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 for it to be that universal way
- High-level use cases for memory synchronization in the industry today (VM live migration, accessing remote ressources transparently)

Technology

The Linux Kernel

- Open-Source kernel created by Linus Torvals in 1991
- Written in C, with recently Rust being added as an additional allowed language[1]
- Powers millions of devices worldwide (servers, desktops, phones, embedded devices)
- Is a bridge between applications and hardware
  - Provides an abstraction layer
  - Compatible with many architectures (ARM, x86, RISC-V etc.)
- Is not an operating system in itself, but is usually made an operating system by a distribution[2]
- Distributions add userspace tools (e.g. GNU coreutils or BusyBox), desktop environments and more, turning into a full operating system
- Is a good fit for this thesis due to it's open-source nature (allows anyone to view, modify and contribute to the source code)

Linux Kernel Modules

- Linux kernel is monolithic, but extensible thanks to kernel modules[2]
- Small pieces of kernel-level code that can be loaded and unloaded as kernel modules
- Can extend the kernel functionality without reboots
- Are dynamically linked into the running kernel
- Helps keep kernel size manageable and maintainable
- Kernel modules are written in C
- Interact with kernel through APIs
- Poorly written modules can cause signficant kernel instability
- Modules can be loaded at boot time or dynamically (modprobe, rmmod etc.)[3]
- Module lifecycle can be implemented with initialization and cleanup functions

UNIX Signals and Handlers

- Signals
  - Are software interrups that notify a process of important events (exceptions etc.)
  - Can originate from the kernel, user input or different processes
  - Function as an asynchronous communication mechanism between processes or the kernel and a process
  - Have default actions, i.e. terminating the process or ignoring a signal[4]
- Handlers
  - Can be used to customize how a process should respond to a signal
  - Can be installed with sigaction()[5]
- Signals are not designed as an IPC mechanism, since they can only alert of an event, but not of any additional data for it

Memory Hierarchy

- Memory in computers can be classified based on size, speed, cost and proximity to the CPU
- Principle of locality: The most frequently accessed data and instructions should be in the closest memory[6]
- Locality is important mostly due to the “speed of the cable” - throughput (due to dampening) and latency (due to being limited by the speed of light) decreases as distance increases
- Registers
  - Closest to the CPU
  - Very small amount of storage (32-64 bits of data)
  - Used by the CPU to perform operations
  - Very high speed, but limited in storage size
- Cache Memory
  - Divided into L1, L2 and L3
  - The higher the level, the larger and less expensive a layer
  - Buffer for frequently accessed data

Memory Management in Linux

- Memory management is a crucial part of every operation system - maybe even the whole point of an operating system
- Creates buffer between applications and physical memory
- Can provide security guarantees (e.g. only one process can access it's memory)
- Kernel space
  - Runs the kernel, kernel extensions, device drivers
  - Managed by the kernel memory module
  - Uses slab allocation (groups objects of the same size into caches, speeds up memory allocation, reduces fragmentation of the memory)[8]
- User space
  - Applications (and some drivers) store their memory here[9]
  - Managed through a paging system
  - Each application has it's own private virtual address space
  - Virtual address space divided into pages of 4 KB

Swap Space

- A portion of the secondary storage is for virtual memory[9]
- Essential for systems running multiple applications
- Moves inactive parts of ram to secondary storage to free up space for other processes
- Implementation in Linux
    - Linux uses a demand paging system: Memory is only allocated when it is needed
    - Can be either a swap partition (separate area of the secondary storage) or file (regular file that can be expanded/trucnated)
    - Swap paritions and files are transparent to use
    - Kernel uses a LRU algoruithm for deciding which pages
- Role in hiberation
    - Before hibernating, the system saves the content of RAM into swap (where it is persistent)
    - When resuming, memory is read back from swap
- Role on performance

