buffer cache is not compatible with M^3, an appropriate solution must be found. In this Chapter, several solutions and their viability shall be introduced and discussed. Section 3.1 starts with the buffer cache, and why the traditional design is not appropriate. Afterwards, the problem of asynchronous disk requests is addressed separately.

3.1 The Buffer Cache

The buffer cache is the key feature implemented during this thesis. One issue is, that the buffer cache typically operates block-wise. The goal is to provide as much data to accelerators as possible, so they can operate autonomously for as long as possible without sending a message to m3fs requesting the next extent. But since memory capabilities only provide access to one extent, in other words a range of contiguous blocks, the data has to be contiguous in main memory as well. For an in-memory filesystem this is not a problem, since everything is already allocated extent-wise in main memory. However, if the buffer cache were to be kept block-based, it could not be guaranteed that sequential blocks on disk would appear sequentially in main memory.
Before discussing ideas on how to restructure the buffer cache, so it would use extents rather than single blocks, it is noteworthy to say that only file data is accessed by the clients themselves. Meta data, such as inodes, bitmaps and directory blocks, will only be written by m3fs on behalf of the clients. The distinction in handling these two data types already becomes apparent in the client sessions of m3fs: a file session and a meta session. This distinction can be applied to the buffer cache as well. As explained in 3.2, requests to m3fs can be suspended at any time, to allow other requests to be handled, whilst a disk requests is active. This means that, while a request is suspended, buffer entries used by said request can be evicted. For file-data this is no problem, since the client gets access to the entries over a capability, which can be revoked. Meta-data however is copied into local memory and accessed via a pointer, which evidently cannot be marked invalid as easily. Therefore it must be ensured that the maximum amount of (meta-) blocks used by the maximum amount of parallel requests can fit inside the buffer without being evicted. Since that is not the case for file-data, separate buffer caches for file and meta-data can be created. Each of them will be discussed separately in the following subsections.

3.1.1 The Meta Buffer

The meta buffer will replace the currently used cache, as the cache was only used for meta-data. Its purpose is to cache the blocks in the local memory of m3fs. It uses a fixed number of block heads or (meta) buffer heads, each pointing to the location where the data is stored. Just like the original cache enough memory will be allocated on the heap to write all the blocks in it, each to its own offset. Instead of the disk server writing the data directly into m3fs' memory, global memory is allocated as a kind of shared memory. This shared memory will consist of one extent large enough for all blocks to fit inside, basically corresponding to the local memory. This allows that the exact same offsets are being reused as for the local buffer, and only one capability has to delegated to the disk server.

3.1.2 The File Buffer

Whilst the block-based design with a fixed number of buffer entries works for meta-data, it is unsuitable for file-data. The main problem is, that extents have an arbitrary length, and the delegated extent should be as long as possible. There are several options to resolve this:

Buffer entries must have the same size. Therefore, only parts of extents larger than the entry size can be loaded, or space would be wasted when loading a smaller one. This basically would be a block based cache, but the used "block size" would be a multiple of m3fs' block size. The advantage would be that a constant number of fixed-sized blocks can be allocated, which are easy to handle, similar to the current cache. Furthermore, if the buffer size stays constant, one piece of global memory can be allocated for the entire buffer. Like this, only one capability must be delegated to the disk server to establish shared memory, instead of one for each entry. The problem with this proposed solution is undoubtedly finding the right cache block size. If a small block size is chosen, only small parts of larger extents can be loaded, and clients can only access these small parts. This would make the buffer cache inefficient as M^3, or m3fs in particular, needs to provide large amounts of data for accelerators to let them work autonomously. On the other hand, a large block size would waste a lot of memory loading blocks that will not get accessed. Therefore fixed sized blocks do not seem to be suitable for m3fs.

Another option would be to use two cache block sizes: one for small and one for larger extents to reduce the amount of wasted memory. To keep the number of cache entries constant, the file buffer must be split into one for large and one for small blocks. For this scenario, it must be ensured that no block appears in both file buffers. Another problem is to determine in which buffer an entry will be loaded into. An application would need to pass information on to m3fs, about how much of an extent it intends to use, or the extents will be assigned to a buffer depending on their size. But since extents are usually large they will be assigned to the larger buffer most of the time, even if the client only needs a small piece.
If both cache block sizes were to be kept in one buffer to avoid doubled entries, the number of cache entries would not be constant anymore, as the same memory could be mapped to either

