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

TODO: Add subtitle

Felicitas Pojtinger (Stuttgart Media University)

2023-08-04

Abstract

TODO: Add abstract

Contents

1 Introduction

TODO: Add introduction

2 Technology

2.1 The Linux Kernel

The Linux kernel, an open-source kernel, was originally brought into existence by Linus Torvalds in 1991. Developed primarily in the C programming language, it has recently seen the addition of Rust as an approved language for further expansion and development[1]. The kernel's influence is extensive, powering millions of devices across the globe, including servers, desktop computers, mobile phones, and a myriad of embedded devices.

In essence, the Linux kernel serves as an intermediary between hardware and applications. Its role is to offer an abstraction layer that simplifies the interaction between these two entities. Moreover, it is engineered for compatibility with a wide array of architectures, such as ARM, x86, RISC-V, and others, which increases its versatility and broadens its reach.

The Linux kernel, though foundational, does not function as a standalone operating system. This role is fulfilled by distributions, which build upon the Linux kernel to create fully-fledged operating systems[2]. Distributions supplement the kernel with additional userspace tools, examples being GNU coreutils or BusyBox. They further enhance functionality by integrating desktop environments and other software. This process transforms the kernel into a complete operating system ready for user interaction.

The open-source nature of the Linux kernel signifies its suitability for academic exploration and usage. It offers transparency, allowing anyone to inspect the source code in depth. Furthermore, it encourages collaboration by enabling anyone to modify and contribute to the source code. This transparency, coupled with the potential for customization and improvement, underscores the value of the Linux kernel for this thesis.

2.2 Linux Kernel Modules

At the heart of the Linux system resides its kernel, an extensible, monolithic component that allows for the integration of kernel modules[2]. Kernel modules are small pieces of kernel-level code that can be dynamically incorporated into the kernel, presenting the advantage of extending kernel functionality without necessitating system reboots.

The dynamism of these modules comes from their ability to be loaded and unloaded into the running kernel as per user needs. This functionality aids in keeping the kernel size both manageable and maintainable, thereby promoting efficiency. Kernel modules are developed using the C programming language, like the kernel itself, ensuring compatibility and consistent performance.

Kernel modules interact with the kernel via APIs (Application Programming Interfaces), mechanisms that facilitate communication between software components. Despite their utility, kernel modules do carry a potential risk. If not written with careful attention to detail, they can introduce significant instability into the kernel, adversely affecting the overall system performance and reliability.

The lifecycle of a kernel module is not strictly confined to its time spent inside the kernel. It can be managed and controlled at different stages, starting from boot time, and be manipulated dynamically when the system is already running. This is facilitated by utilities like `modprobe` and `rmmod`[3].

In the lifecycle of a kernel module, two key functions are of significance: initialization and cleanup. The initialization function is responsible for setting up the module when it's loaded into the kernel. Conversely, the cleanup function is used to safely remove the module from the kernel, freeing up any resources it previously consumed. These lifecycle functions provide a structured approach to module management, further enhancing the efficiency and stability of the Linux kernel.

2.3 UNIX Signals and Handlers

UNIX signals are an integral component of UNIX-based systems, including Linux. They function as software interrupts, notifying a process of significant occurrences, such as exceptions. Signals may be generated from various sources, including the kernel, user input, or other processes, making them a versatile tool for inter-process communication.

Despite their notification role, signals also serve as an asynchronous communication mechanism between processes or between the kernel and a process. As such, they have an inherent ability to deliver important notifications without requiring the recipient process to be in a specific state of readiness[4]. Each signal has a default action associated with it, the most common of which are terminating the process or simply ignoring the signal.

To customize how a process should react upon receiving a specific signal, handlers can be utilized. Handlers are routines that dictate the course of action a process should take when a signal is received. Using the `sigaction()` function, a handler can be installed for a specific signal, enabling a custom response to that signal[5].

However, it is important to note that signals are not typically utilized as a primary inter-process communication (IPC) mechanism. This is primarily due to their limitation in carrying additional data. While signals effectively alert a process of an event, they are not designed to convey any supplemen-

tary data related to that event. Consequently, they are best used in scenarios where simple event-based notifications are sufficient, rather than for more complex data exchange requirements.

2.4 Memory Hierarchy

The memory hierarchy in computers is an organized structure based on factors such as size, speed, cost, and proximity to the Central Processing Unit (CPU). It follows the principle of locality, which suggests that data and instructions that are accessed frequently should be stored as close to the CPU as possible[6]. This principle is crucial primarily due to the limitations of "the speed of the cable", where both throughput and latency decrease as distance increases due to factors like dampening and the finite speed of light.

At the top of the hierarchy are registers, which are closest to the CPU. They offer very high speed, but provide limited storage space, typically accommodating 32-64 bits of data. These registers are used by the CPU to perform operations.

