MittOS: Supporting Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface

Mingzhe Hao, Huaicheng Li, Michael Hao Tong, Chrisma Pakha, Riza O. Suminto,
Cesar A. Stuardo, Andrew A. Chien, and Haryadi S. Gunawi

University of Chicago

ABSTRACT

MittOS provides operating system support to cut millisecond-level tail latencies for data-parallel applications. In MittOS, we advocate a new principle that operating system should quickly reject IOs that cannot be promptly served. To achieve this, MittOS exposes a fast rejecting SLO-aware interface wherein applications can provide their SLOs (e.g., IO deadlines). If MittOS predicts that the IO SLOs cannot be met, MittOS will promptly return `EBUSY` *signal, allowing the application to failover (retry) to another less-busy node without waiting. We build MittOS within the storage stack (disk, SSD, and OS cache managements), but the principle is extensible to CPU and runtime memory managements as well. MittOS' no-wait approach helps reduce IO completion time up to 35% compared to wait-then-speculate approaches.*

CCS CONCEPTS

• **Computer systems organization → Real-time operating systems**; *Distributed architectures*;

KEYWORDS

Data-parallel frameworks, low latency, operating system, performance, real-time, SLO, tail tolerance.

ACM Reference Format:
Mingzhe Hao, Huaicheng Li, Michael Hao Tong, Chrisma Pakha, Riza O. Suminto, Cesar A. Stuardo, Andrew A. Chien, and Haryadi S. Gunawi. 2017. MittOS: Supporting Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface. In *ACM SIGOPS 26th Symposium on Operating Systems Principles.* Shanghai, China, October 28-31, 2017, 16 pages.

SOSP '17, October 28, 2017, Shanghai, China

ACM ISBN 978-1-4503-5085-3/17/10…$15.00
https://doi.org/10.1145/3132747.3132774

1 INTRODUCTION

Low and stable latency is a critical key to the success of many services, but variable load and resource sharing common in cloud environments induces resource contention that in turn produces "the tail latency problem." Early efforts to cut latency tails focused on coarse-grained jobs (tens to hundreds of seconds) [20], where there is sufficient time to wait, observe, and launch extra speculative tasks if necessary. Such a "wait-then-speculate" method has proven to be highly effective; many variants of the technique have been proposed and put into widespread use [11, 51, 60]. More challenging are applications that generate large numbers of small requests, each expected to finish in milliseconds. For these, techniques that "wait-then-speculate" are ineffective, as the time to detect a problem is comparable to the delay caused by it.

One approach to this challenging problem is *cloning*, where every request is cloned to multiple replicas and the first to respond is used [11, 55]; this proactive speculation however *doubles* the IO intensity. To reduce extra load, applications can delay the duplicate request and cancel the clone when a response is received (a *"tied requests"*) [19]; to achieve this, IO queueing and revocation management must be *built in* the application layer [15]. A more conservative alternative is *"hedged requests"* [19], where a duplicate request is sent after the first request is outstanding for more than, for example, the 95^{th}-percentile expected latency; but the slow requests (5%) must *wait* before being retried. Finally, *"snitching"* [1, 52] – the application monitoring request latency and picking the fastest replica – can be employed; however, such techniques are *ineffective* if noise is bursty.

All of the techniques discussed above attempt to minimize tail in the *absence* of information about underlying resource busyness. While the OS layer may have such information, it is *hidden* and *unexposed*. A prime example is the `read()` interface that returns either success or error. However, when resources are busy (disk contention from other tenants, device garbage collection, etc.), a `read()` can be stalled inside the OS for some time. Currently, the OS does not have a direct way to indicate that a request may take a long time, nor is there a way for applications to indicate they would like "to know the OS is busy."

To solve this problem, we advocate a new philosophy: *the OS should be aware of application SLOs and quickly reject IOs with unmet SLOs* (due to resource busyness). The OS arguably knows "everything" about its resources, including which resources suffer from contention. If the OS can quickly inform the application about a long service latency, applications can better manage impacts on tail latencies. If advantageous, they can choose *not* to wait, for example performing an *instant* failover to another replica or taking other corrective actions.

To this end, we introduce MITTOS (pronounced "mythos"), an OS that employs a fast rejecting SLO-aware interface to support millisecond tail tolerance. We materialize this concept within the storage software stack, primarily because storage devices are a major resource of contention [15, 28, 36, 42, 50, 53, 58]. In a nutshell, MITTOS provides an SLO-aware read interface, "`read(...,slo)`," such that applications can attach SLOs to their IO operations (*e.g.*, "`read()` should not take more than 20ms"). If the SLO cannot be satisfied (*e.g.*, long disk queue), MITTOS immediately rejects the IOs and returns `EBUSY` (*i.e.*, no wait), hence allowing the application to quickly failover (retry) to another node.

The biggest challenge in supporting a fast rejecting interface is the development of *latency prediction* used to determine whether the IO request should be accepted or rejected (returning `EBUSY`). Such prediction requires understanding the nature of contention and queueing discipline of the underlying resource (*e.g.*, disk spindles vs. SSD channels/chips, FIFO vs. priority). Furthermore, latency prediction must be fast; the computing effort to produce good predictions should be negligible to maintain high request rates. Finally, prediction must be accurate; vendor variations and device idiosyncrasies must be incorporated.

We demonstrate that these challenges can be met; we will describe our MITTOS design in four different OS subsystems: the disk noop scheduler (MITTNOOP), CFQ scheduler (MITTCFQ), SSD management (MITTSSD), and OS cache management (MITTCACHE). Collectively, these cover the major components that can affect an IO request latency. Our discussion will also cover the key design challenges and exemplar solutions.

To examine MITTOS can benefit applications, we study data-parallel storage such as distributed NoSQL systems. Examination shows that many NoSQL systems (*e.g.*, MongoDB) do not adopt tail-tolerance mechanisms (§2), and thus can benefit from MITTOS support.

To evaluate the benefits of MITTOS in a real multi-tenant setting, we collected statistics of memory, SSD, and disk contentions in Amazon EC2, observed from the perspective of a tenant (§6). Our most important finding is that the "noisy neighbor" problem exhibits sub-second burstiness, hence coarse latency monitoring (*e.g.*, snitching) is not effective, but timely latency prediction in MITTOS will help.

	Def. TT	TO Val.	Fail-over	Clone	Hedged/ Tied
Cassandra	×	12s	✓	×	×
Couchbase	×	75s	×	×	×
HBase	×	60s	✓	✓	×
MongoDB	×	30s	×	×	×
Riak	×	10s	×	×	×
Voldemort	×	5s	✓	✓	×

Table 1: Tail tolerance in NoSQL. *(As explained in §2).*

We evaluate our MITTOS-powered MongoDB in a 20-node cluster with YCSB workloads and the EC2 noise distribution. We compare MITTOS with three other standard practices (basic timeout, cloning, and hedged requests). Compared to hedged requests (the most effective among the three), MITTOS reduces the completion time of individual IO requests by 23-26% at p95[1] and 6-10% on average. Better, as tail latencies can be amplified by scale (*i.e.*, a user request can comprise S parallel requests and must wait for all to finish), with S=5, MITTOS reduces the completion time of hedged requests up to 35% at p95 and 16-23% on average. The higher the scale factor, the more reduction MITTOS delivers.

In summary, our contributions are: the finding of low tail-tolerance in some popular NoSQL systems (§2), the new concept and principles of MITTOS (§3), design and working examples of MITTOS design in disk, SSD, and OS cache managements (§4), the statistics of sub-second IO burstiness in Amazon EC2 (§6), and demonstration that MITTOS-powered storage systems (MongoDB and LevelDB) can leverage fast IO rejection to achieve significant latency reductions in compared to other advanced techniques (§7). We close with discussion, related work, and conclusion.

