

Paper Review (I/O)

Advances in Operating System Design

Presented by:

Atishay Jain 20CS30008

Gaurav Malakar 20CS10029

Roopak Priydarshi 20CS30042

Siripuram Bhanu Teja 20CS10059

Efficient and Safe I/O Operations For Intermittent Systems

Included in the Proceedings of 2023 EuroSys

Intermittent Computing

With the Predictions of more than a trillion IOT things by 2035,the concerns for cost and maintenance have been raised.

These devices doesn't rely on a continuous and reliable power supply,instead they harvest energy from environment and compute only when enough power is buffered up ,they compute intermittently.

Power failures can occur at any time during the execution, so they have to restart the execution from the saved state which often leads to several problems.

This paper aims to address those challenges

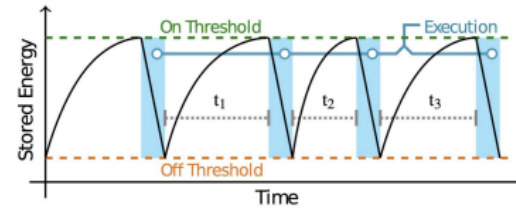
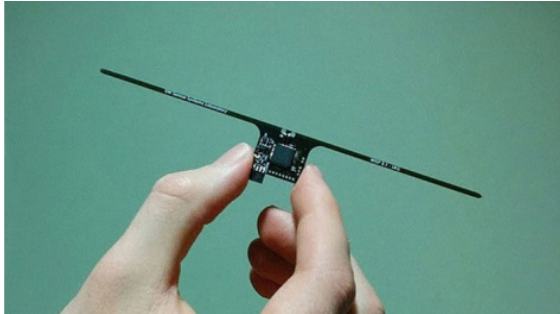


Figure 1. Energy harvesting devices compute intermittently with execution times unpredictably apart from each other. This makes it important to ensure efficient use of energy in order to ensure maximum program progress.

Challenges Faced by Intermittent Systems

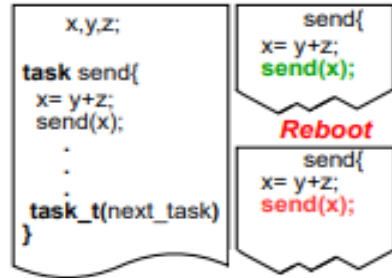
In task-based intermittent systems programs are divided into tasks, which have all or nothing semantics. So in case of power failure it will restart the task.

This leads to following challenges

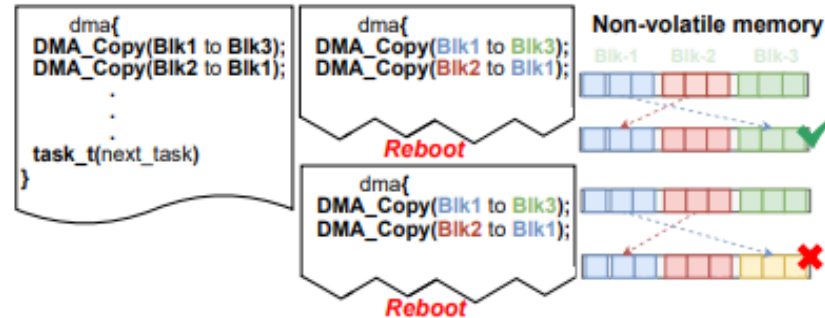
Wasteful Repetitive I/O Operations

Idempotence Bugs.

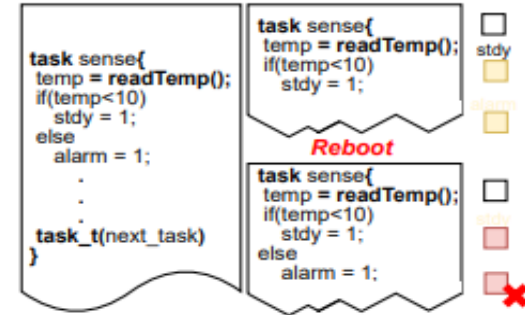
Unsafe Program Execution.



(a) Wasteful I/O Operation



(b) Idempotence Bugs



(c) Unsafe Program Execution.

Ease IO Overview

- The current intermittent runtimes lack programming language support to capture re-execution semantics ,leading to repeated I/O.
- Ease I/O provides programming language semantics that allows programmer to specify the I/O re-execution semantics and a run time that uses this information avoiding repetitive I/O and memory inconsistencies.
- Ease I/O introduces 3 keywords to specify re-execution semantics
 - **Single:** This conveys the runtime that the I/O operation should only execute only once .
 - **Timely:** This tells the runtime that the given I/O operation has timeliness constraints .In case the I/O operation succeeded previously and if the result is valid the I/O operation is need not be executed again.
 - **Always:** This directs the runtime that the given I/O operation should be executed every time or failure.

```
Task T1(){  
    int temp,humd;  
  
    _IO_block_begin ("Timely",10)  
    ..  
    pres= _call_IO(Pres(),"Single");  
    ..  
    _IO_block_end  
  
    temp= _call_IO(Temp(),"Timely",50)  
    humid= _call_IO(Humd(),"Timely",20)  
    _call_IO(Send(temp,humd),"Single")  
}
```

C Source File

- Ease I/O exposes programming interface for programmers to use the semantics mentioned before

__call_IO :

This interface lets users to specify the re-execution semantics for the I/O operation .

__IO_block_begin/_IO_block_end :

These interfaces define the the boundaries of atomic execution for multiple I/O functions.

__DMA_copy:

This handles the block data copy via DMA peripheral ,dynamically adjusting the re-execution semantics based on memory types to prevent inconsistency.

Implementation details of Re-execution Semantics

- Single semantics are implemented using a dedicated boolean flag that keeps track of the completion of I/O operation. And to restore the correct return value of the function incase if it fails to get re-executed (i.e it completed execution in previous cycle) it avoids data inconsistencies by maintaining a non volatile copy of the return value.
- Timely semantics are implemented by maintaining two non volatile variables one storing the timestamp of last execution and other flag tracking the I/O completion.
- Always semantics relies on task-based models and doesn't need new logic to implement re-execution semantics.

```
time_flg_tmp = GetTime()-time_temp)<50;
if(!(flag_temp)||!(time_flg_tmp) {
    temp_priv = Temp();
    time_temp = GetTime();
    flag_temp = SET;
}
temp = temp_priv;
```

Evaluation

1. Energy Consumption: Energy consumed to finish the single execution of target application.

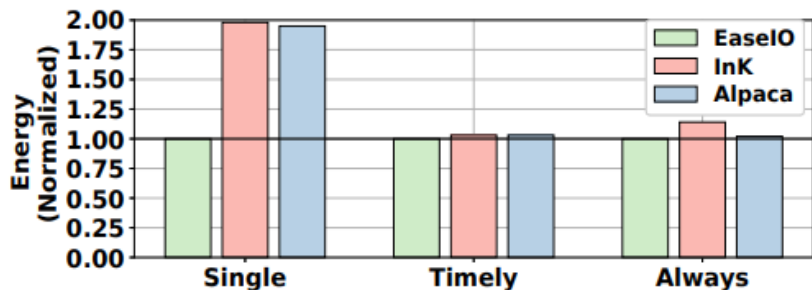
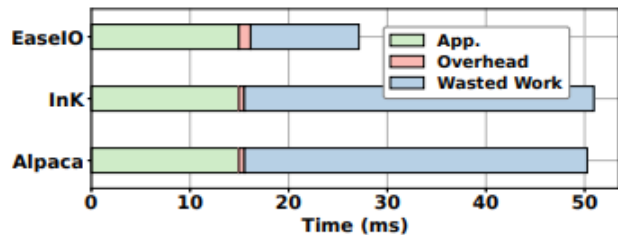


Figure 8. Average energy consumption of for re-execution I/O semantics with controlled power failures.

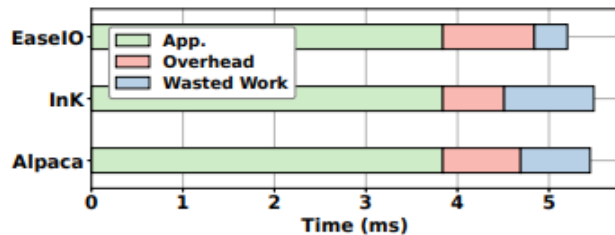


Figure 11. Average energy consumption of multi-task applications with controlled power failures.

