Improving Scalability of Database Systems by Reshaping User Parallel I/O

Anonymous Author(s)
Submission Id: 167*

Abstract

Modern database systems suffer from compromised throughput, persistent unfair I/O processing and unpredictable, high latency variability of user requests as a result of mismatches between highly scaled user parallel I/O and the I/O capacity afforded by the database and its underlying storage I/O stack. To address this problem, we introduce an efficient user-centric QoS-aware scheduling shim, called AppleS, for user-level fine-grained I/O regulation that delivers the right amount and pattern of user parallel I/O requests to the database system and supports user SLOs with high-level performance isolation and reduced I/O resource contention. It is designed to enable database systems to proactively regulate user request behaviors based on runtime conditions to reshape user access pattern to hide excessive user parallelism from the I/O stack that has a limited concurrent processing capability. This helps achieve scalable throughput for multi-user workloads in a fair and stable manner. AppleS is implemented as a user-space shim for transparent user-differentiated I/O scheduling, making it highly flexible and portable. Our extensive evaluation, run on real databases (MySQL and MongoDB), demonstrates that, by incorporating AppleS in the existing database systems, our solution can not only improve the throughput (up to 39.2%) in a fairer (3.2× to 40.6× fairness improvement) and more stable (up to 2× lower latency variability) manner, but also support user SLOs with less I/O provisioning.

1 Introduction

Although significant progress has been made in improving the transaction throughput scalability for database systems running on multi-core architectures [30, 36, 38, 49, 50, 53, 56], including relational database management systems (RDBMS) for Online Transaction Processing (OLTP), e.g., MySQL [17], Oracle [18] and DB2 [3], and key-value stores, e.g., MongoDB [15], Dynamo [29] and Bigtable [25], modern databases still fail to effectively exploit the available parallelism of concurrent transactions for high-contention multi-user workloads in a fair and stable manner. This makes it challenging for a "database-as-a-service" [46] in a modern datacenter to meet the Service-Level Objectives (SLOs) [24, 42, 47].

Today's database systems, including relational databases [3, 17, 18] and key-value stores [15, 25, 29],

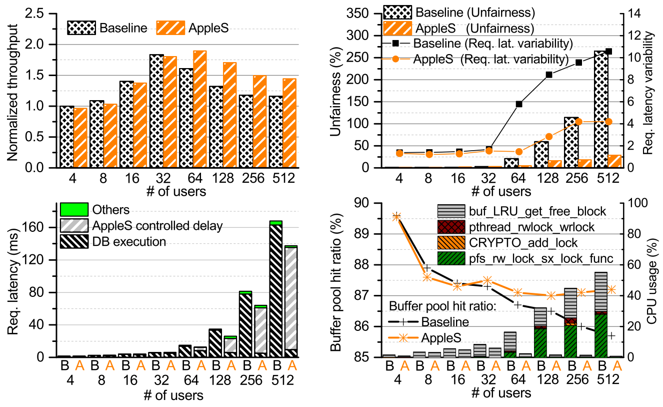

Figure 1. Baseline ("B") results show that request contentions on buffer cache, locks and I/O stack are primary culprits for modern database systems inability to provide sufficient transaction throughput, fair I/O processing, low latency, and low latency variability of user requests at high user parallelism. The AppleS ("A") results show that by temporarily holding back excessive requests outside the database (AppleS controlled delay) and controlling the amount and pattern of requests entering the database, such internal contentions can be substantially curbed (DB execution), leading to increased throughput, decreased user unfairness, latency and latency variability. The *unfairness* metric measures the maximum deviation in individual user throughput from the average, defined in Section 3.1, which is, the smaller the better.

highly rely on buffer cache, lock mechanisms and the storage I/O stack to handle concurrent requests. There user requests that are often arbitrarily delivered through simultaneous user TCP connections may lead to a *user parallelism* beyond the I/O capacity afforded by the database system. Excessive user parallelism can severely compromise database's I/O performance with *inefficient I/O processing*, *persistent user-level I/O unfairness*, and *unstable request latency.* To demonstrate this, we present, in Fig. 1, the I/O performance of a representative relational database, MySQL 8.0.15 [17], under the baseline, the latest default I/O scheduler (i.e., mq-deadline [16]) by scaling users. As one can see, the transaction throughput peaks at a specific user parallelism (i.e., around 32 for B, top-left part) when both user I/O unfairness and request latency variability start to deteriorate precipitously (Baseline, top-right part) along with the throughput decline. While the peaking of throughput with increased user parallelism is inevitable and expected at some point, the sudden jump in user unfairness and latency variability is puzzling and begs for more explanations and closer examinations. Given the aforementioned importance of buffer cache, locks and I/O stack in handling user requests, we suspect that request contentions on them at high user-parallelism should be a main culprit, which is confirmed by the baseline results given in the bottom two parts of Fig. 1. That is, starting at user parallelism of 32, the overall request latency and DB execution time rise rapidly (bottom-left part) while the buffer cache hit ratio drops and lock related processing increases notably (black curve, bottom-right part). In other words, the excessive user parallelism significantly lengthens database's processing time, which is largely spent on the code paths related to locks and buffer cache required to control request concurrency (B bars, bottom-right part). This, along with unbalanced buffer caching across users and queuing effects along the storage I/O stack, directly leads to unstable and uneven user request latency, resulting in persistent user-level I/O unfairness (see Section 2). Our intuitive solution idea, based on these observations and insight, is to hold back excessive user requests temporarily and control the amount and pattern of requests allowed to enter the database system so that the request contentions on buffer cache, locks and I/O stack are prevented to allow for a "minimally congested" database "internal pipe". This high-throughput pipe in turn enables all the requests including those held-back requests to get through the database system faster ("AppleS controlled delay") than if they were not held back ("DB execution"), while the substantially reduced internal request contentions lead to much improved user fairness and latency variability, as evidenced by the "A" results in Fig. 1.

Based on this basic idea, our proposed system in this paper, an *Appl*ication *le*vel *S*cheduler (AppleS), aims to hide excessive user parallelism by strategically holding back the requests surpassing the concurrency afforded by the database system, resulting in lower and stabler request latency, higher throughput, and improved user-level I/O fairness under the given resources.

Unlike the existing solutions, including kernel-level [8, 9, 12, 14, 16, 57] and database-specific approaches [34, 37, 40, 43, 44, 48, 52], that passively react to one or more bottlenecks resulting from highly scaled user parallelism by improving some internals of a database or instrumenting OS-kernel modules, AppleS is designed to nip the aforementioned problem in the bud by proactively reshaping user request streams in such a way that it prevents excessive I/O workload (beyond the I/O capacity afforded by the database and its underlying storage I/O stack) from reaching and overloading the database system. In other words, AppleS trades controllable and predictable delays of excessive user requests waiting outside the database system for an uncongested and resource-efficient runtime database system without which their I/O delays inside the database system can become unpredictably long and variable, harmful to user I/O fairness and efficiency. Specifically, to capture the relationship between user behaviors and a database systems performance, we regulate the user parallel I/O perceived by and delivered to the database system in two dimensions, i.e., *delivered user parallelism* and *scheduling-round I/O quota*. The former is the number of concurrent users that are seen by the database system actively contending for I/O resources, while the latter determines the number of consecutive requests issued in a scheduling round for each user according to its priority. The intuition is that, a value of the former matching the inherent concurrency of the database system allows the most effective usage of the system resources and hence, optimize throughput, whereas, a value of the latter based on user priority would help achieve this throughput without sacrificing user-level fairness by exploiting per-user sequentiality and locality (e.g., for reduced cache misses). By optimizing the values of delivered user parallelism and scheduling-round I/O quota at runtime, AppleS is able to effectively avoid or alleviate the performance bottlenecks due to superfluous user parallel I/O. This enables AppleS to support user QoS with effective user-level isolation and reduced I/O provisioning. In addition, AppleS's non-intrusive implementation, without the need to modify databases and OS kernels, makes it compatible with and portable to different types/versions of databases and different versions of OS kernels.

We implement an AppleS prototype and evaluate its effectiveness under four versions of the Linux kernel and compare it against nine state-of-the-art or widely accepted I/O schedulers and a Linux resource management tool (i.e., Cgroup(v2) [2]) with extensive experiments driven by two types of databases, including a RDBMS, MySQL [17] (version 8.0.15 and 8.0.23), for OLTP evaluation, and a key-value store, MongoDB [18] (version 3.6.0 and 4.4.3), for cloud service workloads [27]. Our prototype evaluation results demonstrate that AppleS is able to provide high transaction throughput, significantly improve on user-level I/O fairness (up to 40.6×) and maintain a low request latency and latency variability when the number of concurrent users scales. More importantly, AppleS can integrate other QoS guarantee algorithms (e.g., PSLO [41]) to enforce customized user-level SLOs with reduced I/O provisioning (saving up to 53.1% transaction throughput) by supporting fairer and more stable I/O resource allocation to concurrent users. The code for the AppleS prototype will be made publicly available upon the publication of the paper.

