SERICO: Scheduling Real-Time I/O Requests in Computational Storage Drives

1st Yun Huang
City University of Hong Kong

2nd Nan Guan
City University of Hong Kong

3rd Shuhan Bai
City University of Hong Kong
Huazhong University of Science and Technology

4th Tei-wei Kuo
National Taiwan University

5th Chun Jason Xue
City University of Hong Kong

***Abstract*—The latency and energy consumption incurred by I/O accesses are significant in data-centric computing systems. Computational Storage Drive (CSD) can largely reduce data movement, and thus reduce I/O latency and energy consumption by offloading data-intensive processing to processors inside the storage device. In this paper, we study the problem of how to efficiently utilize the limited processing and memory resources of CSD to simultaneously serve multiple I/O requests from various applications with different real-time requirements. We proposed SERICO, a system of scheduling computational I/O requests in CSD. The key idea of SERICO is to perform admission control of real-time computational I/O requests by online schedulability analysis, to avoid wasting the processing resources and memory capacity of CSD in doing meaningless work for those requests deemed to violate the timing constraints. Each admitted computational I/O request is served in a controlled manner with carefully designed parameters, to meet its timing constraint with minimal memory cost. We evaluate SERICO with both synthetic workloads on simulators and representative applications on realistic CSD hardware. Experiment results show that SERICO significantly outperforms the default method used in the CSD device and the standard deadline-driven scheduling approach.**

I. Introduction

Data need to be transferred from the storage to main memory for processing, which incurs large time delay and energy consumption. Although the bandwidth of SSD and PCIe interface has increased a lot in recent years, the I/O stack latency is still significant in data-intensive applications. Data transfer between the storage and main memory is often the critical performance bottleneck of data-centric computing systems.

Computational Storage Drive (CSD) can perform near-data processing on processors inside the storage device. Offloading some data processing to CSD reduces the amount of data movement between the storage and main memory, and thus reduces I/O latency and energy consumption, which is appealing for data-centric computing systems. Recently, many researchers studied how to use CSD to accelerate specific applications, e.g., accelerating SQL [8] and list intersection [19] in large databases, or developing application-specific hardware acceleration engines [14], [18].

Besides optimizing the software/hardware design to accelerate a specific application, another important aspect to fully unlocking the capability of CSD is how to manage the limited processing and memory resources of the CSD to serve the computational I/O requests of *multiple* applications simultaneously, which has received little attention in existing research. In reality, many applications have real-time constraints to their computational I/O requests, i.e., if the CSD cannot finish fetching and processing of the requested data before a certain deadline, the result will become useless. Therefore, a good CSD resource manager should both maximize the general throughput and meet the specific timing constraint of each individual request of different applications.

This paper presents SERICO, a framework to schedule multiple computational I/O requests in CSD. The main idea of SERICO is to perform admission control of computational I/O requests by online schedulability analysis, to avoid wasting the processing resources and memory capacity of CSD in doing useless work for those requests deemed to violate the timing constraints, which is supported by several key design points:

- SERICO exploits ping-pong buffers to "desynchronize" the scheduling and analysis of data fetching and processing on different cores, which improves the timing predictability and facilitates accurate online schedulability analysis for effective admission control;
- SERICO divides the serving tasks of each request into periodic jobs, which reduces the memory consumption (since the jobs of the same task can reuse the same buffer area). However, while dividing a task into jobs saves memory, it imposes higher pressure to system schedulability. SERICO will search for the period setting to minimize memory consumption while meeting the timing constraints;
- SERICO uses an EDF-based policy to schedule the jobs, and the online schedulability test guarantees that the admitted requests can meet their deadlines. The schedulability test is based on the classical Demand Bound Function (DBF) technique [2] while enhanced to handle the special characteristics of our problem;
- SERICO applies an *early-release* mechanism to further improve the chance to admit more requests by utilizing available resources which cannot be allocated to the periodically executed jobs.

We evaluate SERICO with synthetic workloads on simulators. Experiment results show that SERICO outperforms the baseline method currently used by the CSD device and the standard deadline-driven scheduling approach consistently with different parameter settings. We also implement SERICO on a realistic

978-3-9819263-7-8/DATE23/© 2023 EDAA

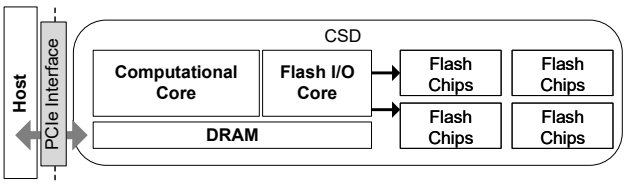

Figure 1: Hardware platform.

