Dynamic Chip Clustering and Task Allocation for Real-time Flash

Gyeongtaek Kim, Sungjin Lee, and Hoon Sung Chwa†
Information and Communication Engineering
DGIST
Daegu, Republic of Korea
{xor0925, sungjin.lee, chwahs}@dgist.ac.kr

2021 58th ACM/IEEE Design Automation Conference (DAC) | 978-1-6654-3274-0/21/$31.00 ©2021 IEEE | DOI: 10.1109/DAC18074.2021.9586252

***Abstract*—The goal of this paper is to provide worst-case timing guarantees for real-time I/O requests, while fully utilizing the potential bandwidth for non real-time I/O requests in NAND flash storage systems. We identify a trade-off between flash chip sharing and I/O workload isolation in terms of timing guarantees and bandwidth. By taking such a trade-off into account, we propose a new real-time I/O scheduling framework that enables dynamic isolation between real-time I/O requests to meet all timing constraints and co-scheduling of real-time and non real-time I/O requests to provide high bandwidth utilization. Our in-depth evaluation results show that the proposed approach outperforms existing isolation approaches significantly in terms of both schedulability and bandwidth.**

I. Introduction

Thanks to a small form factor, high shock resistance, low power consumption, and fast access time, NAND flash-based solid-state drives (SSDs) have been widely used as storage devices in safety-critical real-time embedded systems. In typical real-time embedded systems, real-time I/O workloads that require to meet their timing constraints are colocated with non real-time I/O workloads on the shared NAND flash storage. The main design goal for real-time NAND flash storage systems is to provide worst-case timing guarantees for real-time I/O requests, while fully utilizing the potential bandwidth for non real-time I/O requests.

However, flash-based SSDs often exhibit highly unpredictable worst-case I/O latency. Data written in a flash page (which is a basic unit of reading and writing data) cannot be updated unless the page is erased. The erase operation can only be performed in the unit of a block composed of many pages. Before updating data of a specific page in a block, other pages containing valid data should be moved elsewhere; otherwise, their data are permanently erased. Since the erase operation involves many page reads and writes, flash-based SSDs redirect write requests to free pages and run a garbage collection (GC) task which reclaims obsolete flash pages. While it improves overall I/O throughput, the GC task often blocks user I/O requests, causing unpredictable I/O latency, which are not acceptable to real-time applications.

To make the worst-case I/O latency predictable, a vast amount of work has been done by the real-time community. The most recognized effort is to develop real-time GC mechanisms to reduce and/or to bound the blocking time by the GC operation [1]–[3]. They estimated the worst-case GC cost and then schedule GC operations preemptively with real-time I/Os. Owing to frequent GC interference by other tasks, however, the worst-case latency bound is set too conservatively, which result in underutilization of SSD throughput. To overcome this problem, one suggested an idea of partitioning a set of NAND flash chips into separate read and write sets [4]. By isolating read and write requests within separate chips, it guarantees that read requests are never blocked by write requests or GC operations. But, this benefit comes at the cost of degraded write throughput. Some proposed resource isolation techniques [5]–[9] that assign a given task set to dedicated NAND chips depending their I/O requirements. Since all the tasks are completely isolated, GC interference among tasks are minimized. However, those techniques are designed for non real-time tasks and thus cannot be applied to real-time tasks.

In this paper, we propose a new real-time I/O scheduling framework that enables dynamic resource isolation between real-time I/O requests to meet all timing constraints and co-scheduling of real-time and non real-time I/O requests to provide high bandwidth utilization. First, we compare, via a case study, two typical flash chip isolation approaches — *shared* and *fully-isolated* — and demonstrate the effect of different levels of isolation on the worst-case latency. Under the shared approach where all I/O requests are allowed to read/write data across all flash chips, each write request can fully utilize all flash chips, triggering less frequent GC invocations. This benefit, however, comes at the large amount of interference imposed by other I/O requests. On the other hand, under the fully-isolated approach where each I/O request is statically assigned to a single chip and is allowed to read/write data on that chip only, it receives less interference than the case of no isolation. However, it comes at the cost of frequent GC invocations without utilizing free pages in other chips.

Motivated by this, we consider another resource isolation approach, called *cluster-based isolation*, that is a generalization of shared and fully-isolated approaches. In this approach each I/O request is statically assigned to a cluster that is a subset of flash chips and is allowed to read/write data within the cluster. To support such a cluster-based isolation approach, we develop dynamic chip clustering and request-to-cluster allocation algorithms that determine a cluster configuration and a mapping from I/O requests to clusters to satisfy all timing constraints in the worst case. Building upon the request-to-cluster allocation, we also propose a co-scheduling algorithm to schedule non real-time I/O requests to be executed with real-time I/O requests in the clusters to achieve high bandwidth without violating any timing constraints.

Our in-depth evaluation results show that the proposed cluster-based isolation approach outperforms the existing isolation approaches significantly in terms of the schedulability for real-time I/O requests and the bandwidth for non real-time I/O requests. Our approach is shown to make 43% and 278% more real-time task sets schedulable and also improve the average bandwidth of non real-time I/O requests by up to 96%, respectively, over the fully-isolated and shared approaches.

This paper makes the following main contributions:

- An insightful case study that reveals the effect of different levels of flash chip isolation on the worst-case latency of real-time I/O requests (Section III);
- Development of task-to-cluster allocation with dynamic chip clustering that significantly improves the schedulability of real-time I/O requests by considering a trade-off between sharing chips and isolating chips (Section IV-C);
- Development of a scheduling framework for a mixed-set of real-time and non real-time I/O requests that enables high bandwidth of non real-time I/Os while guaranteeing the timing constraints of real-time I/Os (Section IV-D); and

† Hoon Sung Chwa is the corresponding author.