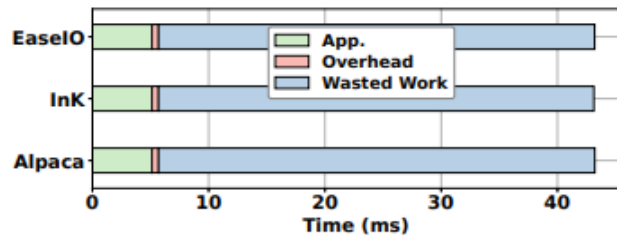
2.Wasted Work :



(a) Single semantic - NVM to NVM DMA



(b) Timely Semantic - Temperature Sensing



(c) Always Semantic - LEA

Figure 7. Total execution time, runtime overhead, and wasted work with controlled power failures.

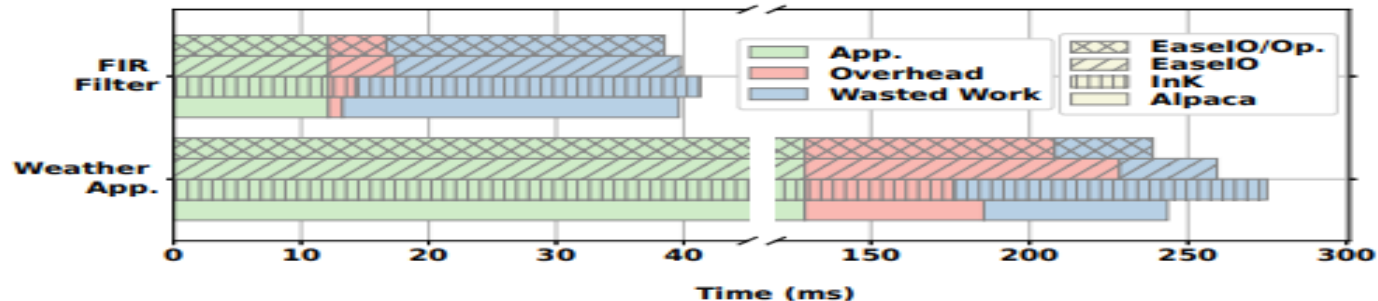


Figure 10. The execution time, runtime overhead and wasted work of weather classifier and FIR filter with controlled power failures.

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

Included in the
Proceedings of the 2019
USENIX Annual Technical
Conference.

(USENIX ATC '19)

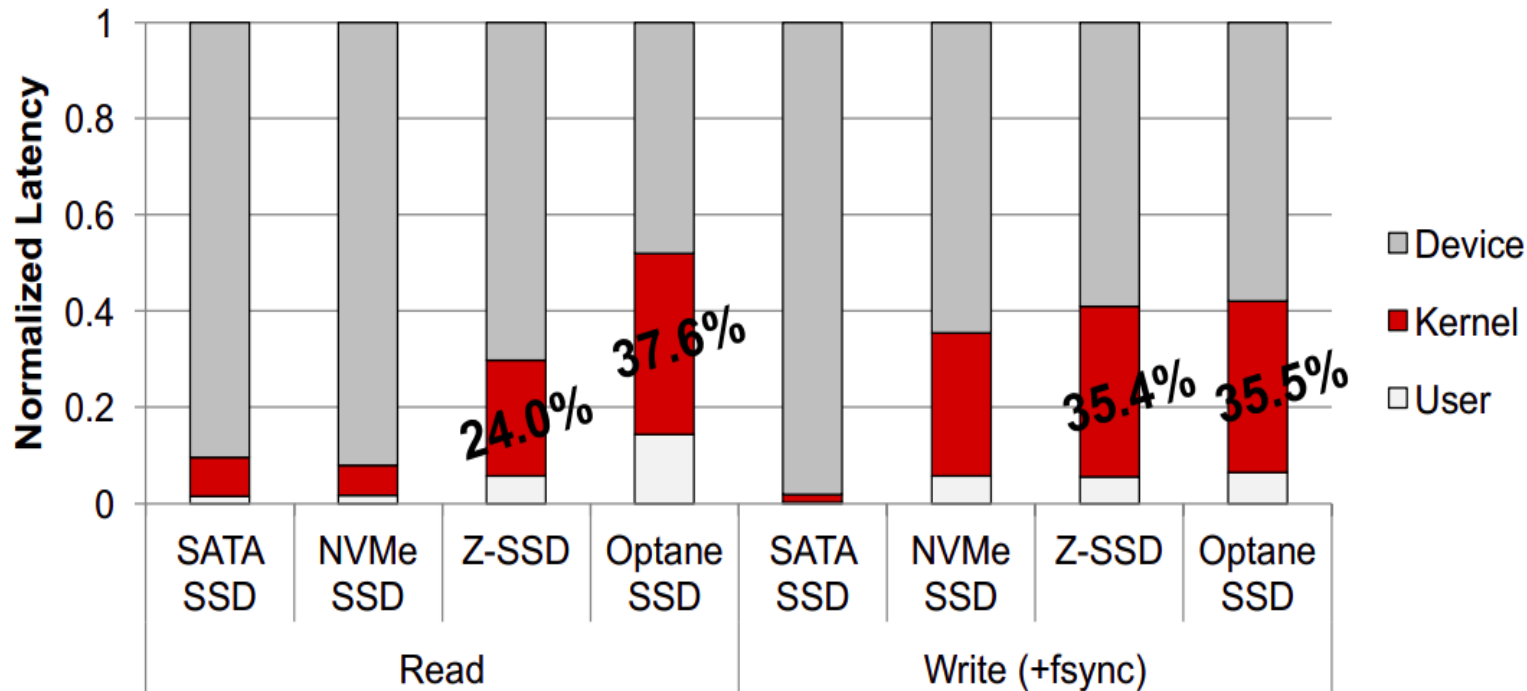


How an I/O request is handled

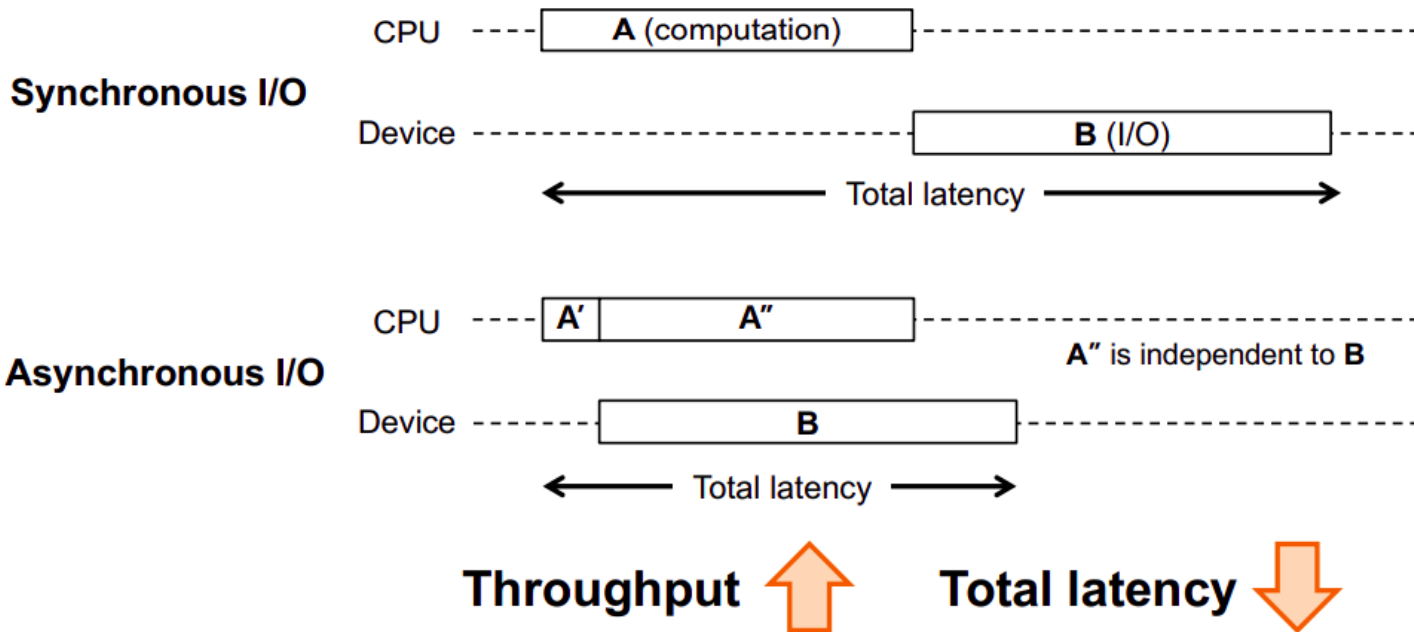
- **Layered Process:** An I/O request traverses multiple layers, including the file system, block layer, and device driver.
- Handling of a single I/O request needs completion of multiple kernel level operations before that.
- **Synchronous Operations:** Key operations like page allocation and DMA mapping occur in a synchronous manner before the actual I/O command.
- The device driver handles the final submission and completion of I/O commands.