several small, or one large entry. If an arbitrary number of entries was the only solution, any cache block size could be allowed instead of just two. Like this, small extents can be loaded into the buffer without wasting memory. At the same time, the aim is to load as much of an extent as possible, which is very much desirable. With different sized entries, the memory for the entire buffer cache can no longer be allocated at once. Since an entry needs a contiguous piece of memory large enough to fit inside, arbitrary entries cannot be replaced anymore. Enough sequential entries must be evicted, preferably those which were not used recently.
Instead, the memory for the buffer cache is to be allocated on demand. For each entry a new piece of global memory is requested, which has exactly the size of the extent. Should an evict occur, the least recently used ones are chosen, their memory freed and a new piece gets requested. Due to this, the memory capability for each new entry must be delegated to the disk server, and the capabilities for evicted ones must be revoked. Since it is anticipated that loading and evicting entries is rather the exception, and cache hits are much more frequent, this still is feasible. Not reusing the memory has advantages as it allows reordering reads and writes from or to the disk to some extent. If enough main memory is available, reading from the disk can be performed before, writing back allowing the clients to access the data earlier.

3.2 Asynchronous Disk Requests

Reading and writing from or to the disk takes a lot of time. Actively waiting until the disk request has finished wastes a lot of resources. Instead of waiting m3fs, could handle another request, which in the best case could be a cache hit, and therefore can be finished without another disk request. Applying this, the request to m3fs gets suspended when a disk request occurs, and is continued when the disk operation is finished. Doing so, another issue arises: Requests handled by m3fs can now be interrupted at any point. This is especially dangerous in terms of the meta buffer cache. If the meta buffer cache were to be kept be kept similar to the current cache, it could mean that if one request already using a meta buffer entry (via a pointer) gets interrupted, another request evicts the same entry and overrides the space with another block. The first request would later continue with a pointer to a block that is completely different. This wrong block could have a different type than the evicted one and the request would simply fail, or it could have the same type and the request can finish, but with a wrong result. Either way, blocks that are used by any request cannot be allowed to be evicted. To achieve this, three changes have to be made: A link-count to the meta buffer entries must be added to keep track of how many requests use a certain block. The m3fs operations must store the block numbers of the meta blocks they used, so they can decrement the link-count of all of them when the operation finishes. The size of the buffer cache must be increased, so it can store all blocks needed for the maximum amount of requests possible. The maximum number of blocks needed by a request is at most 16, the size of the current cache where request could not be interrupted. M3fs can handle up to 32 requests at once, since its receive gates buffer can only store 32 messages, which remain inside the receive buffer until a response is sent. With 32 request and up to 16 blocks, the size for the meta buffer cache

adds up to 512 blocks, which is a lot more than the current cache, but it is needed to keep the buffer usable.

Chapter 4

Implementation

In this chapter the most important parts of the implemented prototype are described in detail. Firstly, a simple setup for the prototype, including m3fs, the new disk service and an example application is presented. In the next Section, the issues and possible solutions for asynchronous disk requests shall be discussed. Furthermore, the implementation of the extent-based buffer cache as per its design described in 3.1 is outlined. In the last section of this chapter, the changes made to the existing file and meta sessions are illustrated.

4.1 Setup

For this prototype the target platform is an X86 architecture built by the gem5 simulator. The setup consists of:

- The kernel
- The disk server running as a daemon
- M3fs running as a daemon
- An application

Each one occupies its own PE and both the disk server and m3fs registers a service at the kernel, named "m3fs" and "disk" respectively. M3fs allocates multiple pieces of main memory for the buffer cache and then delegates the according capabilities to the disk server, to which it previously connected to. The application which connects to m3fs, obtains a subset of the capabilities owned by m3fs, according to the data it has requested.

