Enlightening the I/O Path: A Holistic Approach for Application Performance

Sangwook Kim, *Apposha and Sungkyunkwan University;* Hwanju Kim, *Sungkyunkwan University and Dell EMC;* Joonwon Lee and Jinkyu Jeong, *Sungkyunkwan University*

https://www.usenix.org/conference/fast17/technical-sessions/presentation/kim-sangwook

This paper is included in the Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17).

February 27–March 2, 2017 • Santa Clara, CA, USA

ISBN 978-1-931971-36-2

Enlightening the I/O Path: A Holistic Approach for Application Performance

Sangwook Kim[†§], Hwanju Kim[§*], Joonwon Lee[§], Jinkyu Jeong[§]
[†]*Apposha,* [§]*Sungkyunkwan University*
sangwook@apposha.io, hwandori@gmail.com, joonwon@skku.edu, jinkyu@skku.edu

Abstract

In data-intensive applications, such as databases and key-value stores, reducing the request handling latency is important for providing better data services. In such applications, I/O-intensive background tasks, such as checkpointing, are the major culprit in worsening the latency due to the contention in shared I/O stack and storage. To minimize the contention, properly prioritizing I/Os is crucial but the effectiveness of existing approaches is limited for two reasons. First, statically deciding the priority of an I/O is insufficient since high-priority tasks can wait for low-priority I/Os due to *I/O priority inversion*. Second, *multiple independent layers* in modern storage stacks are not holistically considered by existing approaches which thereby fail to effectively prioritize I/Os throughout the I/O path.

In this paper, we propose a *request-centric I/O prioritization* that dynamically detects and prioritizes I/Os delaying request handling at all layers in the I/O path. The proposed scheme is implemented on Linux and is evaluated with three applications, PostgreSQL, MongoDB, and Redis. The evaluation results show that our scheme achieves up to 53% better request throughput and 42× better 99^{th} percentile request latency (84 ms vs. 3581 ms), compared to the default configuration in Linux.

1 Introduction

In data-intensive applications, such as databases and key-value stores, the response time of a client's *request* (e.g., key-value PUT/GET) determines the level of application performance a client perceives. In this regard, many applications are structured to have two types of tasks: *foreground tasks*, which perform essential work for handling requests, and *background tasks*, which conduct I/O-intensive internal activities, such as checkpointing [31, 14, 3], backup [11, 9, 24, 18], compaction [21, 34, 38, 13], and contents filling [36]. The main reason for this form of structuring is to reduce request handling latency by taking off the internal activities from the critical path of request execution. However, background tasks are still interfering foreground tasks since they inherently share the I/O path in a storage stack. For example, background checkpointing in relational database has known to hinder delivering low and predictable transaction latency, but the database and operating system (OS) communities have no reasonable solution despite their collaborative efforts [12].

The best direction to resolve this problem in OS is to provide an interface to specify I/O priority for a differentiated storage I/O service. Based on this form of OS support, two important issues should be addressed: 1) deciding which I/O should be given high priority, and 2) effectively prioritizing high priority I/Os along the I/O path. The conventional approaches for classifying I/O priorities are I/O-centric and task-centric. These approaches statically assign high priority to a specific type of I/O (e.g., synchronous I/O [30, 2, 32]) and to I/Os issued by a specific task (e.g., interactive task [42]). This I/O priority is typically enforced at the block-level scheduler or at several layers [42, 43].

The previous approaches, however, have limitations in achieving high and consistent application performance. First, they do not holistically consider *multiple independent layers* including caching, file system, block, and device layers in modern storage stacks. Missing I/O prioritization in any of the layers can degrade application performance due to delayed I/O processing in such layers (Section 2.1). Second, they do not address the *I/O priority inversion* problem caused by runtime dependencies among concurrent tasks and I/Os. Similar to the priority inversion problem in CPU scheduling [35], low-priority I/Os (e.g., asynchronous I/Os and background I/Os) sometimes can significantly delay the progress of a high-priority foreground task, thereby inverting I/O priority (Section 2.2). More seriously, I/O priority inversions can occur across different layers in a storage stack.

[*]Currently at Dell EMC

Due to these limitations, existing approaches are limited in effectively prioritizing I/Os and result in suboptimal performance.

In this paper, we introduce a *request-centric I/O prioritization* (or RCP for short) that holistically prioritizes *critical I/Os* (i.e., performance-critical I/Os) over non-critical ones along the I/O path; we define a critical I/O as an I/O in the critical path of request handling regardless of its I/O type and submitting task. Specifically, our scheme identifies foreground tasks by exposing an API and gives critical I/O priority to the foreground tasks (Section 4.1). To handle I/O priority inversions, critical I/O priority is dynamically assigned to a task or an outstanding I/O on which a foreground task depends to make progress (Section 4.2). Then, each layer in the I/O path is adapted to prioritize the critical I/Os and to support I/O priority inheritance (Section 4.3). We also resolve an I/O priority inversion caused by a transitive dependency, which is a chain of dependencies involving multiple tasks (Section 4.4).

As a prototype implementation, we enlightened the I/O path of the Linux kernel. Specifically, in order to accurately identify I/O criticality, we implemented the I/O priority inheritance to blocking-based synchronization methods (e.g., mutex) in the Linux kernel. Based on the identified I/O criticality, we made the Linux caching layer, ext4 file system, and the block layer understand and enforce I/O criticality. Based on the prototype, we evaluated our scheme using PostgreSQL [10], MongoDB [8], and Redis [22] with TPC-C [17] and YCSB [25] benchmarks. The evaluation results have shown that our scheme effectively improves request throughput and tail latency (99.9th percentile latency) by about 7–53% and 4.4–20×, respectively, without penalizing background tasks, compared to the default configuration in Linux.

2 Motivation and Related Work

Background I/O-intensive tasks, such as checkpointing and compaction, are problematic for achieving the high degree of application performance. We illustrate this problem by running the YCSB [25] benchmark against MongoDB [8] document store on a Linux platform with two HDDs each of which is allocated for data and journal, respectively; see Section 7 for the details. As shown in Figure 1, application performance represented as operations per second is highly fluctuated with the CFQ [2], the default I/O scheduler in Linux, mainly due to the contention incurred by periodic checkpointing (60 seconds by default) [1]. Assigning low priority (idle-priority [7] in CFQ) to the checkpoint task using the existing interface,

[1]The interference is not exactly periodic because the checkpointing occurs 60 seconds after the completion of the previous checkpointing.

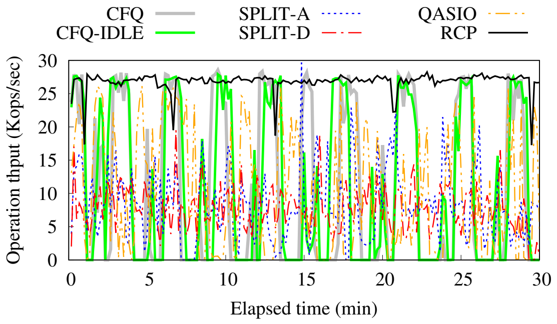

Figure 1: **Limitation of existing approaches.** *Update-heavy workload in YCSB is executed against MongoDB.*

denoted as CFQ-IDLE, is also ineffective in alleviating the performance interference. Moreover, Split-AFQ (SPLIT-A) and Split-Deadline (SPLIT-D), the state-of-the-art cross-layer I/O schedulers [43], also cannot provide consistent application performance even though the checkpoint thread is given lower priority than foreground ones; adjusting the parameters in the SPLIT-A/D (e.g., `fsync()` deadline) did not show any noticeable improvement. Likewise, QASIO [32], which tries to eliminate I/O priority inversions, also shows frequent drops in application performance.