Following registers in the hierarchy is cache memory, typically divided into L1, L2, and L3 levels. As the level increases, each layer becomes larger and less expensive. Cache memory serves as a buffer for frequently accessed data, with predictive algorithms optimizing its usage.

Main Memory, often referred to as Random Access Memory (RAM), provides larger storage capacity than cache but operates at a slower speed. It typically stores running programs and open files.

Below main memory, we find secondary storage devices such as Solid State Drives (SSD) or Hard Disk Drives (HDD). Although slower than RAM, these devices can store larger amounts of data and typically house the operating system. Importantly, they are persistent, meaning they retain data even after power is cut.

Tertiary storage, including optical disks and tape, is slow but very cost-effective. Tape storage can store very large amounts of data for relatively long periods of time. These types of storage are typically used for archiving or physically transporting data, such as importing data from personal infrastructure to a service like AWS.

The memory hierarchy is not static but evolves with technological advancements, leading to some blurring of these distinct layers[7]. For instance, Non-Volatile Memory Express (NVMe) storage technologies can rival the speed of RAM while offering greater storage capacities. Similarly, some research, such as the work presented in this thesis, further challenges traditional hierarchies by exposing tertiary or secondary storage with the same interface as main memory.

2.5 Memory Management in Linux

Memory management forms a cornerstone of any operating system, serving as a critical buffer between applications and physical memory. Arguably, it can be considered one of the fundamental purposes of an operating system itself. This system helps maintain system stability and provides security guarantees, such as ensuring that only a specific process can access its allocated memory.

Within the context of the Linux operating system, memory management is divided into two major segments: kernel space and user space.

Kernel space is where the kernel, kernel extensions, and device drivers operate. The kernel memory module is responsible for managing this segment. Slab allocation is a technique employed in kernel space management. This technique groups objects of the same size into caches, enhancing memory allocation speed and reducing fragmentation of memory[8].

On the other hand, user space is the memory segment where applications and certain drivers store their memory[9]. User space memory management involves a paging system, offering each application its unique private virtual address space.

This virtual address space is divided into units known as pages, each typically 4 KB in size. These pages can be mapped to any location in physical memory, providing flexibility and optimizing memory utilization. The use of a virtual address space further adds a layer of abstraction between the application and the physical memory, enhancing the security and isolation of processes.

2.6 Swap Space

Swap space refers to a designated portion of the secondary storage utilized as virtual memory in a computer system[9]. This feature plays a crucial role in systems that run multiple applications simultaneously. When memory resources are strained, swap space comes into play, relocating inactive parts of the RAM to secondary storage. This action frees up space in primary memory for other processes, enabling smoother operation.

In the Linux operating system, swap space implementation aligns with a demand paging system. This means that memory is allocated only when required. The swap space in Linux can be a swap partition, which is a distinct area within the secondary storage, or it can take the form of a swap file, which is a standard file that can be expanded or truncated based on need. The usage of swap partitions and files is transparent to the user.

The Linux kernel employs a Least Recently Used (LRU) algorithm to determine which memory pages should be moved to swap space. This algorithm effectively prioritizes memory pages based on their usage, transferring those that have not been recently used to swap space.

Swap space also plays a significant role in system hibernation. Before the system enters hibernation, the content of RAM is stored in the swap space, where it remains persistent even without power. When the system is resumed, the memory content is read back from swap space, restoring the system to its pre-hibernation state.

However, the use of swap space can impact system performance. Since secondary storage devices are usually slower than primary memory, heavy reliance on swap space can cause significant system slowdowns. To mitigate this, Linux allows for the adjustment of 'swappiness', a parameter that controls the system's propensity to swap memory pages. Adjusting this setting can balance the use of swap space to maintain system performance while preserving the benefits of virtual memory management.

2.7 Page Faults

Page faults are instances in which a process attempts to access a page that is not currently available in primary memory. This situation triggers the operating system to swap the necessary page from secondary storage into primary memory[10]. These are significant events in memory management, as they determine how efficiently an operating system utilizes its resources.

Page faults can be broadly categorized into two types: minor and major. Minor page faults occur when the desired page resides in memory but isn't linked to the process that requires it. On the other hand, a major page fault takes place when the page has to be loaded from secondary storage, a process that typically takes more time and resources.

To minimize the occurrence of page faults, memory management algorithms such as Least Recently Used (LRU) and the more straightforward clock algorithm are often employed. These algorithms effectively manage the order and priority of memory pages, helping to ensure that frequently used pages are readily available in primary memory.

Handling page faults involves certain techniques to ensure smooth operation. One such technique is prefetching, which anticipates future page requests and proactively loads these pages into memory. Another approach involves page compression, where inactive pages are compressed and stored in memory preemptively[11]. This reduces the likelihood of major page faults by conserving memory space, allowing more pages to reside in primary memory.

In general, handling page faults is a task delegated to the kernel. This critical function is part of the kernel's memory management duties, ensuring that processes can access the pages they require while maintaining efficient use of system memory. This balance between resource availability and system performance underscores the importance of sophisticated page fault management in an operating system's design.