SSD development platform YS9203 [23] and evaluate its performance with real-world applications.

II. Related work

Previous studies on CSD mainly focus on accelerating specified applications on dedicated devices and building novel architectures. CSD has been applied to accelerate data-intensive database applications [4], [8], [22]. Recently, CSD has also been applied to accelerating AI applications, e.g., accelerating both the training [9], [12] and inference [18], [21] of Neural Networks models, and genome sequence analysis applications [6]. Some previous work aims at optimizing the programming model for CSD [7], [10], [15], [17]. In recent years, many works studied novel hardware architectures for CSD, e.g., developing an end-to-end architecture that supports block-oriented near-data processing [1], supports multiple Storage drives with different computation kernels [16], provides high-performance virtualization for CSD by applying hardware-assisted virtualization and constructing and scheduling virtual CSD dynamically [11]. To the best of our knowledge, SERICO is the first work to study how to serve multiple I/O requests of different applications simultaneously in CSD.

III. System Model

A. Hardware Platform

Fig. 1 shows the target hardware platform, where the CSD contains four major components: a computation core P_c, a Flash I/O core P_f, a set of Flash chips and a DRAM which is fully shared by P_c and P_f. P_f can address data blocks on the flash chips in the granularity of 4KB. The transfer of a 4KB data block is non-preemptive because an addressed 4KB data block is transferred to a pre-allocated DRAM area by hardware. We use T_f to denote the time to transfer one 4KB data block from Flash chips to DRAM. The host interacts with the CSD via PCIe interface and the command/data sent from the host can be stored in the DRAM of the CSD.

Although the CSD in Fig. 1 only has *one* computation core and *one* Flash I/O core, it is projected that future CSD may integrate more cores to provide higher processing capacity. In the following section, we will present SERICO in the context of one computation core and one Flash I/O core, but SERICO can be easily applied to the general case of multiple computation cores and Flash I/O cores, as will be discussed in Section IV-F.

B. Computational I/O Request

A *computational I/O request*, called *request* for short, denoted by RQ_i, is served by CSD with two parts. First, a Flash I/O task τ_i^f on P_f transfers the needed data from the Flash chips to DRAM. Second, a computation task τ_i^c on P_c processes these data, the result of which will be sent back to the host via PCIe.

A request RQ_i is characterized by a tuple $\langle r_i, d_i, b_i, n_i, c_i \rangle$. r_i is the *arrival time* and d_i is the *absolute deadline* of the request, i.e., RQ_i arrives (via PCIe) at time r_i, and the CSD should fetch the needed data and finish the required processing by time d_i. Otherwise, the request *misses* its deadline and the result returned after the deadline, if any, is useless. We define $D_i = d_i - r_i$ as the *relative deadline* of RQ_i.

A request may need to fetch and process a large amount of data. However, it is not necessary to transfer all the needed data from Flash to DRAM and process them at once. Instead, the data to be transferred and processed can be divided into *basic blocks*. The data processing must be performed with the granularity of basic blocks. We use b_i to denote the *basic block size* of RQ_i. Different requests have different basic block sizes. As the minimal granularity to transfer data from Flash to DRAM is 4KB, we assume that for any request RQ_i, b_i must be an integral multiple of 4KB. n_i denotes the total number of basic blocks needed by request RQ_i. For example, suppose a request RQ_i has $b_i = 12$KB and $n_i = 10$, then each time the processing algorithm is executed, the input data size (i.e., the basic block size) is $b_i = 12$KB, and in total this request needs to transfer and process $n_i \times b_i = 120$KB data.

We use c_i^c to denote the worst-case execution time (WCET) for τ_i^c to process *one* basic block of RQ_i, and c_i^f denotes the worst-case execution time (WCET) for τ_i^f to fetch *one* basic block from Flash to DRAM. We assume c_i^c is a known parameter which can be obtained by dynamic or static WCET analysis techniques [20]. As the time to transfer a 4KB data block is always T_f, we know $c_i^f = \frac{b_i T_f}{4\text{KB}}$.

