Group No.:4
Topic: I/O

Atishay Jain - 20CS30008
Gaurav Malakar - 20CS10029
Sripuram Bhanuteja - 20CS10059
Roopak Priydarshi - 20CS30042

Rio: Order-Preserving and CPU-Efficient Remote Storage Access, EuroSys 2023:

The research paper discusses the challenges faced by existing networked storage systems in fully utilizing the high bandwidth and concurrency of modern NVMe SSDs and RDMA networks. These challenges primarily result from inefficient storage ordering guarantees, leading to severe synchronous execution that stalls CPU and I/O devices and hampers performance.These high-performance devices have significantly increased data transfer speeds, making **CPU efficiency a critical factor in storage and network systems.**

The **primary focus of the paper is on "storage order"**, which ensures data persistence and is essential for maintaining consistent disk states, especially in the event of a system crash. Traditional networked storage systems use synchronous approaches to ensure storage order, but these approaches underutilize network interface cards (NICs) and SSDs and leave CPUs idle.The paper explores existing approaches from local storage systems but finds that they hinder the efficient use of CPUs and I/O devices in networked storage, making it challenging to scale to multiple servers. For example, the "Horae" approach introduces a synchronous control path for storage order, which wastes CPU cycles and lowers I/O throughput.

To address these issues, the paper **introduces a novel approach called "Rio"** for remote storage access. The key concept behind Rio is to model the storage stack as a CPU pipeline, inspired by the **CPU's out-of-order execution and in-order commitment**. Rio introduces an I/O pipeline that allows internal out-of-order and asynchronous execution for ordered write requests while ensuring that applications still receive data in the correct external storage order. By merging consecutive ordered requests and making these design decisions, Rio achieves write throughput and CPU efficiency that closely matches that of orderless requests. This approach fully exploits NICs and SSDs, ensuring efficient CPU usage.

To address uncertainties and maintain data persistence, the authors introduce techniques like in-order submission and completion and leverage the in-order delivery of the network protocol. These techniques use a special structure called the "ordering attribute" to reconstruct the original storage order, even in asynchronous execution.

The paper presents the design and implementation of Rio. Rio operates across initiator and target servers. Target servers consist of one or more NVMe SSDs, and they **require at least one NVMe SSD with PMR suppor**t or a small region of byte-addressable persistent memory in the initiator server. In the initiator server, the entire storage stack, including the file system, block layer, and device driver, is **revised to allow asynchronous flow of ordering**. A shim layer called "Rio sequencer" is introduced between the file system and the block device to **package the original orderless block abstraction as an ordered one**. Rio's design focuses on controlling the order of ordered write requests at the start and end of their lifetime while permitting some **out-of-order execution** in between. When ordered write requests are initiated, Rio sequencer generates a special ordering attribute for each request and dispatches them asynchronously to the block layer. When completed, the requests are processed in order using these ordering attributes to handle temporary out-of-order execution. This ensures that the file system and applications see the original ordered state. Rio's approach **allows more outstanding requests to be sent to NICs and SSDs**, taking full advantage of the concurrency capabilities of modern NICs and NVMe SSDs. Moreover, Rio enables easy scaling to more target servers, as there are no ordering constraints on the data transfer of ordered write

requests. Asynchronous execution in Rio creates post-crash uncertainty, and Rio addresses this with a persistent ordering attribute that **logs the persistent state of data blocks** of each ordered write request. This allows Rio to speculate on possible post-crash states and recover data blocks to the latest and ordered state in an asynchronous and concurrent manner. Rio's asynchronous design also allows for the **staging and merging of consecutive ordered write requests** to reduce CPU overhead. This optimization reduces the number of commands and operations required, thus conserving CPU cycles at the device drivers.

Authors implemented Rio in the Linux NVMe over RDMA stack and created a file system named RioFS on top of Rio. Their evaluations demonstrate that Rio outperforms both Linux NVMe over RDMA and Horae by significant margins, with up to two orders of magnitude higher throughput for ordered write requests. RioFS also significantly enhances the performance of RocksDB compared to Ext4 and HoraeFS.