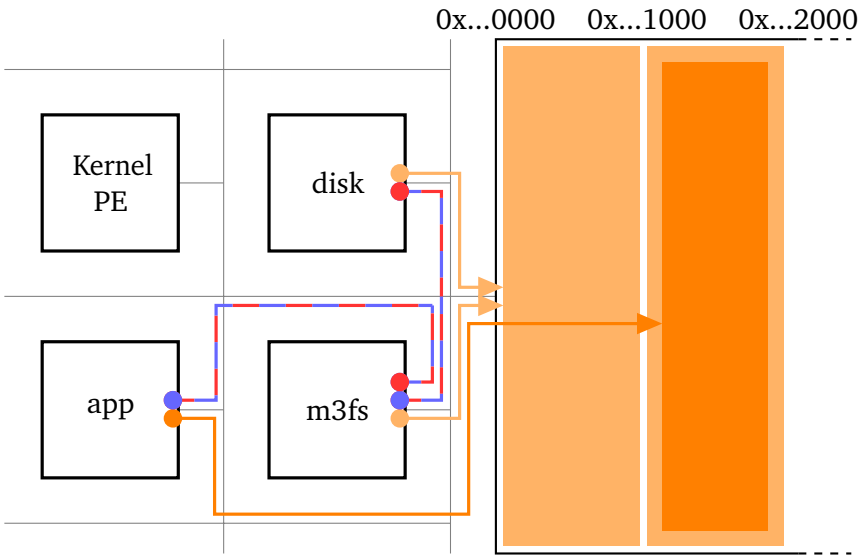

Figure 4.1: Setup with four active PEs. The application connects to the m3fs service and gets a **Send Gate** which sends requests to m3fs' **Recieve Gate**. The receive gate m3fs can send replies to is omitted. The same applies to the connection of m3fs and the disk server. Once the **desired data** is inside the **Buffer Cache**, m3fs derives the appropriate capability to delegate it to the app, where the file back-end loads the data in its own buffer. The actual addresses inside the main memory are irrelevant, but each extent has to be at least one block (4 KiB) large.

4.2 Asynchronous Disk Request

Threads are to be used to be able to suspend requests and continue them later. M^3 utilizes so-called *green threads*. Green threads are user-level threads, which do not get scheduled by the kernel[1]. Instead, it needs to be determined by the applications themselves, when to suspend or switch threads. Threads can either be *running*, *ready*, *sleeping* and *blocked*. Only one thread is running at a time. Blocked threads wait for an event to trigger and can only be scheduled again, after this event happened. Threads become blocked after calling `wait_for`. Sleeping threads have no intend of being scheduled. A thread can become sleeping after calling `yield`, meaning it voluntarily decided to stop. Sleeping threads can be scheduled when no other thread is ready. Ready threads do not currently run, but want to.

[1]The kernel in fact does not know about these threads at all.

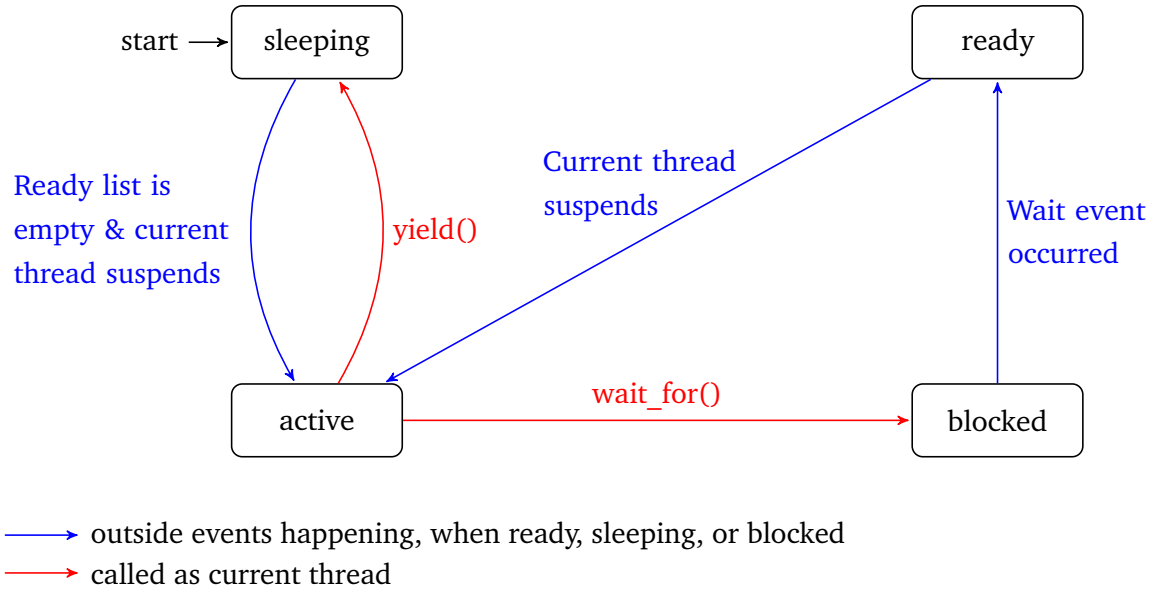

Figure 4.2: State diagram for M^3s threads

The active thread executes a *workloop*: It tries to suspend the DTU until a message is received, runs all *work items* and then checks if another thread is ready. If the ready list is not empty, the active thread yields itself and lets the next ready thread execute the workloop. Most of the time, work items are created through receive gates. When such a work item is run, it checks if the gate received a message, and if it did, it calls the appropriate handler.