At runtime, infinitely many requests, each characterized by the above parameters, arrive at the CSD at any time. Our target is to serve as many requests in time (i.e., meeting their absolute deadlines) as possible.

IV. Design of SERICO

A. Overview

Fig. 2 shows the overall architecture of SERICO. The **task admission module** receives requests from the host, decides whether to admit it or not and, if admitted, dispatches the corresponding computation task τ_i^c to P_c and Flash I/O task τ_i^f to P_f. The **computation task scheduler** and the **Flash I/O task scheduler** manage the computation and Flash I/O tasks on P_c and P_f, respectively.

Both the processing and memory capacity of the CSD are limited. A request will be admitted only if (1) there is enough space on DRAM to be allocated by the admission module and (2) it is assured that the request can meet its deadline. Otherwise, the request is rejected. If a request is rejected, the host will be notified via the returned completion, and then may either cancel this I/O request or resend it later with new timing constraints, depending on the application's requirement. This work focuses on scheduling inside the CSD, and how the host handles the rejected requests is out of the scope of this paper.

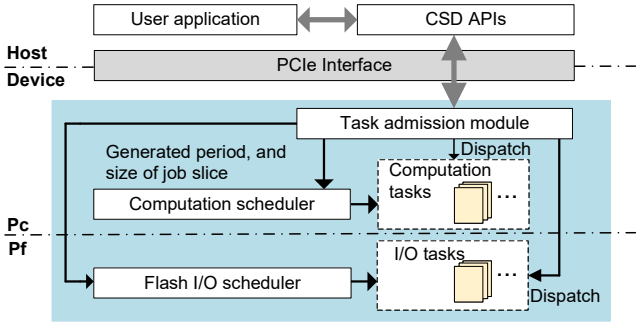

Figure 2: Architecture of SERICO.

B. Admission Control

Recall that as introduced in Section III-B, although a request may need to fetch and process a large amount of data, this can be done in several times instead of all at once. For example, we can fetch and process 1 basic block each time and repeat this for n_i times, or fetch and process 2 basic blocks each time and repeat this for $\lceil \frac{n_i}{2} \rceil$ times. In general, τ_i^f (τ_i^c) can be executed via a number of *jobs*, and each job fetches and processes several basic blocks.

Definition 1 (Job Granularity k_i)**.** *The* job granularity*, denote by k_i, of request RQ_i, is the number of basic blocks fetched/processed by each job of τ_i^f/τ_i^c.*

The jobs will be released periodically with an artificial period, which is also the relative deadline of a job, i.e., a job should finish execution before the release of the next job. In this way, the jobs of a task can reuse the same memory area as their life window do not overlap with each other in time. Therefore, to fully utilize the limited memory, it is preferable to use as-small-as-possible k_i i.e., dividing a task into as small jobs as possible. However, dividing a task into more jobs leads to a smaller timing window to execute a job and thus imposes higher pressure to system schedulability. Therefore, the admission control module will try to find a proper job granularity k_i when requests RQ_i arrive, so that the resulting periodic jobs of τ_i^f and τ_i^c can be assigned to P_f and P_c, without violating the timing and memory constraints. As will be introduced in Section IV-C, SERICO uses a *ping-pong buffer* for τ_i^f and τ_i^c to coordinate with each other, the needed buffer size of RQ_i is $2k_ib_i$ given a k_i value.

As our target is to find the k_i value that minimizes RQ_i's ping-pong buffer size, we will check $k_i = 1 \cdots n_i$ in the *increasing* order, until we find a k_i so that the τ_i^f and τ_i^c can be divided into smaller periodic jobs and the jobs on P_f and P_c are still schedulable. If no $k_i \in [1, n_i]$ can pass the schedulability test, RQ_i is rejected. We will introduce the schedulability test in detail in Section IV-D. If we have found the minimal k_i that can pass the schedulability test, we will check whether there is enough available DRAM space to allocate the ping-pong buffer of size $2k_ib_i$ to RQ_i. If yes, RQ_i is admitted, otherwise, it is rejected and larger values for k_i won't be checked.

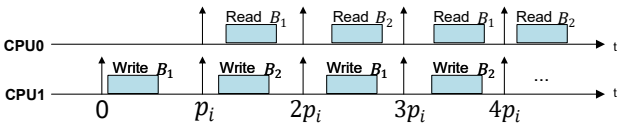

Figure 3: Illustration of execution with ping-pong buffer.