Page Faults

- Page faults occur when the process tries to access a page not available in primary memory, which ause the OS to swap the required page from secondary storage into primary memory[3]
- Types
  - Minor page faults: Page is already in memory, but not linked to the process that needs it
  - Major page fault: Needs to be loaded from secondary storage
- The LRU (and simpler clock algorithm) can minimize page faults
- Techniques for handling page faults
  - Prefetching: Anticipating future page requests and loading them into memory in advance
  - Page compression: Compressing inactive pages and storing them in memory pre-emptively (so that less major faults happen)[10]
- Usually, handling page faults is something that the kernel does

mmap

- Overview
  - UNIX system call for mapping files or devices into memory
  - Multiple possible usecases: Shared memory, file I/O, fine-grained memory allocation
  - Commonly used in applications like databases
  - Is a “power tool”, needs to be used carefully and intentionally
- Functionality
  - Establishes direct link (memory mapping) between a file and a memory region[11]
  - When the system reads from the mapped memory region, it reads from the file directly and vice versa
  - Reduces overhead since no or less context switches are needed
- Benefits:
  - Enables zero-copy operations: Data can be accessed directly as though it were in memory, without having to copy it from disk first
  - Can be used to share memory between processes without having to go through the kernel with syscalls[4]

inotify

- Event-driven notification system of the Linux kernel[12]
- Monitors file system for events (i.e. modifications, access etc.)
- Uses a watch feature for monitoring specific events, e.g. only watching writes
- Reduces overhead and ressource use compared to polling
- Widely used in many applications, e.g. Dropbox for file synchronization
- Has limitations like the limit on how many watches can be established

Linux Kernel Disk and File Caching

- Disk caching
  - Temporarely stores frequently accessed data in RAM
  - Uses principle of locality (see Memory Hierarchy)
  - Implemented using the page cache subsystem in Linux
  - Uses the LRU algorithm to manage cache contents
- File caching
  - Linux caches file system metadata in the dentry and inode caches
  - Metadata includes i.e. file names, attributes and locations
  - This caching accelerates the resoluton of path names and file attributes (i.e. the last change data for polling)
  - File reads/writes pass through the disk cache
- Complexities
  - Data consistency: Between the disk and cache via writebacks. Aggressive writebacks lead to reduced performance, delays risk data loss
  - Release of cached data under memory pressure: Cache eviction requires intelligent algorithms, i.e. LRU[3]

TCP, UDP and QUIC

- TCP
  - Connection-oriented
  - Has been the reliable backbone of internet commmunication
  - Guaranteed delivery and maintained data order
  - Includes error checking, lost packet retransmission, and congestion control mechanisms
  - Powers the majority of the web[13]
- UDP
  - Connectionless
  - No reliability or ordered packet delivery guarantees
  - Faster than TCP due to less guarantees
  - Suitable for applications that require speed over reliability (i.e. online gaming, video calls etc.)[14]
- QUIC
  - Modern transport layer protocol developed by Google and standardized by the IETF in 2020
  - Intents to combine the best aspects of TCP and UDP

Delta Synchronization

- Traditionally, when files are synchronized between hosts, the entire file is transfered
- Delta synchronization is a technique that intents to instead transfer only the part of the file that has changed
- Can lead to reduced network and I/O overhead
- The probably most popular tool for file synchronization like this is rsync
- When a delta-transfer algorithm is used, 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

File Systems In Userspace (FUSE)

- Software interface that allows writing custom file systems in userspace
- Developers can create file systems without having to engage in low-level kernel development
- Available on multiple platforms, mostly Linux but also macOS and FreeBSD
- 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,

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 designed for LAN or localhost usage
- It has two phases: Handshake and transmission[18]
- There are multiple 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[19]

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[20]
- 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