- Demonstration of the effectiveness of the proposed approach in terms of the schedulability for real-time I/Os and the bandwidth for non real-time I/Os (Section V).

II. Background

A NAND flash chip consists of multiple blocks, each of which is composed of several pages. A page is a unit of read/write operations, while a block is a unit of erase operations. To provide high throughput, an SSD controller aggregates multiple NAND chips using a channel-way architecture. Each channel has separate data and control buses. This enables the controller to achieve high I/O throughput by accessing NAND chips on different channels simultaneously through *channel parallelism*. NAND chips on the same channel share the data bus, but are controlled independently by the controller via separate control buses. By utilizing *way interleaving*, the controller can further improve aggregate I/O throughput.

A flash translation layer (FTL) is the firmware running in the controller; it not only provides the typical block I/O interface, but manages multiple NAND chips with unusual physical properties. To hide an out-of-place update nature of flash, the FTL writes incoming data to free pages in a manner that maximizes full throughput of multiple chips. When free space is exhausted, the FTL triggers GC to reclaim free space, which involves a series of page reads, page writes, and block erasures. The FTL usually performs GC operations over multiple channels and ways in parallel to minimize GC time.

Existing FTL designs, however, fail to offer consistent I/O response times, causing unpredictable I/O latency. When real-time I/O requests come while the controller is busy executing GC, they are inevitably delayed until all the GC operations are completed. To address this, some suggested real-time GC scheduling [1]–[3]. It models the GC task as part of real-time tasks by estimating its worst-case cost and partially performs GC I/Os so that they can be preempted by high priority I/Os. It provides a guideline for real-time tasks to meet worst-case timing requirements under GC activities. However, these studies estimate GC cost conservatively and do not consider the channel-way architecture. This results in low utilization of SSD throughput.

Another proposed a partitioned scheduling algorithm that exploits the channel-way architecture for GC isolation [4]. It splits NAND chips into read and write sets and then enforces all the writes to be sent to the write set. This leads GC I/Os to only happen in the write set, preventing read requests from being blocked by GC. This approach makes it possible to provide excellent read throughput, but overall write performance degrades since only a few NAND chips out of many are designated to serve write requests.

Some go one step further by proposing resource isolation techniques [5]–[9]. While detailed designs differ from each other, they attempt to assign a given set of tasks into dedicated flash resources (i.e., channels, ways, and chips). Since each task is completely isolated in physically separated flash resources, it not only minimizes I/O interference among tasks but provides better I/O performance to given tasks. However, all those techniques are designed for non real-time tasks and thus are not able to satisfy the worst-case timing guarantees for real-time I/O requests. They also do not consider the optimal task assignment that satisfies the worst-case latency of I/O tasks while maximing the utilization of SSD throughput.

III. Motivation and Problem Statement

We first present a case study to demonstrate the effect of flash chip isolation on the worst-case I/O latency and describe our goal for the proposed task-to-cluster allocation scheme thereof.

A. Motivation

We illustrate a measurement-based case study to motivate our approach. We use an emulated SSD drive [10] with the configuration shown in Table I. There are three real-time I/O tasks that perform periodic I/O requests on an SSD with three chips[1]: one task writes 24 pages every $30ms$ and reads 40 pages every $36ms$; the other two write 12 pages every $130ms$ and read 80 pages every $25ms$. To consider the worst-case GC cost, all the tasks issue random writes to the SSD. Three tasks are scheduled under non-preemptive earliest deadline first (EDF) scheduling. We measure the worst-case I/O latency for each task under different resource isolation approaches, as presented in Fig. 1. The worst-case I/O latency is decided by three major factors: (*i*) the time taken to read/write pages for a task, (*ii*) the time taken to perform GC for a task, and (*iii*) interference I/Os (i.e., page reads/writes and GC I/Os) from other tasks.

TABLE I: SSD configuration

Parameter	Value	Parameter	Value
Page size	8KB	Read latency (t_r)	50 μs
# of pages/block	256	Write latency (t_w)	500 μs
# of blocks/chip	64	Erase latency (t_e)	5 ms
		Date transfer time (t_d)	40 μs

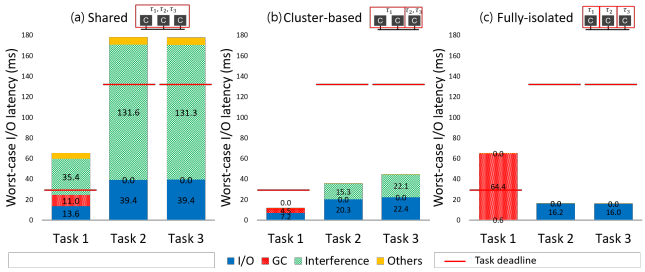


Fig. 1: Motivational case study: the worst-case write latency for each task and its breakdown under different isolation approaches

Via this case study, we examine the effect of different levels of flash chip isolation on the worst-case latency of an real-time I/O request. We first compare two simple approaches for chip isolation over m chips: *shared* (i.e. no isolation) and *fully-isolated* resources. Under the shared approach, all I/O requests are allowed to read/write data across m chips as existing SSDs do. Under the full-isolation approach, each I/O request is statically assigned to a single chip and is allowed to read/write data on that chip only.