“ For traditional storage devices, the time spent on kernel operations is relatively negligible compared to the time consumed by the device itself “

Kernel I/O Stack Overhead



Asynchronous I/O Stack



Kernel operations during I/O are largely device-independent and can be parallelized.



Traditional Read Path

- **Buffered Read System Calls**

- Entry through *read()* and *pread()* calls.
- Leads to Virtual File System (VFS) function which in turn leads to cache layer

- **Page Cache Process**

- Cache miss triggers allocation and indexing of a new page in the page cache.

- **DMA Mapping**

- Creation of Direct Memory Access (DMA) mapping for the allocated pages.

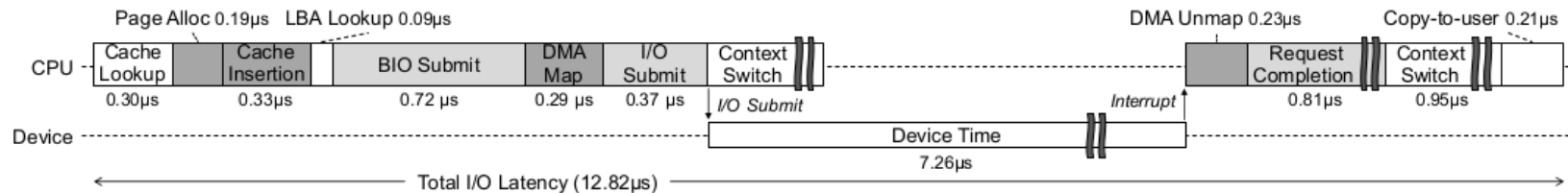
“The synchronous nature of these operations leads to inefficiencies”



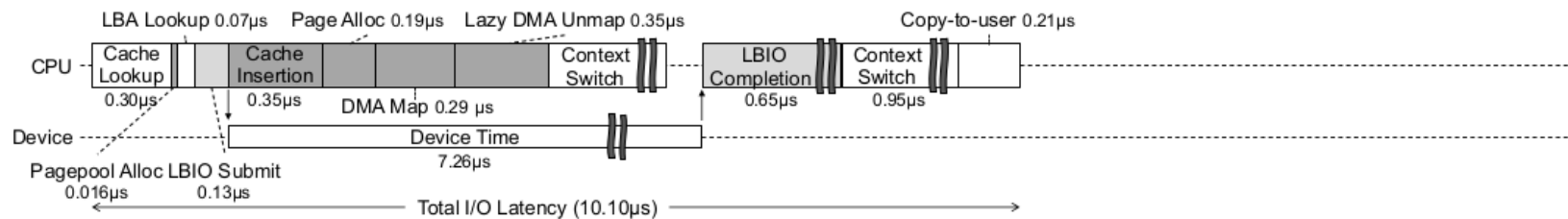
Asynchronous Read Path

- **Shift to Asynchronous Operations**
 - Overlaps CPU and device I/O tasks to reduce end-to-end I/O latency.
- **Efficient Page Management**
 - Asynchronous page allocation and DMA mapping performed parallel to device I/O.
- **Lazy DMA Unmapping**
 - Delays DMA unmapping to idle times, reducing critical path duration.
- **Lightweight Block I/O Layer**
 - Customized LBIO layer for NVMe SSDs minimizes I/O request submission delay.

Comparison of Read Paths



(a) Vanilla read path



(b) Proposed read path

21% latency reduction



Linux Block I/O Layer

- **Multi-Queue Architecture:** Designed for NVMe SSDs, scales well with multi-core CPUs
- **Core Functionalities:** Handles block I/O submission/completion, request merging/reordering, I/O scheduling, and command tagging.
- **I/O Scheduling and Merging:** Supports various I/O schedulers; default often set to 'noop' for fast storage devices.
- **Challenges with Ultra-Low Latency SSDs:** Time spent in block layer becomes significant part of total I/O latency. (About **15%** of the device time)



Light-weight Block I/O Layer

- **Simplified Operations**

- Focuses on essential functions: I/O submission/completion
- Eliminates complex operations like bio-to-request transformation

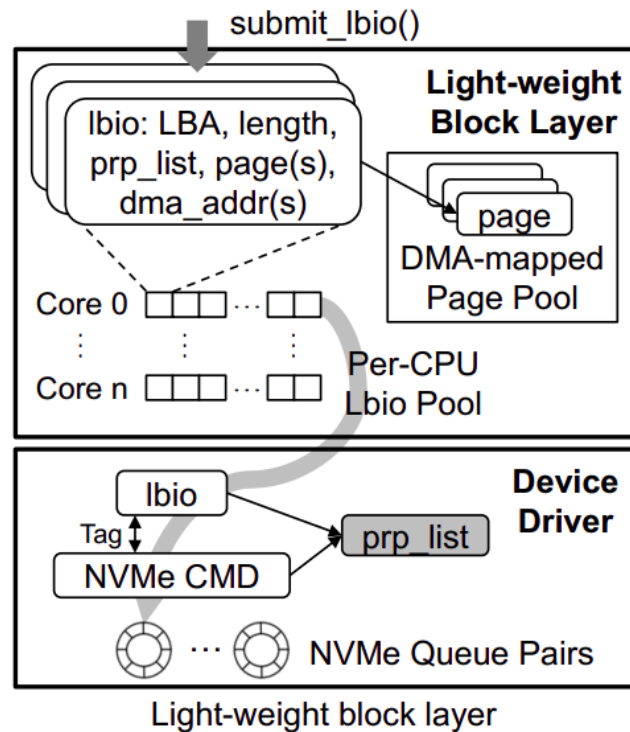
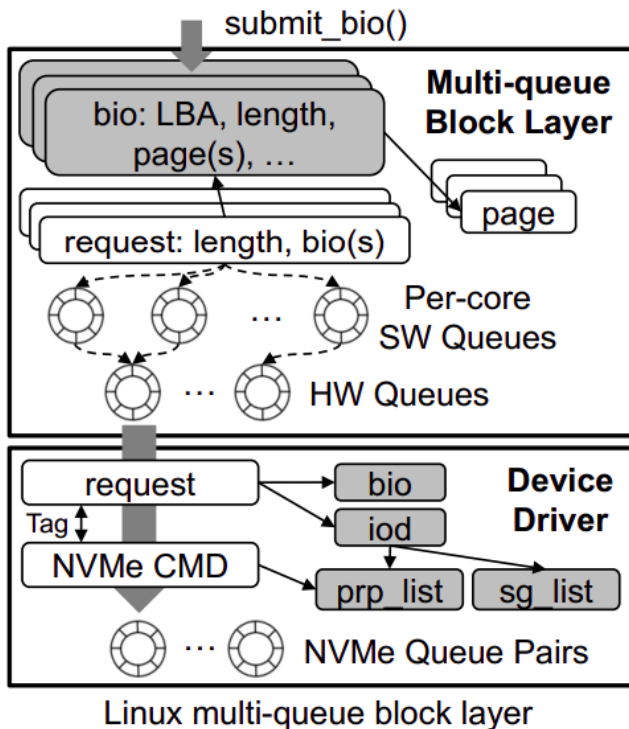
- **Efficient Data Structure**

- Utilizes a single lbio memory object for each block I/O request
- Features a global lbio array, with dedicated rows for each core and NVMe queue pair, so locks are no longer needed

- **Direct Command Dispatch**

- Enables threads to directly dispatch NVMe I/O commands, bypassing I/O merging and scheduling.