Streams and Pipelines

- Fundamental concepts in computer science
- Sequentually process elements
- Allow for the efficient processing of large amounts of data, without having to load everything into memory
- Form the backbone of efficient, modular data processing
- Streams
    - Represent a continous sequence of data
    - Can be a source or destination of data (i.e. files, network connections, stdin/stdout etc.)
    - Allow processing of data as it becomes available
    - Minimized memory consumption
    - Especially well suited for long-running processes (where data gets streamed in for a extended time)[21]
- Pipelines
    - Series of data processing stages: Output of one stage serves as input to the next[22]

gRPC

- Open-Source and high-performance RPC framework
- Developed by Google in 2015
- Features
  - Uses HTTP/2 as the transport protocol to benefit from header compression and request multiplexing
  - Uses protobuf as the IDL and wire format, a high-performance, polyglot mechanism for data serialization (instead of the slower and more verbose JSON of REST APIs)
  - Supports unary RPCs, server-streaming RPCs, client-streaming RPCs and bidirectional RPCs
  - Has pluggable support for load balancing, tracing, health checking and authentication[23]j
- Supports many languages (Go, Rust, JS etc.)
- Developed by the CNCF

Redis

- In-memory data structure store
- Used as a database, cache and/or message broker
- Created by S. Sanfilippo in 2009
- Different from other NoSQL databases by supporting various data structures like lists, sets, hashes or bitmaps
- Uses in-memory data storage for maximum speed and efficiency[24]
- Allows for low-latency reads/writes
- While not intended for persistance, it is possible to store data on disk
- Has a non-blocking I/O model and offers near real-time data processing capabilities
- Includes a pub-sub system to be able to function as a message broker[25]

S3 and Minio

- S3
  - Object storage service for data-intensive workloads
  - Offered by AWS
  - Can be globally distributed to allow for fast access times from anywhere on the globe
  - Range of storage classes with different requirements
  - Includes authentication and authorization
  - Exposes HTTP API for accessing the stored folders and files[26]
- Minio
  - Open-source storage server compatible wiht S3
  - Lightweight and simple, written in Go
  - Can be hosted on-prem and is open source
  - Allows horizontal scalability to storage large amounts of data across nodes[27]

Cassandra and ScylllaDB

- Popular wide-column NoSQL databases
- Combines Amazon's Dynamo model and Google's Bigtable model to create a highly available database
- Apache Cassandra
    - Highly scalable, eventually consistent
    - Can handle large amounts of data across many servers with no single point of failure
    - Consistency can be tuned according to needs (eventual to strong)
    - Doesn't use master nodes due to it's use of a P2P protocol and distributed hash ring design
    - Does have high latency under heavy load and requires fairly complex configuration[28]
- ScyllaDB
    - Launched in 2015
    - Written in C++ and has a shared-nothing architecture, unlike Cassandra which is written in Java
    - Compatible with Cassandra's API and data model[29]

Planning

Pull-Based Synchronization With userfaultfd

- An implementation of post-copy migration
- Memory region is created on the destination host
- Reads to the region by the migrated ressource trigger a page fault
- When we encounter such a page fault, we want to fetch the relevant offset from the remote
- Typically, page faults are resolved by the Kernel
- Here, we want to handle handle the page faults in user space
- Traditionally, it was possible to use SIGSEGV signal handlers to use handle this from the program
- This however is complicated and slow
- Instead we use a new kernel API (“Userfaults”)
- We register the region to be handled by userfault
- Then we start an handler that fetches the offsets from the remote on a page fault
- This handler is connected to the registered region using a file

Push-Based Synchronization With mmap and Hashing

- As mentioned before mmap allows mapping a memory region to a file
- We can put the migratable ressource into this memory region
- If we get writes to the region, we eventually get writes to the region
- If we're able to detect these writes and copy them to the destination, we can implement a pre-copy migration system
- mmaped regions still using caching to speed up reads
- Changes from the region can be flushed to the disk with msync
- Usually we could use inotify to detect changes to the file, but inotify doesn't work with mmaped regions
- As a result, we went for a polling-based solution instead
- Polling has drawbacks, which we tried to work around upon in our implementation