2 NO "TT" IN NOSQL

The goal of this section is to highlight that not all NoSQL systems have sufficient tail-tolerance mechanisms ("no 'TT' in NOSQL"). We analyzed six popular NoSQL systems (listed in Table 1), each ran on 4 nodes (1 client and 3 replicas), generated thousands of 1KB reads with YCSB [18], and emulated a severe IO contention for one second in a rotating manner across the three replica nodes (to emulate IO burstiness; §6), and finally analyzed if there is any timeout/failover.

Table 1 summarizes our findings. First, the "Def. TT" column suggests that all of them (in their default configurations) does not failover from the busy replica to the less-busy ones; Cassandra employs snitching, but is not effective with 1sec

[1] We use "pY" to denote the Y^{th}-percentile; for example, p90 implies the 90^{th}-percentile (y=0.9 in CDF graphs).

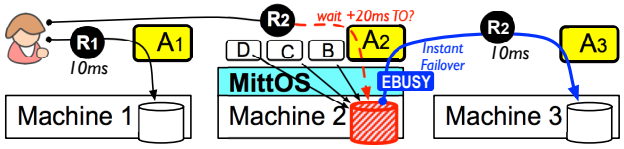

Figure 1: MITTOS Deployment Model (§3.1).

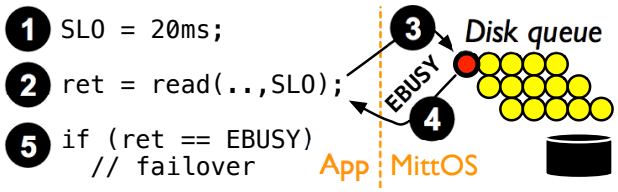

Figure 2: MITTOS use-case illustration (§3.2).

rotating burstiness (as evaluated later in §7.8.3). Second, the "TO Val." column provides the reason; by default, the timeout values are very coarse-grained (tens of seconds), thus an IO can stall for a long time without being retried. Third, to exercise the timeout, we set it to 100ms and surprisingly we observed that three of them do *not* failover on a timeout (the "Failover" column); undesirably, the users receive read errors even though less-busy replicas are available. Finally, we analyzed if more advanced techniques are supported and found that only two employ cloning and none of them employ hedged/tied requests (the last two columns).

3 MITTOS OVERVIEW

3.1 Deployment Model and Use Case

MITTOS suits the deployment model of data-parallel frameworks running on multi-tenant machines, as illustrated in Figure 1. Here, every machine has local storage resources (*e.g.*, disk) directly managed by the *host OS*. On top, different *tenants/applications* (*A*...*D*) share the same machine. Let us consider a single data-parallel storage (*e.g.*, MongoDB) deployed as applications A_1–A_3 across machines #1-3 and the data (key-values) will be replicated three times across the three machines. Imagine a user sending two parallel requests R_1 to A_1 and R_2 to A_2, each supposedly takes only 10ms (the term "*user*" implies the application's users). If the disk in machine #2 is busy because other tenants ($B/C/D$) are busy using the disk, ideally MongoDB should quickly retry the request R_2 to another replica A_3 on machine #3.

In *wait*-and-speculate approaches, request R_2 is only retried after some time has elapsed (*e.g.*, 20ms), resulting in R_2's completion time of roughly 30ms, a tail latency 3x longer than R_1's latency. In contrast, MITTOS will instantly return `EBUSY` (*no wait* in the application), resulting in a completion time of only 10+*e* ms; *e* is a one-hop network overhead.

In the above model, MITTOS is integrated to the *host OS* layer from where applications can get direct notification of resource busyness. However, our model is similar to container- or VM-based multi-tenancy models, where MITTOS can be integrated jointly across the host OS and VMM or container-engine layers. For faster research prototyping, in this paper we mainly focus on direct application-to-host model, but MITTOS can also be extended to the VMM layer. MITTOS principles will remain the same across these models.

3.2 Use Case

Figure 2 shows a simple use-case illustration of MITTOS. ① The application (*e.g.*, MongoDB) creates an SLO for a user. In this paper, we use *latency deadline* (*e.g.*, <20ms) as a form of SLO, but more complex forms of SLO such as throughput or deadline with confidence interval can be explored in future work (§8.1). We use the 95^{th}-percentile latency as the deadline value, which we will discuss more in Sections 7.2 and 8, to what value a deadline should be set. ② The application then tags `read()` calls with the deadline SLO. To support this, we create a new `read()` system call that can accept application SLO (essentially one extra argument to the existing `read()` system call). ③ As the IO request enters a resource queue in the kernel, MITTOS checks if the deadline SLO can be satisfied. ④ If the deadline SLO will be violated in the resource queue, MITTOS will instantly return `EBUSY` error code to the application. ⑤ Upon receiving `EBUSY`, the application can quickly failover (retry) the request to another replica node.

3.3 Goals / Principles

MITTOS advocates the following principles.

- *Fast rejection ("busy is error"):* In the PC era, the OS must be best-effort; returning busy errors is undesirable as PC applications cannot retry elsewhere. However, in tail-critical datacenter applications, best effort interface is insufficient to help applications manage ms-level tails. Datacenter applications inherently run on redundant machines, thus there is no "shame" for the OS to reject IOs. In large-scale deployments, this principle works well, as the probability of all replicas busy at the same time is extremely low (§6).
- *SLO aware:* Applications should expose their SLOs to the OS, such that the OS only rejects IOs whose SLOs cannot be met (due to resource busyness).
- *Instant feedback/failover:* The sub-ms fast rejection gives ms-level operations more flexibility to failover quickly. Making a system call and receiving `EBUSY` only takes <5µs (③ and ④ in Figure 2). Failing over to another machine (⑤ in Figure 2) only involves one more network hop (*e.g.*, 0.3ms in EC2 and our testbed or even 10µs with Infiniband [44]).
- *Keep existing OS policies:* MITTOS' simple interface extensions allow existing OS optimizations and policies to be preserved. MITTOS does *not* negate nor replace all prior advancements in the QoS literature. We only advocate that

applications get notified when OS-level QoS policies fail to meet user deadlines due to unexpected bursty contentions. For example, even with CFQ fairness [2], IOs from high-priority processes occasionally must wait for lower-priority ones to finish. As another example, in SSDs, even with advanced isolation techniques, garbage collection or wear-leveling activities can induce a heavy background noise.

- *Keep applications simple:* Advanced tail-tolerance mechanisms such as tied requests and IO revocation are less needed in applications. These mechanisms are now pushed to the OS layer, which then can be reused by many applications. In MITTOS, the rejected request is *not* queued (step ④ in Figure 2); it is *automatically cancelled* when the deadline is violated. Thus, applications do not need to wait or revoke IOs, nor they add more contentions to the already-contended resources. MITTOS also keeps application failover logic simple and sequential (the sequence of ②-⑤ in Figure 2).

3.4 Design Challenges

The biggest challenge of integrating MITTOS to a target resource and its management is the EBUSY prediction (*i.e.*, whether the arriving IO should be accepted or rejected). There are three major challenges: **(1)** We must understand the contention nature and queueing discipline of the target resource. For example, in disks, the spindle is the resource of contention, but in SSDs, parallel chips/channels exhibit independent queueing delays. Furthermore, the target resource can be managed by different queuing disciplines (noop/FIFO, CFQ [2], anticipatory [32], etc.). Thus, EBUSY prediction will vary across different resources and schedulers. **(2)** In terms of performance overhead, latency prediction should ideally be $O(1)$ for every arriving IO. $O(N)$ prediction that iterates through N pending IOs is not desirable. **(3)** In terms of accuracy, different device types/vendors have different latency characteristics (*e.g.*, varying seek costs across disks, page-level latency variability within an SSD).