2.8 mmap

The mmap is a versatile UNIX system call, used for mapping files or devices into memory, enabling a variety of core tasks like shared memory, file I/O, and fine-grained memory allocation. Due to its powerful nature, it is commonly harnessed in applications like databases. It is indeed a “power tool”, mandating the careful and intentional use to avoid issues or inefficiencies.

One standout feature of mmap is its ability to function by creating a direct memory mapping between a file and a region of memory[12]. This connection means that read operations performed on the mapped memory region directly correspond to reading the file and vice versa, enhancing efficiency by reducing the overhead as the necessity for context switches diminishes.

A key advantage that mmap provides is the capacity to facilitate zero-copy operations. In practical terms, this signifies data can be accessed directly as if it were positioned in memory, eliminating the need to copy it from the disk first. This direct memory access saves time and reduces processing requirements, offering substantial performance improvements.

The mmap is also proficient in sharing memory between processes without having to pass through the kernel with system calls[4]. With this feature, mmap can create shared memory spaces where multiple processes can read and write, enhancing interprocess communication and data transfer efficiency.

However, mmap does come with a notable drawback: it bypasses the file system cache. That can potentially result in stale data when multiple processes are reading and writing simultaneously. This bypass may lead to a scenario where one process modifies data in the mmap region, and another process that is not monitoring for changes via the mmap might remain unaware and continue to work with outdated data.

Despite the potential for stale data, the benefits of using mmap are substantial, affording the implementation of highly efficient applications. Its ability to directly map memory to files and vice versa, avoiding costly file system operations, the potential for zero-copy operations, and memory sharing capabilities make it a significant instrument in the UNIX system call arsenal. Yet, users must always account for the trade-off with the bypass of the cache system and mitigate possible stale data issues. Responsible use of mmap can be a game changer for optimizing memory and file management operations in UNIX-based systems.

2.9 inotify

The inotify is an event-driven notification system of the Linux kernel, designed to monitor the file system for a multitude of events, such as modifications and accesses, among others[13]. It’s unique for its intrinsic ability to heed to specific events through a watch feature. For instance, it can be configured to watch only write operations on certain files. This level of control can offer considerable

benefits in cases where there is a need to focus system resources on certain file system events, and not on others.

Naturally, `inotify` comes with some recognizable advantages. Significantly, it diminishes overhead and resource use when compared to polling strategies. Polling is an operation-heavy approach as it continuously checks the status of the file system, regardless of whether any changes have occurred. In contrast, `inotify` works in a more event-driven way, where it only takes action when a specific event occurs. This differential action is inherently more efficient, reducing overhead where there are infrequent changes to the file system.

Thanks to its efficiency and flexibility, `inotify` has found its utilization across many applications, including Dropbox, a widely used file synchronization service. In such applications, its capability to instantly notify the system of file changes aids in instant synchronization of files, demonstrating how critical its role can be in real-time or near real-time systems that are dependent on keeping data up-to-date.

However, as is the case with many system calls, there is a limit to its scalability. `inotify` is constrained by a limit on how many watches can be established. This limitation can pose challenges in intricate systems where there is a high quantity of files or directories to watch for, and might warrant additional management or fallback to heavier polling mechanisms for some parts of the system.

2.10 Linux Kernel Disk and File Caching

Linux Kernel Disk and File Caching are key elements of the Linux operating system that work to boost efficiency and performance. Within this framework, there are two broad categories: disk caching and file caching.

Disk caching in Linux is a strategic method that temporarily stores frequently accessed data in Random Access Memory (RAM). It is implemented through the page cache subsystem in Linux. This approach leverages the principle of locality, which states that odds are good that data situated near data that has already been accessed will be needed soon. This method of retaining data close to the CPU where it may be swiftly accessed without costly disk reads greatly reduces overall access time. The data within the cache is managed using the Least Recently Used (LRU) algorithm, which prunes the least recently used items first when space is needed.

File caching, on the other hand, is where Linux caches file system metadata in specialized structures known as the `dentry` and `inode` caches. This metadata encompasses varied information such as file names, attributes, and locations. The key benefit of this caching is that it expedites the resolution of path names and file attributes, such as tracking when files were last changed for polling. Notably, file read/write operations are also channeled through the disk cache, further illustrating the intricate interconnectedness of disk and file caching mechanisms in the Linux Kernel.

However, even as disk and file caching improve performance, they also introduce complexities. One such complexity involves maintaining data consistency between the disk and cache through the process known as writebacks. A delicate balance needs to be maintained here: aggressive writebacks, where data is copied back to disk frequently, can lead to reduced performance, while excessive delays may risk data loss if the system crashes before data has been saved.

Another complexity arises from the necessity to release cached data under memory pressure, known as cache eviction. It necessitates sophisticated algorithms, such as the LRU, to ensure effective utilization of available cache space[3]. Prioritizing what to keep in cache when memory pressure builds is no trivial task, as the decision will directly impact the overall system performance.