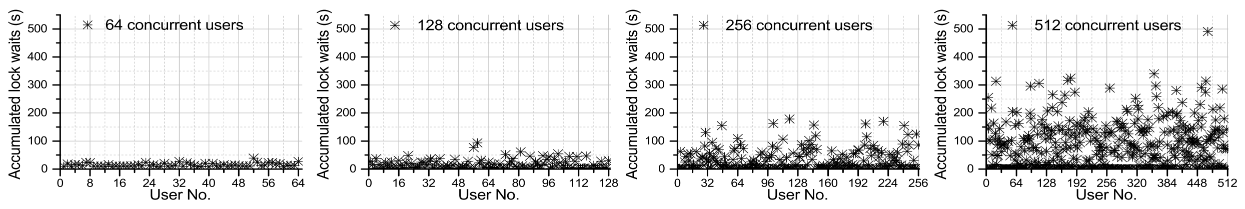

Figure 2. Each data point represents the cumulative wait time experienced by a given user, which issues the TPC-C workload to access a MySQL 8.0.15 instance. The unbalanced lock waits distributed across users can be aggravated by highly scaled user parallelism.

2 Background and Motivation

This section first presents the necessary background to set the stage for the analysis and empirical observations later in the section that motivate the AppleS study.

2.1 Background and Case Studies

Highly scaled *user parallelism*[1] can cause scattered performance bottlenecks (e.g., the buffer cache, lock mechanisms of the database, the kernel-level block layer, etc.) along the whole *I/O stack* that consists of the database and the underlying storage I/O stack. To examine these performance bottlenecks aggravated by high user parallelism, we first discuss their impact on user-level I/O performance isolation, and then study two specific cases of databases, a relational database, MySQL 8.0.15 [17], and a key-value store, MongoDB 3.6.0 [15], to quantify the impact of different levels of user parallelism on transaction throughput, user-level I/O fairness and request-latency variability. This helps us reveal the root causes behind these performance bottlenecks.

User I/O Performance Isolation for Databases: As a key indicator of the lack of user I/O performance isolation, user-level I/O unfairness (defined in Section 3.1) can be attributed to many factors and chief among them are cache conflicts and lock contentions in the database and storage I/O queuing effects. Cache conflicts can induce cache residency imbalance [31] where faster running users are rewarded with more buffer cache space, leading to their increased throughput allocations but at the expense of slower-running users. This vicious cycle causes unacceptable *persistent user-level I/O unfairness.* Moreover, existing transaction-based lock management mechanisms [38, 45, 56] cannot effectively track and prevent likely user-level unfairness resulting from intensified contentions on database locks (e.g., by invoking `pfs_rw_lock_sx_lock_func` in MySQL [17]), as evidenced in Fig. 2. In addition, storage I/O queuing effects can further magnify user I/O unfairness [35]. These three factors often work together to exacerbate the problem of persistent user I/O unfairness under high user parallelism. However, for different types of databases, they can play different roles, as elaborated next.

[1]It refers to the number of concurrent I/O-intensive users contending for I/O resources.

Bottlenecks of MySQL: We investigate the performance of MySQL 8.0.15 to understand its bottlenecks. With the user parallelism continuously increasing beyond the I/O concurrency afforded by the database, lock contentions and buffer cache conflicts become increasingly more frequent and serious (Baseline, Fig. 1), leading to inefficient usage of resources throughout the I/O stack and hence, decreased transaction throughput and increased user-level I/O unfairness and request-latency variability (Baseline, Fig. 1).

The intensified lock contentions and cache conflicts by the scaling of concurrent users cause the transaction throughput to decrease when the number of users surpasses a threshold (i.e., 32), as illustrated in Fig. 1. In the meantime, I/O resources allocated to different users are increasingly unbalanced as user parallelism increases, as indicated by the unfairness measure and request-latency variability in Fig. 1.

It is noted that `pfs_rw_lock_sx_lock_func` consumes more CPU cycles when the number of concurrent users goes beyond a threshold (i.e., 32), which implies more expected processing delay due to row-level shard and exclusive locks [13] with more concurrent users. Similarly, a high CPU usage is observed on `buf_LRU_get_free_block` when the number of users exceeds 32. This means that the buffer pool frequently evicts LRU pages and flushes them to the disk to accommodate new pages, aggravating cache conflicts, resulting in reduced buffer pool hit ratio.

Bottlenecks of MongoDB: I/O queuing effects along the storage I/O stack can become a major bottleneck under high user parallelism for MongoDB [15]. We verify this effect by exponentially increasing the number of I/O-intensive concurrent users imposed on MongoDB 3.6.0 with the YCSB [27] workloads (configurations are given in Section 5) under a state-of-the-art I/O scheduling framework (Split-level [57]). As shown in Fig. 3, a larger number of users can lead to a higher probability of queuing in the storage I/O stack and thus an unstable I/O service. Even with the Actually Fair Queuing (AFQ) scheduler [57], user I/O unfairness and request-latency variability (by measuring the coefficient of variation (CV) [10] averaged across all users) increase by 100× and 3× respectively from 4 users to 1024 users.

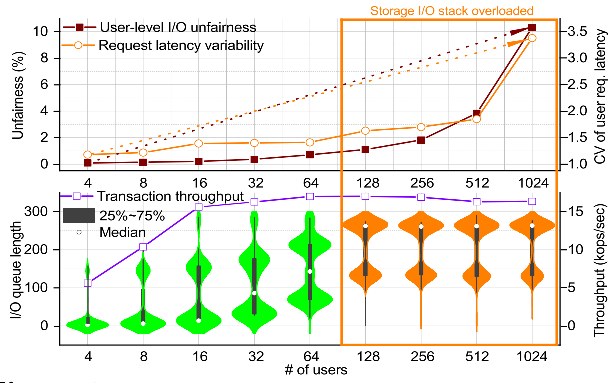

Figure 3. Challenges posed by excessive user parallelism for MongoDB. The violin plots show the distributions of I/O queue length observed in the I/O stack as a function of user parallelism. The median queue length and the interquartile range (from 25% to 75%) for a given user parallelism are represented by the white dot and the thick black vertical bar respectively.

Conclusions: As the storage backend supporting cloud services for multi-tenancy [32, 46] and large-scale industrial services [4, 5], database systems are often faced with high user parallelism. However, for both MySQL [17] and MongoDB [15], highly scaled parallel I/O can significantly weaken user I/O performance isolation and/or degrade I/O efficiency. The key is to fully exploit the potentials of database systems to service more concurrent users by minimizing the side effects, i.e., lock contentions, cache conflicts and/or queuing effects, aggravated by high user parallelism.

2.2 Related Work

In this section, we assess the existing solutions addressing the challenges due to excessive user parallelism for database systems, which further motivates our work.

Kernel-Level I/O Schedulers: The I/O schedulers [8, 12, 14, 16, 57] perform different levels of support for *process/thread-level* fairness and bursty I/O control in a passive manner. However, these schedulers cannot identify the user visiting the application that produced the I/O request and thus cannot effectively track and regulate user-level I/O traffic for databases.

Resource Management Tools: Existing resource management tools, e.g., Cgroups [9], can provide great flexibility in process/thread-level resource allocation. However, like I/O schedulers, Cgroups, as an OS kernel feature, cannot effectively track and regulate user-level I/O traffic that happens on top of the database.

Database-Specific Solutions: Existing efforts on the transaction scalability of OLTP mainly focus on efficiency problems stemming from database internals such as contentions on concurrency control and the buffer cache. For example, many solutions improve the caching or buffering mechanism by optimizing replacement algorithms [34, 37, 44, 48] or buffer cache management [40, 43, 52] for high transaction throughput or low latency. Other solutions [33, 36, 38, 49, 50, 53, 56] aim to enhance the capability of concurrency control for achieving optimized transaction scalability under high user parallelism. However, these solutions are not designed to support user-level fair and stable I/O sharing. Other database-specific solutions (e.g., SILK [23], Rein [51] and DAST [26], etc.) essentially incorporate improved components into a specific database or an evaluation framework (e.g., Janus codebase [6]) for performance improvement, still incapable of proactively alleviating the performance bottlenecks in the storage I/O stack and regulating user parallel I/O behaviors before they enter the database, which is proven to be the key to minimizing the side effects caused by excessive user parallelism (See Fig. 1). More importantly, because databases and OS kernels are frequently updated, it can entail a substantial amount of work (e.g., coding, testing and integration) for a database-specific solution to migrate across different versions of databases and/or OS kernels. This largely prevents database-specific solutions from offering the necessary portability and compatibility to different versions/types of databases and OS kernels.