4 CASE STUDIES

The goal of this section is to demonstrate that MITTOS principles can be integrated to many resource managements such as the disk noop (§4.1) and CFQ (§4.2) IO schedulers, SSD (§4.3) and OS cache (§4.4) managements. In each integration, we describe how we address the three challenges (understanding the resource contention nature and fast and accurate latency prediction).

4.1 Disk Noop Scheduler (MITTNOOP)

Our first and simplest integration is to the noop scheduler. The use of noop for disk is actually discouraged, but the goal of our description below is to explain the basic mechanisms of MITTOS, which will be re-used in subsequent sections.

Resource and deadline checks: In noop, arriving IOs are put to a FIFO dispatch queue whose items will be absorbed to the disk's device queue. The logic of MITTNOOP is relatively simple: when an IO arrives, it calculates the IO's wait time (T_{wait}) given all the pending IOs in the dispatch and device queues. If $T_{wait} > T_{deadline} + T_{hop}$, then EBUSY is returned; T_{hop} is a constant of 0.3ms one-hop failover in our testbed.

Performance: A naive $O(N)$ way to perform a deadline check is to sum all the N pending IOs' processing times. To make deadline check $O(1)$, MITTNOOP keeps track the disk's next free time ($T_{nextFree}$), as explained below. The arriving IO's wait time is simply the difference of the current and next free time ($T_{wait} = T_{nextFree} - T_{now}$). If the disk is currently free ($T_{nextFree} < T_{now}$), the IO is submitted directly.

Accuracy: When an IO is accepted, MITTNOOP adds the next free time with the predicted processing time to serve the new IO ($T_{nextFree} += T_{processNewIO}$). To make $T_{nextFree}$ accurate, $T_{processNewIO}$ must be precise. To achieve that, we must profile the disk's read/write latency, specifically the relationships between IO sizes, jump distances, and latencies. In a nutshell, $T_{processNewIO}$ is a function of the size and offset of the current IO, the last IO completed, and all the IOs in the device queue. We defer the details to Appendix §A. Our one-time profiling takes 11 hours (disk is slow).

$T_{nextFree}$ will automatically be calibrated when the disk is idle ($T_{nextFree} = T_{now} + T_{processNewIO}$). However, under no-idle period, a slight error in $T_{nextFree} += T_{processNewIO}$ will accumulate over time as thousands/millions of IOs are submitted. To calibrate more accurately, we attach $T_{processNewIO}$ and the IO's start time to the IO descriptor, such that upon IO completion, we can measure the "diff" of the actual and predicted processing time ($T_{diff} = T_{processActual} - T_{processNewIO}$) and then calibrate the next free time ($T_{nextFree} += T_{diff}$).

4.2 Disk CFQ Scheduler (MITTCFQ)

Next, we build MITTCFQ within the CFQ scheduler [2], the default and most sophisticated IO scheduler in Linux. We first describe the structure of CFQ and its policy.

Unlike noop, CFQ manages groups with time slices proportional to their weights. In every group, there are three *service trees* (RealTime/BestEffort/Idle). In every tree, there are *process nodes*. In every node, there is a *red-black tree* for sorting the process' pending IOs based on their on-disk offsets. Using `ionice`, applications can declare IO types (RealTime/BestEffort/Idle and 0-7 priority level). CFQ policy always picks IOs from the RealTime tree first, and then from BestEffort and Idle. In the chosen tree, it picks a node in round robin style, proportional to its time slice (0-7 priority level). Then, CFQ dispatch some or all the requests from the node's red-black tree. The requests will be put to a FIFO dispatch queue and eventually to the device queue.

Resource and deadline checks: When an IO arrive, MITTCFQ needs to identify to which group, service tree, and process node, the IO will be attached to. This is for predicting the IO wait time, which is the sum of the wait times of the current pending IOs in the device and dispatch queues as well as the IOs in other CFQ queues that are *in front of* the priority of the new IO's node. This raises the following challenges.

Performance: To avoid $O(N)$ complexity, MITTCFQ keeps track the predicted total IO time of each process node. This way, we reduce $O(N)$ to $O(P)$ where P is the number of processes with pending IOs in the CFQ queues. In our most-intensive test with 128 IO-intensive threads, iterating through all pending IOs' descriptors in the naive $O(N)$ method costs about 10-20µs. Our optimizations above (and more below) bring the overhead down to <5µsper IO prediction.

Accuracy: With the method above, MITTCFQ can reject IOs before they enter the CFQ queues. However, due to the nature of CFQ, some IOs can be accepted initially, but if soon new higher-priority IOs arrive, the deadlines of the earlier IOs can be violated as they are "bumped to the back." To cancel such IOs, the $O(P)$ technique above is not sufficient because a single process (*e.g.*, MongoDB) can have different IOs with different deadlines (from different users). Thus, MITTCFQ adds a hash table where the `key` is a tolerable time range and the `values` are the IOs with the same tolerable time (grouped by 1ms). For example, a recently-accepted IO (supposedly 6ms without noise) has a 25ms deadline but only a 10ms wait time, hence its tolerable time is 9ms. If a new higher-priority IO arrive with 6ms predicted processing time, the prior IO is not cancelled, but its `key` changes from 9ms to 3ms. If another 6ms higher priority IO arrives, the tolerable time will be negative (-3ms); all IOs with negative tolerable time are rejected with `EBUSY`.

4.3 SSD Management (MITTSSD)

Latency variability in SSD is an ongoing problem [5, 6, 19]. Read requests from a tenant can be queued behind writes by other tenants, or the GC implications (more read-write page movements and erases). A 4KB read can be served in 100µs while a write and an erase can take up to 2ms and 6ms, respectively. While there are ongoing efforts to achieve a more stable latency (GC impact reduction [29, 57] or isolation [31, 36]), none of them cover all possible cases. For example, under write bursts or no idle period, read requests can still be delayed significantly [57, §6.6]. Even with isolation, occasional wear-leveling page movements will introduce a significant noise [31, §4.3].

Fortunately, not all SSDs are busy at the same time (§6), a situation that empowers MITTSSD. A read-mostly tenant can set a deadline of <1ms; thus, if the read is queued behind writes or erases then the tenant can retry elsewhere.

Resource and deadline checks: There are two initial challenges in building MITTSSD. First, CFQ optimizations are not applicable as SSD parallelizes IO requests without seek costs; the use of noop is suggested [4]. While we cannot reuse MITTCFQ, MITTNOOP is also not reusable. This is because unlike disks where a spindle (a single queue) is the contended resource [10, 39], an SSD is composed of multiple parallel channels and chips. Calculating IO serving time in the block-level layer will be inaccurate (*e.g.*, ten IOs going to ten separate channels do not create queueing delays). Thus, MITTSSD must keep track of outstanding IOs to *every* chip, which is impossible without white-box knowledge of the device (in commodity SSDs, only the firmware has full knowledge of the internal complexity).

Fortunately, host-managed/software-defined flash [45] is gaining popularity and publicly available (*e.g.*, Linux LightNVM [14] on OpenChannel SSDs [7]). Here, all SSD internal channels, chips, physical blocks and pages are all exposed to the host OS, which also manages all SSD managements (FTL, GC, wear leveling, etc.). With this new technology, MITTSSD in the OS layer is possible.

As an additional note, a large IO request can be striped to sub-pages to different channels/chips. If any sub-IO violates the deadline, `EBUSY` is returned for the entire request; all sub-pages are not submitted to the SSD.