2.11 TCP, UDP and QUIC

TCP (Transmission Control Protocol), UDP (User Datagram Protocol), and QUIC (Quick UDP Internet Connections) are key communication protocols utilized in network communications, each showcasing distinct characteristics and use cases.

TCP has long been the reliable backbone for internet communication due to its connection-oriented nature [14]. This protocol ensures the guaranteed delivery of data packets and their correct order, rendering it a highly dependable means for data transmission. Significantly, TCP incorporates error checking, allowing the detection and subsequent retransmission of lost packets. Moreover, TCP includes a congestion control mechanism to manage data transmission seamlessly during high traffic. Owing to these features, TCP is widely used to power the majority of the web where reliable, ordered, and error-checked data transmission is paramount.

Alternatively, UDP is a connectionless protocol that does not make guarantees about the reliability or ordered delivery of data packets [15]. This lends UDP a speed advantage over TCP since fewer protocols are in place, resulting in less communication overhead. Although it lacks TCP's robustness in handling errors and maintaining data order, UDP finds use in applications where speed takes precedence over reliability. This includes online gaming, video calls, and other real-time communication modes where quick data transmission is crucial even if occasional data packets go astray.

QUIC, a modern transport layer protocol, was created by Google and ratified by the IETF in 2020. It aspires to amalgamate the best qualities of TCP and UDP [16]. Unlike UDP, QUIC ensures the reliability of data transmission and guarantees the ordered delivery of data packets. Simultaneously, it draws inspiration from UDP's speed qualities. One of QUIC's standout features is its ability to reduce connection establishment times, which effectively lowers initial latency. It achieves this by amalgamating connection and security handshakes, reducing the time taken for a connection to be established and secured. Additionally, QUIC is designed to prevent the issue of "head-of-line blocking", allowing for the independent delivery of separate data streams. This means it can handle the delivery of separate

data streams without one stream blocking another, resulting in smoother and more efficient transmission.

2.12 Delta Synchronization

Delta synchronization is an innovative technique that allows for efficient synchronization of files between hosts, aiming to transfer only those parts of the file that have undergone changes instead of the entire file. This incremental approach can remarkably reduce network and I/O overhead, optimally utilizing system resources.

Perhaps the most recognized tool employing this method of synchronization is `rsync`, an open-source data synchronization utility in Unix-like operating systems. The elixir offered by `rsync` lies in its usage of a delta-transfer algorithm, which astutely computes the difference between local and remote files, thereafter synchronizing the changes.

Fundamentally, the algorithm embarks on file block division, dissecting the file on the destination side into fixed-size blocks. For each of these blocks, a quick albeit weak checksum calculation is performed, and these checksums are transferred to the source system.

Concurrently, the source initiates the same checksum calculation process. These checksums are then compared to those received from the destination, a step recognized as matching block identification. The outcome of this comparison allows the source to discern the blocks which have transformed since the last synchronization.

Once the altered blocks are identified, the source proceeds to send the offset of each block alongside the data of the changed block to the destination. Upon receiving a block, the destination writes the chunk to the specific offset in the file. This process results in the reconstruction of the file in accordance with the modifications undertaken at the source.

Finally, after one polling interval is complete, the process reinitiates, perpetuating the cycle of efficient, incremental synchronization[17].

2.13 File Systems In Userspace (FUSE)

File Systems in Userspace (FUSE) is a software interface that enables the creation of custom file systems in the userspace, as opposed to developing them as kernel modules. It liberates developers from the obligatory low-level kernel development normally associated with creating new file systems.

FUSE is available on various platforms. Though mostly deployed on Linux, it can also be found on macOS and FreeBSD. The secret to developing file systems in userspace lies in the FUSE Application Programming Interface (API). In this setup, a userspace program aligns itself with the FUSE kernel

module and provides callbacks for the file system operations. These operations can include tasks like `open`, `read`, `write`, and many others.

So, when a user performs a file system operation on a mounted FUSE file system, the kernel module sends a request for executing that operation to the userspace program. This is followed by the userspace program returning a response, which the FUSE kernel module conveys back to the user. As such, FUSE circumvents the complexity of coding directly in the kernel. It runs comfortably in userspace and does not require a custom kernel module. This approach enhances safety, preventing entire kernel crashes due to errors within the FUSE or the backend.

One commendable attribute of FUSE is its inherent portability. Unlike a file system created as a kernel module, its interaction with the FUSE module rather than the kernel itself grants it this portability. Despite the benefits of FUSE, there is a noticeable performance overhead associated with it. This is largely due to the context switching between the kernel and the userspace that occurs during its operation[18].

Fundamentally, FUSE is widely utilized to mount high-level interfaces to external services. For instance, it can be used to mount S3 buckets in an AWS environment or mount a remote system's disk via Secure Shell (SSH). This functionality goes a long way in facilitating smooth and adaptable access to various resources, lending FUSE its unique versatility in the diverse world of file systems.