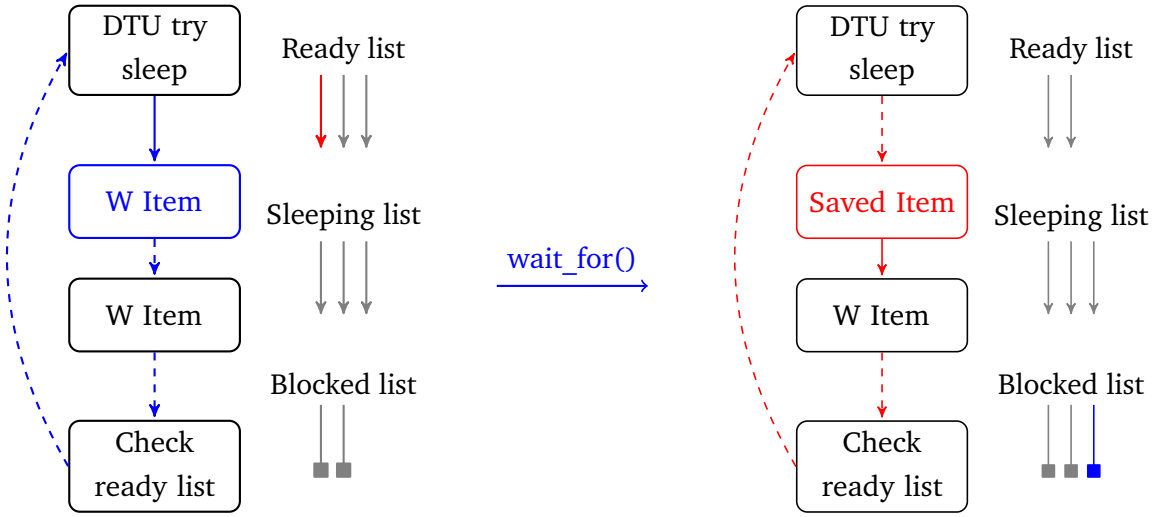

Figure 4.3: Example of M^3s workloop

The important function to enable asynchronous disk requests is `wait_for`. It subscribes the current thread to an event and then blocks it, storing its current state including the current work item. Then the next ready or sleeping thread becomes active. When reading or writing from or to the disk, such an event is created and is sent together with the request to the disk server. Since a block can only be part of one disk request at a time because buffer entries get blocked during disk requests and a block cannot exist in two entries, its request can be identified at any time. Therefore, it is suitable to create the wait-event based on the start

block number of the request. M^3 allows its messages to be labeled. Responses have the same label as the messages they response to. A disk request is labeled with a wait-event and should m3fs get a response, it notifies all threads subscribed to this label, readying them. When scheduled again, the previously suspended thread can continue handling the m3fs requests, as the needed disk request has finished.

4.3 The Buffer Cache

Whilst the traditional buffer cache organizes its buffer heads through a hash table, the buffer cache implemented for this thesis does not. The problem is, that all block numbers in one cache entry need to match said entry. The hash function would have to change every time a new entry is loaded, so that all blocks in one entry match the same bucket. This is definitely not easy to achieve. Instead, the prototype uses an already provided treap, as it can easily be extended by an appropriate matching function. The applicable `matches()` function searching for a block number would look like this:

```
bool matches(b_number) {
  return (entry_start_number <= b_number)
         && (b_number < entry_start_number + entry_length);
}
```

The `BufferHead` base class therefore esesinherits from both `TreapNode` and `DListItem`, so it can be inserted in the treap and the LRU list. Buffer heads can be marked as `locked` if their data is part of a disk operation to temporarily prevent them from being evicted. Writing to the disk could also mean that an entry is currently being evicted. If a client finds a locked entry, it must search the treap again upon unlock, in case the entry was evicted. An example is the file buffer's `get_extent` functions, which looks similar to this:

```
size_t get_extent(requested_block_number bno, requested_length length,
                  target_capability_selector sel) {

  while(true) {
    // searches the treap for the head containing bno
    FileBufferHead *b = get(bno);
    if(b) {
      // found an entry
      if(b->locked)
        ThreadManager::get().wait_for(b->unique_wait_event);
      else {
        lru.remove(b);
        lru.append(b)

        len = min(length, (b->_length - (bno - b->start)));
        // derives a capability (target_selector, offset, length)
        Syscalls::get().derivemem(sel, b->data.sel(),
                                  (bno - b->start) * blocksize,
                                  len * blocksize);

        return len * blocksize;
      }
    } else {
      // no entry found
      break;
    }
  }
  {...
    create an entry (evict entries when needed)
  ...}
}
```

The meta buffer's `get_block` functions works analogous to this, however it does not need a requested length since it gives access to single blocks. Neither does it need to derive a memory capability. Instead of the length of the returned memory capability, it returns a pointer to the block.

4.3.1 The Meta Buffer

As explained in section 3.1, the meta buffer cache uses a fixed amount of memory, allocated on the heap and buffer heads with pointers to this memory with a block-sized offset. The LRU list contains the heads of the entries that are not used and therefore can be overwritten. It should never be empty when a new entry needs to be inserted, because used blocks cannot be

evicted, as explained in 3.2.
A link-counter is an addition exclusively to the meta buffer heads, storing how many unfinished m3fs requests use a certain block. Upon completion each requests decrements the link-count of its used blocks. Should the link-count become zero, the head is appended to the LRU list, as it is no longer used by anyone.