The root causes of this undesirable result in the existing I/O prioritization schemes are twofold. First, the existing schemes do not fully consider multiple independent layers including caching, file system, and block layers in modern storage stacks. Prioritizing I/Os only in one or two layers of the I/O path cannot achieve proper I/O prioritization for foreground tasks. Second and more importantly, the existing schemes do not address the *I/O priority inversion* problem caused by runtime dependencies among concurrent tasks and I/Os. I/O priority inversions can occur across different I/O stages in multiple layers due to transitive dependencies. As shown by RCP in Figure 1, the cliffs in application throughput can be significantly mitigated if the two challenges are addressed. In the rest of this section, we detail the two challenges from the perspective of application performance and discuss existing approaches.

2.1 Multiple Independent Layers

In modern OSes, a storage I/O stack is comprised of multiple and independent layers (Figure 2). A caching layer first serves reads if it has the requested block and it buffers writes until they are issued to a lower layer. If a read miss occurs or a writeback of buffered writes is required, a file system generates block I/O requests and passes them to a block layer. Then, the block layer admits an I/O request into a block-level queue and schedules a queued I/O request to dispatch to a storage device. Finally, a storage device admits an I/O command received from a host into a device-internal queue and

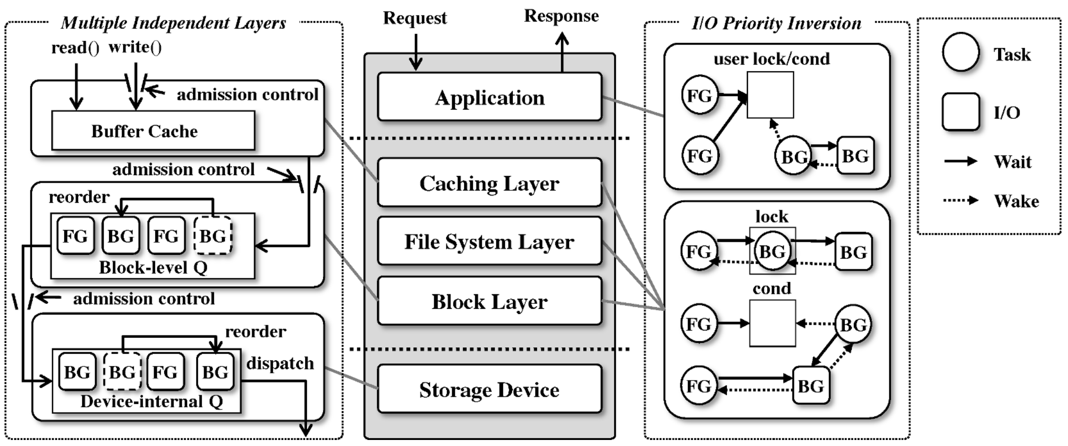

Figure 2: **Challenges in modern storage stacks.** *FG in a circle and a box means a foreground task and an I/O, respectively. Likewise, BG in a circle and a box indicates a background task and an I/O each.*

schedules a queued command to an internal device controller. Though this form of layering with abstraction is an essential part in computer systems for interoperability and independent innovations across the layers, it makes effective I/O prioritization strenuous because each layer has independent policy with limited information.

In the caching layer, priority-agnostic admission control can harm the application performance. Modern OSes, such as Linux [26] and Windows [40], control the admission of buffered writes based on the dirty ratio of available memory for ensuring system stability. However, since a single system-wide dirty ratio is applied to all tasks, a foreground task can be blocked even if most dirty pages are made by background tasks. Giving higher I/O priority to the foreground task is ineffective unless I/O priority is applied to the dirty throttling.

A file system forces specific ordering of writes for consistent updates [39, 41, 19], thereby complicating effective I/O prioritization. For example, ext4 file system, which is the default for most Linux distributions, entangles buffered writes into a single, global, compound transaction that needs to be durable atomically. Since a file system transaction contains dirty pages from any file written by any task, a foreground task calling `fsync()` should wait for the completion of I/Os issued not only by itself, but also by a background task.

In the block layer, many modern block I/O schedulers already reflect the priority of I/Os in their scheduling. However, priority-agnostic admission control can also degrade the application performance. Typically, the size of a block-level queue is limited to restrict memory usage and to control disk congestion [42]. In this case, a burst of background I/Os can significantly delay the processing of a foreground I/O by quickly filling the available slots in a block-level queue. The existing priority schedulers cannot help to mitigate this problem because they have no control of submitted I/Os that are not yet entered the block-level queues.

Even after a foreground I/O becomes ready to dispatch to a storage device, the processing of the foreground I/O can be further prolonged. This is because the size of a device-internal queue (e.g., NCQ [16]) is also limited and a device firmware reorders I/O commands based on the internal geometry of storage media for improving device throughput [44, 20]. Hence, a foreground I/O can be staged because the queue slots are busy handling background I/Os. Furthermore, even if a foreground I/O can be dispatched to the device, the device-internal scheduling can delay the processing of the I/O because of its internal scheduling policy.

2.2 I/O Priority Inversion

The most straightforward way of improving application performance in the existence of background tasks would be to prioritize foreground I/Os over background ones and all I/O layers respect their priorities. However, this simple prioritization is insufficient since I/O priority inversions caused by runtime dependencies can delays the execution of a foreground task (Figure 2). Similar to the priority inversion problem in CPU scheduling [35], I/O priority inversions are problematic because the processing of a background I/O on which a foreground task depend can be arbitrarily delayed by other background I/Os.

Two types of dependencies cause I/O priority inversions: a *task dependency* and an *I/O dependency*. The task dependency occurs when two tasks interact with each other via synchronization primitives, such as a lock and a condition variable. The dependency caused by a lock complicates effective I/O prioritization because a background task can be blocked waiting for an I/O within a critical section that a foreground task needs to enter. For instance, a foreground task attempting to write a file can be blocked on an inode mutex if the mutex is already held by a background task concurrently writing to the different part of that file. Likewise, the dependency caused by a condition variable also cause a similar problem. A foreground task should indirectly wait for the I/Os awaited by a background task that is going to signal the foreground task. For example in Linux ext4, when a foreground task calls `fsync()`, it waits on a specific condition variable which is signaled by jbd2 kernel thread, which could be busy completing journal transactions for background tasks.

Meanwhile, the I/O dependency occurs between a task and an outstanding I/O. Basically, the I/O dependency is generated when a task needs to directly wait for the completion of an ongoing I/O in order to ensure correctness and/or durability. For example, when a foreground task calls `fsync()`, it blocks on the completion of a write I/O that is asynchronously issued by a kernel thread (e.g., pdflush in Linux) for cleaning buffer cache. Once the task and the I/O dependency-induced priority inversions occur, the foreground task should wait for a long time because each layer in the I/O path can arbitrarily prolong the processing of low-priority background I/Os.

Unfortunately, resolving I/O priority inversions is challenging for the following reasons. Firstly, dependency relationships cannot be statically determined since they depend on various runtime conditions, such as execution timing, resource constraint, and client requirement. For example, a foreground task does not always depend on the progress of a kernel thread handling file system transaction since the kernel thread periodically writes out transactions in background [4]. Secondly, dependency occurs in a transitive manner involving multiple concurrent tasks blocked at either synchronization primitives or various I/O stages in multiple layers. We empirically found that a dependency sometimes cascaded in four steps due to the complex interaction between delayed allocation and crash-consistency mechanism in a file system (Section 4.4). Finally, a dependency relationship might not be visible at the kernel-level because of the extensive use of user-level synchronizations (e.g., shared memory mutex) based on kernel-level supports (e.g., Futex [6]) in modern applications.

2.3 Related Work

Table 1 summarizes how the illustrated challenges are addressed (or not) by the existing prioritization schemes. CFQ [2] is a block-level I/O scheduler that supports multiple priority classes (real-time, best-effort, and idle) and priority levels (0 to 7) [7]. However, CFQ prioritizes I/Os only at the block-level queue. It does not consider the I/O priority inversion problem as well as the prioritization at the block queue admission stage.

Redline [42] adapts all I/O layers to limit the interference from background tasks (e.g., virus scanner) for improving responsiveness of interactive applications (e.g., web browser). Redline, however, lacks resolving I/O priority inversions that occur between foreground and background tasks in typical data-intensive applications.