Efficient and Safe I/O for Intermittent Systems, EuroSys 2023:

In recent times, there has been a notable shift towards creating smaller and highly energy-efficient computing devices, used in various fields like IoT gadgets, wearable technology, and even specialised medical sensors that can be placed inside or on the body. These devices leave batteries behind and function entirely on the energy harvested from the surroundings like solar energy,thermal energy.Rather than relying on conventional batteries, these devices store the harvested energy in compact capacitors. This stored energy is then utilised to perform various functions including sensing, actuation, inference, computation, and communication. Operation in these devices is intermittent because energy is not always available to harvest and, even when energy is available, buffering enough energy to do a useful amount of work takes time. Task based Intermittent Systems are systems
Where a task is considered to be atomic.As the tasks have all-or-nothing semantics when on a power failure they re-execute peripheral I/O operations.They waste energy and time when they re-execute these peripheral IO operations. Also there are some challenges faced by these Task based Intermittent
Systems often lead to wasteful I/O on re-execution ,memory inconsistencies when dealing with nonvolatile memory, and unsafe program execution.

In many systems, when a task is interrupted by a power failure, it restarts from the beginning, which means all the input/output (I/O) operations inside the task are unnecessarily repeated. For example, a camera doesn't need to take a picture again if the last attempt was successful. Unfortunately, existing systems lack the capability to recognize when repeating an I/O operation is unnecessary, resulting in a significant waste of time and energy.Redundant re-executions might even lead to a non-termination bug since I/O operations increase the task's energy requirements, which may exceed the energy buffer's capacity.In case of applications that handle large amounts of data, they often transfer data between volatile and non volatile memory requiring peripheral operations.Performing such operation might lead to *memory-inconsistency* if they modify non volatile memory directly.The problem arises from the potential impact of I/O operations on the decision-making process within a task. Specifically, these operations can influence the conditions that determine the branching of a task, where each branch may interact with different parts of non-volatile memory. In the event of a power failure and subsequent task restart, it's possible for the task to take a different branch than it did in the previous energy cycle.This discrepancy arises because a repeated I/O operation might yield a different output.

These are some of the problems faced by intermittent task based systems.There are existing runtimes for intermittent computing .Existing task-based studies face challenges with inefficient resource utilisation due to a lack of comprehensive programming language support. Additionally, they often overlook the critical aspect of re-execution semantics for individual

peripheral operations, leading to unnecessary repetition of I/O tasks .This paper introduces *EaseIO*(EfficientAndSafE I/O*)* a programming language and runtime , allowing programmers to introduce I/O re-execution semantics

Avoiding unnecessary re-execution after reboot. EaseIO provides APIs and compiler annotations that assist programmers in annotating different code regions that may potentially lead to memory inconsistencies across system reboots.It also introduces a new method Regional Privatization, which equips developers with the means to establish protective measures against memory inconsistencies specific to DMA-based I/O operations.

The EaseIO subsystem introduces three keywords to specify peripheral re-execution behaviour:

Single: Instructs the runtime to execute a peripheral operation only once.

Timely: Indicates that the peripheral operation has timeliness constraints. If the data from the last I/O operation is still valid, the runtime skips re-execution.

Always: Directs the runtime to re-execute peripheral operations after each power failure.

EaseIO provides interfaces to simplify the integration of re-execution semantics into intermittent applications. The main interfaces offered by EaseIO are

_call_IO(name,type..) : The _call_IO interface abstracts the execution of peripheral operations, managing re-execution in case of power failure based on annotated semantics at runtime. It can handle both void functions and functions that return values, with the compiler front-end creating a private copy of return values in non-volatile memory.

_IO_block_begin(type,..) ,_IO_block_end : These interfaces define the boundaries of atomic execution for multiple I/O functions, considering EaseIO's semantic annotations.

_DMA_copy(*src,*dst,size): This handles the block data copy via DMA peripheral, dynamically adjusting re-execution semantics based on memory types to prevent inconsistency. For instance, copies between non-volatile memories are treated as Single, while those between volatile memories are marked as Always for guaranteed repetition.