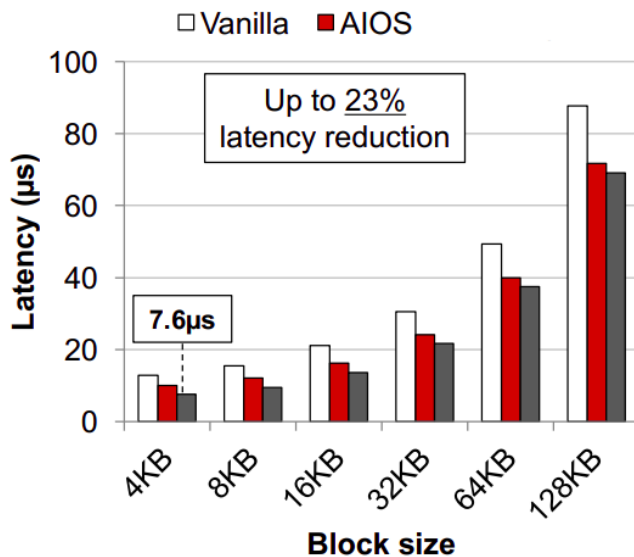
Block I/O Layers Compared



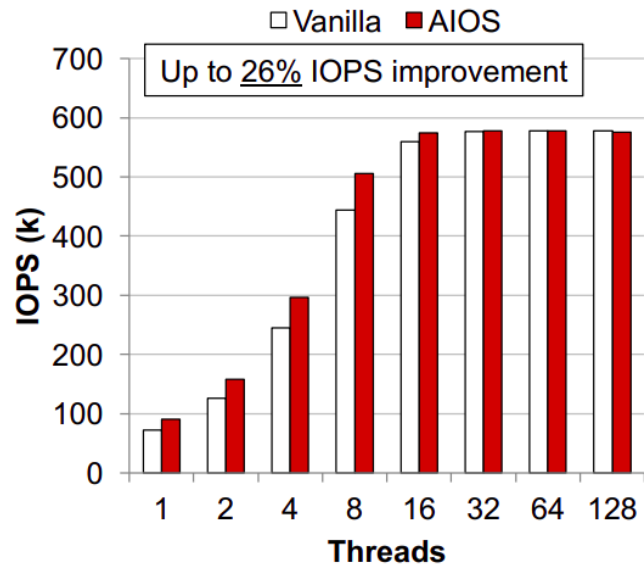


Read Performance Results

- **Single thread**



- **4KB block size**



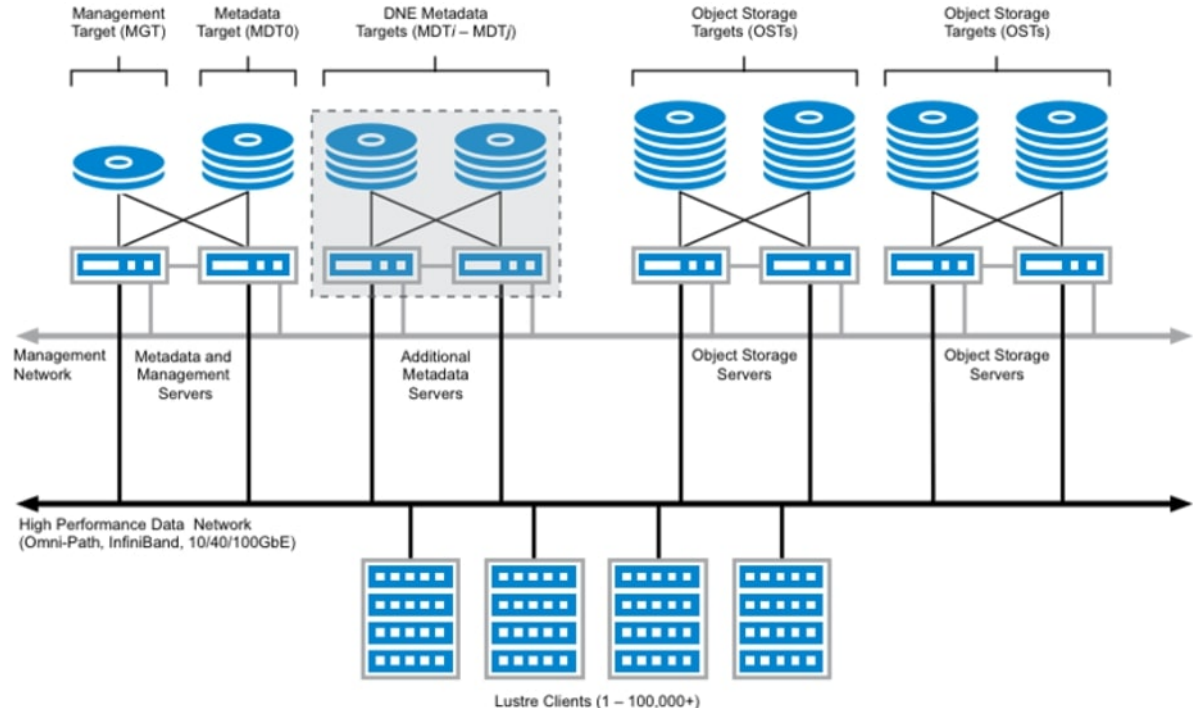
GIFT: A Coupon Based Throttle-and-Reward Mechanism for Fair and Efficient I/O Bandwidth Management on Parallel Storage Systems

Included in the Proceedings of the
18th USENIX Conference on File and
Storage Technologies (FAST '20)

High Performance Computing

Data Intensive parallel applications switch between compute and I/O phases.

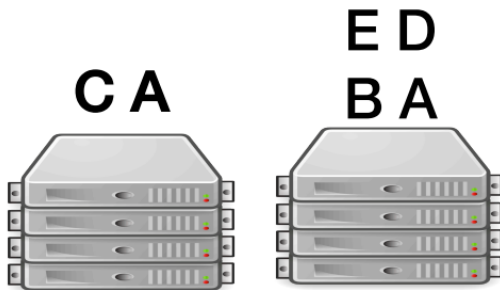
The architecture of a typical parallel storage system of a high performance computing system is given in the diagram.



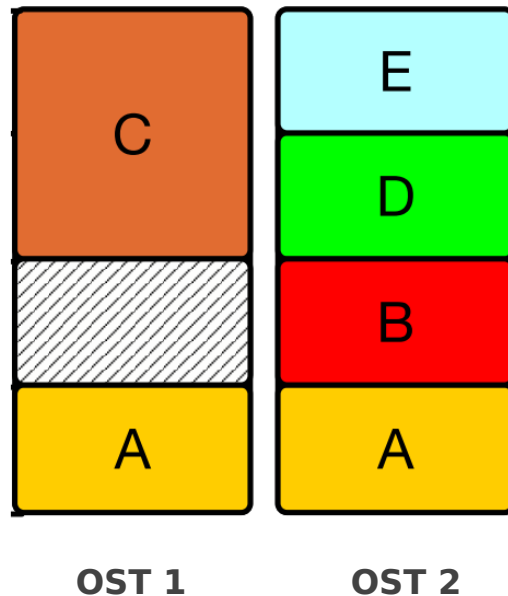
I/O in parallel applications

Parallel applications can cause unmanaged and unpredictable I/O interference!

