Efficient Synchronization of Linux Memory Regions over a Network: A Comparative Study and Implementation (Notes)

A user-friendly approach to application-agnostic state synchronization

Felicitas Pojtinger (Stuttgart Media University)

2023-08-04

Introduction

Introduction

- Research question: Could memory be the universal way to access and migrate state?
- Why efficient memory synchronization is the missing key component
- High-level use cases for memory synchronization in the industry today

Technology

Page Faults

- Page faults occur when a process tries to access a memory region that has not yet been mapped into a process' address space
- By listening to these page faults, we know when a process wants to access a specific piece of memory
- We can use this to then pull the chunk of memory from a remote, map it to the address on which the page fault occured, thus only fetching data when it is required
- Usually, handling page faults is something that the kernel does
- In the past, this used to be possible by handling the SIGSEGV signal in the process

Delta Synchronization

- The probably most popular tool for file synchronization like this is rsync
- When the delta-transfer algorithm for rsync is active, it computes the difference between the local and the remote file, and then synchronizes the changes
- The delta sync algorithm first does file block division
- The file on the destination is divided into fixed-size blocks
- For each block in the destination, a weak and fast checksum is calculated
- The checksums are sent over to the source
- On the source, the same checksum calculation process is run, and compared against the checksums that were sent over (matching block identification)
- Once the changed blocks are known, the source sends over the offset of each block and the changed block's data to the destination

File Systems In Userspace (FUSE)

- In order to implement file systems in user space, we can use the FUSE API
- Here, a user space program registers itself with the FUSE kernel module
- This program provides callbacks for the file system operations, e.g. for open, read, write etc.
- When the user performs a file system operation on a mounted FUSE file system, the kernel module will send a request for the operation to the user space program, which can then reply with a response, which the FUSE kernel module will then return to the user
- This makes it much easier to create a file system compared to writing it in the kernel, as it can run in user space
- It is also much safer as no custom kernel module is required and an error in the FUSE or the backend can’t crash the entire kernel
- Unlike a file system implemented as a kernel module, this layer of

Network Block Device (NBD)

- NBD uses a protocol to communicate between a server (provided by user space) and a client (provided by the NBD kernel module)
- The protocol can run over WAN, but is really mostly meant for LAN or localhost usage
- It has two phases: Handshake and transmission
- There are two actors in the protocol: One or multiple clients, the server and the virtual concept of an export
- When the client connects to the server, the server sends a greeting message with the server's flags
- The client responds with its own flags and an export name (a single NBD server can expose multiple devices) to use
- The server sends the export's size and other metadata, after which the client acknowledges the received data and the handshake is complete
- After the handshake, the client and server start exchanging

Pre-Copy

- While these systems already allow for some optimizations over simply using the NBD protocol over WAN, they still mean that chunks will only be fetched as they are being needed, which means that there still is a guaranteed minimum downtime
- In order to improve on this, a more advanced API (the managed mount API) was created
- A field that tries to optimize for this use case is live migration of VMs
- Live migration refers to moving a virtual machine, its state and connected devices from one host to another with as little downtime as possible
- There are two types of such migration algorithms; pre-copy and post-copy migration
- Pre-copy migration works by copying data from the source to the destination as the VM continues to run (or in the case of a generic migration, app/other state continues being written to)

Post-Copy

- An alternative to pre-copy migration is post-copy migration
- In this approach, the VM is immediately suspended on the source, moved to the destination with only a minimal set of chunks
- After the VM has been moved to the destination, it is resumed
- If the VM tries to access a chunk on the destination, a page fault is raised, and the missing page is fetched from the source, and the VM continues to execute
- The benefit of post-copy migration is that it does not require re-transmitting dirty chunks to the destination before the maximum tolerable downtime is reached
- The big drawback of post-copy migration is that it can result in longer migration times, because the chunks need to be fetched from the network on-demand, which is very latency/RTT-sensitive