4.3.2 The File Buffer

Unlike the meta buffer cache, the file buffer does not have a fixed number of entries. Each new entry allocates global memory via libm3s `create_global` function and stores the resulting memory capability in a new file buffer head. Both the treap and the LRU list contain all current entries. To avoid loading too many blocks from the disk, the file buffer limits the size of the extents loaded from the disk. This limit must be less than or equal to the maximum size of the whole buffer.

4.4 The File Session

With the file buffer cache, a new layer between the file back-end and the blocks of the file system image is introduced. Previously, files consisted of blocks, and the largest, contiguous subsets of all blocks are covered by extents. Files still consist of blocks covered by extents, now managed by the buffer cache. But said extents may not cover as much contiguous memory as possible, since buffer cache entries have a limited size. Therefore, several buffer cache entries can belong to one file-extent, but do not reside sequentially in main memory. Since m3fs was an in-memory file system, the file session could always provide whole extents for the clients. The file session therefore expected whole extents to be present in main memory, which, after introducing the file buffer cache, is not guaranteed anymore. File buffer entries must not contain the whole extent, as it is described in the inodes. They can even contain multiple extents. For example, extents are always allocated, so they contain a huge amount of blocks. Since allocating does not require a disk request, the used buffer cache entry is not limited in size, hence contains the whole extent. Should the file now be truncated, the unused memory can be reused for a different file. However, the buffer cache entry still covers all of the blocks, which are now used for two different extents.
To accommodate these structural changes, the file session must be adapted to no longer require whole file-extents. In addition to the size of the extent, the session must now store how much memory the buffer cache actually returned, since that is the amount, the client can operate on. Furthermore it is important if a provided buffer cache entry belongs at the end of a file-extent, since only then, the session is allowed to step to the next one, if needed. This gets even more important considering that extents can change in size, while their buffer cache entries stay the same until they get evicted.

Chapter 5

Evaluation

After having described the design and implementation of the prototype, in this chapter it shall be evaluated, how limiting the size of file buffer entries affects the performance. Also, it must be assessed if loading from the disk affect cache hits. Finally, it is to be determined how the prototype compares to the current in-memory version of m3fs.

5.1 Limiting the Size of File Buffer Entries

When making a request to m3fs, the information of how much a client actually intents to read or write is omitted entirely. Instead, m3fs tries to grant access to as much contiguous memory as possible to avoid the overhead of future requests. This is a problem as extents need to be loaded from the disk now, since loading takes a substantial amount of time.
In Figure 5.1, it is displayed how much time it takes a client to read a certain number of blocks from a file. In this graph, it is also depicted how limiting the entry size and the consequentially altered distribution of data inside the buffer impacts the performance, because the file consist of only one extent. There are two options on how to set the limit for entries: the static attempt, where the maximum number of blocks loaded is a predefined number, or the dynamic attempt, where the limit is determined by the number of client requests. Re-occurring read or write requests are a sign that the client intents to read or write a larger portion of the file. The optimal solution would be to load exactly the amount of blocks needed by the client, since that would neither load too many blocks, nor make more requests to m3fs than necessary.
From a pure performance-oriented point of view, it seems that loading as few blocks as possible every time is the best practical solution, since loading too many blocks imposes a significantly larger overhead than additional requests to m3fs. Specifically, this can be seen when comparing a static limit of 2 and 512 blocks. However, splitting a file in minimal buffer cache entries basically enforcing a block-based cache has severe drawbacks. M^3 aims to let accelerators run autonomously. Should the client only get access to small portions of memory, it needs to make frequent requests to m3fs forcing m3fs to accompany the accelerator at any time. Figure 5.2 shows how frequent request to m3fs occur. This is tested with an already filled cache, since loading from the disk would add unnecessary overhead. As illustrated below, the dynamic solution performs significantly better compared to the static attempt, at least in regards to compute-heavy benchmarks like sort or sha256sum. Both the number and frequency of requests to m3fs is lower using the dynamic solution. This is definitely the