One application performs I/O concurrently to multiple OSTs:

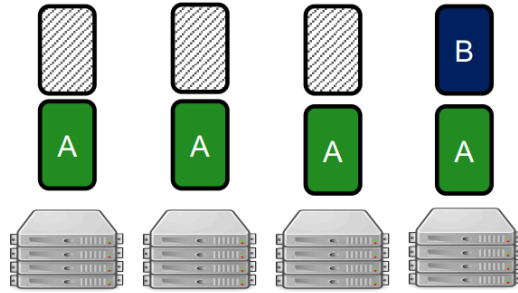


Object Storage Targets (OSTs)

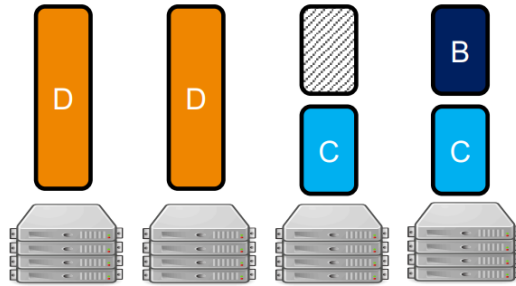


Inefficient I/O bandwidth utilization

Working of GIFT

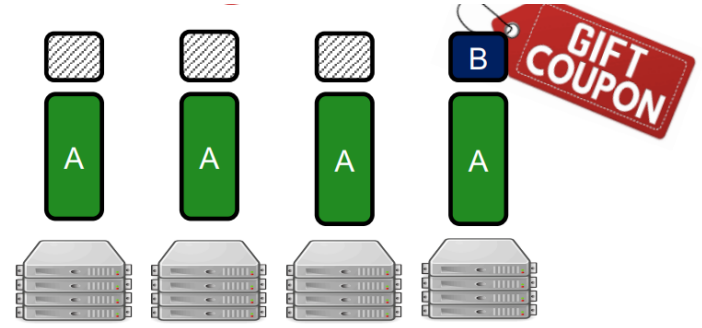


Time t1

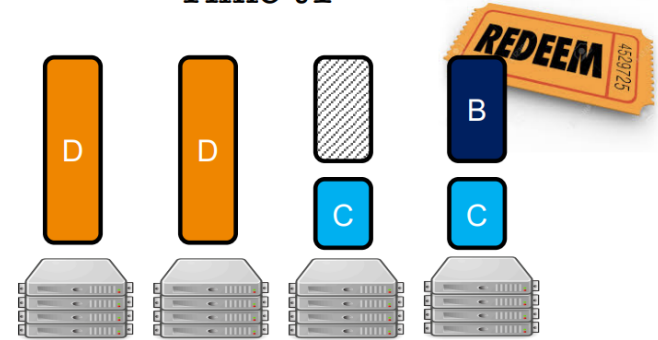


Time t2

Traditional



Time t1

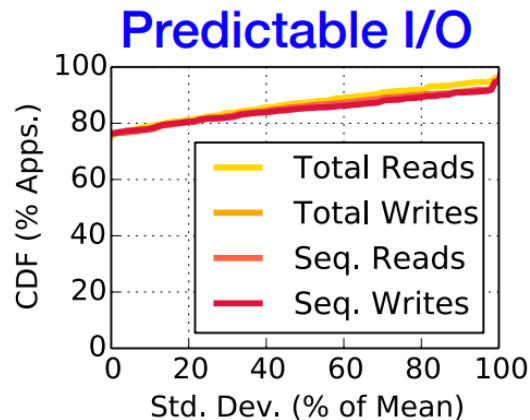
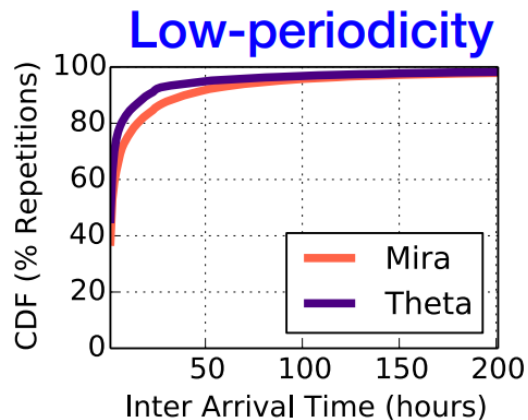
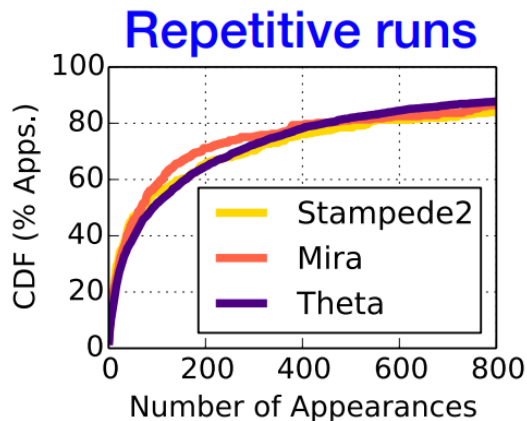


Time t2

GIFT

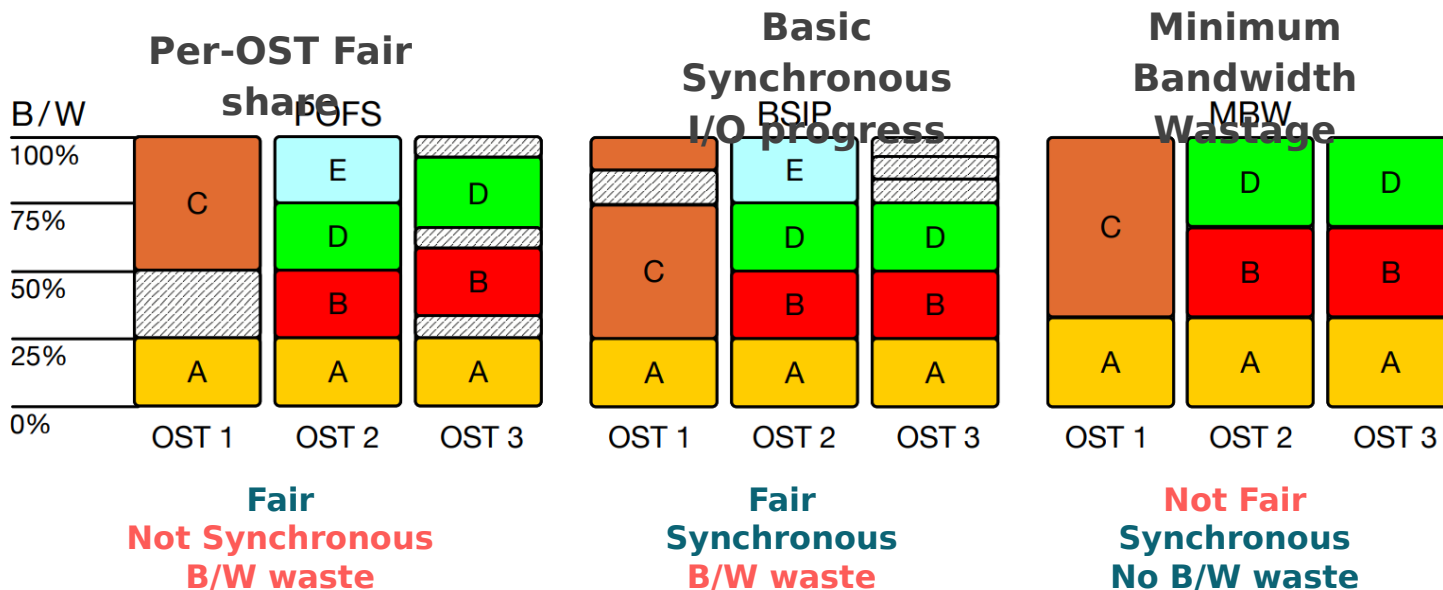
Why does it work?