C. Scheduling

Once a request RQ_i is admitted and k_i is decided, we generate $\lceil \frac{n_i}{k_i} \rceil$ jobs for τ_i^f and τ_i^c and schedule them on P_f and P_c, respectively. A job of τ_i^f and a job τ_i^c need to be synchronized to complete the transfer and processing of k_ib_i data. A job of τ_i^f first transfers data from Flash to DRAM, and then the job of τ_i^c can process these data. In other words, a job of τ_i^c can start execution only after the corresponding job of τ_i^f has finished. The synchronization between τ_i^c and τ_i^F makes the schedulability less predictable as the "effective release time" of a job (i.e., the time when its data is ready and thus it is eligible for execution) depends on the finish time of the corresponding job on the other core, which relies on the scheduling of all running tasks on that core. This unpredictability makes the analysis of system schedulability difficult. Even worse, as the "effective relative deadline" (the time between its effective release time and absolute deadline) becomes shorter, it is more difficult to make the jobs schedulable.

Ping-Pong Buffer. In order to solve the above problem, SERICO uses a *ping-pong buffer* to "desynchronize" the execution of a pair of jobs of τ_i^f and τ_i^c. A ping-pong buffer has two buffer areas (B_1 and buffer B_2), each of size k_ib_i. As shown in Fig. 3, the job of τ_i^f fetches data from Flash and writes them to buffer B_1 and B_2 alternatively. In every period in which the job of τ_i^f writes data to B_1, the job of τ_i^c reads and processes the data in B_2, and vice versa. In this way, the jobs of τ_i^f and τ_i^c are executed on P_f and P_c as if there is no data dependency between them, as long as each job is executed between its release time and absolute deadline, which greatly improves system predictability and simplifies the online schedulability analysis.

Job Parameter Calculation. With the ping-pong buffer, a job of τ_i^c reads and processes the data fetched by the job of τ_i^f in the previous period. Therefore, while the first job of τ_i^f is released immediately at r_i, the first job of τ_i^c is actually released at $r_i + p_i$, where p_i is the *period* of the jobs of τ_i^f and τ_i^c. Similarly, while the period of the last job of τ_i^c ends at the d_i (the absolute deadline of the request), the last job of τ_i^f has its period ending at $d_i - p_i$. Therefore, the period p_i is calculated by

$$p_i = \frac{D_i}{\lceil \frac{n_i}{k_i} \rceil + 1}$$

The worst-case execution time (WCET) of each job of τ_i^f and τ_i^c are calculated by

$$e_i^f = T_f k_i b_i / 4\text{KB}, \quad e_i^c = k_i c_i$$

EDF-based Scheduling. Thanks to the ping-pong buffers, τ_i^f and τ_i^c behave as independent periodic real-time tasks. SERICO

adopts EDF-based scheduling to schedule both computation tasks on P_c and Flash I/O tasks on P_f, since EDF is the optimal single-processor scheduling policy for periodic real-time tasks [13]. However, while EDF is a fully-preemptive scheduling policy (i.e., a job with the earliest deadline can preempt others immediately), in SERICO the scheduling policy used on both P_f and P_c are limited-preemptive EDF (lp-EDF) [3]. On P_f, since the minimal granularity of data transfer is 4KB, the preemption can only take place at the boundaries of transfer of 4KB data blocks. On P_c, we enforce that preemption only happens at the boundary of processing of basic blocks. This is because allowing preemption within a basic block generally leads to higher memory consumption for context saving. As the nested preemption requires to store the context for many tasks at the same time, this will lead to high memory consumption.

D. Schedulability Analysis

As introduced above, using the ping-pong buffer, τ_i^f and τ_i^c can correctly exchange data as long as each job is finished before the next release time. Therefore, the schedulability on P_f and P_c can be analyzed independently. In the following, we first present the schedulability analysis for P_f, and later explain how it can be applied to P_c with slight modification.

Suppose the current time is t_0, at which a new request RQ_i arrives, and currently, we are analyzing whether all jobs can still meet their deadline if the periodic jobs of RQ_i can be admitted so that all jobs are guaranteed to meet their deadlines. Suppose there are deadline misses after t_0, and the first deadline miss happens at time t_d. There are two possible cases:

1) The core is idle at some point between t_0 and t_d.
2) The core is continuously busy between t_0 and t_d.