Performance: Similar to MITTNOOP's approach, MITTSSD maintains the next available time of *every* chip (as explained below), thus the wait-time calculation is $O(1)$. For every IO, the overhead is only 300 ns.

Accuracy: Making MITTSSD accurate involves solving two more challenges. First, MITTSSD needs to know the chip-level read/write latency as well as the channel speed, which can be obtained from the vendor's NAND specification or profiling. For measuring chip-level queueing delay, our profiler injects concurrent page reads to a single chip and for channel-level queueing delay, concurrent reads to multiple chips behind the same channel. As a result, for our OpenChannel SSD: $T_{chipNextFree}$+= 100µs per new page read. That is, a page (16KB) read takes 100µs (chip read and channel transfer); >16KB multi-page read to a chip is automatically chopped to individual page reads. Thus, $T_{wait} = T_{now} - T_{chipNextFree} + (60\mu s \times \#IO_{SameChannel})$. That is, the IO wait time involves the target chip's next available time plus the number of outstanding IOs to other chips in the same channel, where 60µs is the channel queueing delay (consistent with the 280 MBps channel bandwidth in the vendor specification). If there is an erase, $T_{chipNextFree}$+=6ms.

Second, while read latencies are uniform, write latencies (flash programming time) vary across different pages. Pages that are mapped to upper bits of MLC cells incur 2ms programming time, while those mapped to lower bits only incur

1ms. To differentiate upper and lower pages, one-time profiling is sufficient. Our profiled write time of the 512 pages of every NAND block is "`11111121121122...2112`." That is, 1ms write time is needed for pages #0-6, 2ms for page #7, 1ms for pages #8-9, and the middle pages ("...") have a repeating pattern of "`1122`." The pattern is the same for every block (consistent with the vendor specification); hence, the profiled data can be stored in an 512-item array.

To summarize, unlike disks, SSD internal complexity is arguably more complex (in terms of address mapping and latency variability). Thus, accurate prediction of SSD performance requires white-box knowledge of the device.

4.4 OS Cache (MittCache)

A user with accesses to a small working set might expect a high cache hit ratio, hence the use of a small deadline. However, under memory space contention, MittCache can inform the application of swapped-out data and and to retry elsewhere (not wait) while the data is fetched from the disk.

Resource and deadline checks: For applications that read OS-cached data via `read(...,deadline)`, MittCache only adds a slight extension. First, MittCache checks if the data is in the buffer cache. If not, it simply propagates the deadline to the underlying IO layer (§4.1-4.3), where if the deadline is less than the smallest possible IO latency (the user expects an in-memory read), `EBUSY` is returned. Else, the IO layer will process the request as explained in previous sections.

The next challenge is for `mmap()`-ed file, which skips the `read()` system call. For example, a database file can be mmaped to the heap (`myDB[]`). If some of the pages are not memory resident, an instruction such as "`return myDB[i]`" can stall due to a page fault. Since no system call is involved, the OS cannot signal `EBUSY` easily.

We explored some possible solutions including restartable special threads and `EBUSY` callbacks. In the former, MittCache would restart the page-faulting thread and inform the application's master thread to retry elsewhere. In the latter, the page-faulting thread would still be stalled, but the application's main thread must register a callback to MittCache in order to be notified. These solutions keep the `mmap()` semantic but require heavy restructuring of the application.

We resort to a simpler, practical solution: adding an "`addr check()`" system call. Before dereferencing a pointer to an mmap-ed area, the application can make a quick system call (*e.g.*, `addrcheck(&myDB[i], size, deadline)`, which walks through the process' page table and checks the residency of the corresponding page(s).

Performance: We find `addrcheck()` an acceptable solution for three reasons. First, `mmap()` sometimes is used only for the simplicity of traversing the database file (*e.g.*, traversing B-tree on-disk pointers in MongoDB), but not necessarily for performance. Second, in storage systems, users typically read a large data range (>1KB); thus, no system call is needed for every byte access. Third, existing buffer cache and page table managements are already efficient; `addrcheck` traverses existing hash tables in $O(1)$. With these reasons combined, using `addrcheck()` only adds a negligible overhead (82ns per call); network latency (0.3ms) still dominates.

One caveat in MittCache is that OS cache should be treated differently than low-level storage. For fairness, MittCache should continue swapping in the data in the background, even after `EBUSY` is already returned. Otherwise, the OS cache is less populated with data from applications that expect memory residency.

Accuracy: As MittCache only looks up the buffer/page tables, there is no accuracy issues. One other caveat to note, MittCache should return `EBUSY` to signal memory space contention (*i.e.*, swapped-in pages are swapped out again), but not for first-time accesses. Note that MittCache does not modify existing page-eviction policies (§3.3). The OS can be hinted not to swap out the pages that are being checked, to avoid false positives.

4.5 Implementation Complexity

MittOS is implemented in 3440 LOC in Linux v4.10 (MittNoop +MittCFQ, MittSSD, and MittCache in 1810, 1450, and 130 lines respectively, and an additional 50 lines for propagating deadline SLO through the IO stack).

5 APPLICATIONS

Any application that employs data replication can leverage MittOS with small modifications. In this paper, we focus on data-parallel storage such as distributed NoSQL.

• **MittOS-Powered MongoDB:** Our first application is MongoDB; being written in C++, MongoDB enables fast research prototyping of new system calls usage. The following is a series of our modifications. **(1)** MongoDB can create one deadline for every user, which can be modified anytime. A user can set the deadline value to the 95^{th}-percentile (p95) expected latency of her workload (§7.2). For example, if the workload mostly hits the disk, the p95 latency can be >10ms. In contrast, if the dataset is small and mostly hits the buffer cache, the p95 latency can be <0.1ms. **(2)** MongoDB should use MittOS' `read()` or `addrcheck()` system calls to attach the desired deadline. **(3)** If the first two tries return `EBUSY`, the last retry (currently) disables the deadline ($Prob(3NodesBusy)$ is small; §6). Having MittOS return `EBUSY` with wait time, to allow a 4th retry to the least busy node (out of the three), is a possible extension. **(4)** Finally, one last specific change in MongoDB is adding an "exceptionless" retry path. Many

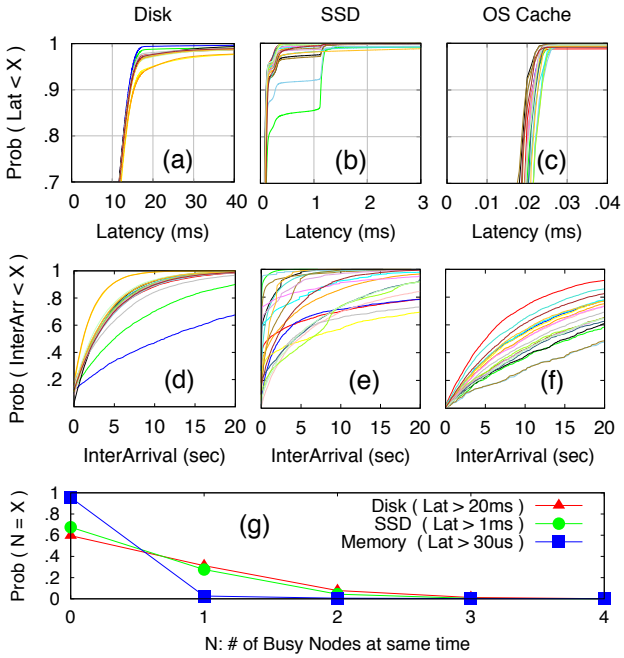

Figure 3: Millisecond-level latency dynamism in EC2. *The figures are explained in Section 6. The 20 lines in Figure (a)-(f) represent the latencies observed in 20 EC2 nodes.*

systems (Java/C++) use exceptions to catch errors and retry. In MongoDB, C++ exception handling adds 200 µs, thus we must make a direct exceptionless retry path.