HPC applications run repeatedly, are frequent, and exhibit similar I/O behavior across different runs:



GIFT challenges

- Parallel applications suffer from non-synchronous I/O progress leading to bandwidth wastage.
- Need for synchronous I/O progress in parallel applications poses new challenges in maintaining efficiency and fairness in I/O bandwidth allocation. Some bandwidth allocation policies:



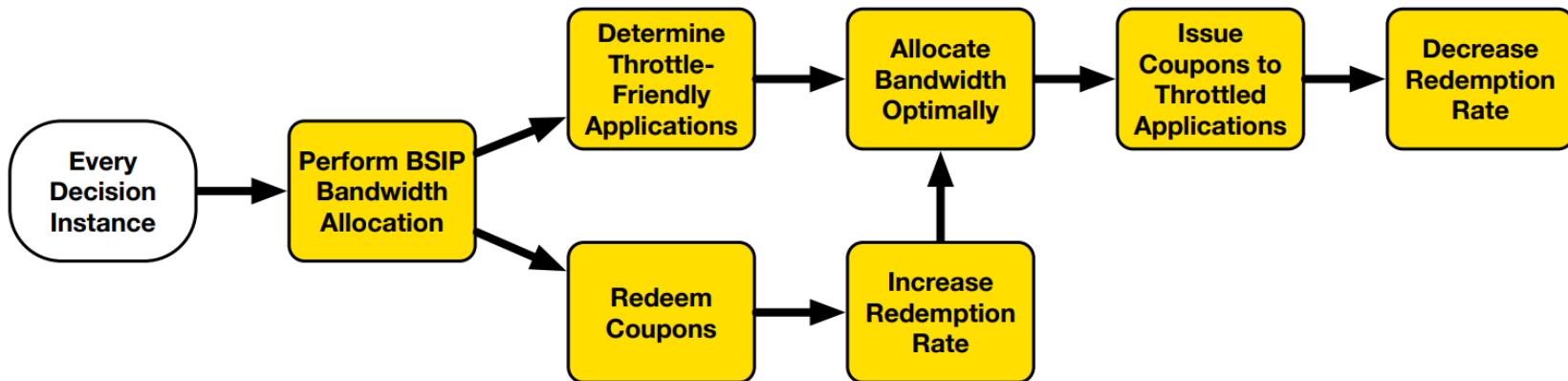


Key ingredients of GIFT:

01	Fairness	GIFT breaks away from instantaneous fairness and maintains fairness over a long time-window.
02	Synchronous I/O Progress	GIFT's initial allocation is the same as BSIP scheme and any subsequent readjustments ensure that this property is preserved.
03	Minimize Bandwidth Wastage	Follows "throttle-and-reward" mechanism that picks "throttle-friendly" applications, issues them coupons to reduce bandwidth wastage at a given time, and "reward" them later.

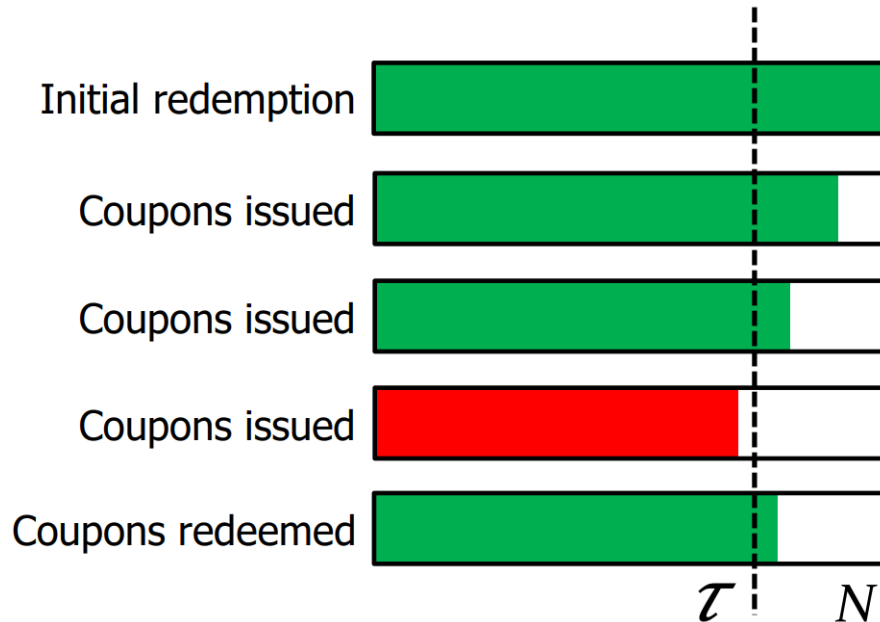


GIFT workflow:



Throttling applications:

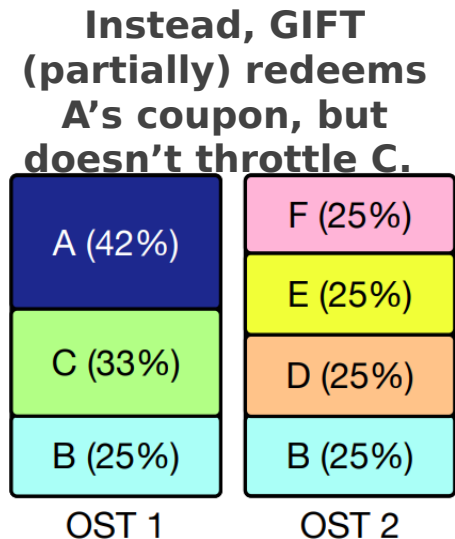
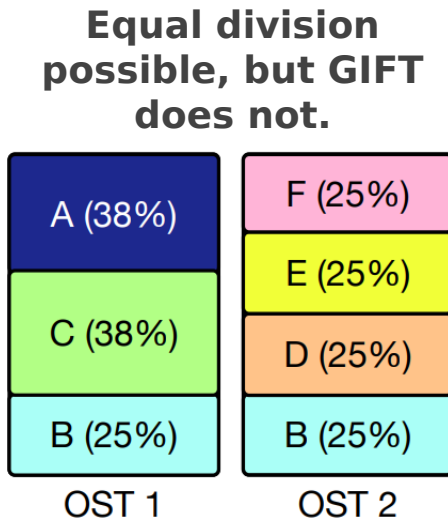
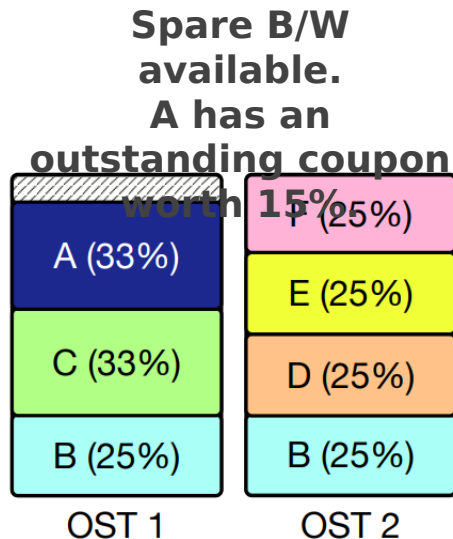
- Careful design leads to minimal system regret budget (compute hours given out due to unfair treatment in long term).
- Throttle-friendly apps can also be expanded if deemed beneficial.
- Set of throttle-friendly applications changes over time.



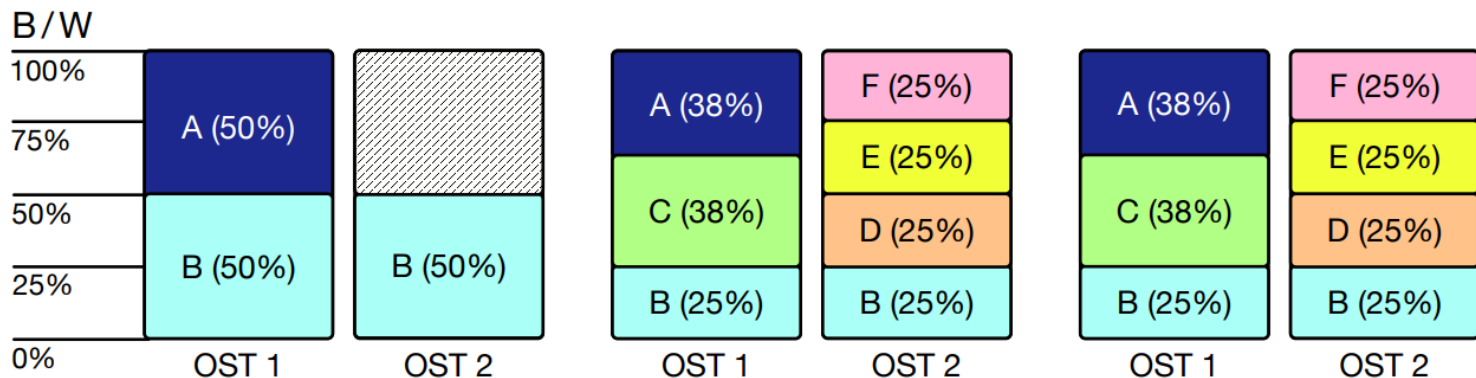


Coupon redemption:

