# REMUS: Efficient Multi-Request Scheduling in Computational Storage Devices

Yun Huang[1,2], Shuhan Bai[3], Qiang Su[4], Heng-Lin Yen[6], Nan Guan[1], Tei-Wei Kuo[5], Xue Liu[2], Chun Jason Xue[2]
[1]City University of Hong Kong
[2]Mohamed bin Zayed University of Artificial Intelligence
[3]Huazhong University of Science and Technology
[4]Xiamen University
[5]Academia Sinica & National Taiwan University
[6]YEESTOR Microelectronics Co., Ltd.

***Abstract*—Data-intensive applications benefit from offloading data processing to storage devices with embedded cores, typically Computational Storage Device (CSD). Co-locating requests from different applications to one CSD offers better performance and power efficiency than dedicating CSDs to a single application. However, current CSD scheduling frameworks fail to handle the contention in CPU, Flash I/O and buffer due to mismatching of resources and the requests, leading to sub-optimal performance.**

**This paper proposes REMUS, a CSD scheduling framework handling multiple requests for CSD platforms with multiple homogeneous cores. The key idea of REMUS is to allocate the workload to multiple cores according to the distribution of the Logical Block Address (LBA) of the requests and mitigate stall time by sorting requests based on their urgency of demand for resources, where urgency is quantified by the current remaining buffer capacity of the requests. Furthermore, a buffer allocation scheme is proposed to avoid programs that exclusively occupy the resources. We conduct experiments on both a simulator and a real CSD platform. The experiment results show that REMUS improved throughput by 1.51× on the simulator and 1.39× on the real platform on average compared to the baselines.**

***Index Terms*—computational storage, scheduling.**

## I. Introduction

The data volume keeps growing explosively and has reached terabyte levels in data-intensive cloud applications [1], [2], where the extensive data traverses between storage devices and host main memory, introducing significant delays and energy consumption from data movement in the I/O stack [3], [4]. **Computational Storage Device (CSD)** [5] emerges as a promising device to offload data-intensive applications, thus reducing such data movement [6]–[8].

Enhancing CSD capacity to handle multiple requests from different applications is essential for the wider adoption of CSDs. In general, the offloaded requests are determined based on their data retrieval amount and computational intensity [9]. Typical applications include databases [6], [10] and machine learning [11], to improve the performance and power efficiency of the data-centric applications. Each offloaded application has a specific resource demand. Equipping one CSD with a dedicated function for each application results in suboptimal performance due to not fully utilising all the resources on each CSD compared to co-locating CSDs.

The corresponding author is Qiang Su. Email: qiangsu@xmu.edu.cn.

However, when sharing CSD resources between multiple requests, resource contention makes efficient scheduling a challenging goal, and the existing CSD scheduler is less efficient in handling this contention due to dependencies between these resources. Classic CPU schedulers [12]–[14] and I/O schedulers [15] do not consider the interaction between flash I/O and CPU time, so requests may have a performance drop due to improper assignment of the flash bandwidth. HORAE [16] and SERICO [17] have explored request scheduling in CSD, but suffer performance declines due to less intelligent stage switching and suboptimal buffer allocation strategies.

Therefore, the fundamental question here is: *how can we build a better request scheduling framework in CSD, that achieves high overall throughput in multi-request scenarios?*

The key challenge of answering this question is perceiving the resource demand for CSD components of requests during scheduling. The inter-dependencies among CPU operations, flash I/O, and buffer allocation, each influencing one another, make it hard to effectively decouple their scheduling. The initiation of data transfer is contingent upon CPU execution, while the buffer inherently limits the volume of data transmitted, with its capacity being determined by both computational requirements and data retrieval processes. Additionally, the unpredictable data fetching times due to flash parallelism further complicate scheduling. Traditional schedulers that focus on optimizing one or two types of resource allocations or that employ fixed resource-sharing schemes are inadequate for the complexities of scheduling multiple requests in CSDs.

To this end, we propose REMUS, an effective resource allocation and scheduling framework that achieves higher overall performance. We first identify the weaknesses of the existing CSD scheduler and then formulate the CSD scheduling problem. Based on it, we propose a workload assignment and a resource-aware scheduler, which perceives resource demand urgency with the current remaining buffer of each request in the runtime. A resource management module is also proposed to batch the workload to minimize the queueing time and allocate a dedicated buffer to further enhance the performance.

We implement REMUS and conduct extensive experiments on both a simulator and a testbed [18]. REMUS outperforms the baselines (CFS, HORAE and SERICO) significantly. In summary, this paper makes the following contributions:

2325-1301/25/$31.00 ©2025 IEEE
DOI 10.1109/RTCSA66114.2025.00024

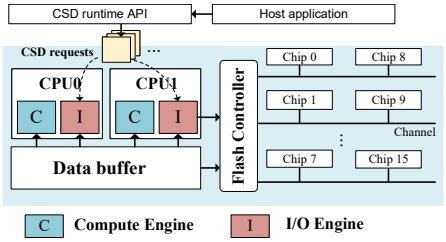


Fig. 1. Architecture of CSD

- We propose REMUS, a novel scheduling framework for the multi-request CSD platforms, which perceives the ever-changing resource demand of requests and effectively schedules resources in the CSD.
- We develop a workload assignment scheme to deliver computation and flash I/O to the cores to balance the workload and reduce flash access conflicts.
- A buffer allocation module is proposed to allocate buffer capacity judiciously to prevent programs from monopolizing the resources.
- REMUS improves throughput by 1.51× on the simulator and 1.39× on the real platform on average compared to the baselines.