Scheme	**Multiple Independent Layers**			**I/O Priority Inversion**	
	Cache	Filesystem	Block	Kernel	User
CFQ [2]	No	No	Yes	No	No
Redline [42]	Yes	Yes	Yes	No	No
Split [43]	Yes	Yes	Yes	No	No
QASIO [32]	No	No	Yes	Yes	No
sCache [33]	No	No	No	Yes	No
RCP	Yes	Yes	Yes	Yes	Yes

Table 1: **I/O prioritization challenges.** *This table shows whether a specific challenge for effective I/O prioritization is addressed or not in each previous work.*

Recently, Split [43], a cross-layer I/O scheduling framework, is introduced to address the limitation of a single-level I/O schedulers. Basically, Split provides additional hooks to several layers for supporting correct cause mapping, cost estimation, and reordering, in the existence of the file system challenges like delayed allocation and journaling [43]. Based on the proposed framework, Split-AFQ and Split-Deadline have been implemented to prove its effectiveness. Split-AFQ, a priority-based scheduler using the Split framework, schedules write I/Os including `write()` and `fsync()` at the system-call layer to avoid the runtime dependencies caused by file system journaling. Different from conventional deadline schedulers, Split-Deadline provide deadline scheduling of `fsync()` calls. In addition, it aggressively writes-back dirty data in background to make the latency of `fsync()` more deterministic by minimizing the file system transaction entanglement. Though Split itself is a generic I/O scheduling framework, its representative schedulers do not specifically consider the I/O priority inversion problem despite its significance.

On the other side, QASIO [32] considers I/O priority inversions for improving system responsiveness. However, QASIO solely focuses on the kernel-level dependencies to asynchronous writes based on the analysis of the several mobile app scenarios. Furthermore, sCache [33] fully considers I/O priority inversions at the kernel-level in order to effectively utilize non-volatile write caches. Both QASIO and sCache, however, do not consider I/O priority inversions at the user-level. Moreover, they do not address the challenges in the I/O path for effective I/O prioritization.

Though several challenges have been separately addressed in the previous work, we argue that only a holistic approach can deliver consistently high application performance as in Figure 1. This is because the I/O priority inversion problem can be worsened when combined with multiple layers as a dependency transitively occurs across layers. Our scheme (RCP) addresses all the challenges in Table 1 by enlightening the I/O path and resolving the kernel- and user-level I/O priority inversions.

3 Our Approach

In this work, we classify I/Os into two priority levels: (performance) *critical* and *non-critical* I/Os. In particular, we define a critical I/O as an I/O in the critical path of request handling since the response time of a request determines the level of application performance.

The proposed classification is distinguished from conventional I/O classification schemes: *I/O-centric* and *task-centric* classifications (Figure 3). The I/O-centric classification differentiates the priority of each I/O based on its operation type (e.g., synchronous I/Os over asynchronous ones [30, 2, 32]). On the other side, the task-centric classification distinguishes the I/Os based on issuing tasks (e.g., foreground I/Os over background ones [42]). These static classification schemes, however, are inadequate for identifying the (performance) criticality of I/Os. In our request-centric viewpoint, synchronous I/Os (e.g., checkpoint writes) and foreground I/Os (e.g., buffered writes) can be non-critical whereas asynchronous I/Os and background I/Os can sometimes be critical due to the runtime dependencies.

Based on the I/O criticality classification, we introduce a *request-centric I/O prioritization* (or RCP) that identifies critical I/Os and prioritizes them over non-critical ones along the I/O path. This form of two-level I/O prioritization is effective for many cases since background tasks are ubiquitous in practice. For example, according to a Facebook developer: "*... There are always cleaners and compaction threads that need to do I/O, but shouldn't hold off the higher-priority "foreground" I/O. ... Facebook really only needs two (or few) priority levels: low and high.*" [5].

Our goals for realizing RCP are twofold: 1) minimizing application modification for detecting critical I/Os, and 2) processing background tasks in a best-effort manner while minimizing the interference to foreground tasks. The following section describes our design for effectively identifying and enforcing I/O criticality throughout the I/O path.

4 I/O Path Enlightenment

4.1 Enlightenment API

The first step to identifying critical I/Os is to track a set of tasks (i.e., foreground tasks) involved in request handling and this can be done in two ways: system-level and application-guided approaches. A system-level approach infers foreground tasks by using information available in the kernel. Though this approach has the benefit of avoiding application modification, it may induce runtime overhead for the inference and the possibility of misidentification. In contrast, an application-guided approach can accurately identify foreground tasks without runtime overheads at the expense of application modification.

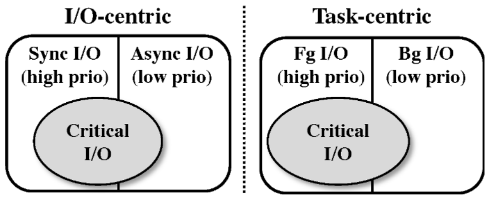

Figure 3: **Comparison with conventional approaches.**

We chose the application-guided approach for accurate detection of foreground tasks without burdening the OS kernel. In particular, we provide an enlightenment interface to user-level so that an application developer (or an administrator if possible) can dynamically set/clear a foreground task based on application-level semantics. The foreground task can be from a short code section to an entire life of a thread depending on where set/clear APIs are called. The simplicity of using APIs makes developers easily prioritize critical I/Os in their applications. We believe the modification cost is also low in practice because typical data-intensive applications already distinguish foreground tasks from background tasks; see Section 6 for the details.

Since the API is solely used for deciding I/O criticality in the OS kernel, a wrong API call does not affect the correct execution of an application. However, API abuse by a malicious or a thoughtless application/tenant may compromise performance isolation among multiple applications/tenants sharing a storage stack. This problem can be solved by integrating RCP to a group-based resource management (e.g., cgroup in Linux [37]). Addressing this issue is out of scope of this paper.

4.2 I/O Priority Inheritance

Based on the enlightenment API, we basically regard a synchronous I/O issued by a foreground task as a critical I/O. This obvious identification, however, is insufficient for detecting all critical I/Os because runtime dependencies cause background I/Os to be awaited by foreground tasks indirectly (Section 2.2). Hence, the next step for critical I/O detection is to handle I/O priority inversions caused by runtime dependencies. To this end, we introduce *I/O priority inheritance* that temporarily gives critical I/O priority to a background task (Section 4.2.1) or a background I/O (Section 4.2.2) on which a foreground task depends to make progress.

4.2.1 Handling Task Dependency

Kernel-level dependency. Resolving the lock-induced dependency has been well-studied in the context of CPU scheduling [35]. Inspired by the previous work, we resolve the lock-induced dependency by inheriting critical I/O priority to a background task when it blocks a foreground task until it leaves a critical section. Specifically, we record an owner task into each lock object (e.g., mutex). When a task is blocked to acquire a lock, the lock owner inherits I/O priority of the waiting task. The inherited priority is revoked when the owner task exits the critical section. This inheritance procedure is repeated until the foreground task acquires the lock. Note that we consider only blocking-based locks since spinning-based locks are not involved with I/O waiting.

Unlike the case of locks, there is no distinct owner in a condition variable at the time of dependency occurs. To deal with the condition varible-induced dependency, we borrow a solution in previous work [27]. In particular, a task is registered as a *helper task* [27] into the corresponding object of a condition variable when the task is going to signal the condition variable. Later, the helper task inherits I/O priority of a task blocked to wait for the condition become true. The inherited I/O priority is revoked when the helper task finally signals the waiting task. In the kernel-level, this approach is viable because only a few condition variables and threads cause the runtime dependencies.