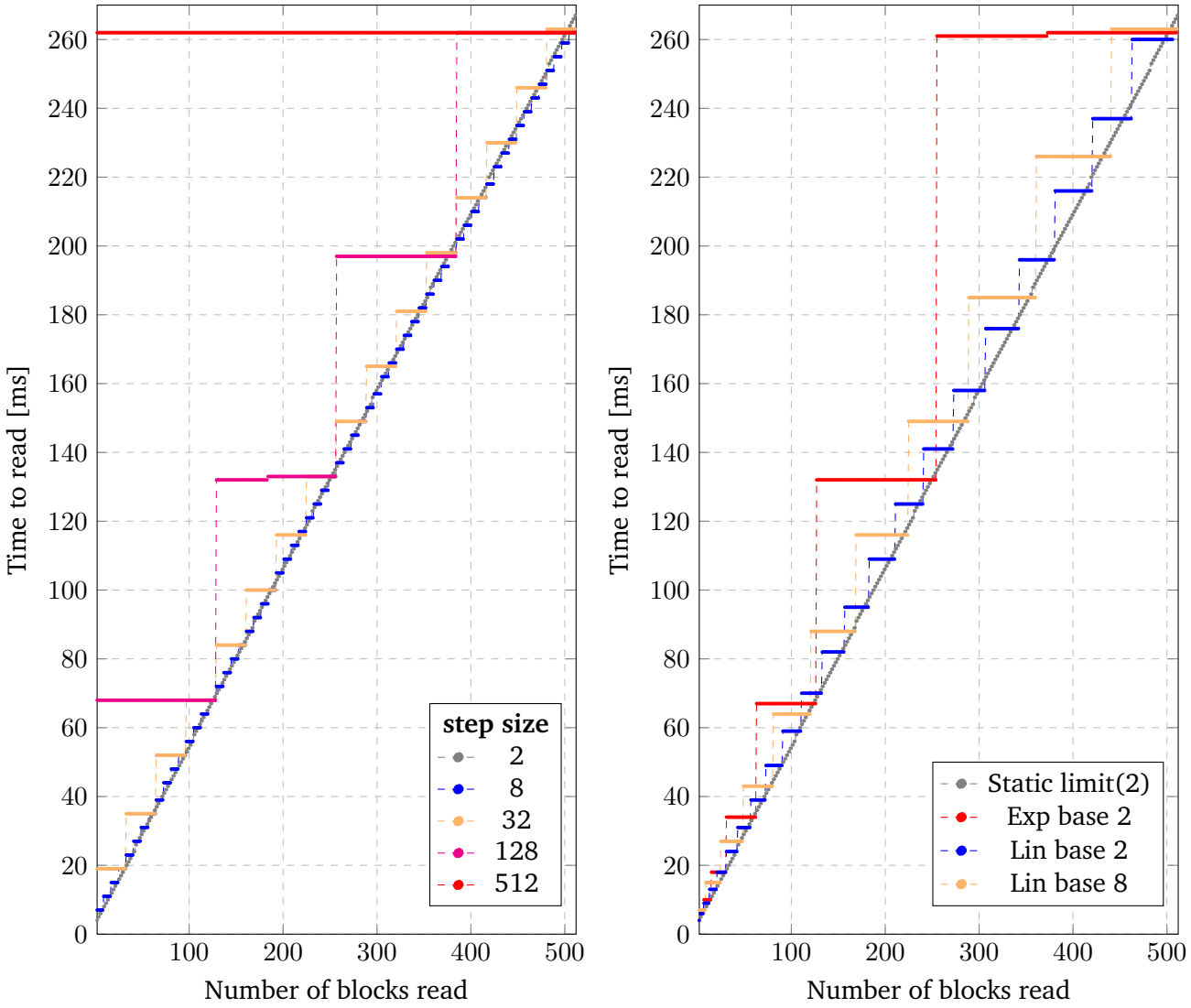

Figure 5.1: Loading time, using different limiting methods

preferred behavior as m3fs is suspended for a longer amount of time. For benchmarks that operate on smaller file segments, or rely on frequent m3fs requests, for example leveldb, sqlite and find, the difference is almost negligible. Since the client either does not get access to longer extents (find), or must frequently invoke other m3fs operations due to consistency reasons (sqlite, leveldb), both solutions perform similarly.

Whilst the static attempt performs better purely considering the load time, the dynamic one excels in real-life applications. Having larger buffer cache entries becomes even more advantageous regarding cache hits. Requests that can be satisfied without loading from the disk take the same time regardless of the amount of data provided. Based on the benchmarks, increasing the limit exponentially appears to be the best solution, as this shows a feasible behavior when loading small amounts of blocks as seen in figure 5.1, and performs on par or significantly better concerning m3fs-downtime.