## II. Background and Motivation

A CSD device integrates several onboard computing units (ARM cores [3], [19] or FPGA circuits [20]), DRAM memory, and a backend flash controller. In this paper, we leverage an ARM-based CSD device equipped with two CORTEX-R4 cores [18], as shown in Fig. 1. Each ARM core runs an I/O engine and a compute engine to issue flash I/O requests and process the retrieved data in the onboard data buffer, respectively. The flash controller has eight channels, each with two flash storage chips, allowing individual flash requests to be processed in parallel [21]. Additionally, each chip contains two *planes* for fetching data in parallel. The minimal unit of a read operation in an SSD is a page, with a size of 4096 Bytes, and each page is associated with a Logical Block Address (LBA) to indicate its location in the chip.

Rather than directly transferring raw data to the host main memory, CSD filters the data first in the embedded cores of the storage device and transferred the results instead. Each request issued by the application for fetching filtered data is considered as a CSD request. The I/O engine initiates the flash request and communicates with the flash controller **(Stage S)**. Data transfer on the dedicated flash chip commences **(Stage F)**, with raw data stored in onboard DRAM and processed by the onboard CPU **(Stage P)**. Selected data is then sent to the host, included in the latency of Stage P.

### A. Sharing CSD

Intuitively, offloading multiple applications simultaneously can make full use of onboard resources, offering benefits in overall performance, energy efficiency, and cost-effectiveness. To illustrate this, we concurrently run two typical CSD applications: SQL-select [22] and the K-Nearest Neighbors (KNN) [23] algorithm, which are common operations in database and data mining systems. The onboard CPU processing time for KNN is longer than the flash access time, which causes the flash controller to be idle and underutilized. In contrast, SQL-select operations have lower computational demands, allowing the onboard CPU to remain idle while waiting for data. All applications issue CSD requests to invoke CSD onboard execution via CSD runtime APIs [24].

TABLE I
Throughput Comparison

| Approaches | Normalized Throughput |
|---|---|
| *Host-only* | 1 |
| *CSD-dedicate* | 0.81 |
| *CSD-colocate* | 1.57 |

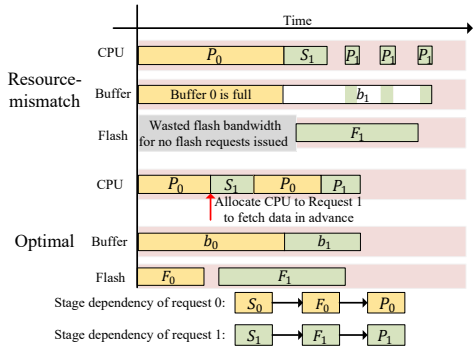


Fig. 2. Weaknesses of current scheduling algorithm (Duration with white color indicates that there is no data ready to be processed in the buffer, while colored duration indicates that there is data ready to process in the buffer.)

We conducted experiments using three approaches: (1) *host-only*, where no requests are offloaded; (2) *CSD-dedicate*, where each request is offloaded to a separate CSD; (3) *CSD-colocate*, which co-locates the two requests on the same CSD. The total processed data of requests issued by SQL and KNN is 3GB and 1GB, respectively. For the CSD-dedicate setting, all SQL requests are allocated to the CSD for SQL and KNN requests are dispatched to the CSD for KNN. For CSD-colocate, both SQL requests and KNN requests are dispatched evenly to the two CSDs.

We measure the overall throughput with the total processed data and the completion time of all the requests. Table I lists the results. The throughput is normalized by *host-only* results. We can see that *CSD-colocate* has the highest overall throughput, which is because (1) compared to *Host-only*, it reduces the time in both host I/O stack and device I/O request processing; (2) compared to *CSD-dedicated*, it utilizes the idle CPU resource in SQL request to process the data of KNN request, and the idle flash bandwidth of KNN to fetch SQL data, improving the overall performance. Hence, co-locating requests with various resource demands can achieve higher overall performance, which motivates our work.

### B. Weaknesses of Existing Frameworks

Fig. 2 exhibits the **mismatching between resources** issue of the current CSD heuristic schedulers. Yellow blocks represent the stages of request 0, while the green block stands for request 1. The specific stage is marked on the blocks, for example, $S_0$ is stage S for request 0, which denotes to sending flash requests. $F_0$ is a flash chip busy fetching data for request 0. And $P_0$ is the process data stage of request 0.