User-level dependency. The key challenge in handling the user-level dependency is that the OS kernel cannot clearly identify a dependency relationship resulted from user-level synchronizations. For user locks, the kernel cannot determine the owner because a user lock variable is located in a shared memory region and modified through atomic operations (e.g., `cmpxchg` in x86) to indicate the lock status. This is an intrinsic optimization to eliminate unnecessary kernel interventions in the uncontended cases. The kernel is involved only when to block or to wake up lock waiters via a system call (e.g., `sys_futex()`). As a consequence, the OS kernel can see only the waiters failed to acquire a user-level lock.

To detect the owner of a user lock, we adjust a user lock primitive (e.g., `pthread_mutex_lock()`) to additionally pass down the information of the owner when a task should block in the kernel due to lock contention. In practice, this can be readily done without modifying existing interfaces; see Section 5 for our implementation. Based on the delivered information, the kernel can properly inherit the I/O priority of a waiting task to a lock-holder task. Note that this modification does not entail the kernel intervention in uncontended cases.

Unlike the condition variable-induced dependency in kernel-level, handling such dependency in user-level is difficult because it is hard to pinpoint helper tasks for condition variables. Modern applications extensively use user-level condition variables for various purposes. For instance, we found over a hundred of user-level condition variables in the source code of MongoDB. Therefore, properly identifying all helper tasks is not trivial even for an application developer.

We adopt an inference technique that identifies a helper task based on usage history of each user-level condition variable. Typically, a background task is dedicated to a specific activity like logging, checkpointing, compaction, and buffer cleaning. Hence, a task signaling a condition is highly likely signal the condition again. Based on this observation, a background task is registered as a helper task when it signals a user-level condition variable. Then, the helper task inherits critical I/O priority of a foreground task when the foreground task needs to block on the user-level condition variable. The helper task is unregistered when it does not signal again for a specific time window.

4.2.2 Handling I/O Dependency

Properly resolving a dependency to an outstanding I/O is complicated because the dependent I/O can be in any stage in the block layer at the time of the dependency occurs. For example, an outstanding I/O can be in admission control stage waiting for the available slots of a block-level queue. Hence, we need to track the status of an ongoing non-critical I/O and appropriately reprioritize it according to the current location when required.

For tracking outstanding non-critical I/Os, we add an `ncio` data structure that stores an I/O descriptor, current location, and the descriptor of a requesting task. An `ncio` object is allocated when an incoming I/O is classified as non-critical at the entrance of the block layer, and inserted to a per-device list indexed by starting sector number. The fields including I/O descriptor and current location in the `ncio` object are properly updated as the corresponding I/O flows along the I/O path. The allocated `ncio` object is freed when the corresponding I/O is reclassified or dispatched to a device.

When a dependency to an ongoing non-critical I/O occurs, the per-device `ncio` list is searched to find the corresponding `ncio` object. Then, the non-critical I/O is reclassified as critical I/O based on the information in the I/O descriptor stored in an `ncio` object if the corresponding `ncio` object is found. In this case, we may need to conduct additional chores according to the current location of the dependent I/O; we present the details in the following subsection.

The runtime overhead for maintaining the `ncio` structure is fairly small. In our implementation, we used a red-black tree for fast lookup. The memory cost is also limited because the number of outstanding non-critical I/Os is limited (128 by default) by the admission control mechanism at the block layer.

4.3 Criticality-Aware I/O Prioritization

As we discussed in Section 2.1, prioritizing critical I/Os only within a single layer (e.g., scheduling in a block-level queue) is ineffective for improving application performance. Hence, we adapt each layer in the I/O path to make it understand and enforce the I/O criticality.

In the caching layer, similar to the approach in [42], we apply separate dirty ratios to tasks issuing critical and non-critical writes, respectively. For tasks issuing non-critical writes, the applied dirty ratio is low (1% by default) to mitigate the interference to foreground tasks. With the lowered limit, a background task writing a large amount of data to buffer cache cannot fill all the available space since it should block until the current dirty ratio drops below the configured ratio. As a consequence, a foreground task is not blocked by a burst of background writes. Moreover, a foreground task calling `fsync()` does not need to depend on a large amount of dirty data generated by background tasks resulting from the file system ordering requirement [42, 43].

In the admission control stage at the block layer, we separately allocate the block queue slots for critical and non-critical I/Os, respectively, so that the admission control is isolated between critical and non-critical I/Os. To resolve the dependency to the I/O blocked at this stage, we transiently give critical I/O priority to the requesting task recorded in the corresponding `ncio` object and wake up the task to make it retry allocation of available slot with critical I/O priority. By doing so, the criticality-inherited I/O can avoid the unnecessary congestion with other non-critical I/Os during the admission.

We also designed a simple priority-based I/O scheduler at the block layer. In particular, our scheduler maintains two FIFO queues that are dedicated for critical and non-critical I/Os each. Then, all I/Os in the critical queue is dispatched first before any I/O in the non-critical queue. To prevent starvation, we use a timer to monitor the non-critical queue and guarantee that at least one non-critical I/O is processed per unit of time (10 ms by default). Furthermore, we added queue promotion support into our scheduler for properly handling a dependency to a non-critical I/O staged at the block-level queue. In order to minimize the interference at the device-level, we conservatively limit the number of non-critical I/Os dispatched to a storage device; this number is configurable and we use one by default. This form of limiting is common in practice for improving responsiveness [2, 36]. Our scheme can be integrated with an existing feature-rich scheduler like Linux CFQ at the expense of additional work to support the I/O priority inheritance.

4.4 Handling Transitive Dependency

Transitive dependencies make effective I/O prioritization more challenging. Consider a dependency chain of tasks $(\tau_1, \tau_2, ..., \tau_n)$ where each task $\tau_i (1 \le i \le n-1)$ is blocked due to a task dependency to τ_{i+1}. The last task τ_n can be in one of the three states: runnable, blocked due to I/O dependency, and blocked at the admission control stages. If τ_n is runnable or blocked due to the I/O dependency, the transitive dependency can be resolved by inheriting the critical I/O priority through the dependency chain.

However, if τ_n is blocked at one of the admission stages, inheriting the critical I/O priority is insufficient because the cause of the blocking should also be resolved. To handle this case, the applied dirty ratio at the caching layer is transiently changed to that of critical I/Os and the blocked task is woken up. At the block layer, similar to the I/O dependency resolution, the critical I/O priority is transiently inherited by the blocked task and the task is woken up. Then, the awakened task retries the admission with the changed I/O priority.

In order to resolve the transitive dependencies, we record a blocking status into the descriptor of a task when the task is about to be blocked. A blocking status consists of blocking cause and an object to resolve the cause. Blocking cause can be one of task dependency, I/O dependency, and blocking at admission control stage. For the task dependency cause, a corresponding object of lock or condition vatiable is recorded. For the I/O dependency cause, an I/O descriptor (i.e., block device identifier and sector number) is recorded. No additional object is recorded for the blocking at admission control stage. Based on the recorded blocking status, a foreground task can effectively track and resolve the blocking causes in the transitive dependencies.

In our experiments, at most four steps of transitive dependency has occurred. In particular, a foreground task is blocked on an inode mutex for file writes. The mutex is held by a background task and the task is also blocked waiting for the signal by a journaling daemon since the task tries to open a new file system transaction. The journaling daemon is also waiting for the completion of updating journal handle by another background task. The last background task is blocked on the admission control stage of the block layer because the task is issuing a burst of writeback for carrying out delayed allocation.

5 Implementation on Linux

We implemented our schemes in Linux 3.13 including around 3100 lines of additional code. A task descriptor has a field indicating whether this task is a foreground task or not. The field is set/cleared by using an existing `setpriority()` system call interface; unused values are used. To denote I/O crticality, a `bi_rw` field in `bio` and `cmd_flags` field in `request` data structures are given an extra flag. These flags are used for the admission control and I/O scheduling at the block layer.