Semantic-aware I/O Re-execution Implementation

To implement the "*single*" semantics a boolean flag is declared to track the I/O completion.Ease IO maintains a nonvolatile copy of the return value.In Case it fails to re-execute for some reason it will simply restore the value using this copy.To implement the "*timely*" semantics it maintains a variable(non-volatile) to store the last time this operation is executed.It checks to see if the last result is still valid else it re-executes the peripheral I/O operation.For "*Always*" semantics it relies on task-based models to help re-execute I/O operations.

Enabling Memory-Safe DMA Operations: DMA can modify nonvolatile memory without the intervention of CPU , it can result in memory-inconsistencies as pointed out earlier.EaseIO decides the semantics of a DMA operation based on source type and destination type.

1. **Volatile to Volatile:** In this case it is re-executed every time as these values don't persist across reboot. Ease IO annotates this as "always".
2. **Volatile/non-volatile to non Volatile:** In this case the destination points to non volatile memory.If the previous execution was successful.It retains this value after a power failure ,so there is no need for re-execution.So the EaseIO annotates this as "single".

Regional Privatisation: In intermittent computing environments, non-volatile variables manipulated by the CPU can pose a challenge for memory consistency, particularly when they are involved in DMA operations. Take the example of a task that copies a value from one non-volatile buffer to another. The runtime would label this operation as "Single," indicating it should not be re-executed. However, bypassing re-execution of such a DMA operation would lead to data inconsistencies.

To ensure safe program execution the easeIO handles this using *Regional Privatisation* .It divides tasks into multiple regions based on DMA operation locations. A task containing 'N' DMAs is split into 'N+1' regions.The compiler establishes regional privatisation and recovery procedures at the outset of each region. Dedicated private copies are generated to ensure

region-specific consistency. A flag is then set at the end of the process, signalling the '_DMA_copy' function for seamless atomic execution of DMA and privatisation. If a power failure occurs post-regional privatisation, EaseIO leverages private copies to restore non-volatile variables. This sequential approach effectively mitigates inconsistencies that might result from DMA non-re-execution in the subsequent energy cycle, enhancing program safety by eliminating unvisited paths in continuous execution.

This paper holds significant value because it introduces EaseIO, a new programming language and system that addresses critical challenges in intermittent computing. It offers efficient handling of I/O operations and addresses the memory challenges faced by Task-based intermittent systems . These advancements are crucial for creating reliable and sustainable batteryless devices, which is an emerging area of interest in OS research.

Asynchronous I/O Stack: A Low-latency Kernel I/O Stack for Ultra-Low Latency SSDs, ATC 19:

The current I/O stack implementation involves numerous steps to handle a single I/O request. For instance, when an application sends a read I/O request, it first allocates memory for data storage and creates an index in a page cache. After that, it establishes a direct memory access (DMA) mapping and manages various auxiliary data structures, like bio, request, and iod in Linux. The problem with this approach is that all these operations happen one after the other in a synchronous manner before the actual I/O command is sent to the storage device.

In the past, older secondary storage devices were pretty slow, so the operations we talked about didn't take much time compared to how long it took to access the actual storage data. But now, with ultra-fast SSDs like the Samsung Z SSD and Intel Optane SSD, the time taken by the storage device itself has decreased a lot and is now close to the time the CPU in the kernel's I/O stack spends on various small tasks during an I/O operation.

The thing is, most of these CPU operations during I/O aren't directly related to the storage device. They can be done simultaneously, or in parallel. The asynchronous I/O stack is based on this very concept. It tries to do as many tasks at the same time as possible to reduce the overall time needed to complete an I/O operation.

The paper introduces a brand-new way of handling the kernel I/O stack. It does a lot of tasks simultaneously with the storage device, which makes things much faster. This new approach includes a lightweight block I/O layer (LBIO), which is a change to the existing multi-queued block layer. LBIO makes use of some of the time-consuming tasks from the original block I/O layer.