The CPU and Flash I/O mismatch baselines allocate CPU resources and Flash I/O resources that are not compatible with the requirements of the requests. Request 0 processes data slower than the flash I/O produces the data. As the buffer of request 0 is full, all the time slot allocated to request 0 is spent on processing data without sending flash requests, which wastes the flash bandwidth. If the shorter request 1 can utilize the wasted bandwidth, as shown in the optimal case, the waiting time of request 1 can be shortened and the occupied buffer can be minimized.

In this work, we adopt three heuristic baselines: CFS (Completely Fair Scheduler) [13], HORAE and SERICO. CFS is a common CPU scheduler used in modern operating systems, which emphasises making each request have a fair share of the CPU. HORAE and SERICO are the state-of-the-art multi-request schedulers in the CSD. HORAE computes the critical path of stages, which is the sum of the time the predecessor stages occupy the resources on each hardware. It prioritises the stage with the longest critical paths, which maximizes the parallelism among the stages. SERICO decouples the computation and Flash I/O scheduling with a ping-pong buffer. It schedules the requests in an earliest-deadline manner, and the deadline is updated periodically.

CFS neglects the corresponding scheduling of flash I/O and buffer, causing CPU and flash I/O mismatch between requests. HORAE tends to give requests with longer execution times higher priority, which will block the flash request sending and downgrade the utilization of the flash parallelism. SERICO calculates the deadline periodically for processing data in buffer and fetching data. However, when Flash I/O and computation are coupled on one core, the deadline will become tighter, resulting in the CPU missing the deadline to send a request and lower flash bandwidth utilization.

### C. Problem Definition

The request and the device can be configured with the parameters shown in Table II:

The Multi-request CSD scheduling problem can be defined as follows. Given as input requests set with $N_r$ requests: $R_{N_r} = \{R_i\}$, the $i^{th}$ request in the set is denoted by $R_i$, and a CSD platform $CSD(N_c, N_{ch}, N_{pl})$, our goal is to generate a schedule that minimize the completion time of the request set. The constraints at any timestamp $t$ for the problem can be expressed as:

$$\sum_{i=1}^{i<N_r} S_i - \sum_{i=1}^{i<N_r} F_i \leq Q \tag{1}$$

TABLE II
NOTATIONS IN THIS WORK

| Notation | Explain |
|---|---|
| $R_i$ | The $i^{th}$ request. |
| $N_c$ | Number of CPU on CSD. |
| $N_{ch}$ | Number of flash chips per channel. |
| $N_{pl}$ | Number of planes. |
| $Q$ | Maximum flash request that can be sent by each core, indicating the maximum parallelism that the flash controller can provide. |
| $Q_{cur}$ | Flash requests remaining buffer to be sent at current time. |
| $T_f$ | Chip busy time for fetching pages. |
| $T_s$ | Time for sending a flash request. |
| $S_i$ | Total sent flash requests of requests $R_i$ on the current core. |
| $F_i$ | Total returned data pages of $R_i$ on the current core. |
| $P_i$ | Total processed pages of $R_i$ on the current core. |
| $c_i$ | Estimated processing time for a data block, which can be profiled before the request is issued. |

$$\sum_{i=1}^{i<N_r} F_i - \sum_{i=1}^{i<N_r} P_i \leq \sum_{i=1}^{i<N_r} B_i \tag{2}$$

$$\sum_{i=1}^{i<N_r} F_i \geq \sum_{i=1}^{i<N_r} P_i \tag{3}$$

$S_i$ indicates the total sent flash requests of requests $R_i$ on the current core, $F_i$ is the total returned data pages of $R_i$ on the current core and $P_i$ is the total processed pages of $R_i$ on the current core by the timestamp $t$. Inequality (1) indicates that at any timestamp, the sent flash requests can advance by $Q$ compared to the returned flash requests, thus not violating the flash quota constraint. Inequality (2) indicates that the volume of returned data that has not yet been processed cannot exceed the allocated buffer size $B_i$ of request $R_i$. Inequality (3) indicates that the processed data is no greater than the returned data. It's a resource assignment problem that involves multiple non-linear constraints, which is difficult to solve in polynomial time.

## III. Design

### A. REMUS Overview

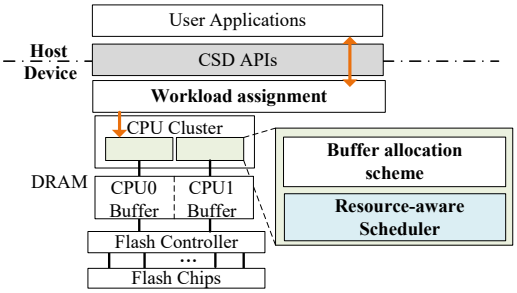


Fig. 3. Overall architecture of REMUS.

*Overall architecture.* Fig. 3 illustrates the overall architecture of REMUS, mainly including four software components inside the CSD device: *Workload assignment*, *Buffer allocation scheme* and *Resource-Aware Scheduler*. The CSD API of the host runtime library follows the design in Insider [24]. The workload assignment scheme is proposed to separate the workload of each request on multiple cores. Each core independently maintains its buffer. The buffer is logically divided into two parts with the same capacity for the two cores as computing buffer, which is denoted by CPU0 Buffer and CPU1 Buffer in Fig. 3.