In contrast, AppleS can effectively alleviate the aforementioned performance bottlenecks without instrumenting databases or OS kernels. Meanwhile, AppleS is orthogonal and complementary to the database internal solutions and hence, can work seamless with different types/versions of databases for further performance enhancement, as demonstrated in Section 5.

3 AppleS Design

AppleS works as a user I/O regulator on top of the *I/O stack*, that consists of the database and its underlying *storage I/O stack*, as illustrated in Figure 4. A database running instance (e.g., for MongoDB [15] and MySQL [17]) typically adopts a multi-thread model to handle user requests wrapped as the corresponding network packets, which are received from users' network links and temporarily stored in the OS socket buffer. The storage I/O stack, including virtual file system, file system layer, block layer and driver layer, etc., leverages multi-threaded I/O requests to fairly and efficiently share the storage backend. *AppleS works as a user-space scheduling shim treating the whole I/O stack as a black box.*

3.1 Design Goals and Principles

"Database-as-a-service" [46] in a modern datacenter demands high *throughput* (i.e., λ) under effective *performance isolation*, e.g., in terms of *user-level I/O unfairness* (i.e., U_f) and/or *request latency variability* (i.e., V_l), to serve concurrent users in a fair, stable and efficient fashion. U_f is a dimensionless metric to assess relative I/O fairness between any two backlogged

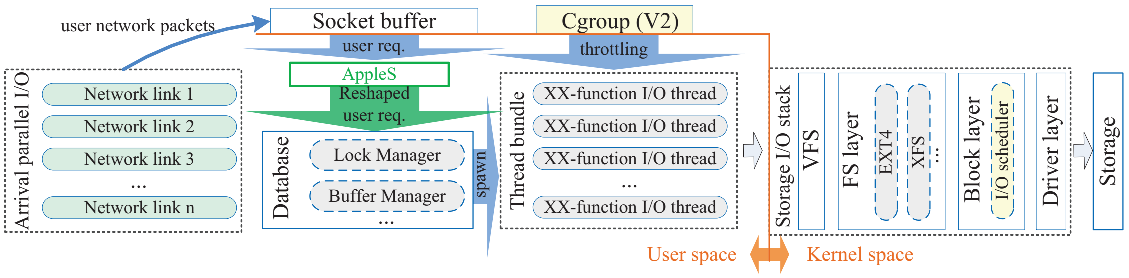

Figure 4. Overview of AppleS. As a user parallel I/O regulator, AppleS reshapes the access pattern of user requests before they enter the database, which makes AppleS potentially orthogonal to and compatible with database-specific and kernel-level solutions.

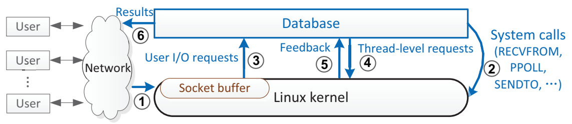

Figure 5. The life cycle of user I/O requests.

users (u_i and $u_j, 1 \le i, j \le n$) during any time interval $[t_1, t_2]$. $U_f = \frac{\max_{i=1}^{n} |N_i(t_1,t_2)/\phi_i - H|}{H}$, where $H = (\sum_{i=1}^{n} \frac{N_i(t_1,t_2)}{\phi_i})/n$, $N_i(t_1, t_2)$ is the number of scheduled requests for user $u_i, 1 \le i \le n$, during $[t_1, t_2]$. Intuitively and informally, U_f is the maximum deviation in individual user throughput from the average. V_l is defined as the coefficient of variation (CV) [10] of request latency averaged across users. Our AppleS targets at maximizing total I/O throughput under strong performance isolation across highly scaled concurrent users. To this end, the AppleS design is guided by the following principles:

`Early intervention`: Highly scaled user parallel I/O can aggravate cache conflicts, lock contentions and the I/O queuing effects, as shown in Fig. 1-3. Rather than individually addressing these bottlenecks, AppleS aims to proactively reshape user parallel I/O before it causes the spread of bottlenecks along the I/O stack.

`Hiding the excessive I/O`: AppleS aims to hide excessive user parallel I/O beyond the I/O-stack capacity by delivering the right amount of user parallelism to and keeping the appropriate per-user consecutive access pattern for the I/O stack at runtime.

`Portability and compatibility`: AppleS aims to be portable to different versions/types of databases (i.e., MongoDB [15] and MySQL [17]) by simple reconfigurations. In addition, AppleS is designed to be compatible with different OS kernels and orthogonal to existing kernel-level resource management tools (e.g., Cgroups [9]) and I/O schedulers [8, 12, 14, 16, 57].

3.2 Early Intervention

It is desirable but challenging to accommodate excessive user requests beyond the I/O-stack capacity before they enter the I/O stack. Without instrumenting the database and OS kernel as traditional designs and implementations do, AppleS regulates user parallel I/O by intervening the process that transforms the network I/O packets from user network links into the user I/O requests to be handled by the database.

Figure 5 illustrates the life cycle of user I/O requests by showing their 6-step process flow. User I/O requests wrapped in I/O packets are first received by the NIC driver and then stored at a socket buffer in the Linux kernel (step 1). The database continuously issues network-I/O system calls to signal if I/O packets are ready for the database (step 2). If yes, I/O packets will enter the database and be transformed into user requests for business-logic processing (step 3). Then, these user requests will be converted into thread-level I/O requests for the OS kernel and the rest of the I/O stack (step 4). Once the results for these requests are fed back from the Linux kernel (step 5), the database will process them and send these results back to the users (step 6). Intuitively, by strategically intercepting and suspending network-I/O system calls, AppleS can effectively shield the user requests that exceed the I/O-stack processing capacity from overloading any part of the I/O stack.

3.3 Hiding Excessive I/O

AppleS reshapes the user parallel I/O delivered to the database system by two control knobs, i.e., *delivered user parallelism*, denoted as P, and *scheduling-round I/O quota*, denoted as ζ, used by a runtime I/O scheduler.

Scheduling: AppleS schedules user requests under a weighted round-robin (WRR) policy [21, 39], aiming to coordinate user parallelism and access pattern in a fine-grained fashion. The I/O scheduling during a period of time can be logically divided into multiple *scheduling rounds*. During each round, the user parallelism seen by the I/O stack is limited by P while the total number of user requests to be issued is bounded by ζ. The *user I/O quota* will be assigned according to their individual weights ϕ_i to implement I/O differentiation under the fairness policy, i.e., $L_i = \phi_i * \zeta$ for the i^{th} user, u_i. In doing so, AppleS can coordinate I/O rate allocation across concurrent users to meet user QoS targets.

The core idea behind the AppleS scheduling is to coordinate the user request latency spent outside of the database system by controlling P, i.e., the AppleS controlled delay, which can considerably reduce the request

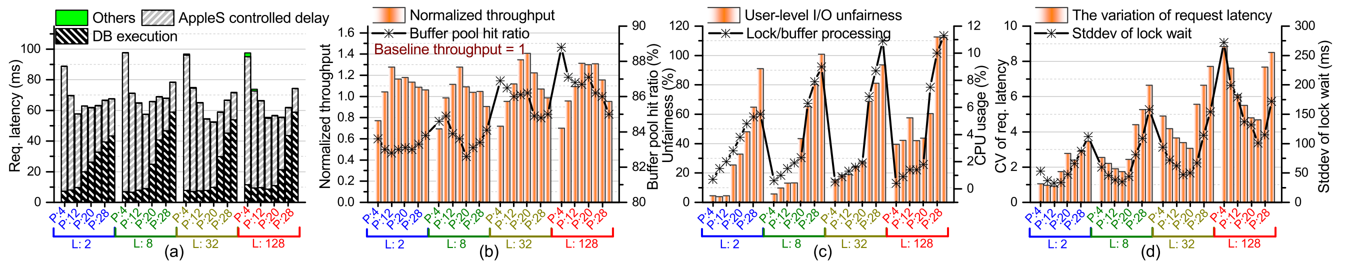

Figure 6. The effectiveness of the AppleS scheduling under different combinations of delivered user parallelism P and scheduling-round I/O quota ζ. For the convenience of observing the impact of per-user I/O sequentiality on I/O performance, which is controlled by L_i for 256 concurrent users, $1 \leq i \leq 256$, each user is assigned the same weight ϕ^0 and thus the same value of $L = \zeta * \phi^0$ for L_i.

delay inside the databased resulting from buffer cache conflicts or lock contentions that can be intensified by the excessive user parallelism inside the database. In addition, AppleS can also enable per-user I/O sequentiality adjustment for exploiting potential I/O locality (e.g., for reduced cache misses) by increasing ζ.