Push-Pull Synchronization with FUSE

- Can serve as the basis for a pre- or post-copy migration
- Similarly to the file-based synchronization we mmap a file into memory
- The file is stored on a custom file system
- We then catch reads (for a post-copy migration) or writes (for a pre-copy migration)
- Implementing a file system in the kernel is possible but cumbersome as described before
- With FUSE we can implement the file system in user space, making this much simpler, as we'll show in the implementation

Mounts with NBD

- Another mmap-based solution for pre- and/or post-copy migration
- Instead of mmaping a file, a device is mmaped
- This device is a block device provided by a NBD client
- We can then connect the NBD client to a remote NBD server, which contains the migratable resource
- Any reads to/from the region go to/from the block device and are resolved by the remote NBD server
- Isn't per se a synchronization methods on it's own, but rather a remote mount
- Unlike a FUSE this means we only need to implement a block device, not a full file system
- Implementation and performance overhead of a NBD server is fairly low as we'll show in the implementation however

Push-Pull Synchronization with Mounts

- Also tracks changes to the memory region of the migratable ressource using NBD
- 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

Pull-Based Synchronization with Migrations

- Also tracks changes to the memory region of the migratable ressource using NBD
- 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

Implementation

Userfaults in Go with userfaultfd

- General functionality
  - By listening to 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 our case, we want to handle page faults in userspace, and implement post-copy with them
  - In the past, this used to be possible from userspace by handling the SIGSEGV signal in the process
  - 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

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

- Implementing a FUSE in Go means tight integration with libraries
- It makes sense to divide the process into two aspects
- Creating a backend for a file system abstraction API like afero.Fs
  - 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 use even very complex and at first view non-compatible backends as a FUSE file system's backend
  - 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)
  - By using an on-disk index and access optimizations, the resulting file system was still performant enough to be used, and supported almost all features required for the average user
  - Creating an adapter between the FS abstraction API and the FUSE

NBD with go-nbd

- Overview
  - 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
- Server
  - The server is completely in user space, there are no kernel components involved here
  - The backend interface for go-nbd is very simple and only requires four methods: ReadAt, WriteAt, Size and Sync
  - The key difference here to the way backends were designed in userfaultfd-go is that they can also handle writes
  - 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)
  - go-nbd exposes a Handle function to support multiple users without depending on a specific transport layer (code snippet from

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

- Overview
  - As mentioned in Pull-Based Synchronization with Migrations before, the mount API is not optimal for a migration scenario
  - Splitting the migration into two separate phases can help a lot to fix the biggest problem, the maximum guaranteed downtime
  - The flexible architecture of the ReadWriterAt components allow the reuse of lots of code for both the mount API and the migration API
- Implementing the seeder
  - The seeder defines a simple read-only RPC API with the familiar ReadAt methods, but also new APIs such as returning dirty chunks from Sync and adding a Track method (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/services/seeder.go#L15-L21)
  - Unlike the remote backend, a seeder also exposes a mount through a path, file or byte slice, so that as the migration is happening, the underlying data can still be accessed by the application
  - This fixes the issue that the mount API had for migrations, where only

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

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 and Concurrent RPCs with Dudirekta

- Overview of the framework and why a custom one was implemented
  - Designed specifically with the hybrid pre- and post-copy scenario in mind
  - Support for concurrent RPCs allows for efficient background pulls since multiple chunks can be pulled at the same time
  - Bi-directional API makes it possible to initiate pre-copy migrations and transfer chunks from the source host without having to make the destination host dialable
    - Dudirekta supports defining functions on both the client and the server
    - This is very useful for implementing e.g. a pre-copy protocol where the source pushes chunks to the destination by simply calling a RPC on the destination, instead of the destination calling a RPC on the source
    - Usually, RPCs don't support exposing or calling RPCs on the client, too, only on the server
    - This would mean that in order to implement a pre-copy protocol with pushes, the destination would have to be dialable from the source
    - In a LAN scenario, this is easy to implement, but in WAN it is complicated and requires authentication of both the client and the