When a new request arrives at the CSD device, the workload assignment module splits the workloads and dispatches them to the dedicated core, balancing the workload and avoiding flash access conflicts between cores. If it's scheduled to be executed, the buffer allocation scheme will allocate the dedicated buffer address and buffer size of CSD DRAM for this request based on the request's computation complexity, aiming to give a constraint on the CPU resources and flash bandwidth that the request can acquire in the runtime. Afterwards, the resource-aware scheduler combines with the overall execution status to identify which request is more likely to cause a stall in the specific resource, assigns a suitable priority, and determines which stage to conduct adaptively.

### B. Workload Assignment

The workload is split into sub-requests in a granularity of pages and assigned to a dedicated CPU according to the chip that the split sub-requests belong to. Each sub-request contains a start LBA and a smaller number of blocks to be processed. Even though there exist Wear-levelling and Garbage Collection procedures inside the CSD, most of the approaches move data inside the same chip [25], [26]. Hence, the designated chip of the flash request in advance can be gained by its LBA:

$$chip_{lba} = \lfloor \frac{LBA}{N_{pl} * page\_size} \rfloor \%(N_{ch} * \text{Channel\_Num}) \quad (4)$$

And we determine the CPU to send this flash request containing this start LBA with Equation (5):

$$cpu\_id = chip_{lba} \% N_c \quad (5)$$

If the sub-requests with start LBA that satisfy: $LBA\%N_{pl} = 0$, which is possible to issue a multi-plane read, the size of data of the sub-request will be set to $N_{pl} * Page\_Size$, otherwise it will be set to 1.

The workload assignment is managed by a primary embedded core, which has access to the host interface. Upon receiving the request from the host, the primary core delivers the sub-requests with its start LBA and the number of LBA to the dedicated CPU according to Equation (5). Once the workload is assigned, Stage S and Stage P of this sub-request will be executed on the same core.

### C. Resource-Aware Scheduler

*1) Priority Assignment:* Recall the problem model described in Sec. II-B; a request with no ready data or a fulfilled buffer may result in insufficient utilisation of CPU or flash bandwidth. Hence, for Stage S, we prioritize the request that is most likely to run out of data. In Stage P, the request that is most likely to exhaust its buffer should be processed first, to prevent it from blocking the overall flash data fetching.

To minimize the impact of resource contention, the demand of each request for each resource should be perceived. Hence, we define a resource demand urgency for each request. The urgency indicates how fast the request will saturate the resources, i.e., how fast they will reach the boundary of the constraints. To estimate the resource demand urgency for a request $R_i$ in the CSD, we first convert the Inequality (2) to:

$$P_i \geq F_i - B_i \quad (6)$$

As we measure the urgency of requests at the current timestamp ( t ), we eliminate the data blocks that have already been processed in the past cycle, denoted by $K$. That is, we deduce the total return pages by K, which satisfy:

$$F_i - K \leq B_i \quad (7)$$

$$F_i \geq K \quad (8)$$

Hence, the inequality can be written in:

$$P_i - K \geq F_i - K - B_i$$

This inequality should be satisfied in the following timestamps. Hence, the larger $F_i - K - B_i$, the more urgent the request should be processed. The $F_i - K$ is the remaining data that can be processed in the buffer at timestamp $t_{rk}$, which is the timestamp of $K^{th}$ block returned. Hence, the term $F_i - K - B_i$ is actually the opposite of the remaining buffer from $t_{rk}$ to $t$. So the urgency of processing is negatively correlated to the remaining buffer. The less remaining buffer that the request possesses, the more urgent it is for the request to process the data.

For Stage S, we convert the Inequality by adding $(B_i - K)$ on both sides of the inequality (3) to:

$$F_i - K + B_i \geq B_i - (K - P_i) \quad (9)$$

The larger $B_i - (K - P_i)$ is, the more urgent it is for the current request to fetch the data, as it will be quicker to hit the constraint boundary. $K - P_i$ denotes the remaining data that can be processed from timestamp $t_{rk}$ to $t$. Therefore, $B_i - (K - F_i)$ is the current remaining buffer. So the urgency of the $F_i$ is positive relative to the remaining buffer, i.e, the more remaining buffer the request has, the more urgent for it to fetch the data.

In the end, the priority assignment of REMUS can be summarised as: In Stage S, the scheduler will pick the request with the most remaining buffer to send flash requests, which indicates that it is most urgent for data fetching. And in Stage P, the scheduler will pick the request with the least remaining buffer to process, which indicates that it is most urgent for data processing.

*2) Preemption Mechanism:* A preemption mechanism is proposed to enable shorter requests to fetch data in advance. We introduce a parameter $Q_{thres}$ to guide the occurrence of the preemption. If there exists an amount of flash quota that is greater than $Q_{thres}$, the current execution of the requests will be preempted. And if there is not enough quota, the CPU will continue to execute Stage P until there is enough data to complete data fetching.