To show the effect of the AppleS scheduling with the two control knobs and provide an intuition for finding their optimal combination later in the paper, we let 256 users, which issue the TPC-C workload [20], concurrently access a MySQL 8.0.15 instance controlled by AppleS under different combinations of P and ζ. To effectively investigate the impact of per-user I/O sequentiality on I/O performance, which is controlled by L_i, each user has the same weight ϕ^0 and thus the same value of $L = \zeta * \phi^0$ for L_i. According to our experimental observations, we can lay out the impact of P-ζ control on I/O performance as follows:

P **Control:** The P control is the key to regulating user-level parallel I/O workload to match the I/O concurrency afforded by the database system. It affects the AppleS controlled delay experienced by those excessive requests temporarily held back because they were considered beyond the I/O-stack capacity based on the P limit. As shown in Fig. 6(a), there is a negative correlation between P and the AppleS controlled delay. A larger P commonly corresponds to a shorter AppleS controlled delay, due to the relaxation on user I/O concurrency limit, but likely causes suddenly surged database execution, largely because of a sharp CPU usage increase on lock/buffer-related processing (see in Fig. 6(c)), e.g., by frequently invoking `pfs_rw_lock_sx_lock_func` and `buf_LRU_get_free_block` in MySQL [17] (recall Section 2.1). Thus, the P control can help seek a *matching point* with the shortest AppleS controlled delay just before database execution time surges, resulting in a maximized throughput (see in Fig. 6(b)). Note that the P value with the maximized throughput commonly corresponds a low-level variability in request latency and lock waits (see in Fig. 6(d)). This means that running at the matching point will likely contribute to an efficient and stable I/O sharing for concurrent users. In addition, for different L (and thus ζ), their matching points basically stay close to each other (i.e. the range of P between 16 and 20) except the case of the smallest I/O sequentiality[2] (i.e., $L = 2$) where the matching point is located at the range of P between 12 and 16. *This indicates a matching point obtained under a specific level of per-user I/O sequentiality can be applied to other levels of I/O sequentiality.*

ζ **Control:** A larger ζ can help exploit per-user I/O sequentiality and locality for a higher buffer cache hit ratio provided that P does not exceed the matching point (see in Fig. 6(b)). However, too large a ζ (e.g., when $L = 128$ and thus $\zeta = 128 * 256$) can lengthen lock waits and offset the throughput gains from the enhanced cache hit ratio (see in Fig. 6(a)(b)). This is because a higher per-user I/O sequentiality indicates that more requests from the same user can enter the database in the same scheduling round, implying a potentially higher data locality among these requests (e.g., accessing the same data row) and resulting in increased row-lock time. Moreover, too large a ζ can also aggravate user I/O unfairness and request latency variability due to aggravated cache residency imbalance [31] (recall Section 2.1) and the intensified variation of lock time, respectively (see in Fig. 6(c)(d)). Finding the right ζ can help maximize throughput under the constraints of user I/O fairness and request latency variability. It is noted that the ζ control has almost negligible impact on database execution and AppleS controlled delay (see in Fig. 6(a)), which makes the ζ control nearly orthogonal to the P control. This is because the ζ control, which regulates the per-user I/O interarrival time distribution, has little impact on the user parallelism (across users) delivered to the database system once P is determined.

As a result, a key ***P*-ζ control principle** is that *the P control that curbs excessive user parallelism takes precedence to and is a precondition for the effectiveness of the ζ control that attempts to increase cache hit ratio by increasing per-user access locality, while the ζ control has*

[2]The case of $L = 1$ is considered no I/O sequentiality.

negligible impact on the delivered user parallelism controlled by P. Based on this principle, we can start by optimizing P control for obtaining the matching point at a specific ζ setting and then further optimize ζ control to maximize throughput constrained by the requirements of user I/O fairness and/or request latency variability.

3.4 Portability and Compatibility

First, AppleS is designed to regulate user parallel I/O on top of the database and the underlying storage I/O stack by intercepting network-I/O system calls, essentially treating the I/O stack as a black box. This separates AppleS's operations from the underlying database processing as well as kernel-level I/O scheduling and the process/thread-level resource throttling executed by resource management tools (e.g., Cgroup (V2) [9]).

Second, from the perspective of AppleS, the difference among different types of databases only lies in the adoption of different types of network-I/O system calls to handle user requests. Thus, to enable AppleS to work for a specific database, we only need to configure AppleS with the identification codes of the system calls associated with user request processing, while the database's internals can change greatly from version to version. As a blackbox solution, AppleS will automatically coordinate the P/ζ controls to optimally adapt to these changes.

Finally, if load balancing middleware is available for the database system, a joint load-balance control in the middleware and the AppleS control in individual backend database servers can be devised to enhance system-level I/O performance. This is because AppleS can support efficient, fair and stable I/O sharing under high user parallelism for each server, thus further improving the system-level capability to service more concurrent I/O intensive users by complementing load balancers.

4 AppleS Implementation

The AppleS implementation consists of three main functional modules, namely, the *User-context builder*, *Scheduler* and *Optimizer*. The user-context builder is responsible for establishing the per-user request queues; the scheduler reshapes user access pattern according to P/ζ controls. As the brain of AppleS, the optimizer tunes P and ζ through a two-timescale tuning process, slow for P and fast for ζ, to provide user-level I/O sharing in a fair, stable and efficient manner.

4.1 User-Context Builder

The user-context builder module builds user-level context by extracting the user-level network connection information hidden behind user requests. The module analyzes the system calls from user requests to extract the peer IP address and port for each user connection. With the user-specific ID information so obtained, a user-level request queue is built for each distinct user by grouping requests from that user in the FIFO order.

4.2 Scheduler

AppleS scheduler, implemented as a user-space module based on a syscall_intercept library [19], intercepts and schedules network-I/O system calls issued by the database. The library allows AppleS to hook the Linux system calls in the user space by hot-patching the machine code of the library in the memory of a process, enabling AppleS to intercept and schedule them. When AppleS delays the system calls, the I/O threads waiting for their feedback will be blocked until AppleS releases its control. Thus, all the user-level scheduling goals can be realized for a specific database by intercepting user requests destined for the I/O stack. The overhead of the system-call interception is found to be negligibly small, i.e., on the order of 100 nanoseconds, which is sufficient in support of soft-realtime applications [28].

4.3 Optimizer

Based on the analysis in Section 3.3, the optimizer workflow can be broadly divided into two parts, namely, optimizations for the control knobs P and ζ respectively. The I/O-stack concurrency capability C (i.e., the optimal value of P) is determined by the resource provision in the I/O stack and thus remains relatively stable since resource configuration changes infrequently. Thus, the P optimization runs at a slow timescale (e.g., minute-level timescale). In contrast, the ζ optimization, which aims to regulate and exploit per-user request sequentiality, must react quickly to the instantaneous I/O bursts to meet the user-level requirements for I/O fairness or request-latency variability, and therefore operates at a fast timescale (e.g., $100ms$-level) provided that each backlogged user can issue at least one request during each scheduling round.

4.3.1 P Optimization. The goal is to find the smallest P where the throughput (i.e., λ) peaks, i.e., the matching point, C, that support high throughput under effective user I/O performance isolation. To keep the measurement complexity low, we resort to a hybrid modeling-and-measurement-based approach. Specifically, by experiment, we first found that the percentage of parallel execution time, $\alpha(P)$, also known as parallel fraction, can be easily modeled as an exponentially decreasing function of P with three parameters. Then with a simple AppleS execution model, we are able to establish $\lambda(P)$ as an explicit function of P via $\alpha(P)$, with the three parameters underpinned by only three throughput samples, resulting in extremely low measurement complexity of O(1). The peak of $\lambda(P)$ is then identified with high accuracy as the experiments show.

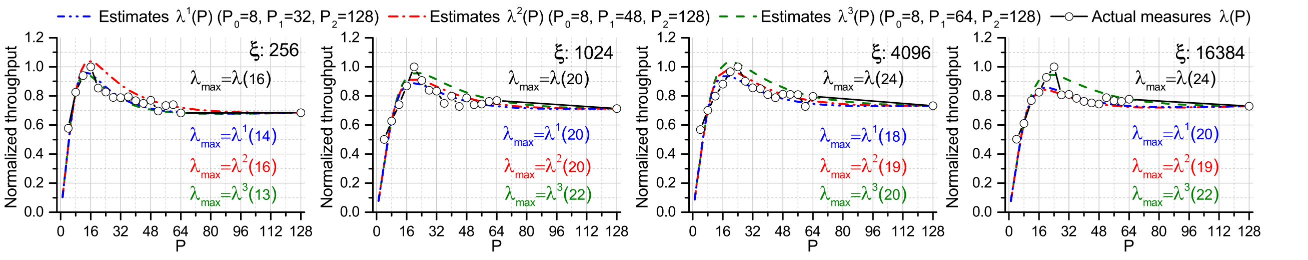

Figure 7. A comparison between AppleS estimates for throughput based on different samplings of the three throughput data points and the actual measurements.