Workload Analysis

- “Reducing Virtual Machine Live Migration Overhead via Workload Analysis” provides an interesting analysis of options on how this decision of when to migrate can be made
- While being designed mostly for use with virtual machines, it could serve as a basis for other applications or migration scenarios, too
- The proposed method identifies workload cycles of VMs and uses this information to postpone the migration if doing so is beneficial
- This works by analyzing cyclic patters that can unnecessarily delay a VM’s migration, and identifies optimal cycles to migrate VMs in from this information
- For the VM use case, such cycles could for example be the GC of a large application triggering a lot of changes to the VMs memory etc.
- If a migration is proposed, the system checks for whether it is currently in a beneficial cycle to migrate in which case it lets the migration proceed; otherwise, it postpones it until the next cycle

Planning

Pull-Based Synchronization With userfaultfd

- In our case, we want to handle page faults in userspace
- In our case however, we can use a recent system called userfaultfd to do this in a more elegant way (available since kernel 4.11)
- userfaultfd allows handling these page faults in userspace
- The region that should be handled can be allocated with e.g. mmap
- Once we have the file descriptor for the userfaultfd API, we need to transfer this file descriptor to a process that should respond with the chunks of memory to be put into the faulting address
- Once we have received the socket we need to register the handler for the API to use
- If the handler receives an address that has faulted, it responds with the UFFDIO_COPY ioctl and a pointer to the chunk of memory that should be used on the file descriptor (code snippet from https://github.com/loopholelabs/userfaultfd-go/blob/master/pkg/mapper/handler.go)

Push-Based Synchronization With mmap and Hashing

- File-based synchronization
  - This approach tries to improve on userfaultfd by switching to push-based synchronization method
  - Instead of reacting to page faults, this uses a file to track changes to a memory region
  - By synchronizing the file representing the memory region between two systems, we can effectively synchronize the memory region itself
  - In Linux, swap space allows Linux to move chunks of memory to disk or other swap partition if the fast speed of RAM is not needed (“paging out”)
  - Similarly to this, Linux can also load missing chunks from a disk
  - This works similarly to how userfaultfd handled page faults, except this time it doesn’t need to go through user space, which can make it much faster
- Detecting file changes
  - inotify allows applications to register handlers on a file’s events, e.g. WRITE or SYNC. This allows for efficient file synchronization, and

Push-Pull Synchronization with FUSE

- File system-based synchronization in the kernel
  - Since the push method requires polling and is very CPU and I/O intensive, and userfaultfd-go has only low throughput, a better solution is needed
  - What if we could still get the events for the writes and reads without having to use userfaultfd-go or hashing?
  - We can create a custom file system in Linux and load it as a kernel module
  - This file system could then intercept reads/writes to/from the mmaped region, making it possible to respond to them with a custom backend
  - But such a system would need to run in the kernel directly, which leads to a lot of potential drawbacks
  - While it is possible to write kernel modules with Rust instead of C these days, a lot of problems remain
  - Kernel modules aren't portable; they are built for a specific kernel, which makes them hard to distribute to users

Pull-Based Synchronization With NBD

- Block Device-based synchronization
  - As hinted at before, a better API would be able to catch reads/writes to a single mmaped file instead of having to implement a complete file system
  - It does however not have to be an actual file, a block device also works
  - In Linux, block devices are (typically storage) devices that support reading/writing fixed chunks (blocks) of data
  - We can mmap a block device in the same way that we can mmap a file
  - Similarly to how a file system can be implemented in a kernel module, a block device is typically implemented as a kernel module/in kernel space
  - However, the same security, portability and developer experience issues as with the former also apply here
  - Instead of implementing a FUSE to solve this, we can create a NBD (network block device) server that can be used by the kernel NBD module similarly to how the process that connected to the FUSE