2.14 Network Block Device (NBD)

Network Block Device (NBD) embodies a protocol that enables communication between a user space-provided server and a Kernel-provided client. Though potentially deployable over Wide Area Networks (WAN), it is primarily designed for Local Area Networks (LAN) or localhost usage. The protocol is divided into two phases: the handshake and the transmission[19].

The execution of the NBD protocol involves multiple participants, notably one or several clients, a server, and the virtual concept of an export. It initiates with a client establishing a connection with the server. The server reciprocates by delivering a greeting message highlighting various server flags. The client responds by transmitting its own flags along with the name of an export to use; a single NBD server can expose multiple devices.

Upon receiving this, the server sends the size of the export and other metadata. The client acknowledges this data, completing the handshake. Post handshake, the client and server exchange commands and replies. A command can embody any of the basic actions needed to access a block device, for instance read, write or flush. These commands might also contain data such as a chunk for writing, offsets, and lengths among other elements. Replies may contain error messages, success status, or data contingent on the reply type.

While powerful in many regards, NBD does exhibit some limitations. Its maximum message size is capped at 32 MB, and the maximum block or chunk size supported by the Kernel is a mere 4096 KB. Thus, it might not be the most optimal protocol for WAN usage, especially in scenarios with high latency.

NBD, being a seasoned protocol, comes with its own set of operational quirks such as multiple different handshake versions and legacy features. As a result, it is advisable to only implement the latest recommended versions and the foundational feature set when considering the NBD for minimal use.

Despite the simplicity of the protocol, there are certain scenarios where NBD falls short. It has limitations when dealing with backing devices that operate drastically different from random-access storage devices, like a tape drive, since it lacks the ability to work with high-level abstractions such as files or directories. For example, it does not support shared access to the same file for multiple clients. However, this shortcoming can be considered as an advantage for narrow memory synchronization, given that it operates on a block level - a case where simplicity and focused functionality triumph over holistic versatility.

2.15 Virtual Machine Live Migration

2.15.1 Pre-Copy

Virtual Machine Live Migration has emerged as a key strategy to optimize systems involving WAN, with a focus on minimizing downtime during data transfer processes. It essentially involves the shifting of a virtual machine, its state, and its connected devices from one host to another, with the objective to minimize disrupted service.

Live migration algorithms can be categorized into two broad types: pre-copy migration and post-copy migration. Here we delve into the specifics of the pre-copy migration technique.

The foremost characteristic of pre-copy migration is its ‘run-while-copy’ nature. In other words, the copying of data from the source to the destination occurs concurrently while the virtual machine (VM) continues to operate. This method is also applicable in a generic migration context where an application or another data state is being updated.

In the case of a VM, the pre-copy migration procedure commences with the transference of the initial state of VM’s memory to the destination host. During this operation, if modifications occur to any chunks of data, they are flagged as ‘dirty’. These modified or ‘dirty’ chunks of data are then transferred to the destination until only a small number remain - an amount small enough to stay within the allowable maximum downtime criteria.

Following this, the VM operation is suspended at the source, enabling the synchronization of the remaining chunks of data to the destination. Once this synchronization process is completed, the operation of the VM is resumed at the destination host.

The pre-copy migration process represents a remarkably resilient strategy, especially in instances where there might be network disruption during synchronization. This resilience arises from the fact that, at any given point during migration, the VM is readily available in full either at the source or the destination.

However, a limitation in the approach is that, if the VM or application alters too many chunks on the source during migration, it may not be possible to meet the maximum acceptable downtime criteria. Furthermore, maximum permissible downtime is inherently restricted by the available round-trip time (RTT)[20].

Despite these constraints, the pre-copy migration strategy offers viable benefits in terms of consistent VM availability and resilience against network disconnections, underlining its potential effectiveness in managing more complex operations incorporating WAN.

2.15.2 Post-Copy

Post-copy migration is an alternative approach to pre-copy migration in Virtual Machine Live Migration. While pre-copy migration operates by copying data before the VM halt, post-copy migration opts for another strategy: it immediately suspends the VM operation on the source, moves it to the destination, and resumes it – all with only a minimal subset of data or 'chunks'.

The process initiates with the prompt suspension of the VM on the source system. It is then relocated to the destination with only some essential chunks of data involved in the transfer. Upon reaching the destination, the VM operation is resumed.

During this resumed operation, whenever the VM attempts to access a chunk of data not initially transferred during the move, a page fault arises. A page fault, in this context, is a type of interrupt generated when the VM tries to read or write a chunk that is not currently present in its address space. This triggers the system to retrieve the missing chunk from the source host, enabling the VM to continue its operations.

The main advantage post-copy migration offers centers around the fact that it eliminates the necessity of re-transmitting chunks of 'dirty' or changed data before hitting the maximum tolerable downtime – quite unlike the pre-copy strategy. This process not only prevents wasteful re-transmission but also economizes data traffic between source and destination to some extent.