By experiment, as shown in Fig. 8, we found that $\alpha(P)$ closely follows an exponentially decreasing function with a non-zero floor. In other words, it can be generally modeled as:

$$\alpha(P) = A + B * e^{-\gamma * P},\ A, B, \gamma > 0 \qquad (1)$$

with only three positive parameters, A, B and γ. Fig. 8 shows the curve fitting of $\alpha(P)$ given by Eq. 1. We observe that both fitting curves closely track real data points (i.e., using throughputs measured at different P values for MongoDB [15] and MySQL [17] databases, under the YCSB [27] and TPC-C benchmark [20] workloads, respectively) and accurately reflect their trend.

With the empirical expression of $\alpha(P)$ in Eq. 1, we show in Appendix that with a simplified execution model that approximately captures the AppleS execution process, the throughput $\lambda(P)$ can be given as,

$$\lambda(P) = \frac{P * \lambda(P_0)}{P_0 + B * P_0 * (e^{-\gamma P_0} - e^{-\gamma P}) * (P - 1)}. \qquad (2)$$

To pin down the parameters in Eq. 2, including $\lambda(P_0)$, P_0, B, and γ, we resort to a measurement approach by sampling three throughput data points. The first data point is $\lambda(P_0)$ at $P = P_0$ $(< C)$. In this case, α is expected to approach 1. This is because with a delivered user parallelism smaller than the I/O concurrency capacity of the I/O stack, the I/O stack may be underutilized and virtually all issued requests can be processed concurrently. Now, there are two parameters, i.e., B and γ, left to be fixed, which calls for two more samples to be measured. Specifically, with two throughput values, $\lambda(P_1)$ and $\lambda(P_2)$, sampled at $P = P_1$ and $P = P_2$, respectively, as input to Eq. 2, we establish two equations which can be jointly solved to uniquely determine B and γ. $\lambda(P_2)$ is sampled without constraining P (i.e., P_2 =the number of users) while $\lambda(P_1)$ can be sampled at a medium value. In principle, the $\lambda(P_1)$ sample can be collected at any P_1 values. Nevertheless, we recommend to sample them (i.e., $\lambda(P_1)$ and $\lambda(P_2)$) far apart to reflect the throughput variation with P. Finally, the matching point, $P = C$, is found as the first local maximum of $\lambda(P)$ given by Eq. 2.

We further verify the precision of Eq. 2 on a MySQL 8.0.15 instance. Specifically, we let 128 users, which issue

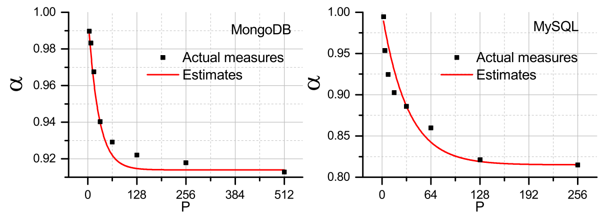

Figure 8. α curve fitting for MongoDB and MySQL.

TPC-C workload [20], concurrently access the MySQL instance controlled by AppleS under distinct ζ settings. We assess the percentage error [1] between the measured throughput at the estimated matching point derived from Eq. 2 and the throughput obtained under the actual matching point, i.e., λ_{max} error. We also evaluate the sensitivity of Eq. 2 by choosing different middle samples, $\lambda(P_1)$. As shown in Fig. 7, Eq. 2 can accurately estimate the trend of throughput as a function of P under different ζ settings, which can be confirmed by the average λ_{max} error (i.e., 3.58%) over all the cases in Fig. 7. More importantly, Eq. 2 is shown to be insensitive to the choice of middle samples as different middle samples, which lead to throughput trends with negligible difference, indicating a satisfactory stability for the output when the input changes (e.g., algorithmic stability [7]). Similarly, we also observe a low λ_{max} error for MongoDB [15] with the YCSB workloads [27] (detailed configurations in Section 5), e.g., for the maximum number of users (i.e., 1024) we use to test a MongoDB instance controlled by AppleS with the smallest per-user I/O sequentiality, the λ_{max} error is 4.46%.

4.3.2 ζ Optimization. AppleS performs fast timescale feedback control to maximize ζ constrained by a user I/O unfairness upper bound U_f^o or a request latency variability threshold V_l^o. To be responsive to rapid changes of I/O unfairness and latency variations, AppleS monitors and adjusts ζ every scheduling round. If both measured user I/O unfairness and the latency variability are lower than half of their respective thresholds, ζ is increased to exploit cache locality to optimize the I/O performance. On the other hand, if the last increment of ζ causes the measured user I/O unfairness or request latency variability to exceed their respective threshold, ζ will be reduced by half.

5 Performance Evaluation

Test Environment: All the evaluation experiments are conducted on a dedicated rack of PowerEdge R630 servers. The storage server is equipped with a RAID-0 SSD array with five 800GB SATA MLC Solid State Drives, consolidating all the logical volumes for databases. The computing server is configured with 2 Intel Xeon E5-2650 processors, 64GB of RAM, a Broadcom NetXtreme II BCM57810 10Gb NIC and 4 $\times$ 1TB SATA HDDs. All the servers are connected by a Dell N4032F switch with peak bandwidth of 10Gb.

Workloads, Databases, I/O Schedulers and Resource Management: We deploy a relational database, MySQL 8.0.15 [17], and a key-value store, MongoDB 3.6.0 [15], as representative databases to verify the effectiveness and robustness of AppleS on enforcing fair, stable and efficient I/O sharing for user-level QoS improvement. Specifically, we use the TPC-C benchmark [20] to establish multiple user connections to concurrently access a database consisting of $1,000$ warehouses (for a total dataset size of 200GB) built on MySQL. Similarly, we run the YCSB Benchmark [27] on MongoDB with multiple user connections, each generating a Zipf distributed key-value request workload. User request workloads include different combinations of GET and SET, and are write-heavy (50% GET, 50% SET) unless otherwise noted, accessing the underlying MongoDB that stores a $150GB$ dataset. Since very small key-value objects (typically smaller than $1KB$) are prevalent in enterprise-level stores [55], we set object size at $1KB$ for the MongoDB dataset. MySQL and MongoDB are each deployed on a computing server and their respective datasets are stored on their own logic volumes provided by a storage server. To verify the portability and compatibility, we evaluate AppleS under 9 I/O schedulers and across 4 versions of Linux kernels, as shown in Table 1. Since CFQ [11] may stall some requests in the queue needlessly and cause poor response times for MySQL [54] and BFQ [8] also exhibits poor performance for relational databases [22], we exclude these two schedulers from the MySQL evaluation.

To compare with the widely adopted resource management mechanism, i.e., Cgroup(v2) [2], we install Cgroup(v2)-enabled CentOS 8.3 with the Linux kernel 5.10.10 to enable full functionality of Cgroup(v2), especially the support for thread-level CPU fairness enforcement. In the experiments based on Cgroup(v2), by default we set the same CPU weight for each work thread forked by the database, assigning equal CPU cycle quota to each associated thread of the database. This enables Cgroup(v2) to effectively regulate thread-level CPU contention and guarantee thread-level CPU fair sharing. For CentOS 8.3, we install MySQL 8.0.23 and MongoDB 4.4.3 from their official repositories.

Evaluation Metrics: For the TPC-C workload [20] that cares more about high transaction throughput for OLTP processing, we use transactions per minute (TpmC) as the metric to assess its throughput while the transaction throughput in terms of ops/s is used for the YCSB [27]. Tail latencies at different percentiles, e.g., P90[3], P99 and P99.9, are used to assess the variability of transaction latency for the YCSB workload, which can significantly impact the user experience of cloud services. To easily discern the comparative results in Figures 11 and 16, pair-wise comparisons (i.e., with or without AppleS) are separated by vertical lines and bars, where AppleS results are highlighted with a yellow background. Unless otherwise specified, all the n users have the same weight ϕ_i.

Baselines and Objectives: Section 5.1 assesses the capability of AppleS to make a desirable tradeoff among transaction throughput, user I/O fairness and request-latency variability for the relational database under different levels of high user parallelism. Section 5.2 verifies the effectiveness of AppleS to enforce fair and stable I/O sharing across different numbers of concurrent users by making full use of I/O resources. Section 5.3 investigates AppleS' ability to reduce I/O provisioning for user QoS guarantees (e.g., enforcing P99.9 tail latency SLOs).

OS	Linux kernel	Resource management or I/O scheduler
CentOS 7.3	3.2.51	Split-AFQ; Split-Deadline; Split-Token
CentOS 7.3	4.9.75	CFQ; Deadline; Noop
CentOS 7.3	5.0.5	mq-deadline; Kyber; BFQ
CentOS 8.3	5.10.10	Cgroup (v2) & mq-deadline

Table 1. The baseline systems and environments.