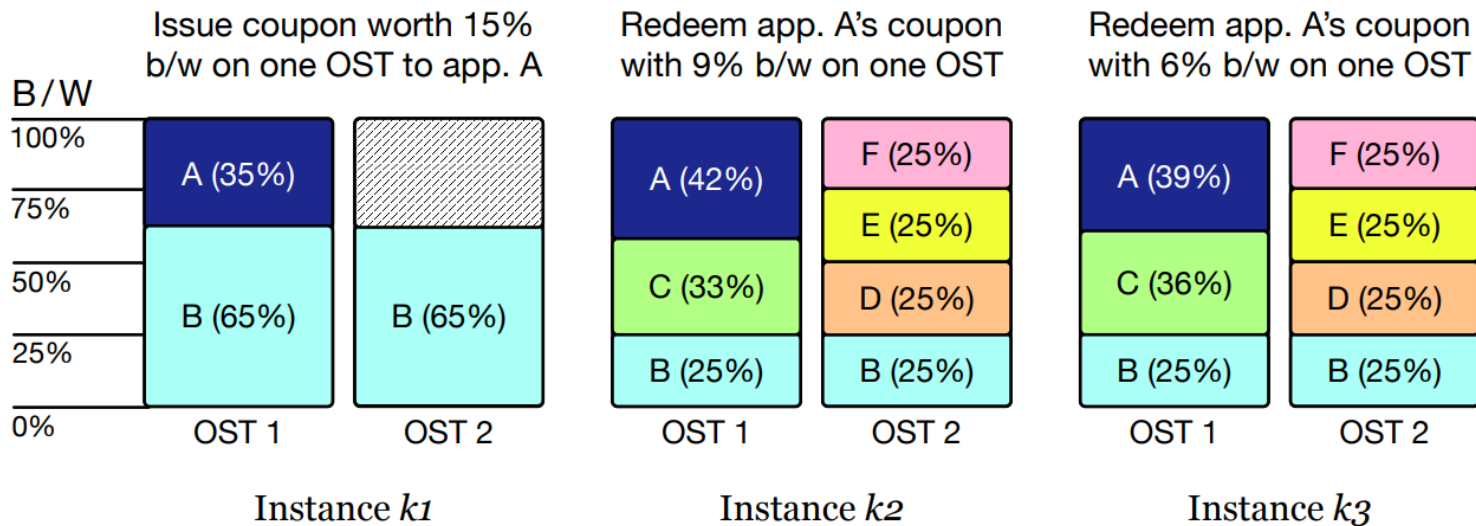
GIFT redeems coupons only when it does not require throttling other applications:



BSIP



GIFT



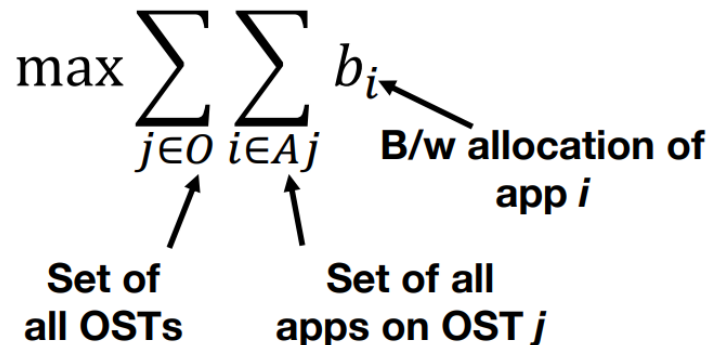
Optimal Bandwidth Allocation:

Formulated as a linear programming optimization problem, subject to some constraints:

- All I/O requests of an application issued across all OSTs should get the same B/W for synch. I/O progress.
- The final B/W allocation should be fair.
- All OSTs are constrained by their full capacity.

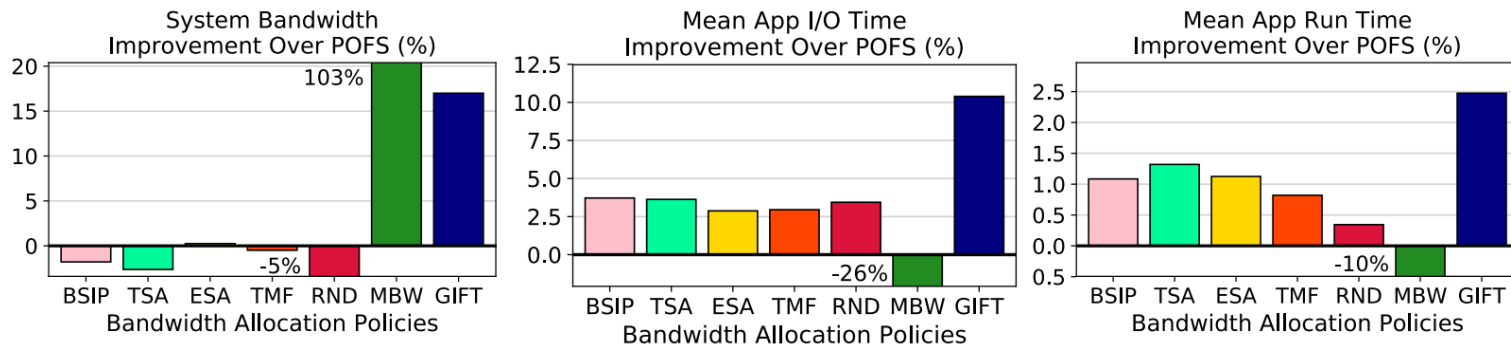
$$\max \sum_{j \in O} \sum_{i \in A_j} b_i$$

Set of all OSTs **Set of all apps on OST j** **B/w allocation of app i**

The diagram shows the mathematical expression for optimal bandwidth allocation. It consists of a maximization problem: max over the sum of b_i for all applications i on all OSTs j. Annotations with arrows point to each part: 'Set of all OSTs' points to the j in the first sum; 'Set of all apps on OST j' points to the i in the second sum; and 'B/w allocation of app i' points to the b_i term.

Evaluation and Analysis:

GIFT real-system prototype improves the system bandwidth by more than 15% and app I/O time by more than 10%, compared to POFS.



GIFT's fairness is comparable to BSIP and is much fairer than MBW.

Simulation-based results (on Stampede2, MIRA and THETA supercomputers) confirm system prototype results.

Rio: Order-Preserving and CPU-Efficient Remote Storage Access

Included in the
Proceedings of the 2023
EuroSys



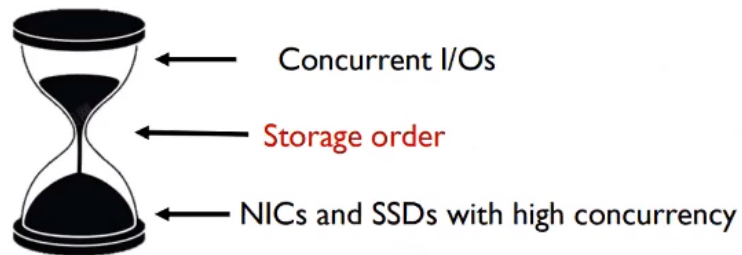
Recent Trend

Remote storage access has been increasingly popular

NVMe SSDs offer unparalleled efficiency compared to traditional storage media

Inefficient storage ordering guarantee

Almost synchronous





Background and Related Work

Remote Storage Access:

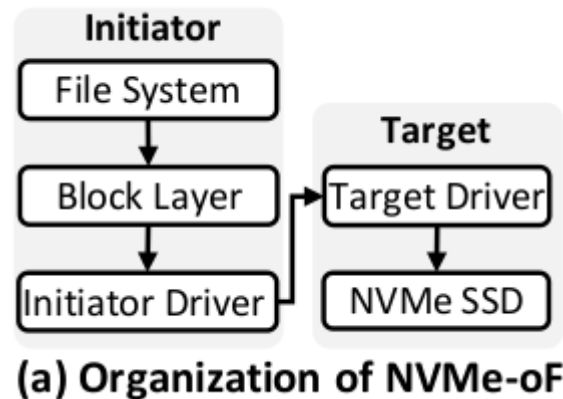
Two Componentes: Initiator and Target

To preserve the storage order they still rely on synchronous execution

Synchronous transfer and FLUSH command

The cost of this traditional approach is expensive

Each layer of NVMe-oF is orderless.