We implemented the I/O priority inheritance to `mutex`, `rw_semaphore`, and `semaphore`. In addition, we resolved the condition variable-induced task dependency in Linux journaling layer (i.e., jbd2). We registered jbd2 kernel thread as a helper task for the condition variables `j_wait_trasnaction_locked` and `j_wait_done_commit`. For the condition variable `j_wait_updates`, a helper task is dynamically assigned and removed since a helper task can be any task having a user context. The three condition variables are specific to ext4, which can hinder the adoption of our scheme to other file systems. A good starting point for identifying condition variables causing runtime dependencies is to inspect the wait queues (i.e., `wait_queue_head_t`) defined in a file system. For example, only nine and three wait queues need to be inspected for integrating with xfs and f2fs, respectively.

The I/O priority inheritance for resolving the user-level dependency is implemented in Futex [6] and System V Semaphore [15] (SysV sem for short). The priority inheritance in Futex (`FUTEX_LOCK_PI`) is exploited and a similar method is implemented in SysV sem with an additional owner field to `sembuf`. A lock owner is recorded at user-level when acquiring a lock and is passed down to the kernel when a waiter is blocked. For the I/O dependencies, we implemented the I/O priority inheritance to `wait_on_bit()` and `wait_on_bit_lock()` methods that are used to synchronize with buffer pages. We attempt I/O prioritization at these methods when a task waits until a specific bit (`PG_locked`, `PG_writeback`, `BH_Lock`, and `BH_Shadow`) is cleared. Note that `BH_Shadow` is used to protect a buffer page that are under journaling I/O for guaranteeing file system consistency.

6 Application Studies

To validate the effectiveness of our scheme, we chose three widely deployed data-intensive applications: PostgreSQL [10] relational database v9.5, MongoDB [8] document store v3.2, and Redis [22] key-value store v3.0. For tagging foreground tasks, we inserted 15, 14, and 2 lines of code to PostgreSQL, MongoDB, and Redis, respectively. This result indicates that adopting the enlightenment API is not complicating for typical data-intensive applications.

PostgreSQL relational database. In PostgreSQL, *backend* is dedicated to client for serving requests while other processes, such as *checkpointer*, *background writer*, *log writer*, and *autovacuum worker*, carry out I/O jobs in background. The checkpointer periodically flushes all dirty data buffers to disk and writes a special checkpoint record to the log file, in order to truncate transaction logs reflected to database, thereby bounding storage space and crash recovery time. The background writer periodically writes some dirty buffers to disk to keep regular backend processes from having to write out dirty buffers. Similarly, the log writer periodically writes out the log buffer to disk in order to reduce the amount of synchronous writes backend processes should perform at commit time. The autovacuum worker reclaims storage occupied by dead tuples since tuples deleted or obsoleted by an update are not physically removed from their table.

We classified the backend processes as foreground tasks. In addition, since PostgreSQL utilizes SysV sem to implement *LWLocks*, which is a user-level mutex, we modified *LWLocks* to pass down lock owner information to the kernel. Note that the information is passed only when a waiter is blocked.

MongoDB document store. In WiredTiger, which is the default storage engine since MongoDB 3.2, background threads, such as *log threads*, *eviction threads*, and *a checkpoint thread*, conduct internal activities, such as logging and checkpointing, while *client threads* handle external requests in foreground. The log threads periodically flush log buffers to an on-disk journal file. Likewise, the eviction threads write dirty pages in the internal cache to OS buffer cache for making free pages. The checkpoint thread periodically flushes all dirty pages in the internal cache to disk for consistency.

We classified the client threads as foreground tasks. MongoDB extensively uses Pthread mutex and condition variable. To handle user-level dependency, we modified the protocol of Pthread mutex to `PTHREAD_PRIO_INHERIT` to distinguish Pthread mutex and condition variable at the kernel-level and to utilize the priority inheritance implemented in Futex.

Redis key-value store. Redis has two options to provide durability: snapshotting and command logging. The snapshotting periodically produces point-in-time snapshots of the dataset but does not provide complete durability since up to a few minutes of data can be lost. Meanwhile, the command logging guarantees the complete durability by synchronously writing an update log to an append-only file before responding back to the command. In the command logging, log rewriting is periodically conducted to constrain the size of the log file.

Similar to the other applications, the snapshotting and log rewriting are conducted by child processes in background while a main server process serves all requests sequentially. Hence, we classified the main server process as a foreground task.

7 Evaluation

7.1 Experimental Setup

The evaluation system is a Dell PowerEdge R530 server that is equipped with two Intel Xeon E5-2620 processors and 64 GB RAM. The CPU frequency is set to the highest level and hyper-threading is enabled. A single 1 TB Micron MX200 SSD is used to store data sets for the evaluation workloads. We used Ubuntu 14.04 with the modified Linux kernel version 3.13 as an OS and ext4 file system mounted with the default options.

We used CFQ as the baseline for our experiments. In addition, we used CFQ-IDLE to prioritize foreground

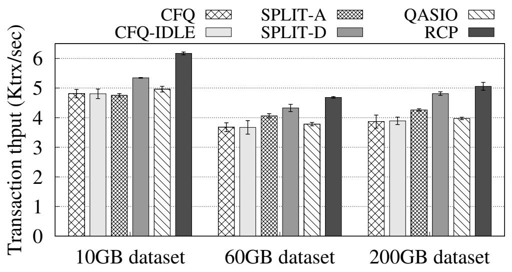

Figure 4: **PotsgreSQL request throughput.**

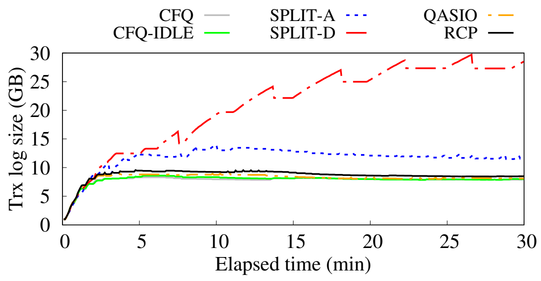

Figure 5: **PostgreSQL transaction log size.**

tasks by putting background tasks (e.g., checkpointer) to idle-priority [7]. We also used the split-level schedulers [43], including Split-AFQ (SPLIT-A) and Split-Deadline (SPLIT-D), and QASIO [32] for comparison.

For RCP, we configured 1% dirty ratio for non-critical writes and 20% dirty ratio (the default ratio in Linux) for critical writes. We separately allocated 128 slots for each critical and non-critical block-level queues. The number of non-critical I/Os outstanding to a storage device is limited to one, and the timeout for non-critical I/Os is set to 10 ms to prevent starvation at the block-level queue.

7.2 PostgreSQL Relational Database

We used the OLTP-Bench [28] to generate a TPC-C [17] workload for PostgreSQL. We simulated 50 clients running on a separate machine for 30 minutes. PostgreSQL was configured to have 40% buffer pool of the size of the initial database and to checkpoint every 30 seconds. For CFQ-IDLE and QASIO, we put the checkpointer to the idle-priority. For SPLIT-A, we set the highest and the lowest I/O priorities to backends and the checkpointer, respectively. For SPLIT-D, we set 5 ms and 200 ms `fsync()` deadlines to backends and the checkpointer, respectively; the configurations are dictated from those in [43]. We report transactions per second and transaction latency as the performance metrics.

Request throughput. Figure 4 shows transaction throughput averaged over three runs on an SSD with 100, 600, and 2000 TPC-C scale factors, which correspond to about 10 GB, 60 GB, and 200 GB of initial databases, respectively. We used unlimited request rate (i.e., zero idle/think time) for this experiment. CFQ-IDLE does not help to improve application throughput though we put the major background task (i.e., checkpointer) to idle-class priority because CFQ-IDLE prioritizes high-priority (foreground) I/Os only at the block-level scheduler. SPLIT-A improves transaction throughput only when read I/Os are dominant as scale factor increases. This is because SPLIT-A does not consider the I/O priority inversion problem and hinders foreground tasks from fully utilizing the OS buffer cache by scheduling writes at the system-call layer. As presented in [43], SPLIT-D is effective in improving the PostgreSQL's performance mainly because it procrastinates the checkpointing task. Though QASIO addresses I/O priority inversions induced by kernel-level dependencies, QASIO does not show any noticeable improvement due to the unresolved dependencies and the inability existed in the block-level scheduler as in CFQ-IDLE. RCP outperforms the existing schemes by about 15%–37%, 8%–28%, and 6%–31% for 100, 600, and 2000 scale factors, respectively.