5.1 AppleS's Optimization for MySQL

In this evaluation, we assess how effectively AppleS optimizes MySQL in terms of the tradeoff among throughput, user I/O fairness and request-latency variability under high user parallelism. Based on the investigation in Section 2.1, a number of concurrent users higher than 64 can generate excessive OLTP workloads on the MySQL running on our testbed. We thus adopt 64 or more users in this assessment. Specifically, AppleS optimizes the scheduling-round I/O quota ζ at the matching point to exploit potential spatial locality to further optimize the throughput performance, subject to two conditions, i.e., user I/O unfairness is less than or equal to 15% and the level of request-latency variability (by measuring the coefficient of variation (CV) [10]) is less than or equal to 90% of the CV of request latency measured for the

[3]PX indicates the X^{th} percentile.

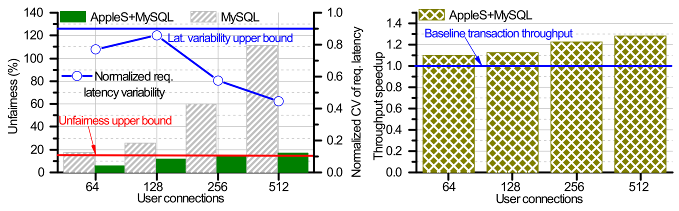

Figure 9. The transaction throughput optimization for MySQL 8.0.15 under the constraints of user I/O unfairness and request-latency variability.

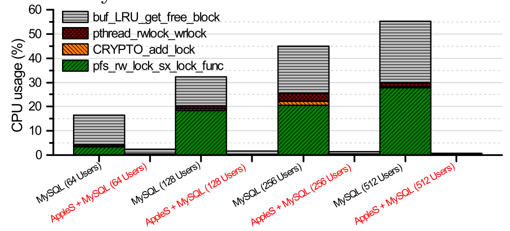

Figure 10. The breakdowns of CPU usage on the code-paths related to lock contention and cache conflict for MySQL 8.0.15.

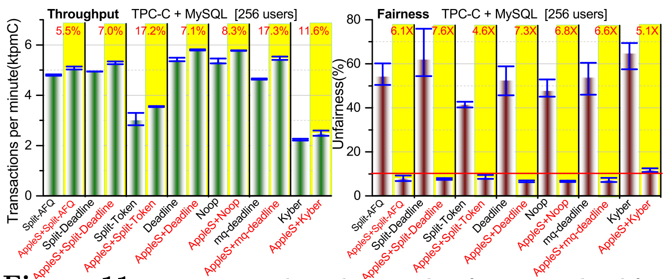

Figure 11. Transaction throughput and unfairness under different schedulers for MySQL 8.0.15.

baseline system (i.e., MySQL running on mq-deadline [16]). Based on the fast-timescale feedback control with 100ms-level monitoring for changes of user I/O unfairness and the CV of request latency, AppleS can effectively respond to each condition-violation event by quickly adjusting ζ. As show in Fig. 9, user I/O unfairness and the CV of request latency can be effectively confined within the expected ranges except for the 512-user case in which the unfairness value is 17% and slightly higher than the target. The throughput improvement ranges from 9.8% to 28.1% when the number of users increases from 64 to 512. This is because AppleS effectively improves the efficiency of the buffer pool by increasing its hit ratio and cutting down the extra disk I/Os for futile page buffering. Also evidenced by Fig. 10, AppleS successfully hides the excessive user parallelism that, otherwise, would aggravate lock contentions and buffer cache conflicts, thus significantly reducing the CPU usage on the relevant code-paths.

We further verify the effectiveness of AppleS in optimizing transaction throughput conditioned by the unfairness target of 10% under 7 I/O schedulers across 3 versions of Linux kernels respectively and compare the transaction throughput and unfairness measured with and without AppleS respectively, giving rise to

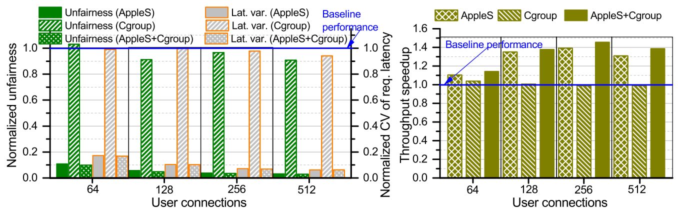

Figure 12. AppleS's effectiveness in reducing user I/O unfairness and request-latency variability, and increasing total throughput for MySQL 8.0.23 under Cgroup(v2)-enabled CentOS 8.3.

corresponding pair-wise comparisons. As demonstrated in Fig. 11, AppleS is able to improve the throughput performance for all the cases studied, with as much as 17.3% improvement for the mq-deadline case. Note that the unfairness values measured under AppleS across all the cases are below or very close to the required limit (i.e., 10%), improving over the cases without AppleS by 4.6× ∼ 7.3×. The highest unfairness incurred by AppleS is merely 11.7% for the AppleS+kyber case.

We also assess AppleS's capability to cooperate with Cgroup(v2) [2] on MySQL 8.0.23. Although the existing Cgroup(v2) can enforce process/thread-level resource distribution hierarchically in a controlled and configurable manner for a subset of resource controllers (e.g., weight-based CPU controller), it is not yet capable of controlling thread-level fair I/O sharing [2]. Thus, MySQL user-level I/O behavior, which typically happens before the corresponding thread I/Os, is actually beyond the I/O control of Cgroup(v2). However, I/O processing conducted by MySQL's work threads still rely on CPU cycles. Thus, we adopt Cgroup(v2)-enforced weight-based CPU controller to enforce thread-level CPU fair sharing among MySQL's work threads. As shown in Fig. 12, Cgroup(v2)-enforced MySQL performs slightly better than the baseline (i.e., MySQL 8.0.23) on user-level I/O fairness and latency variability in some cases (e.g., the case of 512 user connections). However, as shown in Fig. 13, Cgroup(v2) cannot effectively alleviate the contentions on lock and buffer cache of MySQL, which are the dominant factors detrimental to I/O fairness, efficiency and latency performance under high user parallelism. In contrast, AppleS can effectively hide excessive user I/O parallelism from MySQL and its underlying I/O stack, and thus significantly alleviate lock contentions and cache conflicts within MySQL, as shown in Fig. 13. This explains why AppleS can significantly enhance user-level I/O fairness by up to 31.8× with a low latency variability, as well as improve throughput performance by up to 39.2%. More importantly, when combined with Cgroup(v2), AppleS is shown to be orthogonal and complementary to Cgroup(v2)s CPU fair scheduling efforts on MySQL work threads, further improving AppleS's advantages on enhancing user-level I/O fairness (up to 34.5×) and throughput (up to 45.6%).

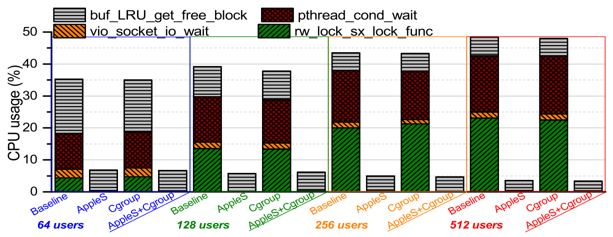

Figure 13. The breakdowns of CPU usage on the code-paths related to lock contention and cache conflict for MySQL 8.0.23 under Cgroup(v2)-enabled CentOS 8.3.

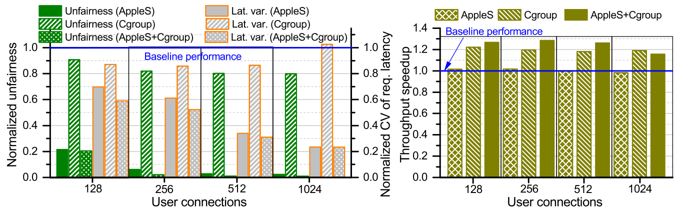

Figure 14. AppleS's effectiveness in reducing user I/O unfairness and request-latency variability, and increasing throughput for MongoDB 4.4.3 under Cgroup(v2)-enabled CentOS 8.3.

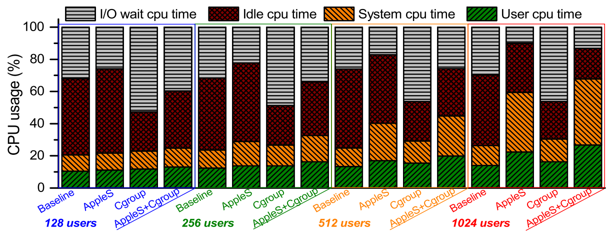

Figure 15. The breakdowns of CPU usage for MongoDB 4.4.3 under Cgroup(v2)-enabled CentOS 8.3.