Push-Pull Synchronization with Mounts

- Limitations of the NBD protocol in WAN
  - Usually, the NBD server and client don’t run on the same system
  - NBD was originally designed to used as a LAN protocol to access a remote hard disk
  - As mentioned before, NBD can run over WAN, but is not designed for this
  - The biggest problem with running NBD over a public network, even if TLS is enabled is latency
  - Individual chunks would only be fetched to the local system as they are being accessed, adding a guaranteed minimum latency of at least the RTT
  - Instead of directly connecting a client to a remote server, we add a layer of indirection, called a Mount that consists of both a client and a server, both running on the local system
- Combining the NBD server and client to a reusable unit
  - We then connect the server to the backend with an API that is better suited for WAN usage

Pull-Based Synchronization with Migrations

- Optimization mounts for migration scenarios
  - We have now implemented a managed mounts API
  - This API allows for efficient access to a remote resource through memory
  - It is however not well suited for a migration scenario
  - For migrations, more optimization is needed to minimize the maximum acceptable downtime
  - For the migration, the process is split into two distinct phases
  - The same preemptive background pulls and parallelized device/syncer startup can be used, but the push process is dropped
  - The two phases allow pulling the majority of the data first, and only finalize the move later with the remaining data
  - This is inspired by the pre-copy approach to VM live migration, but also allows for some of the benefits of the post-copy approach as we'll see later
  - Why is this useful? A constraint for the mount-based API that we haven't mentioned before is that it doesn't allow safe concurrent

Implementation

Userfaults in Go with userfaultfd

- API design for userfault-go
  - Implementing this in Go was quite tricky, and it involves using unsafe
  - We can use the `syscall` and `unix` packages to interact with `ioctl` etc.
  - We can use the `ioctl` syscall to get a file descriptor to the userfaultfd API, and then register the API to handle any faults on the region (code snippet from https://github.com/loopholelabs/userfaultfd-go/blob/master/pkg/mapper/register.go#L15)
  - Passing file descriptors between processes is possible by using a UNIX socket (code snippet from https://github.com/loopholelabs/userfaultfd-go/blob/master/pkg/transfer/unix.go)
- Implementing userfaultfd backends
  - A big benefit of using userfaultfd and the pull method is that we are able to simplify the backend of the entire system down to a `io.ReaderAt` (code snippet from https://pkg.go.dev/io#ReaderAt)
  - That means we can use almost any `io.ReaderAt` as a backend for a userfaultfd-go registered object

File-Based Synchronization

- File-based synchronization
  - We can do this by using mmap, which allows us to map a file into memory
  - By default, mmap doesn't write changes from a file back into memory, no matter if the file descriptor passed to it would allow it to or not
  - We can however add the MAP_SHARED flag; this tells the kernel to write back changes to the memory region to the corresponding regions of the backing file
  - Linux caches reads to such a backing file, so only the first page fault would be answered by fetching from disk, just like with userfaultfd
  - The same applies to writes; similar to how files need to be synced in order for them to be written to disks, mmaped regions need to be msynced in order to flush changes to the backing file
  - In order to synchronize changes to the region between hosts by syncing the underlying file, we need to have the changes actually be represented in the file, which is why msync is critical
  - For files, you can use O_DIRECT to skip this kernel caching if your

FUSE Implementation in Go

- It is possible to use even very complex and at first view non-compatible backends as a FUSE file system's backend
- By using a file system abstraction API like afero.Fs, we can separate the FUSE implementation from the actual file system structure, making it unit testable and making it possible to add caching in user space (code snippet from https://github.com/pojntfx/stfs/blob/main/pkg/fs/file.go)
- It is possible to map any afero.Fs to a FUSE backend, so it would be possible to switch between different file system backends without having to write FUSE-specific (code snippet from https://github.com/JakWai01/sile-fystem/blob/main/pkg/filesystem/fs.go)
- For example, STFS used a tape drive as the backend, which is not random access, but instead append-only and linear (https://github.com/pojntfx/stfs/blob/main/pkg/operations/update.go)