$Q_{thres}$ is initialized equal to $N_{pl}$ and can adjust dynamically at runtime. When the remaining buffer of the current requests in the CSD is lower than the $Q_{cur}$, the CPU workload is much heavier than the flash workload. Hence, the $Q_{thres}$ should be raised, so that the CPU can process more data while fetching less. On the other hand, if there is no data as well as $Q_{cur} < Q_{thres}$, the $Q_{cur}$ should be reduced to the $max[Q_{cur}, N_{pl}]$. The complete resource-aware scheduler is presented in Algorithm 1.

**Algorithm 1** Resource-aware scheduler.

```
CPU is ready to pick the next request
if Q_cur ≥ Q_thres then
    if availble buffer in CSD is less than Q_cur then
        Increase Q_cur to by N_pl and switch to Stage P
    else
        Pick the request R_i with largest remaining buffer
        Enter Stage S and send min[Q_cur, b_ri] requests
    end if
else
    if Data ready for processing greater than zero then
        Pick the request R_i with smallest remaining buffer
        Execute R_i in Stage P
    else
        Waiting for Q_cur > Q_thres to send requests
        Reduce Q_thres reduce to the max[Q_cur, N_pl]
    end if
end if
```

### D. Buffer Allocation

A buffer allocation scheme is proposed to avoid a situation in which the scheduler consistently assigns higher priority to a shorter request. As stated in Sec. III-C1 and III-C2, when $Q_{cur} > Q_{thres}$, the request with more remaining buffer space will send flash requests first. Thus, if there exists a request with a small $c_i$ that can finish processing the data as soon as the data has been transferred (i.e., always with a large remaining buffer space), the scheduler will always pick this request in stage S to send flash requests, leading to the flash bandwidth being occupied.

Initially, we model the waiting time associated with each request to discern a trend in buffer allocation before determining the appropriate buffer size. $T_{fb}$ for a fraction of the request with sequential physical page distribution to transfer $B_i$ pages is predictable, referring to the striping technology [27], denoted by $T_{feq}$. The total flash transferring time of $Q_{cur}$ blocks is:

$$T_{fb} = T_{fseq} B_i, \tag{10}$$

and time spent on sending (the time of sending one flash request is denoted by $T_s$) and processing these $B_i$ blocks is:

$$T_{cpu} = B_i(T_s + c_i). \tag{11}$$

Obviously, if $T_{cpu} < T_{fb}$, data transferring is slower than processing, which will introduce extra waiting time. This boundary execution time of requests can be obtained in the case of $T_{cpu} = T_{fb}$, denoted as $c_b$. The waiting time caused by a request with $c_i < c_b$ can be computed by $T_{fb} - T_{cpu}$. The shorter request with a lower $c_i$ will cause a longer waiting time ($T_{cpu} < T_{fb}$); thus, a lower $B_i$ should be allocated. The longer request with a higher $c_i$ can overlap more waiting time ($T_{cpu} > T_{fb}$); thus, a higher $B_i$ should be allocated.

However, the exact $c_b$ is ambiguous due to the irregular LBA distribution and inevitable conflict of access to the flash chips with other cores. To ensure that the $c_i > c_b$ can be allocated with more buffer, we statically allocate the buffer to the request $R_i$ with:

$$B_i = Q + \lceil c_i \rceil * PageSize. \tag{12}$$

In this way, the longer request is prioritized to fetch data first, and its buffer is filled quickly. Hence, shorter requests can preempt and utilize the wasted bandwidth of the longer request. The buffer allocation scheme can also be further extended to allocate buffers according to the requests' priority.

## IV. Evaluation

### A. Methodology

**Baselines.** We set two types of baselines to evaluate REMUS.
*Online scheduler.* We compare REMUS with CFS [13], HORAE [16] and SERICO [17]. CFS is a common CPU scheduler that guarantees each request has fair access to the CPU. HORAE is the SOTA job-shop scheduling algorithm in CSD that sequences and aligns the stages on different hardware modules with specific functions. SERICO is the SOTA deadline-driven scheduling algorithm that minimizes the deadline miss rate and buffer consumption of CSD. We assign the same deadline for all requests in the experiment. All the baselines are modified to adapt to multiple cores.
*Offline scheduler.* We employ an offline heuristic optimal algorithm to get the optimal result. This optimal scheduler first aligns each Stage S and Stage F for each LBA of the requests iteratively to gain the best performance for fetching data, and then inserts the Stage P between the Stage S to fit into the buffer constraints.
**Simulator.** We developed an in-house event-triggered simulator with Python. This simulator aims to replicate the timing behaviour similar to our real CSD platform YS9203.
**Testbed.** REMUS is implemented and evaluated on YS9203, a realistic commercial SSD platform equipped with two ARM Cortex-R4 cores. The flash capacity of YS9203 is 512 GB. It is constructed with 8 channels, and each channel is connected to 2 chips. Each chip contains 1 die and 4 planes. And the remaining buffer DRAM size for CSD applications to store the intermediate data is 400 MB. The testbed is connected to a

host server equipped with 8-core Intel(R) Core(TM) i7-9700K CPU running at 3.60 GHz with 64 GB of DRAM.