However, this approach is not without its drawbacks. Post-copy migration could potentially lead to extended migration times. This is primarily as a consequence of its 'fetch-on-demand' model for retriev-

ing chunks, a model highly sensitive to network latency and round-trip time (RTT). This essentially means that the time taken for data to travel to the destination and then for an acknowledgement of receipt to return to the source can affect the overall efficiency of the migration[20].

2.15.3 Workload Analysis

The research article “Reducing Virtual Machine Live Migration Overhead via Workload Analysis” explores strategies to determine the most suitable timing for virtual machine (VM) migration[21]. Even though the study chiefly focuses on virtual machines, the methodologies proposed could be adapted for use with various other applications or migration circumstances.

The method proposed in the study identifies cyclical workload patterns of VMs and leverages this knowledge to delay migration when beneficial. This is achieved by analyzing recurring patterns that may unnecessarily postpone VM migration, and identifying optimal cycles within which VMs can be migrated. For instance, in the realm of VM usage, such cycles could be triggered by a large application’s Garbage Collection (GC) that results in numerous changes to VM memory.

When migration is proposed, the system verifies whether it is in an optimal cycle that is conducive to migration. If it is, the migration proceeds; if not, the migration is postponed until the next cycle. This process employs a Bayesian classifier to distinguish between favorable and unfavorable cycles.

Compared to the popular alternative method which usually involves waiting for a significant amount of unchanged chunks to synchronize first, the proposed pattern recognition-based approach potentially offers substantial improvements. The study found that this method yielded an enhancement of up to 74% in terms of live migration time/downtime and a 43% improvement concerning the volume of data transferred over the network.

2.16 Streams and Pipelines

Streams and pipelines are fundamental constructs in computer science, enabling efficient, sequential processing of large datasets without the need for loading the entire dataset into memory. They form the backbone of modular and efficient data processing techniques, each concept having its unique characteristics and use cases.

A stream represents a continuous sequence of data, serving as a conduit between different points in a system. Streams can be either a source or a destination for data. Examples include files, network connections, and standard input/output devices, amongst others. The power of streams comes from their ability to process data as it becomes available. This aspect allows for minimization of memory consumption, making streams particularly impactful for scenarios involving long-running processes where data is streamed over extended periods of time[22].

On the other hand, a pipeline comprises a series of data processing stages, wherein the output of one stage directly serves as the input to the next. It's this chain of processing stages that forms a 'pipeline.' Pipelines allow for stages to often run concurrently—this parallel execution often results in a significant performance improvement due to a higher degree of concurrency.

One of the classic examples of pipelines is the instruction pipeline in CPUs, where different stages of instruction execution—fetch, decode, execute, and writeback—are performed in parallel. This design increases the instruction throughput of the CPU, allowing it to process multiple instructions simultaneously at different stages of the pipeline.

Another familiar implementation is observed in UNIX pipes, a fundamental part of UNIX and Linux command-line interfaces. Here, the output of a command can be 'piped' into another for further processing. For instance, the results from a `curl` command fetching data from an API could be piped into the `jq` tool for JSON manipulation[23].

2.17 gRPC

The gRPC is an open-source, high-performance remote procedure call (RPC) framework developed by Google in 2015. The gRPC framework is central to numerous data communication processes and is recognized for its cross-platform compatibility, supporting a variety of languages including Go, Rust, JavaScript and more. The gRPC maintains its development under the auspices of the Cloud Native Computing Foundation (CNCF), further emphasizing its integration within the realm of cloud services.

One of the notable features of the gRPC is its usage of HTTP/2 as the transport protocol. This allows it to exploit beneficial capabilities of HTTP/2 such as header compression, which minimizes bandwidth usage, and request multiplexing, enabling multiple requests to be sent concurrently over a single TCP connection.

In addition to HTTP/2, gRPC utilizes Protocol Buffers (protobuf) as the Interface Definition Language (IDL) and wire format. Protobuf is a compact, high-performing, and language-neutral mechanism for data serialization. This makes it preferable over the more verbose and slower JSON format often employed in REST APIs. Protobuf's performance and universality facilitate faster data exchange, making it a prime choice for the gRPC framework.

One of the fundamental strengths of the gRPC framework is its support for various types of RPCs. It accommodates unary RPCs where the client sends a single request to the server and receives a single response in return, mirroring the functionality of a traditional function call. It also supports server-streaming RPCs, wherein the client sends a request, and the server responds with a stream of messages. Conversely, in client-streaming RPCs, the client sends a stream of messages to a server in response to a request. Further, it also supports bidirectional RPCs, wherein both client and server can

read and write independently of the other.

What distinguishes gRPC is its pluggable structure that allows for added functionalities such as load balancing, tracing, health checking, and authentication. These features make it a comprehensive solution for efficient and effective communication mechanisms[24].