Figs. 1a and 1c show the worst-case latency of an real-time I/O request under the shared and fully-isolated approaches, respectively. Under the shared approach, GC is triggered less frequently than the fully-isolated approach because free space available in all the chips can be utilized by all the tasks. However, each task suffers from high GC interference by other tasks, which results in the violation of its timing constraints. Under the fully-isolated approach, there is no GC interference of other I/O requests. However, it suffers from high GC overheads to reclaim free pages since free pages in other chips cannot be utilized. This results in violating its timing constraints.

To overcome disadvantages of both approaches, we consider another approach, called *cluster-based isolation*, using a notion of *(chip) cluster*. A cluster is a set of m' chips, where $1 \leq m' \leq m$. Under the cluster-based approach, I/O requests are statically assigned to a cluster and are allowed to share m' chips within the cluster. Fig. 1b shows the worst-case latency of an real-time I/O request under the cluster-based approach. By trading-off between resource contention by sharing chips and garbage collection overhead by isolating chips, we can achieve a lower worst-case latency than both the shared and fully-isolated approaches, satisfying the timing constraint.

[1]As a motivational case study, we consider a simple task set with a small number of chips. Our in-depth evaluation results with various task sets will be presented in Section V.

Moreover, in addition to real-time I/O requests, there might be non real-time I/O requests to be serviced together in a flash storage system. Then, the execution of non real-time I/O requests must not violate timing guarantees of real-time I/O requests, while non real-time I/O requests are intended to obtain high bandwidth.

B. Problem statement

Motivated by our case study, we focus on cluster-based isolation and aim to solve the following task-to-cluster allocation problem for real-time NAND flash storage systems. Let $\pi = \{\pi_1, \pi_2, ..., \pi_m\}$ be the set of m flash chips, let $\tau = \{\tau_1, \tau_2, ..., \tau_n\}$ be the set of n real-time (RT) periodic I/O tasks, and let $J = \{J_1, J_2, ...\}$ be the collection finite or infinite non-RT aperiodic I/O jobs.

Definition 1: Given a RT task set τ and a collection of non-RT jobs J running on a NAND flash storage system π, determine (i) cluster configuration Φ, (ii) task-to-cluster assignment Λ such that

G1. the average bandwidth of non-RT I/O jobs is maximized; and

G2. all the read and write I/O requests of a RT task $\tau_i \in \tau$ meet their deadlines for all legitimate I/O request sequences.

For a cluster configuration, each *cluster* ϕ_k is a disjoint partition of π. Hence, $\pi = \cup\phi_k$, and if $i \neq j$, $\phi_i \cap \phi_j = \emptyset$. There can be multiple possible ways to form a set of clusters for m flash chips, where each of such ways is defined as a cluster configuration Φ.

For a task-to-cluster allocation, we need to find i) a mapping from the RT tasks of τ to the clusters in Φ and ii) a run-time schedule of non-RT jobs to be executed with RT tasks to achieve G1 and G2.

Note that the cluster-based isolation approach can be viewed as a generalization of shared and fully-isolated approaches; it is equivalent to the shared approach at one extreme end where we assign I/O requests to a single cluster of size m, and the fully-isolated approach at the other extreme end where we assign I/O requests to m clusters each of size one.

IV. Cluster-based Resource Isolation

In this section, we first describe our target task model and derive a schedulability analysis to verify whether all RT I/O tasks satisfy their timing constraints under a given task-to-cluster allocation. We then present our approach to determine a task-to-cluster allocation together with a cluster configuration for RT I/O tasks and generate a schedule for non-RT jobs to achieve G1 and G2.

A. Task model

1) Real-time I/O task: Real-time I/O requests are presented by the periodic task model which is widely used in various real-time storage systems [1], [4]. Each RT I/O task $\tau_i \in \tau$ can be specified as $\tau_i = [(r_i, T_i^r), (w_i, T_i^w)]$, where it performs r_i page reads every T_i^r time-units and w_i page writes every T_i^w time-units.[2] Such a task τ_i is assumed to generate a potentially infinite sequence of read and write requests every T_i^r and T_i^w time-units, respectively, with each read (write) request needing to complete r_i page reads (w_i page writes) within a relative deadline of T_i^r (T_i^w) time-units.

Read/write request. For a read/write request, the worst-case scenario is when every read/write operation is sequentially serviced on a flash chip. The worst-case execution times (WCETs) of read and write requests for τ_i (denoted as C_i^r and C_i^w, respectively) are

$$C_i^r = r_i \cdot t_r, \tag{1}$$

$$C_i^w = w_i \cdot t_w, \tag{2}$$

where t_r and t_w are the time to read and write a page on a chip, respectively.

[2]The task model does not account for the CPU computation time. These tasks exist on the FTL and utilize the NAND bus. A task is assumed to be scheduled on the CPU in a way that is able to guarantee the above read and write request rates on the SSD as in [4].