### B. Simulation Results

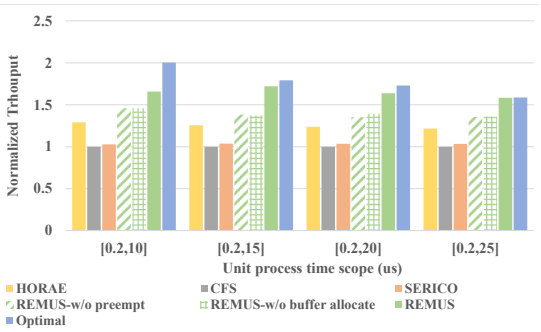


Fig. 4. Normalized throughput Comparison when $T_f = 65\mu$s

In our simulation, we set the value of $T_s$ to 1$\mu$s, and $T_f = 65\mu$s, $N_c = 2, N_{pl} = 2, N_{ch} = 2$ to compare the performance of the schedulers. Each test group contains 50 requests, and the computation time $c_i$ is all generated in a uniform distribution in a ratio of shorter requests: longer requests = 6:4. We mainly measure the throughput. The total buffer size is set to 80MB, and the total data processed in a request group is 1 GB.

*Throughput comparision.* Fig. 4 shows the normalized throughput of the five schedulers. The digit at the bottom of the figure denotes that the $c_i$ of the requests are generated in [0.2, digit] under the uniform distribution. The digit on the left of the figure denotes the normalized throughput reference against the scheduler CFS. Compared to CFS, REMUS achieves a throughput enhancement of 1.65× on average. For HORAE and SERICO, REMUS achieves an improvement of 1.32× and 1.59× on average, respectively.

### C. Real Platform Results

On a real CSD platform, we implemented all three baselines, CFS, HORAE, and SERICO, to compare their performance. The requests are generated with 3 applications, SQL, STAT, and KNN, with request sizes ranging from 100 MB to 1.2 GB. SQL has the lowest computational complexity, KNN has the highest, and STAT is in the middle. STAT abstracts the characteristics of the datasets, such as the minimum and maximum values [28]. The $c_i$ of SQL ranges from 0.25 to 1$\mu$s, STAT is 1 to 4$\mu$s, and KNN is 12 to 14$\mu$s.

We compare the performance under different compositions of the requests, as illustrated in the horizontal axis of Fig. 5. For reference, we have computed the normalized throughput relative to CFS. REMUS achieves an average throughput enhancement over CFS, HORAE and SERICO by 1.42×, 1.33× and 1.41× respectively. The performance improvement differs because the computation workload differs as the ratio of different applications changes.

*Overhead analysis.* We also measure the timing and implementation overhead incurred by the additional sorting operation in REMUS during execution with a CPU timer. Since

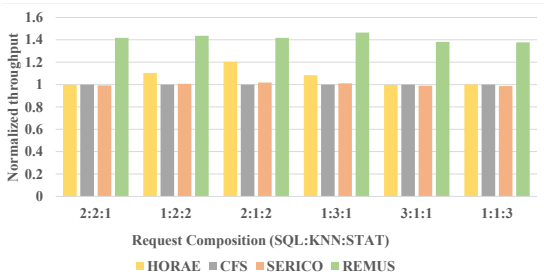


Fig. 5. Performance comparison on a real platform (YS9203).

the request queues are stored in the Tightly Coupled Memory in the ARM-Core, the latency of accessing the request queue is lower. REMUS introduces an overhead of only 0.6% of the total execution time when there are 10 pending requests in the current request queue. Therefore, the time overhead of REMUS is quite small. And the resource overhead that achieves REMUS for storing request queues is 1.8 KB.

## V. Related Work

Prior research efforts focused primarily on advancing specific applications and refining the architectural framework of CSD. Notably, one area of application enhancement lies within the domain of database applications [6], [29], [30]. Furthermore, CSD has demonstrated promise in expediting AI applications [31], [32]. Several researches work on accelerating graph applications on CSD, such as for GNN training [33] and neighbour sampling and feature lookup in GNN [34]. CSD has been employed to diminish the latency associated with embedding lookups [35] and gene sequence analysis [7]. The performance of list operations can be improved through the utilization of CSD [36], [37].

Some previous works have endeavoured to optimize the programming model for CSD [24], [38], aiming to reduce programming complexity and streamline the stack overhead associated with CSD implementation. In recent years, there has been a surge in research investigating novel architectures for CSD. Noteworthy contributions include the development of end-to-end architectures that support block-oriented near-data processing [39], as well as the establishment of high-performance virtualization techniques for CSD through hardware-assisted virtualization and dynamic scheduling of virtual CSD instances [40].