2.18 Redis

Redis, an acronym for 'REmote DIctionary Server,' is an in-memory data structure store, primarily utilized as a database, cache, or message broker. Introduced by Salvatore Sanfilippo in 2009, Redis lends itself to high-speed and efficient data management, thus positioning itself as a key resource in numerous applications demanding rapid data processing.

Setting it apart from other NoSQL databases, Redis supports a multitude of data structures, including lists, sets, hashes, and bitmaps. These differing data types allow for versatile data manipulation capabilities. Thus, in scenarios where a diverse set of data types need to be managed concurrently, Redis supplies a fitting solution[25].

One of the primary drivers for Redis's speed is its reliance on in-memory data storage. By storing data in memory rather than on disk, Redis enables low-latency reads and writes, contributing to the technology's overall capacity for swift, efficient data operations.

But while the primary domain of Redis resides in in-memory operations, the technology also accommodates persistence—it can store data on disk. This feature broadens the use cases for Redis, allowing it to handle applications that require long-term data storage, albeit not being its initial intent.

Another noteworthy aspect of Redis is its non-blocking I/O model. This paradigm facilitates efficient data transfer operations, eliminating the necessity for waiting times during data-processing tasks. As such, Redis can offer near real-time data processing capabilities—a key attribute in many modern digital contexts, especially those involving stream data processing and live analytics.

Further expanding on its versatility, Redis incorporates a publish-subscribe (pub-sub) system. This empowers it to function as a message broker, where messages are published to 'channels' and delivered to all the 'subscribers' interested in those channels. This feature is particularly useful when it comes to real-time event notifications in web applications or the distribution of tasks among worker nodes in high-performance computing setups[26].

2.19 S3 and Minio

Amazon's Simple Storage Service (S3) and Minio are two contemporary solutions providing storage functionality, each with unique characteristics and use cases.

S3 is a scalable object storage service crafted for data-intensive tasks, befitting anything from small deployments to large-scale applications. It is one of the prominent services offered by Amazon Web Services, a leading provider of cloud computing resources. S3's design allows for global distribution, which means the data can be stored across multiple geographically diverse servers. This permits fast access times from virtually any location on the globe, crucial for globally distributed services or applications with users spread across different continents.

S3 offers a variety of storage classes catering to different needs, whether the requirement is for frequent data access, infrequent data retrieval, or long-term archival. This flexibility ensures the service can meet a wide array of business and application demands. S3 also comes equipped with comprehensive security provisions. This includes authentication and authorization mechanisms, critical for data privacy and access control.

Communication with S3 is facilitated through an exposed HTTP API. Users and applications can interact with the stored data—including files and folders—via this API, allowing for seamless data manipulation and retrieval[27].

On the contrary, Minio represents a different approach towards storage solutions. It is an open-source storage server, underlining compatibility with Amazon S3's API. Written in the Go programming language, Minio is lightweight and straightforward, tailored for deployments requiring simplicity without sacrificing core functionality. Being open-source, users have the ability to view, modify, and distribute Minio's source code, fostering community-driven development and innovation.

A critical distinction of Minio is its suitability for on-premises hosting. Organizations with specific security regulations or those preferring to maintain direct control over their data could find Minio a fitting choice. Moreover, Minio supports horizontal scalability, designed to distribute large quantities of data across multiple nodes. This feature aids in optimizing storage efficiency, data access times, and overall capability to handle growing data volumes[28].

2.20 Cassandra and ScylllaDB

Apache Cassandra and ScyllaDB are compelling wide-column NoSQL databases, each tailored for large-scale, distributed data management tasks. They blend Amazon's Dynamo model's distributed nature with Google's Bigtable model's structure, leading to a highly available, heavy-duty database system.

Apache Cassandra is lauded for its scalability and eventual consistency, designed to handle vast amounts of data spread across numerous servers. Unique to Cassandra is the absence of a single point of failure, thus ensuring continuous availability and robustness—critical for systems requiring high uptime.

Cassandra's consistency model is tunable according to needs, ranging from eventual to strong consistency. It distinguishes itself by not employing master nodes due to its usage of a peer-to-peer protocol and a distributed hash ring design. These design choices eradicate the bottleneck and failure risks associated with master nodes.

Despite its robust capabilities, Cassandra does come with certain limitations. Under heavy load, it experiences high latency that can negatively affect system performance. Besides, it demands complex configuration to correctly fine-tune according to specific needs[29].

In response to the perceived shortcomings of Cassandra, ScyllaDB was launched in 2015. It shares design principles with Cassandra—compatibility with Cassandra's API and data model—but boasts architectural differences intended to overcome Cassandra's limitations. It's primarily written in C++, contrary to Cassandra's Java-based code. This contributes to ScyllaDB's shared-nothing architecture, a design that aims to minimize contention and enhance performance.

ScyllaDB was particularly engineered to address Cassandra's issues around latency, specifically the 99th percentile latency that impacts system reliability and predictability. In fact, ScyllaDB's design improvements and performance gains over Cassandra have been endorsed by various benchmarking studies[30].