There are a couple of other important changes too. One is called "Lazy page cache indexing," and the other is "Lazy DMA unmapping." In the old way of doing things, these tasks happened at the same time as everything else. But with this new approach of AIOS (asynchronous I/O stack), these tasks are delayed until the system isn't busy or waiting for another I/O request.

Because of these changes, AIOS can achieve really fast I/O performance, with latency in just a few microseconds when used with an Optane SSD. This kind of speed wasn't possible before because the traditional I/O stack had too much overhead.

In my view, this paper is a significant breakthrough in the field of operating systems research, and it sets the stage for even faster I/O operations in the future. Given the rapid advancements in technology, we can expect more and more high-speed secondary storage devices in the future. This opens up opportunities to further optimize the CPU operations related to these devices, to make the whole process of accessing them incredibly smooth.

Who knows, one day we might even have I/O devices that are as fast as our computer's RAM, making data access almost instantaneous.

"GIFT: A Coupon Based Throttle-and-Reward Mechanism for Fair and Efficient I/O Bandwidth Management on Parallel Storage Systems", FAST 20:

Large-scale parallel applications in high-performance computing (HPC) are known for their intense data processing needs. They regularly perform massive I/O operations, which involve reading from and writing to storage devices. The challenge arises when multiple such applications run simultaneously on a powerful computing system. This concurrent execution can lead to I/O contention, where these applications compete for access to storage resources. I/O contention causes inefficiencies and unfair resource allocation. Some applications may get a disproportionate share of the system's resources, while others are left waiting, hindering the overall efficiency of the HPC system.

The paper recognizes that this problem is exacerbated as computing power continues to increase. Scientific applications are producing even more data and spending a significant portion of their execution time on I/O operations. This situation is expected to worsen, especially with the emergence of exascale systems, which are capable of performing a quintillion calculations per second. These systems are expected to handle even larger workloads, making effective I/O management increasingly critical.

To make matters more complex, many HPC applications require synchronous I/O progress. This means that different parts of an application, running on multiple nodes or processors, need to coordinate their I/O operations. All processes must wait for the slowest one to complete its I/O task before moving forward. This synchronization is necessary for the proper functioning of these applications but can exacerbate I/O contention issues.

To address these challenges, the paper introduces GIFT, a novel approach to manage I/O contention and resource allocation in large-scale parallel applications. GIFT shifts away from the traditional concept of ensuring fairness at each individual I/O operation. Instead, it focuses on fairness over a series of I/O phases and runs of an application. It leverages the fact that HPC applications often exhibit similar behavior across runs. This means GIFT doesn't require instantaneous fairness but seeks fairness over a more extended period. Also, it adopts a strategy where it selectively throttles certain applications temporarily to reduce I/O bandwidth waste. This means slowing down specific applications at times to improve the overall utilization of the system's I/O resources. Importantly, GIFT ensures that the I/O progress within each application remains synchronized, which is crucial for HPC applications.

GIFT allocates I/O bandwidth fairly among all competing applications while ensuring synchronous I/O progress within each application. This design principle eliminates the inefficiency caused by non-synchronous I/O progress. GIFT uses linear programming to optimally allocate bandwidth to applications, aiming to minimize the amount of wasted I/O bandwidth. When an application is temporarily slowed down (throttled), it receives "coupons" proportional to the degree of throttling.

Later, GIFT redeems these coupons to ensure fairness. If, for some reason, a coupon cannot be redeemed for an application, GIFT compensates the application with compute node-hours from a special budget. This approach ensures that fairness is maintained even when the original coupons can't be used. GIFT isn't just a theoretical concept; it has been designed and implemented in a real system prototype based on the FUSE file system, showing that these ideas can work in practice and are available for the broader community.

The evaluation of GIFT demonstrates that it can significantly reduce wasted bandwidth caused by I/O contention, leading to better system performance, fairness among applications, and overall system efficiency. This approach enhances the effective system I/O bandwidth and application I/O times by a substantial margin, even in scenarios with high contention and various application characteristics. GIFT offers a practical solution to address I/O contention in large-scale parallel applications.