## VI. Conclusion

Resource contention brings challenges in scheduling multiple requests in CSD. The existing schedulers fail to handle the contention for resource mismatching. In this work, we propose REMUS, which is a lightweight scheduler to perceive and schedule the resource with the remaining buffer of requests and the current flash controller status. Additionally, a buffer allocation scheme is used to further enhance the performance of REMUS. Evaluation results demonstrate that REMUS is effective for CSD platforms with various workloads.

## REFERENCES


[1] A. J. Thirunavukarasu, D. S. J. Ting, K. Elangovan, L. Gutierrez, T. F. Tan, and D. S. W. Ting, "Large language models in medicine," *Nature medicine*, vol. 29, no. 8, pp. 1930–1940, 2023.

[2] L. Dobos, J. Szüle, T. Bodnár, T. Hanyecz, T. Sebők, D. Kondor, Z. Kallus, J. Stéger, I. Csabai, and G. Vattay, "A multi-terabyte relational database for geo-tagged social network data," in *2013 IEEE 4th International Conference on Cognitive Infocommunications (CogInfoCom)*. IEEE, 2013, pp. 289–294.

[3] S. Seshadri, M. Gahagan, S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu, and S. Swanson, "Willow: A User-Programmable SSD," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 67–80.

[4] Z. Yang, J. R. Harris, B. Walker, D. Verkamp, C. Liu, C. Chang, G. Cao, J. Stern, V. Verma, and L. E. Paul, "Spdk: A development kit to build high performance storage applications," in *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2017, pp. 154–161.

[5] "Computational storage: A new way to boost performance," *https://newsroom.arm.com/blog/computational-storage*.

[6] Z. Cao, H. Dong, Y. Wei, S. Liu, and D. H. Du, "IS-HBase: An In-Storage Computing Optimized HBase with I/O Offloading and Self-Adaptive Caching in Compute-Storage Disaggregated Infrastructure," *ACM Transactions on Storage (TOS)*, vol. 18, no. 2, pp. 1–42, 2022.

[7] N. Mansouri Ghiasi, J. Park, H. Mustafa, J. Kim, A. Olgun, A. Gollwitzer, D. Senol Cali, C. Firtina, H. Mao, N. Almadhoun Alserr, R. Ausavarungnirun, N. Vijaykumar, M. Alser, and O. Mutlu, "Genstore: a high-performance in-storage processing system for genome sequence analysis," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 635–654.

[8] K. K. Matam, G. Koo, H. Zha, H.-W. Tseng, and M. Annavaram, "Graphssd: graph semantics aware ssd," in *Proceedings of the 46th international symposium on computer architecture*, 2019, pp. 116–128.

[9] Z. Ruan, T. He, and J. Cong, "Analyzing and Modeling In-Storage Computing Workloads On EISC — An FPGA-Based System-Level Emulation Platform," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2019, pp. 1–8.

[10] T. Vinçon, A. Bernhardt, I. Petrov, L. Weber, and A. Koch, "nKV: near-data processing with KV-stores on native computational storage," in *Proceedings of the 16th International Workshop on Data Management on New Hardware*, ser. DaMoN '20. New York, NY, USA: Association for Computing Machinery, 2020.

[11] N. Prakriya, Y. Yang, B. Mirzasoleiman, C.-J. Hsieh, and J. Cong, "NeSSA: Near-storage data selection for accelerated machine learning training," in *Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems*, 2023, pp. 8–15.

[12] T. Balharith and F. Alhaidari, "Round Robin Scheduling Algorithm in CPU and Cloud Computing: A review," in *2019 2nd International Conference on Computer Applications & Information Security (ICCAIS)*, 2019, pp. 1–7.

[13] J. Kobus and R. Szklarski, "Completely fair scheduler and its tuning," 2009. https://fizyka.umk.pl/ jkob/prace-mag/cfs-tuning.pdf

[14] D. Faggioli, M. Trimarchi, F. Checconi, M. Bertogna, and A. Mancina, "An implementation of the earliest deadline first algorithm in linux," in *Proceedings of the 2009 ACM Symposium on Applied Computing*, ser. SAC '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 1984–1989.

[15] "Completely fair queueing." https://www.kernel.org/doc/Documentation/block/cfq-iosched.txt

[16] J. Li, X. Chen, D. Liu, L. Li, J. Wang, Z. Zeng, Y. Tan, and L. Qiao, "Horae: A Hybrid I/O Request Scheduling Technique for Near-Data Processing-Based SSD," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 11, pp. 3803–3813, 2022.

[17] Y. Huang, N. Guan, S. Bai, T.-w. Kuo, and C. J. Xue, "SERICO: Scheduling Real-Time I/O Requests in Computational Storage Drives," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2023, pp. 1–6.

[18] YEESTOR. (2021, 11) Yeestor YS9203 PCIe SSD Memory Controller. YEESTOR Microelectronics. https://www.yeestor.com/products/detail/i-16.html

[19] D. Tiwari, S. Boboila, S. Vazhkudai, Y. Kim, X. Ma, P. Desnoyers, and Y. Solihin, "Active Flash: Towards Energy-Efficient, In-Situ data analytics on Extreme-Scale machines," in *11th USENIX Conference on File and Storage Technologies (FAST 13)*. San Jose, CA: USENIX Association, Feb. 2013, pp. 119–132.