Garbage collection request. We consider *cluster-level* parallel garbage collection (GC), in which when a flash chip in a cluster ϕ_k runs out of free pages, GC is triggered on all chips in ϕ_k in parallel to reclaim free pages. Every RT I/O task with $w_i > 0$ will have its corresponding GC task to ensure that enough free pages are reclaimed for τ_i to write w_i pages every T_i^w time-units. When the GC operation starts for each flash chip in a cluster ϕ_k, a victim block with the minimum number of valid pages is chosen, valid pages are copied from the victim block to a free block, and the victim block is erased. Then, the worst-case scenario occurs when all valid pages are evenly distributed to all flash memory blocks in a cluster ϕ_k. For each chip with over-provisioning, the number of valid pages needed to be copied is upper-bounded by $\lceil \lambda \cdot P \rceil$, where λ is a ratio of logical address space to physical address space and P is the number of pages in a block. Then, the number of reclaimed pages in a chip after the GC operation is lower-bounded by $\alpha = P - \lceil \lambda \cdot P \rceil$, yielding $\alpha \cdot |\phi_k|$ reclaimed pages in a cluster ϕ_k in total. The period of a GC task depends on how fast its corresponding τ_i consumes reclaimed pages. For a task τ_i allocated to ϕ_k, if $w_i = \alpha \cdot |\phi_k|$, then GC needs to reclaim a block every write period T_i^w, so the period for the GC task is the same as the write period. If w_i is more than $\alpha \cdot |\phi_k|$, then the GC task needs to guarantee that *at least* two blocks are reclaimed every T_i^w, yielding a shorter GC period than T_i^w. If w_i is less than half of $\alpha \cdot |\phi_k|$, then the GC task only needs to reclaim a block every $2 \cdot T_i^w$. Based on this, a GC task with the worst-case execution time C_i^g and period T_i^g corresponding to task τ_i in ϕ_k is presented as

$$C_i^g = \lceil \lambda \cdot P \rceil \cdot (t_r + t_w) + t_e, \tag{3}$$

$$T_i^g(\phi_k) = \begin{cases} T_i^w / \lceil \frac{w_i}{\alpha \cdot |\phi_k|} \rceil, & \text{if } w_i > \alpha \cdot |\phi_k|, \\ T_i^w \cdot \lfloor \frac{\alpha \cdot |\phi_k|}{w_i} \rfloor, & \text{otherwise,} \end{cases} \tag{4}$$

where t_e is the time to erase a block.

As shown in the case study in Section III, the period of a GC task depends on not only the number of write pages of τ_i but also the number of chips in a cluster ϕ_k; the less the number of write pages with more chips in a cluster, the longer the period of a GC task.

Data transfer delay. Data transfer delay may occur because chips on the same channel share the data bus. The worst-case data transfer delays via channel for read and write requests for τ_i allocated to ϕ_k (denoted as $D_i^r(\phi_k)$ and $D_i^w(\phi_k)$, respectively) are presented as

$$D_i^r(\phi_k) = r_i \cdot (t_d \cdot (m_c - (\lceil \frac{|\phi_k|}{m_c} \rceil + 1))), \tag{5}$$

$$D_i^w(\phi_k) = w_i \cdot (t_d \cdot (m_c - (\lceil \frac{|\phi_k|}{m_c} \rceil + 1))), \tag{6}$$

where t_d is the time to transfer a page from a page register to a controller DRAM and m_c is the number of chips on a channel.

Let $U_i(\phi_k)$ be the utilization of τ_i when allocated to ϕ_k, and it is

$$U_i(\phi_k) = \frac{C_i^r + D_i^r(\phi_k)}{T_i^r} + \frac{C_i^w + D_i^w(\phi_k)}{T_i^w} + \frac{C_i^g}{T_i^g(\phi_k)}. \tag{7}$$

2) Non real-time I/O job: Each non-RT I/O job $J_j \in J$ is modeled as the aperiodic job model [11], and it can be specified as $J_j = (r_j, w_j)$, where it performs r_j page reads and w_j page writes with no deadline. The arrival time of each non-RT job is unknown.

The WCETs (C_j^r and C_j^w) and the worst-case data transfer delays ($D_j^r(\phi_k)$ and $D_j^w(\phi_k)$) for read and write requests of J_j are presented as the same as those of the RT I/O task. The GC operation for non-RT I/O jobs will be discussed in Section IV-D.

B. Schedulability analysis

We now provide a schedulability test to verify whether all *real-time* tasks satisfy their timing constraints under a task-to-cluster allocation Λ. For now, we do not consider the schedule of non-RT jobs, which will be discussed later in Section IV-D. We consider non-preemptive

EDF scheduling with stack resource policy (SRP) [12]. In EDF scheduling, each I/O request of a RT task and its corresponding GC task are assigned their priorities according to their absolute deadlines: the earlier the deadline of a request, the higher its priority. For accessing shared resources, SRP guarantees that each request will not be blocked for the duration of more than one critical section of a lower priority request. Since all flash operations are non-preemptive, and erase operation is the most time-consuming one, the schedulability of RT I/O tasks with a task-to-cluster assignment Λ can be verified by the following theorem.

Theorem 1: For a given RT task set τ, cluster configuration Φ, and task-to-cluster assignment Λ, the RT task set τ is schedulable under EDF scheduling with SRP on a NAND flash storage system π, if for each cluster $\phi_k \in \Phi$, the following inequality holds:

$$\frac{t_e}{min(T)} + \sum_{\forall \tau_i \in \Lambda_{\phi_k}} U_i(\phi_k) \leq 1, \tag{8}$$

where $min(T) = \min_{\tau_i \in \Lambda_{\phi_k}}(T_i^r, T_i^w, T_i^g)$ and Λ_{ϕ_k} is the set of tasks allocated to cluster ϕ_k.