5.2 AppleS's Optimization for MongoDB

In this section, we verify AppleS's efficacy on MongoDB 4.4.3 by adopting the same Cgroup(v2) configuration as that for the MySQL 8.0.23 case (described in Section 5.1) so that thread-level CPU fair sharing is enabled for MongoDB. Compared with the OLTP workload under the relational database MySQL with complex transaction processing but slower I/O speed, YCSB workloads [27] accessing MongoDB can generate a higher key-value request rate that relies more on stable and adequate CPU cycle provisioning. However, thread-level contention on CPU resource can force some threads to forfeit their CPU shares and exhibit deceptive idleness [58], resulting in low utilization of CPU and thus slower I/O speed for the storage I/O stack. As shown in Fig. 14 and Fig. 15, Cgroup(v2)-enforced thread-level CPU fair sharing can help MongoDB's work threads to expose their actual CPU demand to the OS scheduler and put a higher I/O pressure on the storage (e.g., up to 1.77× I/O wait cpu share) than that under the baseline, leading to a higher throughput (up to 22.1% improvement). In addition, although Cgroup(v2) provides some benefit in reducing user-level I/O unfairness and latency variability in most cases, it fails to effectively regulate user-level

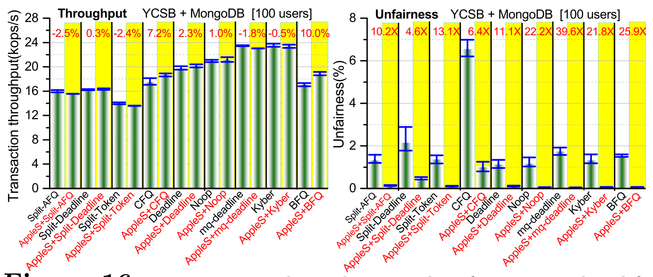

Figure 16. Transaction throughput and unfairness under different schedulers for MongoDB 3.6.0.

I/Os. In contrast, AppleS can effectively hide excessive user parallelism from the I/O stack and significantly reduce the share of I/O wait CPU time by up to 71.3%. This indicates that more portions of the I/O journey will be under AppleS's predicable control with user-level I/O fairness guarantee without sacrificing throughput, effectively preventing unpredictable and harmful I/O delay in the I/O stack. More importantly, AppleS can cooperate well with Cgroup (v2) to combine its efforts on CPU fair sharing and further improve throughput (by up to 28.6%) and user-level I/O fairness (by up to 101.5×) with a low latency variability.

Further, we examine the effectiveness of AppleS for MongoDB [15] under different I/O schedulers. Specifically, we let MongoDB run on 9 I/O schedulers across 3 versions of Linux kernels, with and without AppleS, respectively. As shown in Figure 16, AppleS helps reduce unfairness by 4.6× ∼ 39.6× across all the 9 I/O schedulers. More importantly, the cost on throughput is negligibly small, with a maximum of 2.5% transaction throughput reduction in the Split-AFQ [57] case. AppleS helps improve throughput performance for some other I/O schedulers, e.g., CFQ [11], Deadline [12] and BFQ [8]. Such improvements are likely to be attributed to AppleS's optimization that effectively alleviates I/O blocking in the I/O stack by curbing queuing effects.

5.3 AppleS's Optimization for User QoS

In this assessment, we verify the effectiveness of AppleS on reducing I/O provisioning for user QoS guarantee. AppleS is designed to export the interface for coordinating user-level fairness policy to other QoS guarantee systems, e.g., PSLO [41]. In this case, AppleS actually works as a user-level I/O scheduler for the database to provide fair, stable and efficient I/O scheduling that can be controlled by these QoS-guaranteed systems.

Specifically, we use PSLO as a QoS guarantee system for testing. PSLO can enforce any percentile tail latency SLOs for consolidated virtual machines (VMs) by dynamically coordinating I/O resources allocated among VMs based on a measurement-based feedback control model [41]. We implement the PSLO control model for database users instead of VMs. In this way, PSLO can

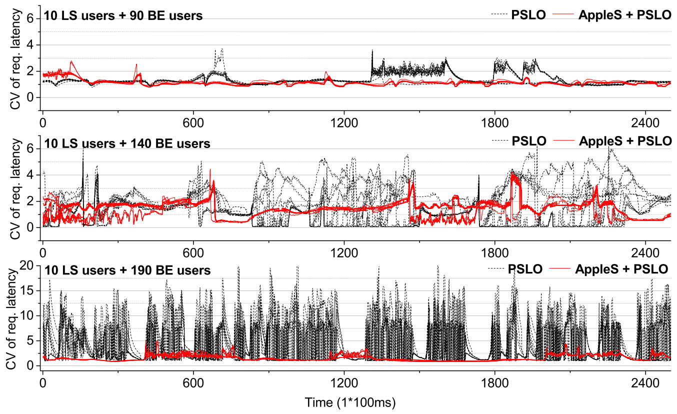

Figure 17. The time-trend curves of request-latency variability for the LS users under different schemes for different combinations of LS users and BE users.

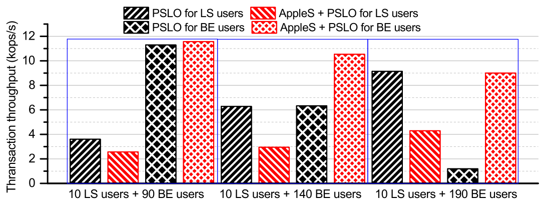

Figure 18. Comparisons in the throughput allocated to the LS users and the throughput allocated to the BE users under different schemes for different combinations of LS users and BE users.

work with AppleS to guarantee P99.9 tail latency SLOs (e.g., $60ms$) for the latency-sensitive (LS) users accessing the database (e.g., MongoDB [15]) by coordinating user weights. Meanwhile, the majority of the users demand high throughput but run in a best-effort mode, especially in cloud computing systems. They are called best-effort (BE) users that often share the same database with the LS users. Thus, cloud service providers would like to use minimum I/O resources needed to meet the LS users' tail latency targets so as to spare as much I/O resources as possible for the BE users. In our evaluation, we use three combinations of LS users and BE users to share a MongoDB. The number of the LS users is kept at 10 while that of the BE users are 90, 140 and 190. Each LS user has a P99.9 tail latency SLO of $60ms$. As shown in Fig. 17, AppleS can effectively reduce request-latency variability for the LS users, which becomes more important when more BE users share the MongoDB. As evidenced in Fig. 18, AppleS can successfully help PSLO to reduce I/O provisioning by cutting down the throughput allocated to the LS users (up to 53.1% reduction) and improve the throughput of the BE users (up to 7.6× improvement).

6 Conclusion

In this paper, we propose an application-level I/O scheduler (AppleS) to support user-level scheduling goals of I/O fairness, latency request variability and throughput for database systems. AppleS can proactively hide excessive user parallelism according to the derived matching point and optimize user I/O quotas under SLO constraints. Our extensive evaluation, driven by MySQL and MongoDB, demonstrates that users of databases can greatly benefit from AppleS, resulting in significantly enhanced user-level I/O fairness by up to 40.6× with a low variability of request latency, as well as an improved throughput by up to 39.2%. In addition, AppleS can help other QoS guarantee systems reduce I/O provisioning and run in a more cost-effective fashion.

A Proof of Eq. 2

Proof. Consider a simplified execution model that approximates the actual AppleS execution process, i.e., parallel execution rounds of P requests each and sequential execution rounds of one request each. Let p and m denote the total numbers of requests in the parallel execution rounds and the sequential execution rounds, respectively. Then for N total requests scheduled, $N = p+m$. We further define $S_k^{(q)}$ as the service time for the k^{th} request of the execution type q, for sequential execution $q = 1$ or parallel execution $q = P$. Then the total service time for the requests in the parallel execution ($q = P$) is $T_p = \sum_{k=1}^{p} S_k^{(P)}$ and the total service time for the requests in the sequential execution ($q = 1$) is $T_s = \sum_{k=1}^{m} S_k^{(1)}$. With this model, the parallel fraction, α, can then be explicitly expressed as, $\alpha = T_p/T$, where $T = T_p + T_s$, the sum of the request execution times for all the requests.

Thus, the total execution time of all N requests can be expressed as, $\alpha*T/P+(1-\alpha)*T$, and hence, we have $\lambda(P) = N/[\alpha(P)*T/P+(1-\alpha(P))*T]$. As P reduces to a certain value P_0 ($< C$), α is expected to approach 1. Substituting $P = P_0$ and $\alpha(P_0) = 1$ into the equation, we have $\lambda(P_0) = P_0 * N/T$. Since $\alpha(P_0) = 1$, from Eq. 1, we have $A = 1 - B * e^{-\gamma * P_0}$, by substituting which into Eq. 1, and then Eqs. 1 and $\lambda(P_0) = P_0 * N/T$ into $\lambda(P) = N/[\alpha(P)*T/P+(1-\alpha(P))*T]$, we finally have Eq. 2.