The core modification in MongoDB to leverage MITTOS support is only 50 LOC, which mainly involves adding user's deadlines and calling `addrcheck` (MongoDB by default uses `mmap()` to read data file). For testing MITTOS' `read()` interface, we also add `read`-based method to MongoDB in 40 LOC. For making one-hop, exceptionless retry path, we add 20 more lines of code. Finally, to evaluate MITTOS with other advanced techniques, we have added cloning and hedged-request features to MongoDB for another 210 LOC.

• **MITTOS-Powered LevelDB+Riak**: To show that MITTOS is applicable broadly, we also integrated MITTOS to LevelDB. Unlike MongoDB, LevelDB is not a replicated system, it is only a single-machine database engine for a higher-level replicated system such as Riak. Thus, we perform a two-level integration: we first modify LevelDB to use MITTOS system calls, and then the returned `EBUSY` is propagated to Riak where the read failover takes place. All of the modifications are only 50 LOC additions.

6 MILLISECOND DYNAMISM

For our evaluation (§7), we first present the case of ms-level latency dynamism in multi-tenant storage due to the noisy neighbor problem. To the best of our knowledge, no existing work shows ms-level dynamism at the local resource level; NAS/SAN latency profiles (*e.g.*, for S3/EBS [22, 55]) exist but the deep NAS/SAN stack prevents the study of dynamism at the resource level. Device-level hourly aggregated latency data [28] also exists but prevents ms-level study.

We ran three sets of data collections for disk, SSD, and OS cache in EC2 instances with directly-attached "instance storage," shared by multiple tenants per machine. Many deployments use local storage for lower latency and higher bandwidth [47]. The instance types are `m3.medium` for SSD and OS cache and `d2.xlarge` for disk experiments (none of "`m`" instances have disk). For each experiment, we spawn **20 nodes** for **8 hours** on a weekday (9am-5pm). For disk, in each node, 4KB data is read randomly every 100ms; for SSD, 4KB data is read every 20ms; for OS cache, we pre-read 3.5GB file (fit in the OS cache) and read 4KB random data every 20ms. Their latencies without noise are expected to be 6-10ms (disk), 0.1ms (SSD), and 0.02ms (OS cache).

We ran more experiments to verify data accuracy. We repeat each experiment 3x on different weekdays (and still obtain highly similar results). To verify the absence of self-inflicted noises, ≥20ms `sleep` is used, otherwise "no-sleep" instances will hit VCPU limit and occasionally freeze. An "idle" instance (no data-collection IOs) only reads/writes 10 MB over 8 hours (0.4 KB/s). Finally, the 3.5GB file always fits the OS cache.

Observation #1: *Long tail latencies are consistently observed.* Figures 3a-c show the latency CDFs from the disk, SSD, and cache experiments, where each line represents a node (20 lines in each graph). Roughly, tail latencies start to appear around p97 (>20ms) for disks, p97 (>0.5ms) for SSD, and p99 (>0.05ms) for OS cache; the last one implies the cached data is swapped out for other tenants (a VM ballooning effect [54]). The tails can be long, more than 70ms, 2ms, and 1ms at p99 for for disk, SSD, and OS cache, respectively. Sometimes, some nodes are more busy than others (*e.g.*, >0.5ms deviation at p85 and p90 in Figure 3b).

A small resource-level variability can easily be amplified by scale. If P fraction of the requests to a node observe tail latencies, and a user request must collect N parallel sub-requests to such nodes, probabilistically, $1-(1-P)^N$ of user requests will observe tail latencies [19]. We will see this scale amplification later (§7.3).

Observation #2: *Contentions exhibit bursty arrivals.* As a machine can host a wide variety of tenants with different workloads, noise timings are hard to predict. Figures 3d-f show the CDF of noise inter-arrival times (one line per node). We define "noisy period", bucketed to windows of 20ms (SSD/cache) or 100ms (disk), when the observed latency is above 20ms, 1ms, and 0.05ms for disk, SSD, and OS cache, respectively. If noises exhibit a high temporal locality, all lines would have a vertical spike at x=0. However,

the figures show that noises come and go at various intervals. Noise inter-arrival distributions also vary across nodes.

Observation #3: *Mostly only 1-2 nodes (out of 20) are busy simultaneously.* This is the most important finding that motivates MITTOS. Figure 3 shows the probability of X nodes busy simultaneously (across the 20 nodes), which diminishes rapidly as X increases. For example, only 1 node is busy in 25% of the time and only 2 nodes are busy in 5% of the time. Thus, almost all the time, there are less-busy replicas to failover to.

7 EVALUATION

We now evaluate MITTCFQ, MITTSSD, and MITTCACHE (with the data set up in the disk, SSD, and OS buffer cache, respectively). We use YCSB [18] to generate 1KB key-value `get()` operations, create a noise injector to emulate noisy neighbors, and deploy 3 MongoDB nodes for microbenchmarks, 20 nodes for macrobenchmarks, and the same number of nodes for the YCSB client nodes. Data is always replicated across 3 nodes; thus, every `get()` request has three choices. For MITTCFQ and MITTCACHE, each node runs on an Emulab d430 machine (two 2.4GHz 8-core E5-2630 Haswell with 64GB DRAM and 1TB SATA disk). For MITTSSD, we only have one machine with an OpenChannel SSD (4GHz 8-core i7-6700K with 32GB DRAM and 2TB OpenChannel SSD with 16 internal channels and 128 flash chips).

All the latency graphs in Figures 4 to 12 show the latencies obtained from the client `get()` requests. In the graphs, "`NoNoise`" denotes no noisy neighbors, "`Base`" denotes vanilla MongoDB running on vanilla Linux with noise injections, and "`MittOS`" or "`Mitt`" prefix denotes our modified MongoDB running on MITTOS with noise injections. In most of the graphs, even in `NoNoise`, there are tail latencies at p99.8-p100 with a max of 50ms; our further investigation shows three causes: around 0.03% is caused by YCSB (Java) stack, 0.08% by Emulab network contention, and 0.09% by disk being slow (all of which we do not control at this point).

7.1 Microbenchmark Results

The goal of the following experiments is to show that MITTOS can successfully detect the contention, return `EBUSY` instantly, and allow MongoDB to failover quickly. We setup a 3-node MongoDB cluster and run our noise injector on one replica node. All `get()` requests are initially directed to the noisy node.

MITTCFQ: Figure 4a shows the results for MITTCFQ in three lines. First, without contention (`NoNoise`), almost all `get()` requests can finish in <20ms. Second, with the noise, vanilla MongoDB+Linux (`Base`) experiences the noise impacts (tail latencies starting at p80); the noise injector runs 4 threads of 4KB random reads, but with less priority than

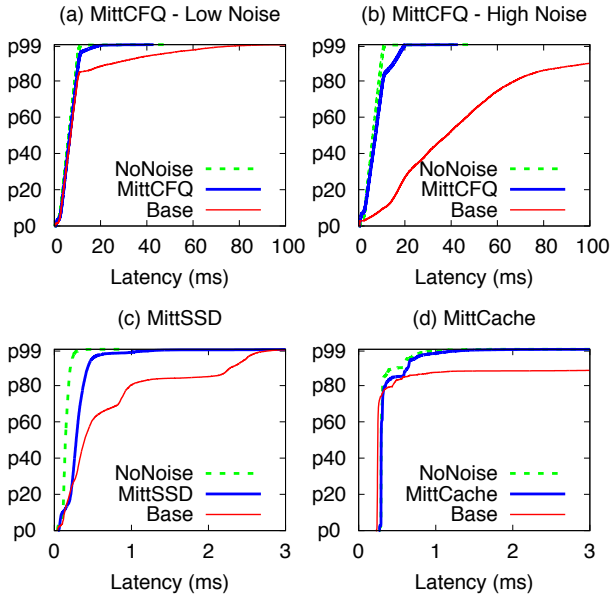

Figure 4: Latency CDFs from microbenchmarks. *The figures are explained in Section 7.1.*

MongoDB. Third, still with the same noise but now with MITTCFQ, MongoDB receives `EBUSY` (with a deadline of 20ms) and retries quickly to another replica, cutting tail latencies towards the `NoNoise` line.