NBD with go-nbd

- Implementing go-nbd
  - Due to the lack of pre-existing libraries, a new pure Go NBD library was implemented
  - This library does not rely on CGo/a pre-existing C library, meaning that a lot of context switching can be skipped
  - The backend interface for go-nbd is very simple and only requires four methods: ReadAt, WriteAt, Size and Sync
  - A good example backend that maps well to a block device is the file backend (code snippet from https://github.com/pojntfx/go-nbd/blob/main/pkg/backend/file.go)
  - The key difference here to the way backends were designed in userfaultfd-go is that they can also handle writes
  - go-nbd exposes a Handle function to support multiple users without depending on a specific transport layer (code snippet from https://github.com/pojntfx/go-nbd/blob/main/pkg/server/nbd.go)
  - This means that systems that are peer-to-peer (e.g. WebRTC), and thus don't provide a TCP-style accept syscall can still be used easily

Chunking, Push/Pull Mechanisms and Lifecycle for Mounts

- The ReadWriterAt pipeline
  - In order to implement the chunking system, we can use a abstraction layer that allows us to create a pipeline of readers/writers - the ReadWriterAt, combining an io.ReaderAt and a io.WriterAt
  - This way, we can forward the Size and Sync syscalls directly to the underlying backend, but wrap a backend's ReadAt and WriteAt methods in a pipeline of other ReadWriterAts
  - One such ReadWriterAt is the ArbitraryReadWriterAt (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/chunks/arbitrary_rwat.go)
  - It allows breaking down a larger data stream into smaller chunks
  - In ReadAt, it calculates the index of the chunk that the offset falls into and the position within the offsets
  - It then reads the entire chunk from the backend into a buffer, copies the necessary portion of the buffer into the input slice, and repeats the process until all requested data is read
  - Similarly for the writer, it calculates the chunk's index and offset
  - If an entire chunk is being written to, it bypasses the chunking system,

Live Migration for Mounts

- Optimization mounts for migration scenarios
  - The flexible architecture of the ReadWriterAt components allow the reuse of lots of code for both use cases
- The migration protocol
  - To fix this, the migration API defines two new actors: The seeder and the leecher
  - The seeder represents a resource that can be migrated from/a host that exposes a migratable resource
  - The leecher represents a client that wants to migrate a resource to itself
  - Initially, the protocol starts by running an application with the application's state on the seeder's mount
  - When a leecher connects to the seeder, the seeder starts tracking any writes to it's mount
  - The leecher starts pulling chunks from the seeder to it's local backend
  - Once the leecher has received a satisfactory level of locally available chunks, it as the seeder to finalize, which then causes the seeder to

Pluggable Encryption and Authentication

- Compared to existing remote mount and migration solutions, r3map is a bit special
- As mentioned before, most systems are designed for scenarios where such resources are accessible in a high-bandwidth, low-latency LAN
- This means that some assumptions concerning security, authentication, authorization and scalability were made that can not be made here
- For example encryption; while for a LAN deployment scenario it is probably assumed that there are no bad actors in the subnet, the same can not be said for WAN
- While depending on e.g. TLS etc. for the migration could have been an option, r3map should still be useful for LAN migration use cases, too, which is why it was made to be completely transport-agnostic
- This makes adding encryption very simple
- E.g. for LAN, the same assumptions that are being made in existing

Optimizing Backends For High RTT

- In WAN, where latency is high, the ability to fetch chunks concurrently is very important
- Without concurrent background pulls, latency adds up very quickly as every memory request would have at least the RTT as latency
- The first prerequisite for supporting this is that the remote backend has to be able to read from multiple regions without locking the backend globally
- For the file backend for example, this is not the case, as the lock needs to be acquired for the entire file before an offset can be accessed (code snippet from https://github.com/pojntfx/go-nbd/blob/main/pkg/backend/file.go#L17-L25)
- For high-latency scenarios, this can quickly become a bottleneck
- While there are many ways to solve this, one is to use the directory backend
- Instead of using just one backing file, the directory backend is a