[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: Communication, concurrency, and threads*. Prentice Hall PTR, 2003.

[6] A. J. Smith, "Cache memories," *ACM Comput. Surv.*, vol. 14, no. 3, pp. 473–530, Sep. 1982, doi: 10.1145/356887.356892.

[7] H. A. Maruf and M. Chowdhury, "Memory disaggregation: Advances and open challenges." 2023.Available: https://arxiv.org/abs/2305.03943

[8] J. Bonwick, "The slab allocator: An Object-Caching kernel," Jun. 1994.Available: https://www.usenix.org/conference/usenix-summer-1994-technical-conference/slab-allocator-object-caching-kernel

[9] M. Gorman, *Understanding the linux virtual memory manager*. Upper Saddle River, New Jersey 07458: Pearson Education, Inc. Publishing as Prentice Hall Professional Technical Reference, 2004.

[10] Q. Li, “User level page faults,” Master’s thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA 15213, 2020.

[11] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating system concepts*, 10th ed. Hoboken, NJ: Wiley, 2018.Available: https://lccn.loc.gov/2017043464

[12] J. Choi, J. Kim, and H. Han, “Efficient memory mapped file I/O for In-Memory file systems,” Jul. 2017.Available: https://www.usenix.org/conference/hotstorage17/program/presentation/choi

[13] M. Prokop, “Inotify: Efficient, real-time linux file system event monitoring,” Apr. 2010. https://www.infoq.com/articles/inotify-linux-file-system-event-monitoring/

[14] “Transmission Control Protocol.” RFC 793; J. Postel, Sep. 1981. doi: 10.17487/RFC0793.

[15] “User Datagram Protocol.” RFC 768; J. Postel, Aug. 1980. doi: 10.17487/RFC0768.

[16] A. Langley *et al.*, “The QUIC transport protocol: Design and internet-scale deployment,” in *Proceedings of the conference of the ACM special interest group on data communication*, 2017, pp. 183–196. doi: 10.1145/3098822.3098842.

[17] H. Xiao *et al.*, “Towards web-based delta synchronization for cloud storage services,” in *16th USENIX conference on file and storage technologies (FAST 18)*, Feb. 2018, pp. 155–168.Available: https://www.usenix.org/conference/fast18/presentation/xiao

[18] B. K. R. Vangoor, V. Tarasov, and E. Zadok, “To FUSE or not to FUSE: Performance of User-Space file systems,” in *15th USENIX conference on file and storage technologies (FAST 17)*, Feb. 2017, pp. 59–72.Available: https://www.usenix.org/conference/fast17/technical-sessions/presentation/vangoor

[19] E. Blake, W. Verhelst, and other NBD maintainers, “The NBD protocol.” https://github.com/NetworkBlockDevice/nbd/blob/master/doc/proto.md, Apr. 2023.

[20] S. He, C. Hu, B. Shi, T. Wo, and B. Li, “Optimizing virtual machine live migration without shared storage in hybrid clouds,” in *2016 IEEE 18th international conference on high performance computing and communications; IEEE 14th international conference on smart city; IEEE 2nd international conference on data science and systems (HPCC/SmartCity/DSS)*, 2016, pp. 921–928. doi: 10.1109/HPCC-SmartCity-DSS.2016.0132.

[21] A. Baruchi, E. Toshimi Midorikawa, and L. Matsumoto Sato, “Reducing virtual machine live migration overhead via workload analysis,” *IEEE Latin America Transactions*, vol. 13, no. 4, pp. 1178–1186, 2015, doi: 10.1109/TLA.2015.7106373.

[22] T. Akidau, S. Chernyak, and R. Lax, *Streaming systems*. Sebastopol, CA: O’Reilly Media, Inc., 2018.

[23] J. D. Peek, *UNIX power tools*. Sebastopol, CA; New York: O'Reilly Associates; Bantam Books, 1994.

[24] gRPC Authors, "Introduction to gRPC." 2023.Available: https://grpc.io/docs/what-is-grpc/introduction/

[25] Redis Ltd, "Introduction to redis." https://redis.io/docs/about/, 2023.

[26] Redis Ltd, "Redis pub/sub." https://redis.io/docs/interact/pubsub/, 2023.

[27] Amazon Web Services, Inc, "What is amazon S3?" https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html, 2023.

[28] MinIO, Inc, "Core administration concepts." https://min.io/docs/minio/kubernetes/upstream/administration/concepts.html, 2023.

[29] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010, doi: 10.1145/1773912.1773922.

[30] P. Grabowski, J. Stasiewicz, and K. Baryla, "Apache cassandra 4.0 performance benchmark: Comparing cassandra 4.0, cassandra 3.11 and scylla open source 4.4," ScyllaDB Inc, 2021.Available: https://www.scylladb.com/wp-content/uploads/wp-apache-cassandra-4-performance-benchmark-3.pdf