Proof: By [13], all real-time tasks scheduled by EDF are schedulable if their total utilization is less than or equal to 1. $\frac{t_e}{min(T)}$ is the ratio of utilization sacrificed due to the blocking of non-preemptive operations. In the worst-case scenario, a real-time task might be blocked by erase operation. The maximum blocking factor is contributed by the task which has the shortest period. Eq. (8) is derived in a similar way as that for Theorem 2 in [12]. ■

C. Task-to-cluster allocation for RT I/O tasks

We now discuss how to determine a cluster configuration and task-to-cluster allocation for RT I/O tasks focusing on G2. The task-to-cluster allocation problem is NP-hard, since finding a feasible mapping from the RT I/O tasks to the clusters is equivalent to the bin-packing problem which is known to be NP-hard in the strong sense [14]. Thus, we need to look for heuristics. Focusing on feasibility, a typical task-to-cluster allocation is to apply variants of well-known bin-packing algorithms, including Best-Fit Decreasing (BFD) and Worst-Fit Decreasing (WFD) [15]. These algorithms process tasks one-by-one in the order of non-increasing utilization, assigning each task to a cluster according to the heuristic function that determines how to break ties if there are multiple clusters that can accommodate the new task. Whether a cluster can accommodate each task or not is determined by the schedulability test in Theorem 1. In addition, feasibility also depends on a cluster configuration (i.e., the total number of clusters and the number of chips in each cluster).

Considering this, we propose a new task-to-cluster allocation algorithm together with a cluster configuration, called Clustering-BFD, as presented in Algorithm 1. At first, we set an initial cluster configuration as assigning one chip per each cluster (lines 2–3). RT tasks are sorted in non-increasing order of their utilizations (Line5). Clustering-BFD then seeks to allocate each RT task τ_i to a cluster in every iteration step (Lines 6–19). In the i-th iteration step with the current cluster configuration, the algorithm finds a subset of feasible clusters on which the allocation of τ_i can preserve feasibility by the schedulability test in Theorem 1 (Line 7). Let $U_{\phi_c}(\Lambda_{\pi_c})$ be the total utilization of tasks allocated to ϕ_c, and it is calculated as

$$U_{\phi_c}(\Lambda_{\pi_c}) = \frac{t_e}{min(T)} + \sum_{\forall \tau_i \in \Lambda_{\phi_c}} U_i(\phi_c). \tag{9}$$

If there is any feasible cluster, then Clustering-BFD assigns τ_i to the cluster with the *highest total utilization* similar to BFD (Lines 17–18). If there is no feasible cluster, then Clustering-BFD considers a new cluster configuration by merging two clusters into one (Lines 12–14). Among all possible combinations of merging two clusters,

Algorithm 1 Clustering-BFD (τ, π)

```
1: Φ ← ∅
2: for π_c ∈ {π_1, ..., π_m} do
3:    φ_c ← π_c, Φ ← Φ ∪ φ_c, Λ_{φ_c} ← ∅
4: end for
5: τ' ← Sort(τ by non-increasing U_i(φ_c))
6: for τ_i ∈ τ' do
7:    Φ' ← {φ_c ∈ Φ : feasible-assignment(Λ_{π_c} ∪ τ_i) by Eq. (8)}
8:    if Φ' = ∅ then
9:       if |Φ| = 1 then
10:         return Failed to assign
11:      end if
12:      Φ* ← {φ_x ← merge(φ_c1,φ_c2) : φ_c1, φ_c2 ∈ Φ}
13:      φ_y ← arg min_{φ_x ∈ Φ*} U_{φ_x}(Λ_{φ_x})
14:      Φ ← (Φ \ {φ_c1, φ_c2}) ∪ φ_y
15:      go back to line 6 and repeat for τ_i
16:   end if
17:   φ_k ← arg max_{φ_c ∈ Φ'} U_{φ_c}(Λ_{φ_c} ∪ τ_i)
18:   Λ_{φ_k} ← Λ_{φ_k} ∪ τ_i
19: end for
20: return Λ, Φ
```

the algorithm selects the one with the *lowest total utilization* to accommodate as many tasks as possible. Clustering-BFD then tries to allocate τ_i with the new cluster configuration (Line 15).

Note that Clustering-BFD considers the feasibility of RT tasks and the schedule of non-RT jobs together in task-to-cluster assignment. Focusing on the feasibility of RT tasks, Clustering-BFD tries to pack as many tasks as possible on one cluster while keeping the other clusters empty to accommodate other unassigned tasks, similarly with BFD. In general, BFD have shown better feasibility than other bin-packing algorithms, such as WFD [16]. Focusing on the schedule of non-RT jobs, Clustering-BFD sets an initial cluster configuration to have the maximum number of clusters and merge clusters only when necessary. In this way, Clustering-BFD tries to reserve as many clusters as possible to schedule non-RT jobs runtime in parallel, maximizing the bandwidth of non-RT jobs.

Runtime complexity. Algorithm 1 first sorts the RT I/O tasks with $O(n \cdot logn)$ complexity. Then, the algorithm allocates each task to a feasible cluster by starting from the cluster with the highest total utilization with $O(n \cdot m)$. For each number of clusters k in $1 \leq k \leq m$, the algorithm considers all combinations of two clusters to merge, and the total number of combinations $\forall k$ in $1 \leq k \leq m$ is $O(m^3)$. Thus, the total complexity is $O(\max(n \cdot logn, n \cdot m, m^3))$.

D. Schedule generation for non-RT I/O jobs

Building upon the task-to-cluster allocation (derived by Clustering-BFD) for RT I/O tasks, we now discuss how to schedule non-RT I/O jobs so as to achieve G1 and G2. Focusing on G2, the execution of non-RT aperiodic jobs must not violate timing guarantees given to RT tasks. One simple approach to address G2 is to schedule non-RT jobs at lowest priority in each cluster, where non-RT jobs are executed only at times when there is no real-time I/O request ready for execution. Such a background scheduling is fairly straightforward, however, non real-time tasks may suffer from very long latency when the utilization of RT tasks in each cluster is high, resulting in low bandwidth of non-RT jobs (hard to achieve G1).