Optimizing The Transport Protocol For Throughput

- Despite these benefits, gRPC is not perfect however
- Protobuf specifically, while being faster than JSON, is not the fastest serialization framework that could be used
- This is especially true for large chunks of data, and becomes a real bottleneck if the connection between source and destination would allow for a high throughput
- This is where fRPC, a RPC library that is easy to replace gRPC with, becomes useful
- Because throughput and latency determine the maximum acceptable downtime of a migration/the initial latency for mounts, choosing the right RPC protocol is an important decision
- fRPC also uses the same proto3 DSL, which makes it an easy drop-in replacement, and it also supports multiplexing and connection polling
- Because of these similarities, the

Using Remote Stores as Backends

- Using key-value stores as ephemeral mounts
  - RPC backends provide a way to access a remote backend
  - This is useful, esp. if the remote resource should be protected in some way or if it requires some kind of authorization
  - Depending on the use case however, esp. for the mount API, having access to a remote backend without this level of indirection can be useful
  - Fundamentally, a mount maps fairly well to a remote random-access storage device
  - Many existing protocols and systems provide a way to access essentially this concept over a network
  - One of these is Redis, an in-memory key-value store with network access
  - Chunk offsets can be mapped to keys, and bytes are a valid key type, so the chunk itself can be stored directly in the KV store (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/backend/redis.go#L36-)

Bi-Directional Protocols With Dudirekta

- Another aspect that plays an important role in performance for real-life deployments is the choice of RPC framework and transport protocol
- As mentioned before, both the mount and the migration APIs are transport-independent
- A simple RPC framework to use is dudirekta
- Dudirekta is reflection-based, which makes it very simple to use to iterate on the protocol quickly
- To use it, a simple wrapper struct with the needed RPC methods is created (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/services/backend.go#L41-L61)
- This wrapper struct simply calls the backend (or seeder etc.) functions
- The wrapper struct is then passed as the local function struct into a

Connection Pooling For High RTT Scenarios

- This does however come at the cost of not being able to do connection pooling, since each client dialing the server would mean that the server could not reference the multiple client connections as one composite client without changes to the protocol
- While implementing such a pooling mechanism in the future could be interesting, it turned out to not be necessary thanks to the pull-based pre-copy solution described earlier
- Instead, only calling RPCs exposed on the server from the client is the only requirement for an RPC framework, and other, more optimized RPC frameworks can already offer this
- Dudirekta uses reflection to make the RPCs essentially almost transparent to use
- By switching to a well-defined protocol with a DSL instead, we can gain further benefits from not having to use reflection and generating code instead

Results

Userfaults

- Benchmark: Sensitivity of userfaultfd to network latency and throughput

File-Based Synchronization

- Benchmark: Performance of different Go hashing algorithms for detecting writes
- Benchmark: Hashing the chunks individually vs. hashing the entire file
- Benchmark: Throughput of this custom synchronization protocol vs. rsync (which hashes entire files)

Mounts and Live Migration

- Benchmark: Local vs. remote chunking
- Benchmark: Parallelizing startups and pulling n MBs as the device starts
- Benchmark: File vs. directory backend performance
- fRPC is 2-4x faster than gRPC, and especially in terms of throughput (insert graphics from https://frpc.io/performance/grpc-benchmarks)
- Benchmark: Effect of tuning the amount of push/pull workers in high-latency for these three backends on latency till first n chunks and throughput
- Benchmark: These three backends on localhost and on remote hosts, where they could be of use
- Benchmark: Latency and throughput of all benchmarks on localhost and in a realistic latency and throughput scenario (direct mounts can outperform managed mounts in tests on localhosts)
- Benchmark: Maximum acceptable downtime for a migration scenario