Motivation

Alleviating the overhead of storage barrier instructions

Making data transfer asynchronous

Reducing CPU cycles whenever possible



Rio

- Makes the storage stack conceptually similar to the CPU pipeline
- Introduced the I/O pipeline that allows internal out-of-order and asynchronous execution for ordered write requests while offering intact external storage order to applications
- It interfaces with the block layer of the operating system, making it compatible with various file systems and storage protocols.

Rio Architecture

- Revised Storage Stack
- Rio Sequencer
- Ordered Write Requests
- Concurrency and Scalability
- Crash Consistency

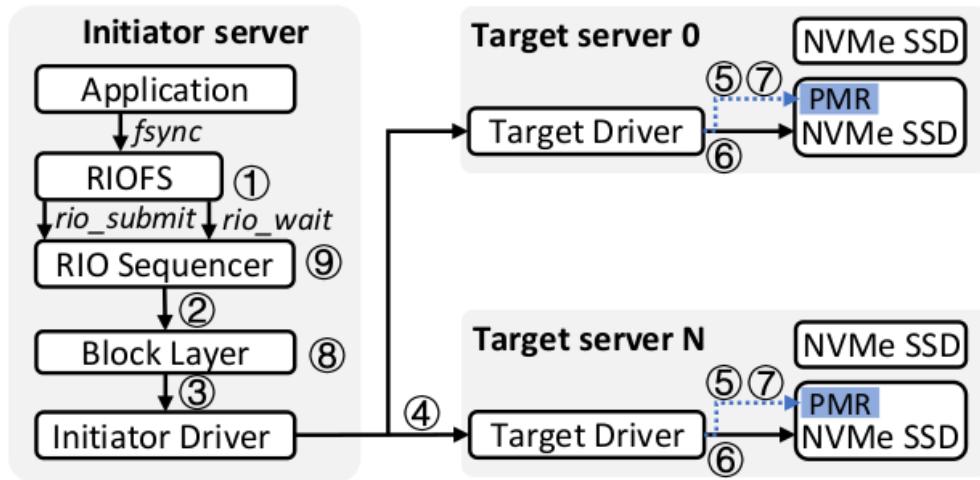
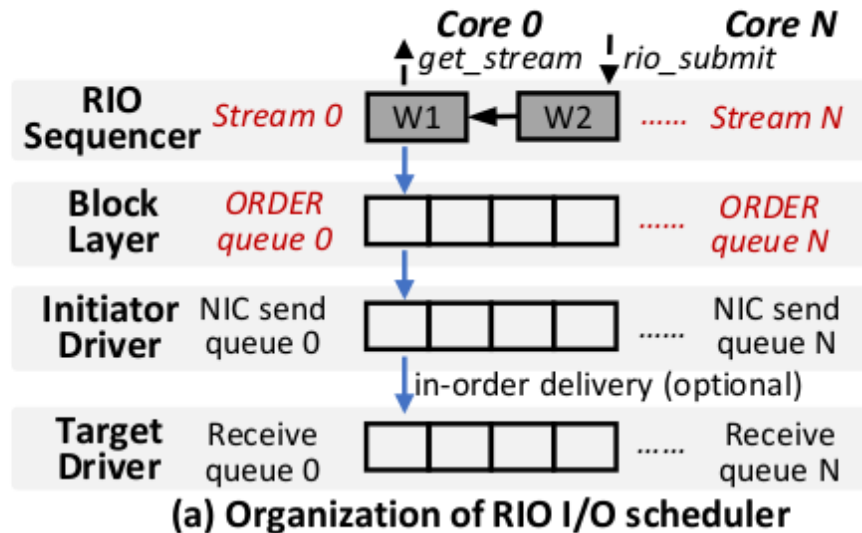


Figure 4. Rio architecture.



Rio Sequencer

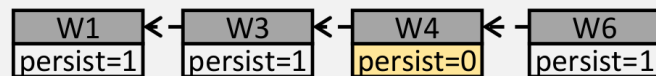
- Generates a special persistent ordering attribute which is an identity of ordered request, and then dispatches the requests to the block layer asynchronously
- When finished and returned to Rio sequencer, Rio completes the requests in order using the ordering attributes





Crash Recovery

target server 1: $1 \leftarrow 3$



target server 2: $2 \leftarrow 5$

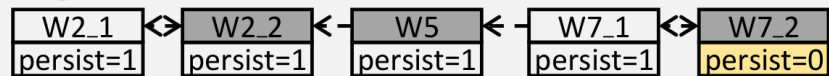


Figure 6. A recovery example. *Other fields of the ordering attributes are omitted to facilitate the description.*

Initiator Recovery: Collects per-server ordering lists

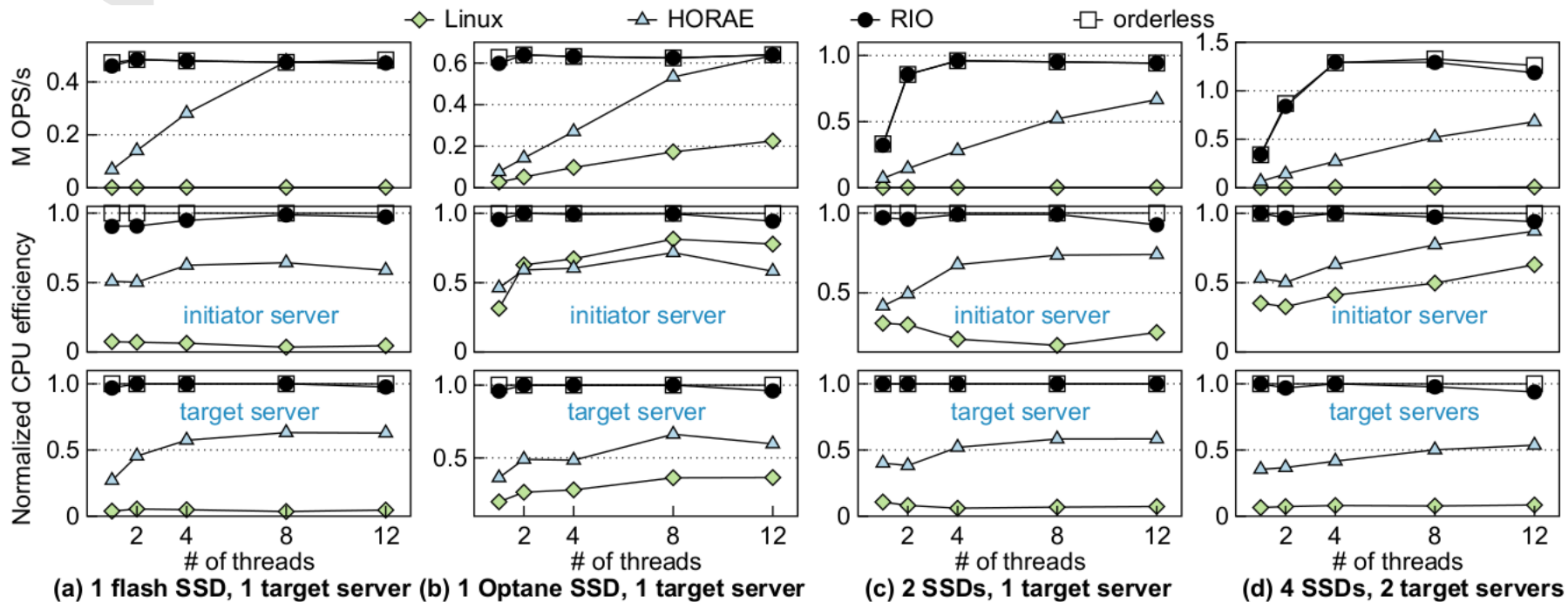
Creates a global ordering list and send back

Target Recovery: Initiator reconnects and create the ordering list

Connects the failed server until successful



Evaluation





Limitations

It is primarily tailored for NVMe SSDs equipped with Persistent Memory Regions (PMR)

The system's behavior in extremely high-load scenarios or in highly scalable environments has not been thoroughly explored

Incorporating it into existing systems may present complexities.