Figure 4b uses the same setup above but now the noise IOs have a higher priority than MongoDB's. The performance of vanilla MongoDB (`Base`) is severely impacted (a deviation starting at p0). However, MITTCFQ detects that the in-queue MongoDB's IOs are often less picked than the new high-priority IOs, hence quickly notifying MongoDB of the disk busyness.

MITTSSD: Figure 4c shows the results for MITTSSD (note that we use our lab machine for this one with a local client). First, SSD can serve the requests in <0.2ms (`NoNoise`). Second, when read IOs are queued behind write IOs (the noise), the latency variance is high (`Base`); the noise injector runs a thread of 64KB writes. Third, with MITTSSD, MongoDB instantly reroutes the IOs that cannot be served in 2ms (the small gap between `Base` and `MittSSD` lines is the cost of software failover).

MITTCACHE: Figure 4d shows the results for MITTCACHE. First, in-memory read can be done in µs, but the network hop takes 0.3ms (`NoNoise`). Second, as we throw away about 20% of the cached data (with `posix_fadvise`), some memory accesses trigger page faults and must wait for disk access (tail latencies at p80). Finally, with MITTCACHE, MongoDB can first check whether the to-be-accessed memory region is in the cache and rapidly retry elsewhere if the data is not in the cache, removing the 20% long tail.

7.2 MittCFQ Results with EC2 Noise

Our next goal is to show the potential benefit of MittOS in a real multi-tenant cluster. We note that MittOS is targeted for deployment at the host OS (and VMM) level for full visibility of resource queues. For this reason, we do not run experiments on EC2 as there is no access to the host OS level (running MittOS as a guest OS will not be effective as MittOS cannot observe the contention from other VMs). Instead, to mimic a multi-tenant cluster, in this evaluation section, we apply EC2 noise distributions (§6) to our testbed. Later (§7.8.1), we will also inject noises with macrobenchmarks and production workloads.

Methodology: We deploy a 20-node MongoDB disk-based cluster, with 20 concurrent YCSB clients sending `get()` requests across all the nodes. For the noise, we take a 5-minute timeslice from the EC2 disk latency distribution across the 20 nodes (Figure 3a). We run a multi-threaded *noise injector* (in every node) whose job is to emulate busy neighbors at the right timing. For example, if in node n at time t, the EC2 data shows a 30ms latency (while no noise is around 6ms), then the noise injector will add IO noises that will make the disk busy for 24ms (*e.g.*, by injecting two concurrent 1MB reads, where each will add 12ms delay).

Other techniques compared: Figure 5a shows the comparisons of MittCFQ with other techniques such as hedged requests, cloning, and application timeout. **Base:** As usual, we first run vanilla MongoDB+Linux under a typical noise condition; we will use *13ms*, the *p95* latency (Figure 5a) for deadline and timeout values below. **Hedged requests:** This is a strategy where a secondary request is sent after "the first request [try] has been outstanding for more than the 95^{th}-percentile expected latency, [which] limits the additional load to approximately 5% while substantially shortening the latency tail" [19]. More specifically, if the first try does not return in 13ms, MongoDB will make a 2nd try to another replica and take the first one to complete (the first try is *not* cancelled). **Cloning:** Here, for every user request, MongoDB duplicates the request to two random replica nodes (out of three choices) and picks the first response. **Application timeout (TO):** Here, if the first try does not finish in 13ms, MongoDB will cancel the first try and make a second try, and so on. With MittCFQ and application timeout, the third try (rare, as alluded in Figure 3g) disables the timeout; otherwise, users can undesirably get IO errors.

Results: We discuss Figure 5a from right- to left-most lines. First, as expected, `Base` suffers from long tail latencies (>40ms at p98), as occasionally the requests are "unlucky" and hit a busy replica node.

Second, application timeout (`AppTO` line) must *wait* at least 13ms delay before reacting. The 2nd try will take at least another few ms for a disk read, hence `AppTO` still exhibits around >20ms tail latencies above p95.

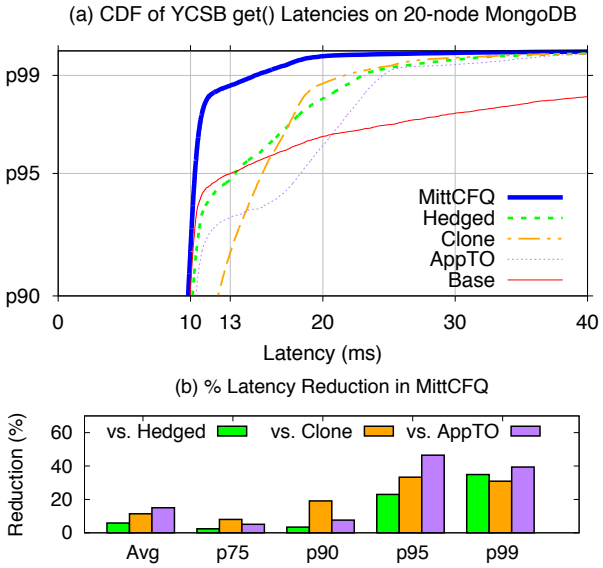

Figure 5: MittCFQ results with EC2 noise. *The figures are explained in Section 7.2.*

Third, cloning is better than timeout (`Clone` vs. `AppTO`) but only above p95. This is because cloning can pick the faster of the two concurrent requests. However, below p93 to p0, cloning is worse. This is because cloning increases the load by 2x, hence creating a self-inflicting noise in common cases.

Fourth, hedged strategy proves to be effective. It does not significantly increase the load (below p95, `Hedged` and `Base` are similar), but it effectively cuts the long tail (the wide horizontal gap between `Hedged` and `Base` lines above p95). However, we can still observe that hedged's additional load slightly delays other requests (`Hedged` is slightly worse than `Base` between p92 and p95).

Finally, MittCFQ *is shown to be more effective.* Our most fundamental principle is that the first try does *not* need to wait if the OS cannot serve the deadline. As a result, there is a significant latency reduction above p95. To quantify our improvements, the bar graph in Figure 5b shows (at specific percentiles) the % of latency reduction that MittCFQ achieved compared to the other techniques.[2] For example, at p95, MittCFQ reduces the latency of `Hedged`, `Clone`, and `AppTO` by 23%, 33%, and 47%, respectively. There is also a pattern that the higher the percentiles, MittCFQ's latency reductions are more significant.