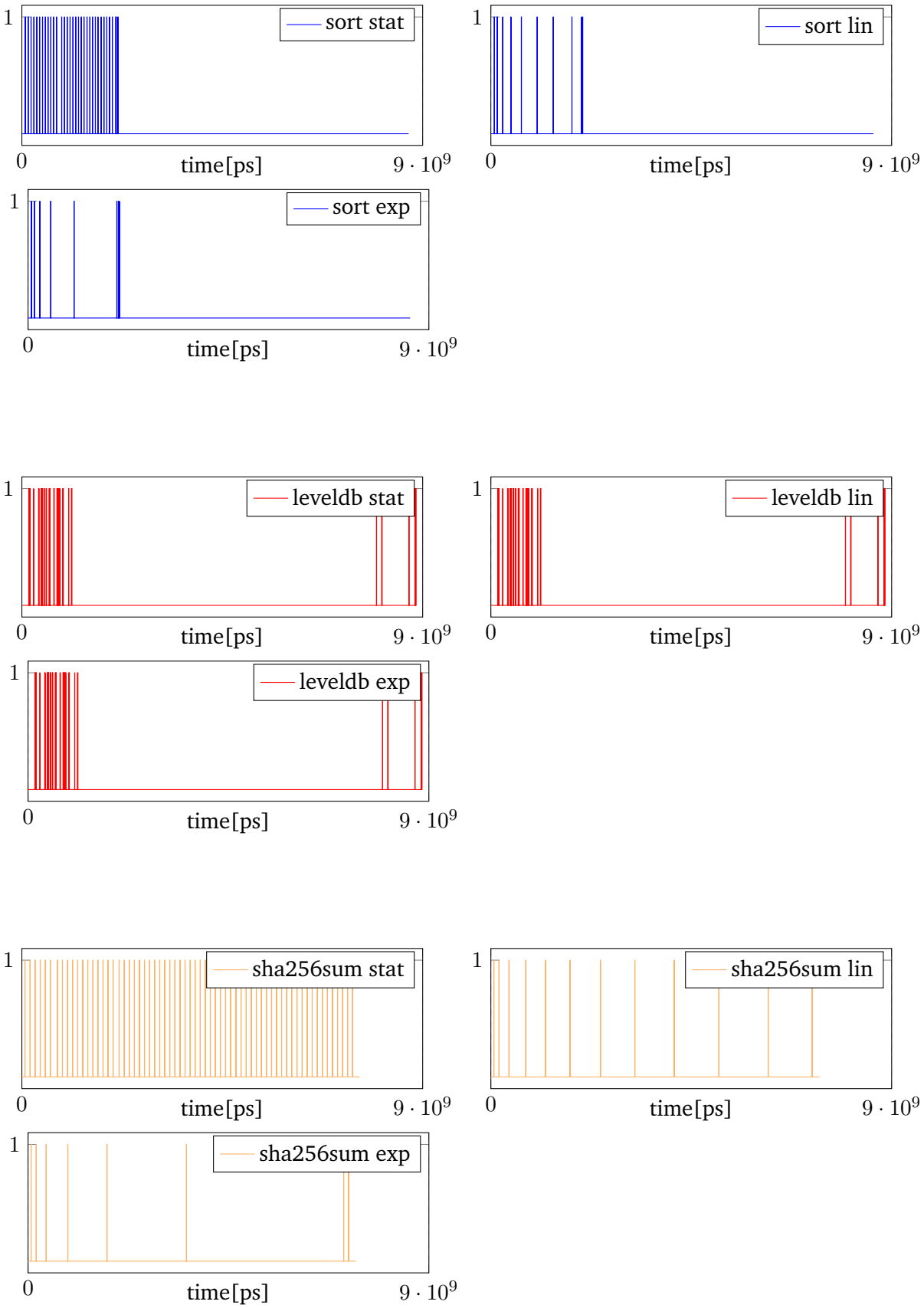

Figure 5.2: Frequency analysis of requests to m3fs

5.2 Parallel requests

The asynchronous design allows m3fs to handle other requests while a previous one waits for a load from the disk to finish. Since the thread executing the first requests is blocked and the disk server runs on a different PE, the performance of a second request should not be affected. The assumption is, that the second one can be served directly from the buffer cache.

mode	cycles/iteration	σ
alone	107533	± 0
parallel	108600	± 0

Table 5.1: Comparison of cache hits, with and without a parallel load from disk

In Table 5.1 is shown how long it takes a client to read a 27 KiB file. As expected, the difference between both modes are negligibly small. The slight difference of about 1000 cycles might be caused by hardware caches or additional entries in the buffer cache, which affect searching the treap.