Connection Pooling with gRPC

- Drawbacks of Dudirekta
  - While the dudirekta RPC serves as a good reference implementation of the basic RPC protocol, it does not scale particularly well
  - This mostly stems from two aspects of how it is designed
  - JSON(L) is used for the wire format, which while simple and easy to analyze, is slow to marshal and unmarshal
  - Dudirekta's bi-directional RPCs do 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

Optimizing RPC Throughput and Latency with fRPC

- 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

Results

Testing Environment

- Hardware specifications
- Benchmark scripts are reproducible (add link here)
- Multiple runs were done for each measurement to ensure consistenly

Access Methods

- Latency for technologies
  - Compared to disk and memory, all three other access methods (userfaultfd, direct mounts, managed mounts) are slow
  - Latency of accessing a chunk on userfaultfd is 15x slower than on disk
  - Latency of accessing a chunk on direct mounts is 28x slower than on disk, almost double the latency of userfaultfd
  - Managed mounts is 40x slower than disk
  - It is however important to consider the total scale: All of these are in the scale of µs
  - Interesting to see that managed mounts have a significantly higher latency
  - Looking at the distribution, we can see a similar pattern
  - Spread for managed mount is the smallest, but there are significant outliers up until 1750 µs
  - Direct mounts has the highest spread of the access methods
  - Up until now we took a look at 0ms RTT, meaning that we connected the backend to the mount/userfault handler directly

Initialization

- Mount initialization time
  - There are two diffent ways to check for whether a device is ready: Polling or subscribing to the udev events
  - Initialization time for udev is higher on average than the polling method
  - Spread is similar for both methods
- Pre-emptive pulls
  - When looking at pre-emptive pulls as RTT increases, the role of workers becomes apparent
  - The higher the worker count is, the more data can be pulled
  - While for 4096 workers, almost 40 MB can be pulled pre-emptively at 7ms, this drops to 20 MB for 2048 workers, 5 MB for 512 workers and continues to drop
  - Even for a RTT of 0ms, more workers mean more pre-emptively pulled data in general

Chunking

- Server vs. client-side chunking
  - Chunking can be done on either client- or server-side for both direct and managed mount
  - It is clear that unless the RTT is 0, managed mounts yield significantly higher throughput than direct mounts of both server- and client side chunking
  - When looking at the direct mounts, server-side chunking is very fast for 0ms RTT at almost 500 MB/s throughput, but drops very quickly to 75MB/s at 1ms, 40MB/s at 2ms, and continues to drop down to just over 5MB/s at 20ms RTT
  - For client-side chunking, the result is lower at just 30MB/s even at a 0ms RTT, and then continues to drop steadily it reaches 4.5MB/s at 20ms RTT
  - Looking at managed mounts the scenario is very different
  - Throughput also declines as RTT increases, but less drastically
  - Server-side chunking has much higher throughputs, at 450MB/s at 0ms RTT vs. 230MB/s at 0ms for the client-side chunking

RPC frameworks

- Throughput for RPC frameworks
  - Looking at the performance for dudirekta, gRPC and fRPC for managed and direct mounts, quite drastic throughput differences can be seen as the RTT increases
  - While for 0 RTT, the direct mounts provide the best throughput in line with the measurements for the different access technologies, the throughput drops drastically as RTT increases compared to managed mounts
  - In general, dudirekta has much lower throughput than both gRPC and fRPC
  - When looking at direct mounts specifically, the sharp difference between dudirekta and gRPC/RPC is visible
  - While at 0 RTT, fRPC is at 390MB/s and gRPC is at 500MB/s, dudirekta reaches just 50MB/s
  - At 2ms, throughput for all RPC frameworks drop drastically as the RTT increases, with fRPC and gRPC both reaching 40MB/s, and Dudirekta dropping to just 20MB/s