Discussion

Userfaults

- As we can see, using userfaultfd we are able to map almost any object into memory
- This approach is very clean and has comparatively little overhead, but also has significant architecture-related problems that limit its uses
- The first big problem is only being able to catch page faults - that means we can only ever respond the first time a chunk of memory gets accessed, all future requests will return the memory directly from RAM on the destination host
- This prevents us from using this approach for remote resources that update over
- Also prevents us from using it for things that might have concurrent writers/shared resources, since there would be no way of updating the conflicting section
- Essentially makes this system only usable for a read-only “mount” of

File-Based Synchronization

- Similarly to userfaultfd, this system also has limitations
- While userfaultfd was only able to catch reads, this system is only able to catch writes to the file
- Essentially this system is write-only, and it is very inefficient to add hosts to the network later on
- As a result, if there are many possible destinations to migrate state too, a star-based architecture with a central forwarding hub can be used
- The static topology of this approach can be used to only ever require hashing on one of the destinations and the source instead of all of them
- This way, we only need to push the changes to one component (the hub), instead of having to push them to each destination on their own
- The hub simply forwards the messages to all the other destinations

FUSE

- FUSE does however also have downsides
- It operates in user space, which means that it needs to do context switching
- Some advanced features aren't available for a FUSE
- The overhead of FUSE (and implementing a completely custom file system) for synchronizing memory is still significant
- If possible, the optimal solution would be to not expose a full file system to track changes, but rather a single file
- As a result of this, the significant implementation overhead of such a file system led to it not being chosen

NBD

- Limitations of NBD and ublk as an alternative
  - NBD is a battle-tested solution for this with fairly good performance, but in the future a more lean implemenation called ublk could also be used
  - ublk uses io_uring, which means that it could potentially allow for much faster concurrent access
  - It is similar to NBD; it also uses a user space server to provide the block device backend, and a kernel ublk driver that creates /dev/ublkb* devices
  - Unlike as it is the case for the NBD kernel module, which uses a rather slow UNIX or TCP socket to communicate, ublk is able to use io_uring pass-through commands
  - The io_uring architecture promises lower latency and better throughput
  - Because it is however still experimental and docs are lacking, NBD was chosen
- BUSE and CUSE as alternatives to NBD

Mounts and Live Migration

- Overhead of using managed mounts
  - Another interesting aspect of optimization to look at is the overhead of managed mounts
  - It is possible for managed mounts (and migrations) to deliver signficantly lower performance compared to direct mounts
  - This is because using managed mounts come with the cost of potential duplicate I/O operations
  - For example, if memory is being accessed linearly from the first to the last offset immediately after it being mounted, then using the background pulls will have no effect other than causing a write operation (to the local caching backend) compared to just directly reading from the remote backend
  - This however is only the case in scenarios with a very low latency between the local and remote backends
  - If latency becomes higher, then the ability to pull the chunks in the background and in parallel with the puller will offset the cost of duplicate I/O

Remote Swap With ram-dl

- ram-dl is a fun experiment
- Tech demo for r3map
- Uses the fRPC backend to expand local system memory
- Can allow mounting a remote system's RAM locally
- Can be used to inspect a remote system's memory contents
- Is based on the direct mount API
- Uses mkswap, swapon and swapoff (code snippet from https://github.com/pojntfx/ram-dl/blob/main/cmd/ram-dl/main.go#L170-L190)
- Enables paging out to the block device provided by the direct mount API
- ram-ul "uploads" RAM by exposing a memory, file or directory-backed file over fRPC
- ram-dl then does all of the above
- Not really intended for real-world usecases, but does show that this

Mapping Tape Into Memory With tapisk

- tapisk is an interesting usecase because of how close it is to STFS, which provided the inspiration for the FUSE-based approach
- Very high read/write backend latency (multiple seconds, up to 90s, due to seeking)
- Linear access, no random reads
- Can benefit a lot from asynchronous writes provided by managed mounts
- Fast storage acts as a caching layer
- Backend is linear, so only one read/write possible at a time
- With local backend, writes are de-duplicated automatically and both can be asynchronous/concurrent
- Writes go to fast (“local”) backend first, syncer then handles in both directions
- Chunking works on tape drive records and blocks
- Only one concurrent reader/writer makes sense