5.3 Prototype vs. Current M3FS

With a fragmented buffer cache and the necessity of searching buffer cache entries, it is expected that the prototype does not perform as well as the in-memory version of m3fs. However, the overhead must be kept minimal since the file system is a frequently used service, that should be optimized.

trace	buffer lin2	buffer exp	in-memory
find	81468933	81954583	-
	13549836	14485825	6705697
sqlite	24897023	25346795	-
	13103449	13625845	5954521
sort	107296570	107251005	-
	390515	371454	223111
tar	6218841016	[1]	-
	17195783	[1]	14469981
untar	6217386164	[1]	-
	16001329	[1]	15083121
sha	200104550	200034006	-
	526186	481588	352926
leveldb	15861922	15885025	-
	1823331	1844843	1047061

Table 5.2: Average amount of cycles it takes the traces to complete, without the application time

In Figure 5.2, values for a cold cache and filled cache, regarding several benchmarks are presented. However, only the ones with the filled cache are worth comparing to the in-memory

file system. Read and write operations did not gain a substantial overhead, so compute heavy benchmarks like tar, untar and sha256sum performed quite well. Specifically tar and untar only gained an overhead of 18.8% and 6%. Benchmarks that frequently use the open or fstat operation, such as find, sqlite and leveldb performed below expectations. With an increase of 228.8%, sqlite performed the worst. Both open and find have to iterate through every block in a directory. That means, for every block the treap of the meta buffer cache must be searched. Specifically the open operation takes approximately 2.5 times longer and must be optimized further. It should be noted, that the prototype is built with an older version of m3fs. The newer version, to some degree allows file manipulations without the creation of a file session, which consumes a substantial amount of time.

[1] There is a unique bug occurring for these two benchmarks forcing the kernel to panic. Closing the written files is prevented by a fault in the pagetables, which technically are not in contact with the buffer cache. The missing values are expected to be lower than the ones for the linear version.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this thesis, one possible solution to extent m3fs for persistent storage was presented. The designed prototype as described in Chapter 3 uses the well-known buffer cache structure. It is modified to meet the design goals of M^3 and to be compatible with m3fs. The previous cache was replaced by a block-based meta buffer cache with similar functionality. A new addition is the extent-based file buffer cache, which stores file data in main memory, as the actual file system image has been moved to a disk. Out of several possible solutions, a design was chosen involving on-demand, dynamically sized file buffer entries. Both buffer caches are strictly separated, with the exception of directory blocks. To connect the provided disk driver to m3fs, a new asynchronous interface was introduced. User-level threads are used to hide high latency disk requests. Measurements presented in Chapter 5 have proven, that the average performance of the prototype is acceptable. In terms of improving energy efficiency through suspending m3fs, the results justify the chosen design, and verify that the specified upper limit is feasible. However, some operations like open gained an unfeasible overhead, and should be optimized as soon as possible.

6.2 Future Work

Apart from optimizing the above mentioned operations, other aspects of the prototype need to be adjusted as well. The disk driver must definitely be extended to enable parallel disk requests. Nesting the meta buffer cache inside the file buffer cache is a desirable adjustment to avoid copying blocks that can be used by both the clients and m3fs.
Considering that M^3 allows the usage of multiple m3fs instances, it would be interesting to find a solution for consistency problems. Two possible solutions worth investigating could be to either implement a consistency protocol while each m3fs instance has its own buffer cache, or to split the buffer cache from m3fs and moving it to its own service.
Now that persistent storage is possible, other features already implemented in well known file systems, like a journaling mechanism, or coherency protocols, may be of use. However, those features, just like the buffer cache, need to be adjusted to fit the requirements and purpose of M^3.

Bibliography

[Asm] Nils Asmussen. *Escape*. https://github.com/Nils-TUD/Escape. (last accessed: 02.09.2018).

[AVN+16] Nils Asmussen, Marcus Volp, Benedikt Nothen, Hermann Hartig, and Gerhard Fettweis. M3: A Hardware/Operating-System Co-Design to Tame Heterogeneous Manycores. *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 189–203, 2016.

[BEF+12] Francisco J. Ballesteros, Noah Evans, Charles Forsyth, Gorka Guardiola, Jim McKie, Ron Minnich, and Enrique Soriano-Salvador. NIX: A case for a manycore system for cloud computing. *Bell Labs Technical Journal*, 17:41–54, 2012.

[DGR+74] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, Oct 1974.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 20–43, Bolton Landing, NY, 2003.

[JS08] K. U. Järvinen and J. O. Skyttä. High-speed elliptic curve cryptography accelerator for koblitz curves. In *2008 16th International Symposium on Field-Programmable Custom Computing Machines*, pages 109–118, April 2008.

[LCL+15] Daofu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Teman, Xiaobing Feng, Xuehai Zhou, and Yunji Chen. Pudiannao: A polyvalent machine learning accelerator. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 369–381, New York, NY, USA, 2015. ACM.

[MCBD07] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, and Andreas Dilger. The new ext 4 filesystem : current status and future plans. *Proceedings of the Linux Symposium, Volume 2*, pages 21–33, 2007.

[TB15] Andrew Tanenbaum and Herbert Bos. *Modern Operating Systems*. Pearson plc, 4 edition, 2015.