In the following, we will derive sufficient conditions to guarantee that there is not enough workload to cause a deadline miss in each of the above two cases.

We start with case 1). Let t_s denote the *first* time point before t_d so that P_f is continuously busy between t_s and t_d. In other words, t_s is the starting time of the *busy period* [2] containing t_d. To analyze the schedulability in this busy period, we can use the well-known demand bound functions (DBF) [2]:

$$\mathrm{DBF}_i(\Delta) = \left(\left\lfloor \frac{\Delta - p_i}{p_i} \right\rfloor + 1 \right) e_i^f \tag{1}$$

which represents the workload of jobs of task τ_i^f with both release times and absolute deadlines within a time interval of length Δ. Note we replace the relative deadline by period in the definition $\mathrm{DBF}_i(\Delta)$ as in our problem a job's relative deadline equals to its period. A periodic task system is schedulable under lp-EDF with non-preemptive blocking time B_i if:

$$\forall \Delta : \sum_{\text{all } \tau_i}^{n} \mathrm{DBF}_i(\Delta) + B_i \leq \Delta$$

In our problem, since τ_i^f only releases $\lceil \frac{n_i}{k_i} \rceil$ jobs, rather than infinitely many jobs as in the standard periodic task model, we should refine DBF to fit our problem:

$$\mathrm{DBF}_i^*(\Delta) = \min \left(\left\lceil \frac{n_i}{k_i} \right\rceil e_i^f, \mathrm{DBF}_i(\Delta) \right) \tag{2}$$

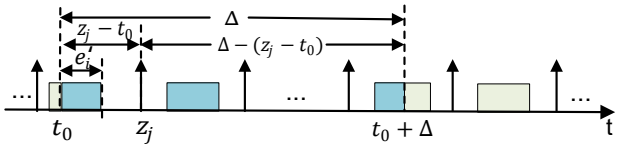

Figure 4: Illustration of the intuition of (4).

and obtain a sufficient condition to guarantee that a deadline miss at t_d is impossible in case 1):

$$\forall \Delta : \sum_{\text{all } RQ_i} \mathrm{DBF}_i^*(\Delta) + B_i \leq \Delta$$

where B_i equals T_f on P_f.

Then we consider case 2). For this case, we want to derive an upper bound for the total demand in $[t_0, t_d]$. We first consider the newly arrive request RQ_i. The first job of τ_i^f is immediately released at t_0, so the demand of jobs of τ_i^f in time interval $[t_0, t_0 + \Delta)$ is calculated by

$$W_i^f(\Delta) = \left\lfloor \frac{\Delta}{p_i} \right\rfloor e_i^f \tag{3}$$

Then we consider existing requests that are already admitted to the system. For each existing request RQ_j, we use e_j' to denote the maximal remaining execution time of the current job, and z_j the next job release time, and x_j the number of unreleased jobs of this task, which are all available information at the current time point. In particular, we can estimate e_j' by counting the amount of data to be finished, instead of using timers to keep track of the actual executed time of each running job. Then we can compute the total amount of workload in time interval $[t_0, t_0 + \Delta)$:

$$W_j(\Delta) = e_j' + \min \left(x_j, \max \left(0, \left\lfloor \frac{\Delta - (z_j - t_0)}{p_j} \right\rfloor \right) \right) e_j^f \tag{4}$$

The intuition of (4) is shown in Fig. 4.

Therefore, a sufficient condition to guarantee that a deadline miss at t_d is impossible in case 2) for P_f is

$$\forall \Delta : \sum_{RQ_j \in \mathbf{S}} W_j(\Delta) + W_i^f(\Delta) \leq \Delta \tag{5}$$

where $\mathbf{S}$ denotes the set of requests currently being served.