Backends

- Latency for backends
  - Average first chunk latency for memory, file, directory, Redis and Cassandra backends at 0ms RTT are all similar and within µs range
  - The latency for accessing Redis is the lowest at 2.5µs
  - Looking at the latency distribution, Redis once again is visible as having both the smallest amount of spread and the lowest amount of latency
  - Memory and S3 stick out for having a low amount of outlyers when it comes to first chunk latency, while the directory backend is notable for it's significant spread
- Throughput for backends
  - When looking at throughput, the backends have significantly more different characteristics compared to latency
  - File and memory consistently have high throughput
  - For direct mounts, file throughput is higher than memory througput at 2081 MB/s vs 1630MB/s on average respectively
  - For managed mounts, this increases to 2372MB/s vs. 2178MB/s

Discussion

Userfaults

- By 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 time
- 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
- Only a viable solution for pre-copy migration
- 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

FUSE

- FUSE can provide both a solution for pre- and post-copy migration
- FUSE also has downsides
- It operates in user space, which means that it needs to do context switching, esp. compared to a file system in kernel space
- Some advanced file system features aren't available for a FUSE
- The overhead of FUSE (and implementing a completely custom file system) for synchronizing memory is significant
- 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, since NBD is available as an alternative

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

Direct Mounts

- Have a high spread/are unpredictable when from a first chunk latency perspective at 0ms RTT, but is more predictable when it comes to throughput
- First chunk latency grows linearly like userfaultfd as RTT increases, because there are no pre-emptive pulls
- Has the highest throughput at 0ms RTT, even higher than managed mounts, because the duplicate I/O operations for the background pull are not necessary
- Throughput doesn't drop as rapidly as userfaultfd, but still significantly because there is no background pulls, all data needs to be accessed one by one
- Write speed is subpar because writes need to be delivered directly remotely, as there are no background pushes compared to managed mounts
- Is a good solution if the internal overhead of managed mounts is

Managed Mounts

- Have an internal overhead due to duplicate I/O operations for background pull, resulting in worse throughput for low (esp. 0ms) RTT scenarios and higher inital latencies
- As soon as the RTT reaches levels more typical for a WAN deployment, this becomes
- Tuning the background workers can substantially increase the throughput values, since data can be accessed in parallel
- Pull priority function can allow for even more optimized pulls
- Pre-emptive pulls can significantly reduce initial chunk latency, since multiple MB can be pulled before the device is open
- Higher worker counts can significanlty increase the amount of chunks that can be pre-emptively pulled
- Generally speaking, polling is almost always the better choice for an initialization algorithm in order to reduce the overall Open() time and thus reduce the overhead of mounting a remote ressource

Chunking

- In general, server-side chunking should always be preferred due to the much better throughput
- For direct mounts, due to the linear access, the throughput is low for both server- and client-side chunking due to linear access as RTT increases, but server-side chunking is much more performant in this circumstance for low RTTs
- For managed mounts, client-side chunking can half the throughput of a mount compared to server-side chunking, mostly because if the chunks are smaller than the NBD block size, it decreases the amount of chunks that can be pulled relative if the worker count stays the same, while server-side chunking doesn't require and extra worker on the client for each extra chunk that needs to be pulled, making it possible for the background pull system to pull more, increasing the throughput

RPC Frameworks

- Dudirekta
  - Consistently has lower throughput than the alternaties
  - Performs better for managed mounts than direct mounts thanks to support for concurrent RPCs
  - Less sensistive to RTT compared to gRPC and fRPC for managed mounts
  - Even with managed mounts, much worse throughput compared to both due to no connection pooling support
  - Remains a good option for prototyping due to lower development overhead and transport layer independence
- gRPC
  - Considerably faster than Dudirekta for managed and direct mounts
  - Has support for connection pooling, giving it a significant performance benefit over Dudirekta for managed mounts
  - Very good performance for direct mounts with 0ms RTT
  - Is an industry standard, and has good tooling and known scalability characteristics