7.3 Tail Amplified by Scale (MittCFQ)

The previous section only measures the latency of every individual IO, which reflects the "component-level variability"

[2] $\%\ of\ Latency\ Reduction = (T_{Other} - T_{MittCFQ}) \ / \ T_{Other}$

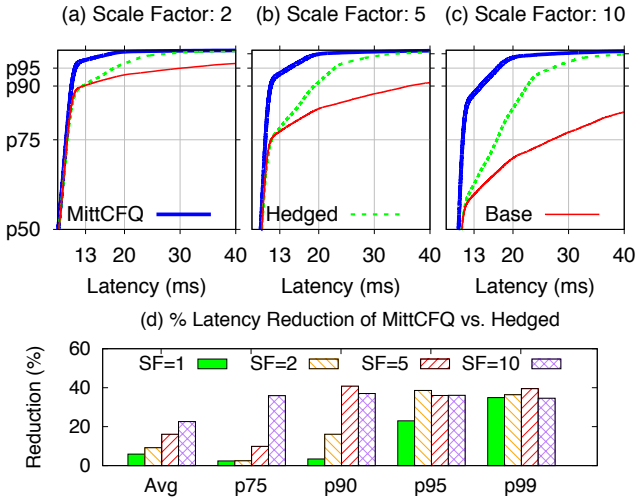

Figure 6: Tail amplified by scale (MittCFQ vs. Hedged). *The figures are explained in Section 7.3.*

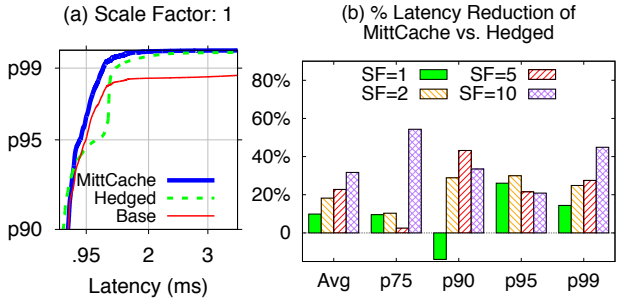

Figure 7: MittCache vs. Hedged. *The figures are explained in Section 7.4. The left figure shows the same CDF plot as in Figure 5a and the right figure the same %reduction bar plot as in Figure 6b.*

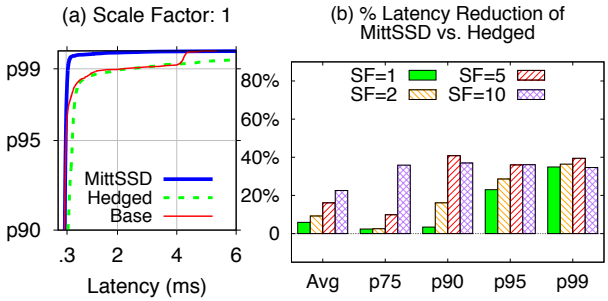

Figure 8: MittSSD vs. Hedged. *The figures are explained in Section 7.5. The left figure shows the same plot as in Figure 5a and the right figure the same %reduction bar plot as in Figure 6b.*

in every disk/node. In other words, so far, a user request is essentially a single `get()` request. However, component-level variability can be amplified by scale [19]. For example, a user request might need to fetch N data items by submitting S parallel `get()` requests and then must wait until all the S items are fetched.

To show latency tail amplified by scale, we introduce "*SF*," a scale factor of the parallel requests; for example, *SF*=5 means a user request is composed of 5 parallel `get()` requests to different nodes. Figure 5a in the previous section essentially uses *SF*=1. Figures 6a-c show the same figures as in Figure 5a, but now with scaling factors of 2, 5, and 10. Since, hedged requests are more optimum than cloning or application timeout, we only compare MittCFQ with `Hedged`. We make two important observations from Figure 6.

First, again, hedged requests must wait before reacting. If before, with *SF*=1 (Figure 5a), there are 5% (p95) of requests that must wait for 13ms, now with *SF*=2, there are roughly 10% (p90) of requests that cannot finish by 13ms (the `Hedged` line in Figure 6a). Similarly, with *SF*=5 and 10, there are around 25% (p75) and 40% (p60) of requests that must wait for 13ms, as shown in Figures 6b-c.

Second, MittCFQ initially already cuts the tail latencies of the individual IOs significantly; with *SF*=1, only 1% (p99) of the IOs are above 13ms (Figure 5a). Thus, with scaling factors of 5 and 10, only 5% and 10% of IOs are above 13ms (`MittCFQ` lines in Figures 6b-c).

The bar graph in Figure 6d summarizes the % of latency reduction achieved by MittCFQ from the `Hedged` strategy across the three scaling factors. With *SF*=10, MittCFQ already reduces the latency by 36% starting at *p75*. Thus, MittCFQ reduces the overall average latency of `Hedged` by 23% (x=`Avg`, *SF*=10 bar); note that "average" denotes the average latencies of *all* IOs (not just the high-percentile ones).

7.4 MittCache Results with EC2 Noise

Similarly, Figure 7 shows the success of MittCache (20-node results). All the data were originally in memory, but we swapped out *P*% of the cached data, where *P* is based on the cache-miss rate in Figure 3c. Another method to inject cache misses is by running competing IO workloads, but such setup is harder to control in terms of achieving a precise cache miss rate. For this reason, we perform manual swapping. In terms of the deadline, we use a small value such that `addrcheck` returns `EBUSY` when the data is not cached. In Figure 7b, at p90 and *SF*=1, our reduction is negative; our investigation shows that this is from the uncontrollable network latency (which dominates the request completion time). Similarly, in Figure 7a, between p95 and p98, `Hedged` is worse than `Base` due to the same networking reason.

7.5 MittSSD Results with EC2 Noise

Unlike in prior experiments where we use 20 nodes, for MittSSD, we can only use our single OpenChannel SSD in one machine with 8 core-threads. We carefully (a) partition the SSD into 6 partitions with no overlapping channels,

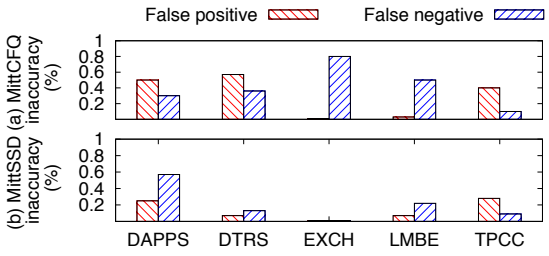

Figure 9: Prediction inccuracy. *(As explained in §7.6).*

hence no contention across partitions, (b) set up 6 MongoDB nodes/processes on a single machine serving only 6 concurrent client requests, each mounted on one partition, (c) pick noise distributions only from 6 nodes in Figure 3b, and (d) set the deadline to the p95 value, which is 0.3ms (as there is no network hop).

While latency is improved with MITTOS (the gap between `MittSSD` and `Base` in Figure 8a), we surprisingly found that hedge (`Hedged` line) is worse than the baseline. After debugging, we found another limitation of hedge (in MongoDB architecture). In MongoDB, the server creates a request handler for every user, thus 18 threads are created (for 6 clients connecting to 3 replicas). In stable state, only 6 threads are busy all the time. But for 5% of the requests (after the timeout expires), the workload intensity doubles, making 12 threads busy simultaneously (note that SSD is fast, thus processes are not IO bound). These hedge-induced CPU contentions (12 threads on a 8-thread machine) cause the long tail. Figure 8b shows the resulting % of latency reduction.

7.6 Prediction Accuracy

Figure 9 shows the results of MITTCFQ and MITTSSD accuracy tests. For a more thorough evaluation, we use 5 real-world block-level traces from Microsoft Windows Servers (the details are publicly available [35, §III][3]), choose the busiest 5 minutes, and replay them on just one machine. For a fairer experiment, as the traces were disk-based, we re-rate the trace 128x more intensive (128 chips) for SSD tests. For each trace, we always use the p95 value for the deadline.