References

[1] [n. d.]. Approximation error. https://en.wikipedia.org/wiki/Approximation_error.
[2] [n. d.]. Control Group v2. https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html.
[3] [n. d.]. DB2. http://www.ibm.com/software/db2/.
[4] [n. d.]. Google Earth. https://www.google.com/earth/.
[5] [n. d.]. Google Finance. https://www.google.com/finance.
[6] [n. d.]. Janus codebase. https://github.com/NYU-NEWS/janus.
[7] [n. d.]. Stability (learning theory). https://en.wikipedia.org/wiki/Stability_(learning_theory).
[8] 2019. BFQ. https://www.kernel.org/doc/html/latest/block/bfq-iosched.html.

[9] 2019. Cgroups v2. https://www.kernel.org/doc/Documentation/cgroupv2.txt.

[10] 2019. Coefficient of variation. https://en.wikipedia.org/wiki/Coefficient_of_variation.

[11] 2019. Completely Fair Queuing. https://www.kernel.org/doc/Documentation/block/cfq-iosched.txt.

[12] 2019. Deadline. https://www.kernel.org/doc/Documentation/block/deadline-iosched.txt.

[13] 2019. InnoDB Locking. . https://dev.mysql.com/doc/refman/8.0/en/innodb-locking.html.

[14] 2019. Kyber. https://www.kernel.org/doc/Documentation/block/kyber-iosched.txt.

[15] 2019. MongoDB. http://www.mongodb.org/.

[16] 2019. mq-deadline. https://github.com/torvalds/linux/blob/master/block/mq-deadline.c.

[17] 2019. MySQL. http://www.mysql.com.

[18] 2019. Oracle. https://www.oracle.com.

[19] 2019. The system call intercepting library . https://github.com/pmem/syscall_intercept.

[20] 2019. TRANSACTION PROCESSING PERFORMANCE COUNCIL. The TPC-C home page. . http://www.tpc.org/tpcc/.

[21] 2019. Weighted round robin. https://en.wikipedia.org/wiki/Weighted_round_robin.

[22] 2020. Linux 5.6 I/O Scheduler Benchmarks: None, Kyber, BFQ, MQ-Deadline. https://www.phoronix.com/scan.php?page=article&item=linux-56-nvme&num=3.

[23] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. 2019. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *Proceedings of the USENIX Annual Technical Conference (ATC)*.

[24] A. Bouch, A. Kuchinsky, and N. Bhatti. 2000. Quality is in the eye of the beholder: meeting users' requirements for Internet quality of service. In *In Proceedings of the SIGCHI conference on Human factors in computing systems (CHI'00) (The Hague, The Netherlands, April 2000)*.

[25] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2006. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*.

[26] Xusheng Chen, Haoze Song, Jianyu Jiang, Chaoyi Ruan, Cheng Li, Sen Wang, Gong Zhang, Reynold Cheng, and Heming Cui. 2021. Achieving low tail-latency and high scalability for serializable transactions in edge computing. In *Proceedings of the 3th European conference on Computer systems (EuroSys)*.

[27] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the ACM symposium on Cloud computing (SoCC)*.

[28] S. S. Craciunas, C. M. Kirsch, and H. Röck. 2008. I/O Resource Management Through System Call Scheduling. *SIGOPS Oper. Syst. Rev* 42, 5 (2008), 44–54.

[29] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swami Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*.

[30] Akon Dey, Alan Fekete, and Uwe Röhm. 2013. Scalable transactions across heterogeneous NoSQL key-value data stores. In *Proceedings of the VLDB Endowment*.

[31] Dave Dice, Virendra J. Marathe, and Nir Shavit. 2014. Persistent unfairness arising from cache residency imbalance. In *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures*.

[32] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsai. 2012. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[33] Takashi Horikawa. 2013. Latch-free data structures for DBMS: design, implementation, and evaluation. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, SIGMOD'13*.

[34] Song Jiang and Xiaodong Zhang. 2002. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*.

[35] W. Jin, J. S. Chase, and J. Kaur. 2004. Interposed proportional sharing for a storage service utility. *ACM SIGMETRICS Performance Evaluation Review* 32, 1 (2004), 37–48.

[36] Ryan Johnson, Ippokratis Pandis, and Anastasia Ailamaki. 2009. Improving OLTP scalability using speculative lock inheritance. In *Proceedings of the VLDB Endowment*.

[37] T. Johnson and D. Shasha. 1994. 2Q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the VLDB Endowment*.

[38] Hyungsoo Jung, Hyuck Han, Alan Fekete, Gernot Heiser, and Heon Y. Yeom. 2014. A scalable lock manager for multicores. *ACM Transactions on Database Systems (TODS)* 39, 4 (2014), 1–29.

[39] M. Katevenis, S. Sidiropoulos, and C. Courcoubetis. 1991. Weighted round-robin cell multiplexing in a general-purpose ATM switch chip. *IEEE Journal on Selected Areas in Communications* 9, 8 (1991), 1265–1279.

[40] H. Kim and S. Ahn. 2008. BPLRU: A buffer management scheme for improving random writes in flash storage. In *Proccedings of the conference on File and storage technologies (FAST)*.

[41] N. Li, H. Jiang, D. Feng, and Z. Shi. 2016. PSLO: Enforcing the X^{th} Percentile Latency and Throughput SLOs for Consolidated VM Storage. In *Proceedings of the 3th European conference on Computer systems (EuroSys)*.

[42] H. C. Lim, S. Babu, and J. S. Chase. 2010. Automated Control for Elastic Storage. In *IEEE International Conference on Autonomic Computing (ICAC)*.

[43] Yanfei Lv, Bin Cui, Bingsheng He, and Xuexuan Chen. 2011. Operation-aware buffer management in flash-based systems. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, SIGMOD'11*.

[44] Nimrod Megiddo and Dharmendra S. Modha. 2003. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proccedings of the conference on File and storage technologies (FAST)*.

[45] Barzan Mozafari, Carlo Curino, Alekh Jindal, and Samuel Madden. 2013. Performance and resource modeling in highly-concurrent OLTP workloads. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, SIGMOD'13*.

[46] Barzan Mozafari, Carlo Curino, and Samuel Madden. 2013. DBSeer: Resource and Performance Prediction for Building

a Next Generation Database Cloud. In *CIDR*.

[47] L. N.Bairavasundaram, G. Soundararajan, V. Mathur, K. Voruganti, and S. Kleiman. 2011. Italian for beginners: the next steps for SLO-based management. In *Proceedings of the USENIX symposium on Hot Topics in Storage and File Systems (HotStorage)*.

[48] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. 1993. The LRU-K page replacement algorithm for database disk buffering. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, SIGMOD'93*.

[49] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, and Anastasia Ailamaki. 2010. Data-oriented transaction execution. In *Proceedings of the VLDB Endowment*.

[50] Ippokratis Pandis, Pinar Tözün, Ryan Johnson, and Anastasia Ailamaki. 2011. PLP: page latch-free shared-everything OLTP. In *Proceedings of the VLDB Endowment*.

[51] W. Reda, M. Canini, L. Suresh, D. Kostić, and S. Braithwaite. 2017. Rein: Taming Tail Latency in Key-Value Stores via Multiget Scheduling. In *Proceedings of the 3th European conference on Computer systems (EuroSys)*.

[52] Kun Ren, Jose M. Faleiro, and Daniel J. Abadi. 2016. Design principles for scaling multi-core oltp under high contention. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data, SIGMOD'16*.

[53] Kun Ren, Alexander Thomson, and Daniel J. Abadi. 2012. Lightweight locking for main memory database systems. In *Proceedings of the VLDB Endowment*.

[54] Baron Schwartz, Peter Zaitsev, and Vadim Tkachenko. 2012. *High performance MySQL: optimization, backups, and replication.* O'Reilly Media, Inc., Sebastopol, CA.

[55] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. 2015. LSM-trie:An LSM-tree-based Ultra-Large Key-Value Store for Small Data. In *Proceedings of the USENIX Annual Technical Conference (ATC)*.

[56] Cong Yan and Alvin Cheung. 2016. Leveraging lock contention to improve OLTP application performance. In *Proceedings of the VLDB Endowment*.

[57] S. Yang, T. Harter, N. Agrawal, S. S. Kowsalya, A. Krishnamurthy, S. Al-Kiswany, R. T. Kaushik, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. 2015. Split-Level I/O Scheduling. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*.

[58] Yong Zhao, Kun Suo, Xiaofeng Wu, Jia Rao, Song Wu, and Hai Jin. 2019. Preemptive multi-queue fair queuing. In *Proceedings of the ACM International Symposium on High Performance Distributed Computing (HPDC)*.