Improving File Storage Solutions

- Another potential usecase is using r3map to create a mountable remote filesystem with unique properties
- Currently there are two choices on how these can be implemented
- Google Drive, Nextcloud etc. listen to file changes on a folder and synchronize files when they change
- The big drawback is that everything needs to be stored locally
- If a lot of data is stored (e.g. terabytes), the locally available files would need to be manually selected
- There is no way to dynamically download files this way as they are required
- It is however very efficient, since the filesystem is completely transparent to the user (writes are being synced back asynchronously)
- It also supports an offline usecase easily
- The other option is to use a FUSE, e.g. s3-fuse

Universal Database, Media and Asset Streaming

- Another usecase is accessing a remote database locally
- While using a database backend is one option of storing chunks, an actual database can also be stored in a mount as well
- Particularly interesting for in-memory or on-disk databases like SQLite
- Instead of having to download the entire SQLite database before using it, it can simply be mounted, and accessed as it is being used
- This allows very efficient network access, as only very rarely the entire database is needed
- Since reads are cached with the managed mount API, only the first read should potentially have a performance impact
- Similarly so writes to the database will be more or less the same throughput as to the local disk, since changes are written back asynchronously
- If the full database should eventually be accessible locally, the

Universal App State Synchronization and Migration

- Synchronization of app state is hard
- Even for hand-off scenarios a custom protocol is built most of the times
- It is possible to use a database sometimes (e.g. Firebase) to synchronize things
- But that can't sync all data structures and requires using specific APIs to sync things
- What if you could mount and/or migrate any ressource?
- Usually these structures are marshalled, sent over a network, received on a second system, unmarshalled, and then are done being synced
- Requires a complex sync protocol, and when to sync, when to pull etc. is inefficient and usually happens over a third party (e.g. a database)
- Data structures can almost always be represented by am `[]byte`
- If the data structures are allocated from on a `[]byte` from the block

Summary

Summary

- Looking back at all synchronization options and comparing ease of implementation, CPU load and network traffic between them
- Summary of the different approaches, and how the new solutions might make it possible to use memory as the universal access format

Conclusion

Conclusion

- Answer to the research question (Could memory be the universal way to access and migrate state?)
- What would be possible if memory became the universal way to access state?
- Further research recommendations (e.g. ublk)

References

Structure

- Abstract
- Tables
  - Content
  - Graphics
  - Abbrevations
- Introduction: Research question/goal
- Technology: Current state of research and introductions to related topics (like a reference book)
- Planning: Ideas on how to solve the research question/goal
- Implementation: Description of how the ideas were implemented
- Results: Which results each of the methods yielded (i.e. benchmarks)
- Discussion: How the individual results can be interpreted and what use cases there are for the methods
- Summary: How the discussion can be interpreted as a whole
- Conclusion: Answer to the research question and future outlook, future research questions

Sources

- FluidMem: Full, Flexible, and Fast Memory Disaggregation for the Cloud (userfaultfd)
- Memory Disaggregation: Advances and Open Challenges (general memory hierarchy etc.)
- User Level Page Faults (sigaction & signal-based page fault handlers)
- Towards Web-based Delta Synchronization for Cloud Storage Services (rsync)
- To FUSE or Not to FUSE: Performance of User-Space File Systems (implementing a FUSE)
- The NBD protocol (NBD protocol documentation from the sources; info on implementing a NBD server and client)
- Remote regions: a simple abstraction for remote memory (Kernel filesystem for remote memory access)
- Reducing Virtual Machine Live Migration Overhead via Workload Analysis (when to trigger the finalization process for a live VM