Impact on background task. To analyze the impact of each scheme on the background task (i.e., checkpointer), we measured the size of transaction logs during the workload execution in the case of 100 scale factor. As shown in Figure 5, CFQ, CFQ-IDLE, and QASIO complete checkpointing task regularly as intended, thereby bounding the size of transaction logs to 8GB in total. SPLIT-A increases the size of transaction logs to 12GB. On the other hand, SPLIT-D increases the size of transaction logs by 3.6× over CFQ since it penalizes the regular checkpoint by delaying every `fsync()` calls made by the checkpointer until the dirty data of the requested file drops to 100 pages. As a result, SPLIT-D leads to using more storage space and possibly taking longer recovery time, which are undesirable in practice [12]. RCP completes the checkpointing task as frequent as CFQ while improving request throughput. Note that the log sizes with the other scale factors show similar trend.

Request latency. In order to show the effectiveness of RCP in terms of request latency, we ran rate-controlled TPC-C workload (i.e., fixed number of transactions) with 100 scale factor. Figure 6 demonstrates a complementary cumulative distribution function (CCDF), and so the point (x,y) indicates that y is the fraction of requests that experience a latency of at least x ms. This form of representation helps visualizing latency tails, as y-axis labels correspond to the 0^{th}, 90^{th}, 99^{th} (and so on) percentile latencies. Though CFQ-IDLE, SPLIT-D, and QASIO achieves better latency than CFQ, all the existing schemes induce several seconds request latency after 99^{th} percentile. This is because the critical path of request execution is arbitrarily prolonged at the various stages in the I/O path. On the other hand, RCP bounds request latency up to around 300 ms. We omit the latency results with 600 and 2000 scale factors because it

Latency (ms)	CFQ	CFQ-IDLE	SPLIT-A	SPLIT-D	QASIO	RCP
semtimedop	1351.8 (3.9)	3723.0 (4.2)	3504.4 (3.4)	1951.1 (3.4)	2241.7 (2.7)	247.8 (1.0)
j_wait_done_commit	1282.9 (55.9)	1450.1 (66.0)	342.4 (6.4)	1886.1 (10.6)	198.0 (3.8)	23.9 (2.5)
PG_writeback	490.2 (0.2)	1677.6 (0.1)	454 (0.2)	458.0 (0.2)	454.7 (0.1)	243.2 (0.1)
get_request	481.8 (3.0)	3722.8 (22.0)	405.2 (1.3)	240.1 (2.8)	2241.1 (3.7)	1.3 (0.1)
j_wait_transaction_locked	306.5 (68.8)	229.4 (53.2)	0.4 (0.1)	2.4 (0.2)	0.3 (0.1)	0.2 (0.1)
BH_Lock	201.3 (40.3)	1339.7 (356.7)	1.1 (0.1)	53.9 (11.1)	0.0 (0.0)	1.7 (0.5)
rwsem_down	92.4 (8.9)	357.1 (179.7)	0.8 (0.1)	33.4 (1.4)	0.0 (0.0)	2.3 (0.2)
BH_Shadow	46.5 (2.9)	15.9 (2.9)	208.9 (3.8)	236.1 (14.1)	1294.0 (36.9)	2.4 (0.2)
mutex_lock	32.7 (7.0)	16.3 (3.1)	18.6 (2.5)	944.3 (53.3)	53.3 (3.1)	0.4 (0.1)
write_entry	N/A	N/A	1703.2 (1.8)	0.0 (0.0)	N/A	N/A
fsync_entry	N/A	N/A	1084.4 (0.5)	0.0 (0.0)	N/A	N/A

Table 2: **PostgreSQL system latency breakdown.** *This table shows the maximum latency incurred at each kernel method in milliseconds; the corresponding average latency is presented in parenthesis.*

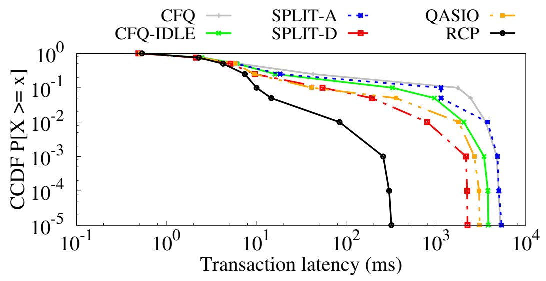

Figure 6: **PostgreSQL request latency.**

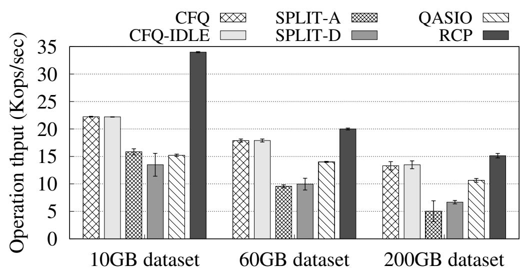

Figure 7: **MongoDB request throughput.**

is similar to that with 100 scale factor.

To analyze the reason behind the performance differences, we measured the maximum and average wait time of foreground tasks (i.e., PostgreSQL backends) in kernel functions using LatencyTOP [29]. As shown in Table 2, CFQ incurs tens to a thousand milliseconds latency at various synchronization methods. In particular, foreground tasks suffer from excessive latency at SysV sem (`semtimedop`) and `j_wait_done_commit` condition variable. CFQ-IDLE additionally causes several seconds latency waiting for the allocation of a block-level queue slot (`get_request`), the writeback of cache pages (`PG_writeback`), and the acquisition of a buffer lock (`BH_Lock`). SPLIT-A and -D are effective in resolving the file system-induced dependencies by scheduling writes at the system-call layer. However, SPLIT-A causes over one second latency at `write_entry` and `fsync_entry` because it prevents foreground tasks from fully utilizing the OS buffer cache. SPLIT-D also causes about one second latency at inode mutex (`mutex_lock`) due to the I/O priority inversion. Though QASIO resolves some dependency-induced latencies ocurred in CFQ-IDLE, it still incurs excessive latencies at `semtimedop`, `get_request`, and `BH_Shadow`. On the contrary, RCP bounds all the latencies below 250 ms.

7.3 MongoDB Document Store

For MongoDB, we used the update-heavy workload (Workload A) in the YCSB [25] benchmark suite. We simulated 150 clients running on a separate machine for 30 minutes. MongoDB was configured to have internal cache of size 40% of the initial data set. We applied the identical configuration to client threads and checkpoint thread as the PostgreSQL case for CFQ-IDLE, SPLIT-A, SPLIT-D, and QASIO. We report operations per second and operation latency as the performance metrics.

Request throughput. Figure 7 plots request throughput averaged over three runs with 10, 60, and 200 million objects, which correspond to about 10 GB, 60 GB, and 200 GB of initial data sets, respectively. As in the PostgreSQL case, CFQ-IDLE and QASIO do not help to mitigate the interference from background tasks. Unlike the case of PostgreSQL, SPLIT-D degrades request throughput rather than improving due to the different application design. MongoDB stores a collection of documents into a single file whereas PostgreSQL splits a database into multiple 1 GB-sized files. Hence, the checkpoint thread in MongoDB writes whole data set to the collection file and then calls `fsync()` to the file. In this case, SPLIT-D cannot help to prevent write entanglement since it does not schedule writes at the system-call layer. Meanwhile, scheduling writes at the system-call layer (SPLIT-A) is not also effective because buffered writes are handled slowly as in the PostgreSQL case. On the other hand, RCP improves request throughput by about 53%–152%, 12%–136%, and 12%–201% for 10, 60, and 200 million objects, respectively, compared to the existing schemes.