To address this, we use the concept of *server*, that is, a periodic task whose purpose is to service aperiodic requests as soon as possible. Like any periodic task, a server is characterized by a *fixed* utilization and generates a sequence of jobs. The server is scheduled as any other RT tasks, and once active, it serves non-RT aperiodic requests within the limit of its utilization. In particular, we use the *Total Bandwidth Server* (TBS), a well-known server mechanism designed for EDF

scheduling [17]. Each cluster has its own TBS to schedule non-RT jobs. Now, we discuss how to utilize the TBS for each cluster and how to assign non-RT jobs to clusters.

How to utilize the TBS for each cluster. A TBS $\tau_s(\phi_k)$ in a cluster ϕ_k is specified by its utilization factor $U_s(\phi_k)$ (i.e., its bandwidth). The bandwidth $U_s(\phi_k)$ is determined as the remaining utilization on ϕ_k after allocating RT tasks, and it is calculated as

$$U_s(\phi_k) = 1 - U_{\phi_k}(\Lambda_{\pi_k}). \tag{10}$$

Then, each time a non-RT job arrives, the total bandwidth of the server is assigned to the job. Suppose a non-RT job $J_j = (r_j, w_j)$ is arrived at time $t = a_j$. Then, it receives a deadline

$$d_j = \max(a_j, d_{j-1}) + \frac{C_j}{U_s(\phi_k)}, \tag{11}$$

where d_{j-1} is the deadline of previous non-RT job scheduled on ϕ_k ($d_0 = 0$ by definition) and C_j is the execution time of the job (presented in Eq. (12)). A TBS $\tau_s(\phi_k)$ has its corresponding GC task to handle the GC operation for non-RT jobs assigned to ϕ_k. Thus, every non-RT job need to take into account the time for the GC operation C_j^g (presented in Eq. (3)) when calculating C_j. The worst-case execution time C_j of J_j is presented as

$$C_j = C_j^r + D_j^r(\phi_k) + C_j^w + D_j^w(\phi_k) + B \cdot C_j^g, \tag{12}$$

where

$$B = \begin{cases} \lceil \frac{w_j + w(\phi_k)}{\alpha \cdot |\phi_k|} \rceil, & \text{if } w_j + w(\phi_k) > \alpha \cdot |\phi_k|, \\ 0, & \text{otherwise,} \end{cases}$$

where B is the number of blocks to be reclaimed and $w(\phi_k)$ is the the number of pages written by $\tau_s(\phi_k)$ since its previous GC operation.

With the TBS in each cluster, non-RT jobs can be scheduled with the guaranteed bandwidth without violating any timing constraints of RT tasks, as stated in the following theorem.

Theorem 2: Given a set of RT tasks τ with cluster configuration Φ and task-to-cluster assignment Λ, and a collection of non-RT jobs J with a TBS $\tau_s(\phi_k)$ for each cluster ϕ_k, the whole set is schedulable under EDF scheduling with SRP on a NAND flash storage system π, if for each cluster $\phi_k \in \Phi$, the following inequality holds:

$$U_{\phi_k}(\Lambda_{\pi_k}) + U_s(\phi_k) \leq 1. \tag{13}$$

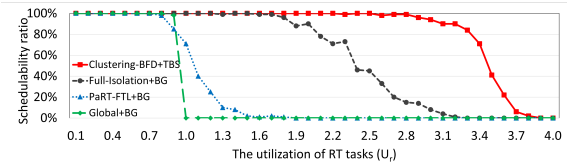
Proof: According to Lemma 2 in [17], the actual utilization of the non-RT jobs scheduled by $\tau_s(\phi_k)$ does not exceed $U_s(\phi_k)$ in any interval of time. Then, combined with Theorem 1, the theorem holds as for Theorem 3 in [17]. ■

How to assign non real-time jobs to clusters. We now discuss how to assign non-RT jobs to clusters. Upon arrival of a non-RT job J_j, it is allocated to the cluster on which it receives the earliest deadline d_j. Then, it is scheduled with RT tasks in the cluster according to EDF scheduling. In this way, we can provide the maximum bandwidth to non-RT jobs under a cluster configuration without compromising the schedulability of RT tasks.

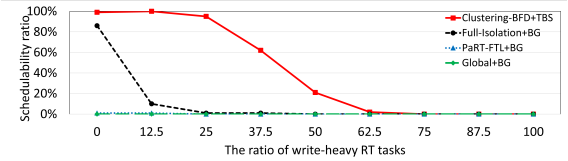
V. Evaluation

We demonstrate the capability of the proposed task-to-cluster allocation and scheduling algorithms. We use two metrics: *schedulability ratio* for RT tasks and *average bandwidth* for non-RT jobs. The schedulability ratio is defined to be the percentage of schedulable RT task sets of the total number of generated RT task sets. The average read/write bandwidth is defined as the number of read/write requests performed in one second (i.e., IOPS).

We first show the extensive simulation results for synthetic task sets with randomly generated parameters. We then show the results for a case study. We compare the following four different approaches:



(a) Schedulability ratio with different utilizations of RT tasks



(b) Schedulability ratio with different ratios of write-heavy tasks

Fig. 2: Schedulability ratio of RT task sets

- Global+BG: a single cluster of size m with background execution of non-RT jobs (i.e., the shared approach),
- Full-Isolation+BG: m clusters each of size one with background execution of non-RT jobs (i.e., the fully-isolated approach),
- PaRT-FTL+BG: partitioned read and write chip sets with background execution of non-RT jobs, and
- Clustering-BFD+TBS: our cluster configuration and task-to-cluster allocation by Clustering-BFD with a TBS per cluster.