Putting the above discussions together, the following theorem summarizes the schedulability test condition for P_f:

Theorem 1. *All the jobs on P_f can meet their deadlines if the newly arrived request is admitted with k_i (based on which p_i, e_i^f and e_i^c are determined) if $\forall \Delta$:*

$$\sum_{all\ RQ_i} \mathrm{DBF}_i^*(\Delta) + B_i^f \leq \Delta \ \wedge \sum_{RQ_j \in \mathbf{S}} W_j(\Delta) + W_i^f(\Delta) \leq \Delta \tag{6}$$

The schedulability condition for P_c can be derived similarly, with several differences. First, e_i^f in (1), (2) and (4) should be replaced by e_i^c. Second, the non-preemptive blocking time on P_c is the maximal c_i among all τ_i^c currently running on P_c, as on P_c the preemption happens at the boundary of processing basic blocks. Third, the calculation of the demand of τ_i^c is

different from (3). Recall that, since τ_i^f and τ_i^c exchanges data with a ping-pong buffer, and thus the first job of τ_i^c is actually released at $t_0 + p_i$. Therefore, we should replace $W_i^f(\Delta)$ by $W_i^c(\Delta)$, which is calculated by

$$W_i^c(\Delta) = \max\left(0, \left\lfloor \frac{\Delta - p_i}{p_i} \right\rfloor e_i^c\right)$$

Note that (6) only needs to be checked for Δ up to a certain upper bound, which can be derived by the standard technique in real-time scheduling: we over-approximate $\mathrm{DBF}_i'(\Delta)$, $W_j(\Delta)$, $W_i^f(\Delta)$ and $W_i^c(\Delta)$ by, e.g., linear functions with respect to Δ and compute the smallest values of Δ at which the sum of these over-approximated functions intersect with the diagonal, which can be used as the upper bound of Δ values to be checked. The details are omitted due to the page limit.

E. Early Release Mechanism

The jobs of τ_i^f and τ_i^c are released periodically since the schedulability of periodic release patterns is relatively easier to analyze. At runtime, it is possible that the core is temporally idle while the next job of each task isn't released yet due to the periodic release pattern. SERICO uses an *early release* mechanism to reclaim such idle time so that the current tasks can finish sooner, which increases the chance of admitting more requests in the future.

The early release mechanism allows a job to be released before its original scheduled release time, while still using its original deadline. However, early releasing a job may cause problems in data exchange between τ_i^c and τ_i^f with the ping-pong buffer. Recall that, with the ping-pong buffer, a job of τ_i^c reads the data fetched by the job of τ_i^f in the previous period. If a job of τ_i^c is released and starts to execute early, and its needed data may not have been ready since the job of τ_i^f in the last period may not have finished. Similarly, if a job of τ_i^f is released and starts to execute early, its target buffer may not be available for it to write since the job of τ_i^f in the last period may not have finished.

To solve the above problem, when a core is idle, SERICO will select a job to be early-released only if the job of its counterpart in the other core in the previous period has finished. If multiple jobs can be early released, SERICO selects the one with the earliest deadline. Note that each early-released job still uses its original deadline. Thus the early-release mechanism does not invalidate the schedulability guarantee present in the last subsection since it simply reclaims the idle time without introducing extra interference or blocking to any job.

F. Further Remarks

For simplicity of presentation, we did not consider the memory requirement to store the results of τ_i^c in our problem model. In reality, the memory requirement to store the results of τ_i^c s is usually much smaller than the raw data, which can be easily taken into account by SERICO when admitting a request.

So far we have introduced the design of SERICO assuming only one computation core and one Flash I/O core. However, SERICO can be easily extended to the case of multiple computation cores and/or multiple Flash I/O cores, by adding a policy to decide which computation core and Flash I/O core to be used to execute the τ_i^c and τ_i^f, e.g., using real-time task partitioning heuristics [5]. After the tasks are allocated, the scheduling and analysis of jobs on each core are independent from other cores.

Although this paper only considers real-time requests, SERICO can also be extended to handle non-real-time requests. We can serve these non-real-time requests with lower priority than real-time tasks, i.e., only when currently there is no workload for serving real-time requests.

V. Evaluation

We conduct experiments with both synthetic workloads on a simulator and a couple of applications on a realistic CSD hardware platform with an SSD controller YS9203 [23].

A. Evaluation with Simulation

We developed an event-triggered simulator to simulate the high-level scheduling behavior on CSD, then conduct experiments with synthetic workloads with various parameter settings. SERICO is compared with following baseline methods:

- **FCFS** (First-Come-First-Served): The system maintains a list of all arrived requests and serves them in the order of their arrival times. This is the method in the default system software with YS9203 platform;
- **Simple-EDF**: The system maintains a list of all arrived requests and serve them in the order of their absolute deadlines. It will drop the requests once the request exceeds its deadline.