Backends

- Redis
  - Has the smallest amount of spread and lowest amount of initial chunk latency at 0ms RTT
  - Direct mounts have a lower throughput compared to managed mounts, showing good support for concurrent chunk access
  - Has the highest throughput for direct mounts by a significant margin due to its optimized protocol and fast key lookups
  - Is also the fastest network-capable backend for managed mounts, once again due to the good support for concurrent access
  - Good choice for ephermeral data like caches, where speed or the need for the direct mount API is important
- Cassandra
  - Has highest throughput in 0ms RTT deployments for direct mounts, showing very good concurrent access performance
  - Falls short when it comes to usage in direct mounts, where the performance is worse than any other backend, showing a high read latency overhead for looking up individual chunks

RegionFS

- Comparing this API to RegionFS, an existing remote memory system
  - A similar approach was made in RegionFS[33]
  - RegionFS is implemented as a kernel module, but it is functionally similar to how this API exposes a NBD device for memory interaction
  - In RegionFS, the regions file system is mounted to a path, which then exposes regions as virtual files
  - Instead of using a custom configuration (such as configuring the amount of pushers to make a mount read-only), such an approach makes it possible to use chmod on the virtual file for a memory region to set permissions
  - By using standard utilities like open and chmod, this API usable from different programming languages with ease
  - Unlike the managed mounts API however, the system proposed in Remote Regions is mostly intended for private usecases with a limited amount of hosts and in LAN, with low-RTT connections
  - It is also not designed to be used for a potential migration scenarios, which the modular approach of r3map allows for

Language Limitations

- Issues with the r3map implementation
  - While the managed mounts API mostly works, there are some issues with it being implemented in Go
  - This is mostly due to deadlocking issues; if the GC tries to release memory, it has to stop the world
  - If the mmap API is used, it is possible that the GC tries to manage the underlying slice, or tries to release memory as data is being copied from the mount
  - Because the NBD server that provides the byte slice is also running in the same process, this causes a deadlock as the server that provides the backend for the mount is also frozen (https://github.com/pojntfx/r3map/blob/main/pkg/mount/slice_managed.go L93)
  - A workaround for this is to lock the mmaped region into memory, but this will also cause all chunks to be fetched, which leads to a high Open() latency
  - This is fixable by simply starting the server in separate process or

Remote Swap With ram-dl

- ram-dl is a fun experiment
- Is a tech demo for r3map
- Uses the fRPC backend to expand local system memory
- Allows 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

- Overview
  - 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
  - Only one concurrent reader/writer makes sense
  - Syncing intervals to/from can maybe be minutes or more to make it more efficient (since long, connected write intervals prevent having to

Improving File System Synchronization Solutions

- Existing 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, similarly to the file synchronization approach to memory synchronization
  - 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
  - This means that files can be fetched on demand

Universal Database, Media and Asset Streaming

- Streaming access to remote databases
  - 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 background pullers can be used

Universal App State Synchronization and Migration

- Modelling state
  - 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 device, we can use the managed mount/direct mount/migration APIs to send/receive them or mount them

Summary

Summary

- Due to the flexible nature of r3map, it is possible to bring the semantics of VM live migration and snapshots to anything that has state
- 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 presented might make it finally 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? - Yes!)
- What would be possible if memory became the universal way to access state (short recap of the ideas for use cases)
- Further research recommendations (e.g. using ublk instead of NBD)

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

Citations

[1]
T. kernel development community, “Quick start.” https://www.kernel.org/doc/html/next/rust/quick-start.html, 2023.

[2]
R. Love, *Linux kernel development*, 3rd ed. Pearson Education, Inc., 2010.

[3]
W. Mauerer, *Professional linux kernel architecture*. Indianapolis, IN: Wiley Publishing, Inc., 2008.

[4]
W. R. Stevens, *Advanced programming in the UNIX environment*. Delhi: Addison Wesley Logman (Singapore) Pte Ltd., Indian Branch, 2000.

[5]
K. A. Robbins and S. Robbins, *Unix™ systems programming: Commu-*