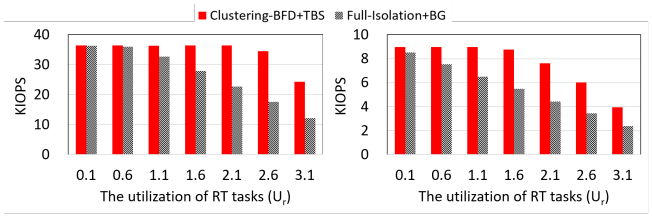
To check the schedulability of RT tasks, all approaches except PaRT-FTL+BG use the schedulability analsysis presented in Theorem 1, while PaRT-FTL+BG uses its own presented in [4]. To handle non-RT jobs, Clustering-BFD+TBS uses a TBS per cluster, while the others use background execution, where non-RT jobs are executed only when there is no real-time I/O request ready for execution.

A. Extensive simulations

Task set generation. We generate RT tasks by using the UUniFast algorithm [18], which has been widely used for the generation of RT task sets. We generate 4,000 RT task sets in total while varying their total utilization of RT tasks from 0.1 to 4.0 with an incremental step of 0.1. Given the total utilization (U_r) for a RT task set and the number of RT tasks as 8, each task is generated as follows. The utilization U_i of each task τ_i is randomly generated such that $\sum U_i = U_r$. T_i^r and T_i^w are uniformly chosen in [100, 500]ms, and C_i^r and C_i^w are determined as satisfying $U_i = \frac{C_i^r}{T_i^r} + \frac{C_i^w}{T_i^w}$.

Schedulability. Fig. 2a compares the percentage of schedulable task sets by four approaches when the number of flash chips is 16 with the parameter shown in Table I.[3] Clustering-BFD+TBS exhibits high capability in finding schedulable task sets in that Clustering-BFD+TBS finds 43%, 225%, and 278% more schedulable task sets than Full-Isolation+BG, PaRT-FTL+BG, and Global+BG, respectively. The performance gap between Clustering-BFD+TBS and the others becomes larger as U_r increases. Under Global+BG, all RT tasks in a task set share a single cluster, so Global+BG cannot find any schedulable task sets when $U_r > 1$. Under PaRT-FTL+BG, there are two clusters (one for read requests and another for write requests), so PaRT-FTL+BG cannot find any schedulable task sets when $U_r > 2$. Although Full-Isolation+BG shows better performance than Global+BG and PaRT-FTL+BG by distributing/isolating RT tasks among 4 clusters, GC tasks exhibit high utilization, making it harder to find schedulable task sets as U_r increases. On the other hand, using Clustering-BFD+TBS, 94% of the task sets are schedulable at $U_r = 3.0$, while only 8% of the task sets are schedulable by Full-Isolation+BG. We interpret such a gap as the benefit of flexible chip

[3]Note that, to compare the schedulability ratio, we only consider RT tasks because execution of non-RT jobs does not affect the schedulability of RT tasks with background scheduling and TBS.



(a)Average bandwidth of read-only non-RT jobs (b)Average bandwidth of write-only non-RT jobs

Fig. 3: Average bandwidth of non-RT jobs

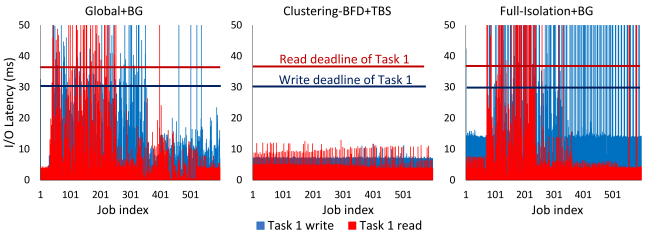


Fig. 4: Experimental results for a case study

clustering and task allocation by trading-off the number of clusters (total resource capacity) and GC overhead.

We also varied the ratio of write-heavy RT tasks to identify their effects on schedulability. We consider tasks with $\frac{C_i^w}{T_i^w} \geq 0.5$ as write-heavy tasks. We generate 900 additional RT task sets in total while varying the ratio of write-heavy RT tasks from 0 to 100% when the number of tasks is 8 as shown in Fig. 2b. Clustering-BFD+TBS is shown to outperform the other approaches for all ratios. This is because Clustering-BFD+TBS effectively reduces GC overhead by merging clusters when the ratio of write-heavy RT tasks increases.

Bandwidth. We now compare the average bandwidth of aperiodic non-RT jobs. With the generated RT task sets, we only consider the schedulable task sets by both Clustering-BFD+TBS and Full-Isolation+BG[4] and simulate aperiodic arrivals of non-RT jobs. In simulation, read-only and write-only non-RT jobs request 20 read and 5 write pages, respectively with their period uniformly chosen in [100, 1000]μs. Fig. 3 compares the average bandwidth of non-RT jobs by Clustering-BFD+TBS and Full-Isolation+BG while varying U_r from 0.1 to 3.1. Clustering-BFD+TBS is shown to outperform Full-Isolation+BG for all U_r values. The performance gap between Clustering-BFD+TBS and Full-Isolation+BG becomes larger as the utilization of RT tasks increases. For example, when $U_r = 2.6$, Clustering-BFD+TBS handles 16,950 (2,590) more read (write) pages than Full-Isolation+BG in one second. As U_r increases, less resources become available for non-RT jobs. Then, under Full-Isolation+BG, non-RT jobs have less chance to be executed since it is only available at times when there is no RT task ready for execution. However, Clustering-BFD+TBS uses the TBS and executes non-RT jobs together with RT tasks as soon as possible within the limit of TBS's utilization.