We generate requests with parameters randomly distributed in the following ranges: $r_i \in [0, 150000]$, $D_i \in [1000, 150000]$, $b_i \in [1, 10]$ and $n_i \in [10, 400]$. We change the range of c_i for the x-axis in Figure 5, where with a certain x value, c_i is randomly distributed in $[5, x]$. The y-axis represents the value of T_f. The z-axis is the *reject ratio*, which is the ratio between the number of requests that are rejected (with SERICO) or miss their deadlines (with FCFS and Simple-EDF), and the total number of requests. The experimental results in Figure 5 show that, as c_i and T_f increase, the reject ratio of all the 3 methods in comparison increase. Overall, SERICO has a much lower reject ratio than FCFS and simple-EDF.

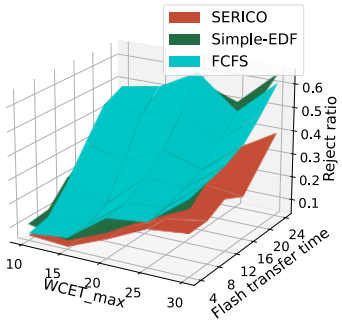

Figure 5: Reject ratio in simulation (lower is better).

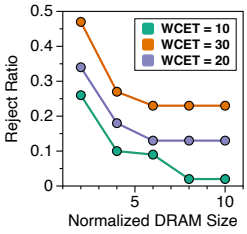

Figure 6: Impact of DRAM size on reject ratio.

Figure 6 shows the reject ratio with different normalized DRAM sizes, which is defined as the total buffer size divided

by the maximum b_i of all requests. For each curve, c_i is randomly generated in $[5, \text{WCET}]$. For this setting, the reject ratio decreases as the available buffer size increase in the CSD.

B. Evaluation on Realistic Hardware

We implement SERICO on CSD platform with an SSD controller YS9203 [23] equipped with 2 Cortex-R4 cores (one computation core and one Flash I/O core), both with 600MHz frequency. The available DRAM size is 1GB. Each experiment has 10 applications, and each application executes either the SQL query function or the statistic analysis function as the computational task for each I/O request in CSD. Each application generates requests periodically with a period of 100ms. The amount of data to be fetched and processed by each request is randomly chosen between 70MB and 1GB. The deadline of each request is randomly distributed in $[5000, 50000]$ms. We assume both functions have a basic block size of 512KB. The WCET for processing one basic block, i.e., c_i, is 12ms for the SQL query function and 8ms for the statistic analysis function. Figure 7 shows the reject ratio of the three methods with X applications executing the SQL query function and Y applications executing the statistic analysis function, where X:Y are set to be 3:7, 5:5 and 7:3, respectively.

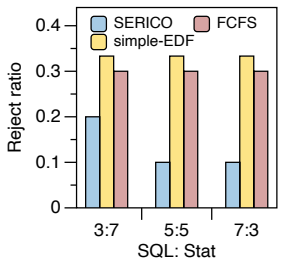

Figure 7: Reject ratio of real workloads (lower is better).

We measure the memory and time overhead of SERICO on YS9203. The memory overhead for maintaining the 4KB data blocks on DRAM is 655KB, about 0.06% of the total available DRAM on YS9203. The time overhead of SERICO includes the admission control and timer updating to keep track of the job periods. The total time overhead is 4.12% of the total execution time in the above experiments with 10 applications. Note that the significant improvement by SERICO shown in Figure 7 already includes these timing overheads. Although we do not include the time to transfer the results back to the host in our problem model, we also measure this time cost in our experiments. The speed of transferring results from the CSD to host via PCIe is 10us/32KB, which is negligible compared with the time to fetch and process data in the CSD.

VI. Conclusion

This paper studies how to manage the limited processing and memory resources in CSD to serve real-time I/O requests of multiple applications. We present SERICO, a system of scheduling computational I/O requests in CSD. SERICO performs online admission control to avoid wasting the processing resources and memory capacity of CSD in handling the requests that cannot meet their deadlines anyway. The Flash I/O and computation workloads of a request is divided into periodic jobs to meet its timing constraint while minimizing memory consumption. We evaluate SERICO with both synthetic workloads on the simulator and representative applications on realistic CSD hardware. Experiment results show that SERICO significantly outperforms the default method used in the CSD device and the standard deadline-driven scheduling approach.

Acknowledgement

We sincerely thank anonymous reviewers for their feedback. This work is partially supported by the Research Grants Council of Hong Kong, China, under Grant CityU 11219319.