[20] Y. Lee, J. Chung, and M. Rhu, "SmartSAGE: Training Large-Scale Graph Neural Networks Using in-Storage Processing Architectures," ser. ISCA '22, 2022.

[21] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and S. Zhang, "Performance impact and interplay of ssd parallelism through advanced commands, allocation strategy and data granularity," in *Proceedings of the international conference on Supercomputing*, 2011, pp. 96–107.

[22] Microsoft. Sql server. https://learn.microsoft.com/en-us/sql/sql-server/

[23] Microsoft. Query spatial data for nearest neighbor. https://neo4j.com/docs/graph-data-science/current/algorithms/knn/

[24] Z. Ruan, T. He, and J. Cong, "INSIDER: Designing In-Storage computing system for emerging High-Performance drive," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 379–394.

[25] C. Gao, L. Shi, M. Zhao, C. J. Xue, K. Wu, and E. H.-M. Sha, "Exploiting parallelism in i/o scheduling for access conflict minimization in flash-based solid state drives," in *2014 30th Symposium on Mass Storage Systems and Technologies (MSST)*, 2014, pp. 1–11.

[26] N. Elyasi, M. Arjomand, A. Sivasubramaniam, M. T. Kandemir, C. R. Das, and M. Jung, "Exploiting Intra-Request Slack to Improve SSD Performance," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 375–388.

[27] L.-P. Chang and T.-W. Kuo, "An adaptive striping architecture for flash memory storage systems of embedded systems," in *Proceedings. Eighth IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 2002, pp. 187–196.

[28] Z. Ruan, T. He, and J. Cong. Statisitcs in-storage application. https://github.com/zainryan/INSIDER-System/tree/master/apps/device/statistics

[29] I. Jo, D.-H. Bae, A. S. Yoon, J.-U. Kang, S. Cho, D. D. G. Lee, and J. Jeong, "YourSQL: A High-Performance Database System Leveraging in-Storage Computing," *Proc. VLDB Endow.*, 2016.

[30] L. Woods, Z. István, and G. Alonso, "Ibex: an intelligent storage engine with support for advanced sql offloading," *Proc. VLDB Endow.*, vol. 7, no. 11, p. 963–974, jul 2014.

[31] S. Kim, Y. Jin, G. Sohn, J. Bae, T. J. Ham, and J. W. Lee, "Behemoth: A Flash-centric Training Accelerator for Extreme-scale DNNs," in *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, Feb. 2021, pp. 371–385.

[32] J. Kim, M. Kang, Y. Han, Y.-G. Kim, and L.-S. Kim, "Optimstore: In-storage optimization of large scale dnns with on-die processing," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 611–623.

[33] F. Niu, J. Yue, J. Shen, X. Liao, and H. Jin, "Flashgnn: An in-ssd accelerator for gnn training," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2024, pp. 361–378.

[34] Y. Wang, X. Pan, Y. An, J. Zhang, and G. Reinman, " Beacon-GNN: Large-Scale GNN Acceleration with Out-of-Order Streaming In-Storage Computing ," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. Los Alamitos, CA, USA: IEEE Computer Society, Mar. 2024, pp. 330–344.

[35] X. Sun, H. Wan, Q. Li, C.-L. Yang, T.-W. Kuo, and C. J. Xue, "RM-SSD: In-Storage Computing for Large-Scale Recommendation Inference," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022, pp. 1056–1070.

[36] J. Wang, D. Park, Y.-S. Kee, Y. Papakonstantinou, and S. Swanson, "SSD in-storage computing for list intersection," in *Proceedings of the 12th International Workshop on Data Management on New Hardware*, ser. DaMoN '16. New York, NY, USA: Association for Computing Machinery, 2016.

[37] C. Li, Y. Wang, C. Liu, S. Liang, H. Li, and X. Li, "GLIST: Towards In-Storage graph learning," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, Jul. 2021, pp. 225–238.

[38] Z. Yang, Y. Lu, X. Liao, Y. Chen, J. Li, S. He, and J. Shu, "λ-IO: A unified IO stack for computational storage," in *21st USENIX Conference*

*on File and Storage Technologies (FAST 23)*. Santa Clara, CA: USENIX Association, Feb. 2023, pp. 347–362.

[39] A. Barbalace, M. Decky, J. Picorel, and P. Bhatotia, "BlockNDP: Block-Storage Near Data Processing," in *Proceedings of the 21st International Middleware Conference Industrial Track*, ser. Middleware '20, 2020, p. 8–15.

[40] D. Kwon, D. Kim, J. Boo, W. Lee, and J. Kim, "A Fast and Flexible Hardware-based Virtualization Mechanism for Computational Storage Devices," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, Jul. 2021, pp. 729–743.