B. Case study

In addition to our simulation study, we implement the proposed techniques in the Linux operating system with a DRAM-emulated SSD [10] that models a real-world SSD. We then carry out a case study using RT tasks. Our SSD has four NAND flash chips which are configured with the parameters shown in Table I. Fig. 4 shows the experimental results. We consider four RT tasks and one non-RT job: $\tau_1 = [(r_1 = 40, T_1^r = 36), (w_1 = 24, T_1^w = 30)]$, $\tau_2 = \tau_3 = \tau_4 = [(80, 36), (12, 130)]$, and $J_1 = (r_1 = 0, w_1 = 128)$ every $12.5ms$. Under Clustering-BFD+TBS, no deadline is missed for RT tasks, while a considerable amount of read and write requests for RT tasks miss deadlines under Global+BG and Full-Isolation+BG.[5] For the average bandwidth of the non-RT job, Clustering-BFD+TBS achieves 950 (IOPS), while Global+BG and Full-Isolation+BG show 345 and 202 (IOPS), respectively.

VI. Conclusion and Discussion

In this paper, we investigated the effect of cluster-based resource isolation on schedulability and bandwidth in real-time NAND flash storage systems. By taking a trade-off between flash chip sharing and I/O workload isolation, we developed dynamic chip clustering, task-to-cluster allocation, and co-scheduling algorithms to achieve high bandwidth for non real-time I/Os while guaranteeing all timing constraints of real-time I/Os. Our evaluation results demonstrated that the proposed dynamic cluster-based resource isolation and scheduling approach made a substantial improvement of both schedulability and bandwidth over existing isolation approaches. In future, we would like to extend our approach to handle wear-leveling issues.

[4]This is because PaRT-FTL+BG and Global+BG find few schedulable task sets, which is insufficient for comparison.

[5]We only show the results of τ_1 due to space limitation.

Acknowledgement

This work was supported in part by the National Research Foundation of Korea (NRF) grant (2020R1F1A1076058, 2018R1A5A1060031, 2017M3A9G8084463) and the Institute for Information and communications Technology Promotion (IITP) grant (2014-0-00065, Resilient Cyber-Physical Systems Research) funded by the Korea government (MSIT), as well as the DGIST R&D Program of MSIT (20-CoE-IT-01).

References

[1] S. Choudhuri and T. Givargis, "Deterministic service guarantees for NAND flash using partial block cleaning," in *CODES+ISSS*, 2008.

[2] Z. Qin, Y. Wang, D. Liu, and Z. Shao, "Real-time flash translation layer for NAND flash memory storage systems," in *RTAS*, 2012.

[3] Q. Zhang, X. Li, L. Wang, T. Zhang, Y. Wang, and Z. Shao, "Optimizing deterministic garbage collection in NAND flash storage systems," in *RTAS*, 2015.

[4] K. Missimer and R. West, "Partitioned real-time NAND flash storage," in *RTSS*, 2018.

[5] D.-W. Chang, H.-H. Chen, and W.-J. Su, "VSSD: Performance isolation in a solid-state drive," *ACM Transactions on Design Automation of Electronic Systems*, vol. 20, no. 4, pp. 51:1–51:33, 2015.

[6] J. Huang, A. Badam, L. Caulfield, S. Nath, S. Sengupta, B. Sharma, and M. K. Qureshi, "FlashBlox: Achieving both performance isolation and uniform lifetime for virtualized ssds," in *FAST*, 2017.

[7] J. Liu, F. Wang, and D. Feng, "CostPI: Cost-effective performance isolation for shared NVMe SSDs," in *ICPP*, 2019.

[8] M. Kwon, D. Gouk, C. Lee, B. Kim, J. Hwang, and M. Jung, "DC-Store: Eliminating noisy neighbor containers using deterministic I/O performance and resource isolation," in *FAST*, 2020.

[9] R. Liu, X. Chen, Y. Tan, R. Zhang, L. Liang, and D. Liu, "SSDKeeper: Self-adapting channel allocation to improve the performance of SSD devices," in *IPDPS*, 2020.

[10] S. Lee, M. Liu, S. Jun, S. Xu, J. Kim, and Arvind, "Application-Managed Flash," in *FAST*, 2016, pp. 339–353.

[11] B. Sprunt, L. Sha, and J. Lehoczky, "Aperiodic task scheduling for hard real-time systems," *Real-Time Systems*, vol. 1, no. 1, pp. 27–60, 1989.

[12] T. Baker, "A stack-based resource allocation policy for realtime processes," in *RTSS*, 1990.

[13] C. Liu and J. Layland, "Scheduling algorithms for multi-programming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[14] T. A. AlEnawy and H. Aydin, "Energy-aware task allocation for rate monotonic scheduling," in *RTAS*, 2005.

[15] E. G. Coffman, G. Galambos, S. Martello, and D. Vigo, "Bin packing approximation algorithms: Combinatorial analysis," in *Handbook of combinatorial optimization*, 1999, pp. 151–207.

[16] H. Aydin and Q. Yang, "Energy-aware partitioning for multiprocessor real-time systems," in *IPDPS*, 2003.

[17] M. Spuri and G. C. Buttazzo, "Scheduling aperiodic tasks in dynamic priority systems," *Real-Time Systems*, vol. 10, no. 2, pp. 179–210, 1996.

[18] E. Bini and G. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems*, vol. 30, no. 1-2, pp. 129–154, May 2005.