References

[1] Antonio Barbalace, Martin Decky, Javier Picorel, and Pramod Bhatotia. BlockNDP: Block-Storage Near Data Processing. In *Middleware '20*.

[2] Sanjoy K Baruah, Aloysius K Mok, and Louis E Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *RTSS,1990*.

[3] Marko Bertogna and Sanjoy Baruah. Limited preemption EDF scheduling of sporadic task systems. *IEEE Trans. on Industrial Informatics*, 2010.

[4] Zhichao Cao, Huibing Dong, Yixun Wei, Shiyong Liu, and David H. C. Du. IS-HBase: An In-Storage Computing Optimized HBase with I/O Offloading and Self-Adaptive Caching in Compute-Storage Disaggregated Infrastructure. *ACM Trans. Storage*, 2022.

[5] Robert I. Davis and Alan Burns. A Survey of Hard Real-Time Scheduling for Multiprocessor Systems. *ACM Comput. Surv.*

[6] Mansouri Ghiasi et al. GenStore: A High-Performance in-Storage Processing System for Genome Sequence Analysis. In *ASPLOS '22*.

[7] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. Biscuit: A framework for near-data processing of big data workloads. In *ISCA '16*.

[8] Insoon Jo, Duck-Ho Bae, Andre S. Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel D. G. Lee, and Jaeheon Jeong. YourSQL: A High-Performance Database System Leveraging in-Storage Computing. *Proc. VLDB Endow.*

[9] Shine Kim, Yunho Jin, Gina Sohn, Jonghyun Bae, Tae Jun Ham, and Jae W Lee. Behemoth: A Flash-centric Training Accelerator for Extreme-scale DNNs. In *USENIX FAST '21*.

[10] Gunjae Koo, Kiran Kumar Matam, Te I., H.V. Krishna Giri Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, and Murali Annavaram. Summarizer: Trading Communication with Computing Near Storage. In *MICRO '17*.

[11] Dongup Kwon, Dongryeong Kim, Junehyuk Boo, Wonsik Lee, and Jangwoo Kim. A Fast and Flexible Hardware-based Virtualization Mechanism for Computational Storage Devices. In *USENIX ATC '21*.

[12] Yunjae Lee, Jinha Chung, and Minsoo Rhu. SmartSAGE: Training Large-Scale Graph Neural Networks Using in-Storage Processing Architectures. ISCA '22.

[13] C. L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM*, 1973.

[14] Shuyi Pei, Jing Yang, and Qing Yang. REGISTOR: A Platform for Unstructured Data Processing Inside SSD Storage. *ACM Trans. Storage*, 2019.

[15] Zhenyuan Ruan, Tong He, and Jason Cong. INSIDER: Designing In-Storage computing system for emerging High-Performance drive. In *USENIX ATC '19*.

[16] Sahand Salamat, Armin Haj Aboutalebi, Behnam Khaleghi, Joo Hwan Lee, Yang Seok Ki, and Tajana Rosing. NASCENT: Near-Storage Acceleration of Database Sort on SmartSSD. FPGA '21.

[17] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A User-Programmable SSD. In *USENIX OSDI '14*.

[18] Xuan Sun, Hu Wan, Qiao Li, Chia-Lin Yang, Tei-Wei Kuo, and Chun Jason Xue. RM-SSD: In-Storage Computing for Large-Scale Recommendation Inference. In *HPCA '22*.

[19] Jianguo Wang, Dongchul Park, Yang-Suk Kee, Yannis Papakonstantinou, and Steven Swanson. SSD In-Storage Computing for List Intersection. In *DaMoN '16*.

[20] Wilhelm et al. The Worst-Case Execution-Time Problem—Overview of Methods and Survey of Tools. *ACM Trans. Embed. Comput. Syst.*

[21] Mark Wilkening, Udit Gupta, Samuel Hsia, Caroline Trippel, Carole-Jean Wu, David Brooks, and Gu-Yeon Wei. RecSSD: near data processing for solid state drive based recommendation inference. In *ASPLOS '21*.

[22] Zsolt Woods, Louis István and Gustavo Alonso. Ibex: An Intelligent Storage Engine with Support for Advanced SQL Offloading. *Proc. VLDB Endow.*

[23] YEESTOR. Yeestor YS9203 PCIe SSD Memory Controller, 11 2021.