Request latency. Figure 8 shows CCDF of request latency measured during the execution of the rate-controlled YCSB workload (i.e., fixed number of operations) with 10 million objects. CFQ and CFQ-IDLE

Latency (ms)	CFQ	CFQ-IDLE	SPLIT-A	SPLIT-D	QASIO	RCP
futex_lock_pi	6092.6 (3.3)	11849.3 (3.4)	8300.6 (3.6)	12292.6 (3.7)	3717.4 (3.1)	305.6 (3.2)
j_wait_done_commit	6067.3 (2.3)	11846.2 (2.4)	4066.9 (2.4)	10598.4 (2.6)	3652.1 (2.0)	246.5 (2.0)
pg_writeback	239.5 (0.1)	240.1 (0.1)	426.2 (0.2)	241.0 (0.2)	241.8 (0.2)	64.2 (0.1)
get_request	35.0 (18.4)	48636 (26.6)	17.3 (5.5)	0.0 (0.0)	852.5 (6.8)	0.0 (0.0)
j_wait_transaction_locked	2.2 (1.0)	1790.3 (44.1)	1.4 (0.1)	0.0 (0.0)	1942.6 (24.0)	2.0 (0.9)
mutex_lock	0.0 (0.0)	3296.6 (544.2)	0 (0.0)	1.2 (0.1)	992.0 (77.2)	0.0 (0.0)
write_entry	N/A	N/A	7884.9 (27.4)	0.0 (0.0)	N/A	N/A
fsync_entry	N/A	N/A	8273.1 (26.2)	0.0 (0.0)	N/A	N/A

Table 3: **MongoDB system latency breakdown.** *This table shows the maximum latency incurred at each kernel method in milliseconds; the corresponding average latency is presented in parenthesis.*

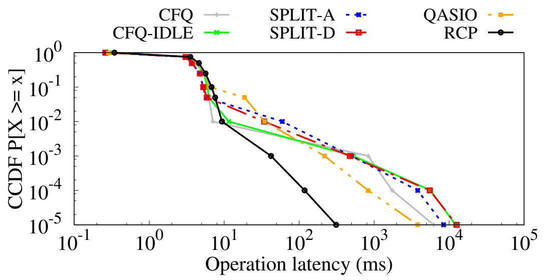

Figure 8: **MongoDB request latency.**

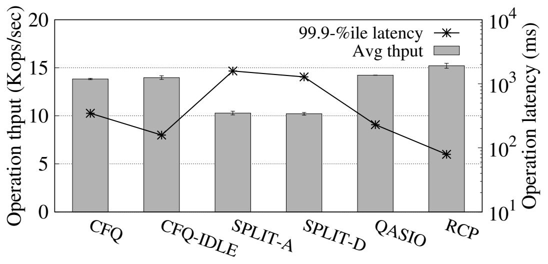

Figure 9: **Redis performance.**

significantly increase latency tail especially after 99th percentile. SPLIT-A and SPLIT-D also cannot help to bound latency; about five seconds latency at 99.99th percentile. Though QASIO improves request latency compared to the other existing schemes, it still incurs about four seconds latency at 99.999th percentile. On the contrary, RCP completes all the requests within 310 ms by carefully handling the critical path of request execution. Note that the latency results with 60 and 200 million objects is similar to that of 10 million objects.

Table 3 shows the maximum and average system latencies in foreground tasks. All the existing schemes induce excessive latencies at various synchronization methods, such as Pthread mutex (`futex_lock_pi`) and `j_wait_done_commit` condition variable. Unlike the case of PostgreSQL, CFQ-IDLE and QASIO cause over a second latency at `j_wait_transaction_locked` condition variable, which is a barrier for starting a new file system transaction. RCP largely reduces dependency-induced system latencies .

7.4 Redis Key-Value Store

For Redis, we used the same workload as the MongoDB's except that we concurrently ran ten YCSB benchmarks against ten Redis instances to utilize our multicore testbed due to the single threaded design of Redis [23]. We enabled both snapshotting and command logging for data safety [24]. We report operations per second and operation latency as the performance metrics.

Figure 9 plots operation throughput averaged over three runs and 99.9th percentile operation latency. CFQ-IDLE and QASIO slightly improves application performance by putting the background tasks including the snapshotting and log rewriting to idle-class priority. SPLIT-A and SPLIT-D deteriorate application performance because SPLIT-A does not fully utilize write buffer in the caching layer and SPLIT-D does not protect non-critical writes from file system-level entanglement. By handling the limitations in the existing prioritization schemes in terms of application performance, RCP improves request throughput by about 7%–49% compared to the existing schemes. In addition, RCP shows 78 ms request latency at 99.9th percentile, thereby achieving 2×–20× improvement over the existing schemes.

7.5 Need for Holistic Approach

In order to show why a holistic approach is crucial for application performance, we selectively disabled one of the components in our scheme, the caching layer, the block layer, the kernel-level dependency handling, the user-level dependency handling, and the transitive dependency handling. Figure 10 shows average request throughput normalized to that of all the components are enabled in the 10 GB data set configurations. Disabling any one of the component degrades application throughput by about 7–33% and 6–45% for PostgreSQL and MongoDB each. This result justifies our claim that only a holistic approach can guarantee high degree of application performance.

7.6 Impact of Limiting I/Os

Due to hidden and unpredictable I/O scheduling inside storage, we limited the number of sojourn non-critical I/Os to one. This may lead to low utilization of storage devices when there is low foreground activity. To quantify the impact on system throughput, we concurrently

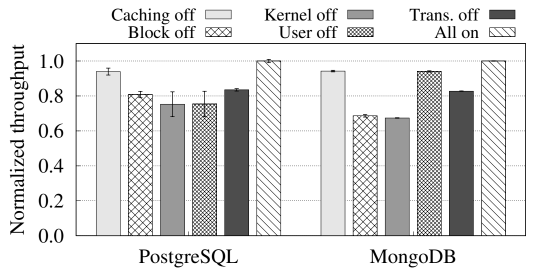

Figure 10: **Normalized request throughput with turning off one of the components in RCP.**

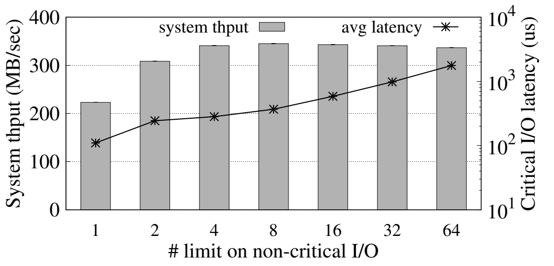

Figure 11: **Tradeoff between system throughput and foreground latency.**

ran a sequential write workload with 5 ms thinktime as a lightly-loaded foreground task and a random write workload without thinktime as a I/O-intensive background task. As shown in Figure 11, the system-wide throughput increases from 223 MB/sec to 345 MB/sec (55% improvement) while relaxing the limitation on non-critical I/Os. However, the improvement in system throughput comes with the degraded latency of foreground I/O. In particular, the average latency of foreground I/O gradually increases from 110 us to 1771 us (over 16× slowdown). If storage devices implement priority-based scheduling feature in storage interface standards (e.g., SCSI, ATA, and NVMe), this tradeoff would be minimized by exploiting the priority feature.

8 Discussion

Penalizing background tasks. Whether a foreground task really does not rely on the progress of background tasks depends on the semantic of an application. For example in MySQL, when the size of transaction logs is below a preconfigured maximum size, a foreground task does not wait for a checkpointer. However, the foreground task stops accepting updates when the log size exceeds the threshold. One workaround is to provide another threshold which is a point to give the critical I/O priority to the checkpointer. This sort of application modification requires understanding of application semantic. We believe application developers are likely willing to conduct such modifications since our scheme brings superior performance with a simple API.

User-level threading library. An application may use a user-level threading library, such as Fiber [1], though it is uncommon for data-intensive applications we targeted. In this case, our scheme cannot detect the user-level dependency. If using a user-level threading library is prevalent, implementing the I/O priority inheritance to the library based on the enlightenment API may be necessary.

User-level condition variable. Our scheme uses a simple history-based inference technique to track a helper task of a user-level condition variable. In the tested applications, this method was sufficient since observed helpers were mostly static. However, if an application has a complex relationship between condition variables and helpers, a more sophisticated inference technique is desired, which is our future work.

Implementation practicality. Our prototype implementation involves modifications to all the layers and the synchronization methods in the I/O path, thereby hindering our scheme from wide adoption. The most promising direction to be practically viable is exploiting the Split framework [43]. It provides a collection of handlers for an I/O scheduler to operate across all layers in the I/O path. We believe our scheme can be cleanly implemented based on the framework by controlling non-critical writes at the system-call level, before the caching and file system layer generates complex dependencies, and non-critical reads at the block-level.

9 Concluding Remarks

In this paper, we have shown that all the layers in the storage I/O path should be considered as a whole with I/O priority inversion in mind for effective I/O prioritization. Our experiments demonstrate that the proposed scheme can provide low and predictable request latency while minimally penalizing background tasks. We believe that our scheme can contribute to reducing total cost of ownership by alleviating the contention introduced by a burst of background I/Os and thereby relaxing the need for over-provisioning storage resources.

To handle the fairness issue which results from sharing a storage stack among multiple applications/tenants, we plan to explore integrating our scheme with an existing group-based I/O scheduler (e.g, CFQ). We also plan to investigate request handling in a distributed system with replicated data stores.

10 Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Raju Rangaswami, for their valuable comments. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (No. NRF-2014R1A2A1A10049626) and by Samsung Electronics.

References

[1] Boost Fiber. `http://olk.github.io/libs/fiber/doc/html`.

[2] Completely Fair Queuing. `https://www.kernel.org/doc/Documentation/block/cfq-iosched.txt`.

[3] Database checkpoints (SQL server). `http://msdn.microsoft.com/en-us/library/ms189573.aspx`.

[4] Ext4 filesystem. `https://www.kernel.org/doc/Documentation/filesystems/ext4.txt`.

[5] Facebook and the kernel. `https://lwn.net/Articles/591780`.

[6] futex(2) - Linux man page. `http://linux.die.net/man/2/futex`.

[7] ionice(1) - Linux man page. `http://linux.die.net/man/1/ionice`.

[8] MongoDB. `https://www.mongodb.org`.

[9] MongoDB backup methods. `https://docs.mongodb.org/manual/core/backups`.

[10] PostgreSQL. `http://www.postgresql.org`.

[11] PostgreSQL continuous archiving and point-in-time recovery. `http://www.postgresql.org/docs/current/static/continuous-archiving.html`.

[12] PostgreSQL pain points. `https://lwn.net/Articles/591723`.

[13] PostgreSQL routine vacuuming. `http://www.postgresql.org/docs/9.5/static/routine-vacuuming.html`.

[14] PostgreSQL WAL configuration. `http://www.postgresql.org/docs/9.5/static/wal-configuration.html`.

[15] semop(2) - Linux man page. `http://linux.die.net/man/2/semop`.

[16] Serial ATA native comminad queueing: An exciting new performance feature for serial ATA. Intel Corporation and Seagate Technology.

[17] The TPC-C benchmark. `http://www.tpc.org/tpcc`.

[18] AMVROSIADIS, G., AND BHADKAMKAR, M. Identifying trends in enterprise data protection systems. In *Proceedings of the 2015 USENIX Annual Technical Conference - USENIX ATC '15*.

[19] AVANTIKA MATHUR, MINGMING CAO, SUPARNA BHATTACHARYA, ANDREAS DILGER, ALEX TOMAS, LAURENT VIVIER. The new ext4 filesystem: Current status and future plans. In *In Proceedings of the Ottawa Linux Symposium - OLS '07*.

[20] BLAGOJEVIĆ, F., GUYOT, C., WANG, Q., TSAI, T., MATEESCU, R., AND BANDIĆ, Z. Priority IO scheduling in the Cloud. In *Proceedings of the 5th USENIX Workshop on Hot Topics in Cloud Computing - HotCloud '13*.

[21] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation - OSDI '06*.

[22] CITRUSBYTE. Redis. `http://redis.io/`.

[23] CITRUSBYTE. Redis latency problems troubleshooting. `http://redis.io/topics/latency`.

[24] CITRUSBYTE. Redis persistence. `http://redis.io/topics/persistence`.

[25] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud computing - SoCC '10*.

[26] CORBET, J. No-I/O dirty throttling. `https://lwn.net/Articles/456904`.

[27] CUCINOTTA, T. Priority inheritance on condition variables. In *Proceedings of the 5th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications - OSPERT '09*.

[28] DIFALLAH, D. E., PAVLO, A., CURINO, C., AND CUDRE-MAUROUX, P. OLTP-Bench: An extensible testbed for benchmarking relational databases. In *Proceedings of the VLDB Endowment - VLDB '13*.

[29] EDGE, JAKE. Finding system latency with LatencyTOP. `http://lwn.net/Articles/266153/`.

[30] GANGER, G. R., AND PATT, Y. N. The process-flow model: Examining I/O performance from the system's point of view. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems - SIGMETRICS '93*.

[31] GRAY, J., AND REUTER, A. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, Inc., 1992.

[32] JEONG, D., LEE, Y., AND KIM, J.-S. Boosting quasi-asynchronous I/O for better responsiveness in mobile devices. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies - FAST '15*.

[33] KIM, S., KIM, H., KIM, S.-H., LEE, J., AND JEONG, J. Request-oriented durable write caching for application performance. In *Proceedings of the 2015 USENIX Annual Technical Conference - USENIX ATC '15*.

[34] LAKSHMAN, A., AND MALIK, P. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating Systems Review 44*, 2 (2010), 35.

[35] LAMPSON, B. W., AND REDELL, D. D. Experience with processes and monitors in Mesa. *Communications of the ACM 23*, 2 (Feb. 1980), 105–117.

[36] LOSH, M. W. I/O Scheduling in CAM. In *Proceedings of the Technical BSD Conference - BSDCan '15*.

[37] MENAGE, P. B. Adding generic process containers to the Linux kernel. In *Proceedings of the Ottawa Linux Symposium - OLS '07*.

[38] MIKHAIL BAUTIN, GUOQIANG JERRY CHEN, PRITAM DAMANIA, PRAKASH KHEMANI, KARTHIK RANGANATHAN, NICOLAS SPIEGELBERG, LIYIN TANG, M. V. Storage infrastructure behind Facebook messages using HBase at scale. *IEEE Data Engineering Bulletin 35*, 2 (2012), 4–13.

[39] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles - SOSP '91*.

[40] RUSSINOVICH, M. E., SOLOMON, D. A., AND IONESCU, A. *Windows Internals, Part 2: Covering Windows Server 2008 R2 and Windows 7*, 6th ed. Microsoft Press, 2012.

[41] SWEENEY, A., DOUCETTE, D., HU, W., ANDERSON, C., NISHIMOTO, M., AND PECK, G. Scalability in the XFS file system. In *Proceedings of the 1996 USENIX Annual Technical Conference - USENIX ATC '96*.

[42] TING YANG, TONGPING LIU, EMERY D. BERGER, SCOTT F. KAPLAN, J. ELIOT, B. M. Redline: First class support for interactivity in commodity operating systems. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation - OSDI '08*.

[43] YANG, S., HARTER, T., AGRAWAL, N., KOWSALYA, S. S., KRISHNAMURTHY, A., AL-KISWANY, S., KAUSHIK, R. T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Split-level I/O scheduling. In *Proceedings of the 25th Symposium on Operating Systems Principles - SOSP '15*.

[44] YU, Y. J., SHIN, D. I., EOM, H., AND YEOM, H. Y. NCQ vs. I/O scheduler. *ACM Transactions on Storage 6*, 1 (Mar. 2010), 1–37.