The % of inaccuracy includes: false positives (`EBUSY` is returned, but $T_{processActual} \leq T_{deadline}$) and false negatives (`EBUSY` is not returned, but $T_{processActual} > T_{deadline}$). During accuracy tests, `EBUSY` is actually *not* returned; if error is returned, the IO is not submitted to the device, hence the actual IO completion time cannot be measured, which is also the reason why we cannot report accuracy numbers in real experiments. Instead, we attach `EBUSY` flag to the IO descriptor, thus upon IO completion, the accuracy can be measured.

Figure 9 shows the % of false positives and negatives over all IOs. In total, MITTCFQ inaccuracy is only 0.5-0.9%. Without our precision improvements (§4.2), its inaccuracy can be as high as 47%. MITTSSD inaccuracy is also only up to 0.8%. Without the improvements (§4.3), its inaccuracy can rise up to 6% (no hard-to-predict disk seek time). The next question is how far our predictions are off *within* the inaccurate IO population. We found that all the "diff"s are <3ms and <1ms on average, for disk and SSD respectively. We leave further optimizations as future work.

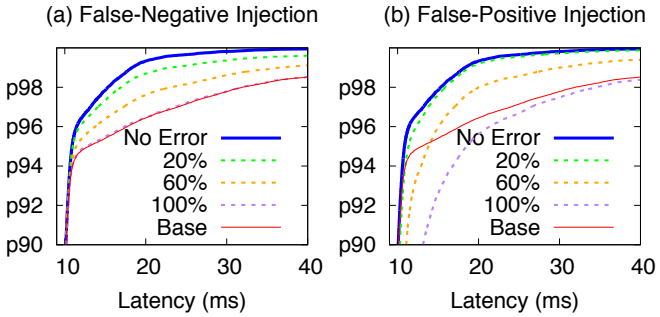

Figure 10: Tail sensitivity to prediction error. *The figures are described in Section 7.7.*

7.7 Tail Sensitivity to Prediction Error

High prediction accuracy depends on detailed and complex device performance model. This raises the question whether a simpler model (but lower accuracy) can also be effective. To investigate this, we use the same MITTCFQ experiment in Section 7.2 but now we vary MITTCFQ's prediction accuracy by injecting false-negative and false-positive errors: **(a)** False-negative injection implies that when MITTOS decides to cancel an IO and return `EBUSY`, it has *E*% chance to let the IO continue and not return `EBUSY`. Figure 10a shows the latency implications when the false-negative rate is varied (*E*=20-100). "No Error" denotes the current MITTCFQ's accuracy and 100% false negative reflects the absence of MITTOS (*i.e.*, similar to `Base`). **(b)** False-positive injection implies that when IO actually can meet the deadline, MITTOS instead will return `EBUSY` at *E*% rate. Figure 10b shows the latency results with varying false-positive rates.

In both cases, results show that higher accuracy produces shorter latency tail. False-negative errors only affect slow requests, thus even 100% error rates only reduce MITTOS performance to that of `Base`. On the other hand, false-positive errors matter less at 20% rates, but at higher rates they trigger large numbers of unnecessary failovers. In Figure 10b, with 100% false-positive rate, all IOs are retried, creating much worse latency tail than `Base`.

7.8 Other evaluations

7.8.1 Workload Mix. We ran experiments that colocate MITTOS + MongoDB with filebench and Hadoop Facebook workloads [16]. We deployed filebench's fileserver, varmail,

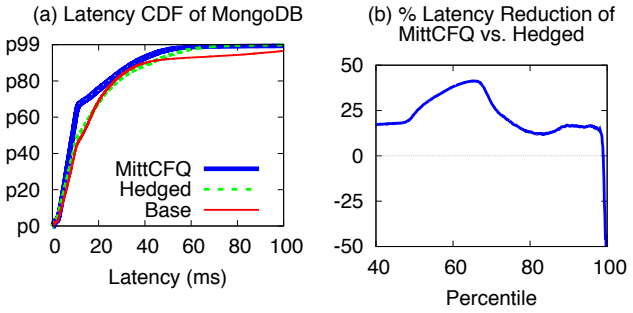

Figure 11: MITTCFQ with macrobenchmarks and production workloads. *The figures are discussed in Section 7.8.1.*

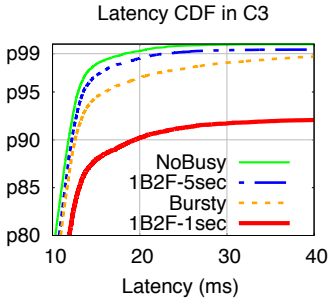

Figure 12: C3 and bursty noises. *The figure is described in Section 7.8.3.*

and webserver macrobenchmarks on different nodes (creating different levels of noise) and the first 50 Hadoop jobs from the Facebook 2010 benchmark [16]. Figure 11a shows the resulting performance of MITTCFQ, `Hedged` and `Base`. The `Base` line shows that almost 15% of the IOs experience long tail latencies (x>40ms above p85). `Hedged` shortens the latency tail, but MITTCFQ is still more effective.

The y-axis of Figure 11b shows the % of latency reduction achieved by MITTCFQ compared to `Hedged` at every percentile (*i.e.*, an interpose layout of Figure 11a). The reduction is positive in overall (up to 41%), but above p99, `Hedged` is faster. This is because the intensive workloads make our MongoDB perform 3rd retries (with deadline disabled) in 1% of the IOs, but the 3rd choices were busier than the first two. On the other hand, in the `Hedged` case, it only tries to the 2nd replica, and in this 1% case, the 2nd choices were less busy than the 3rd ones. This problem can be addressed by extending MITTOS interface to return the expected wait time, with which MongoDB can choose the shortest wait time when all replicas return `EBUSY`.

7.8.2 vs. Tied Requests. One evaluation that we could not fully perform is the comparison with the "tied requests" approach [19, pg77]. In this approach, a user request is cloned to another server with a small delay and both requests are tagged with the identity of the other server ("tied"). When one of them "*begins execution*," it sends a cancellation message to the other one. This approach was found very effective for a cluster-level distributed file system [19].

We attempted to build a similar mechanism, however MongoDB does not manage any IO queues (unlike the distributed file system mentioned above). All incoming requests are submitted directly to the OS and subsequently to the device queue, which raises three complexities. First, we found that most requests do not "linger" in the OS-level queues (§4.1-4.2); instead, the device quickly absorbs and enqueues them in the device queue. As an implication, it is not easy to know when precisely an IO is served by the device (*i.e.*, the "begin execution" time). Device queue is in fact "invisible" to the OS; MITTOS can only record their basic information (size, offset, predicted wait time, etc.). Finally, it is not easy to build a "begin-execution" signal path from the OS/device layer to the application; such signal cannot be returned using the normal IO completion/`EBUSY` path, thus a callback must be built and the application must register a callback. For these reasons, we did not complete the evaluation with tied requests.

7.8.3 vs. Snitching/Adaptivity. Many distributed storage systems employ a "choose-the-fastest-replica" feature. This section shows that such existing feature is not effective in dealing with millisecond dynamism. Specifically, we evaluated Cassandra's snitching [1] and C3's adaptive replica selection [52] mechanisms. As C3 improves upon Cassandra [52], we only show C3 results for graph clarity. Figure 12 shows that under sub-second burstiness, unlike MITTOS, neither Cassandra nor C3 can react to the bursty noise (the gap between `NoBusy` and `Bursty` lines). When we create another scenario where one replica/disk is extremely busy (1B) and two are free (2F) in 1-second rotating manner, the tail latencies become worse (the `1B2F-1sec` line is below p90). We then decrease the noise rotating frequency and found that they only perform well if the busyness is stable (a busy rotation in every 5 seconds), as shown in the `1B2F-5sec` line.

7.8.4 MittOS-powered LevelDB+Riak. Figure 13 shows the result of MITTOS integration to LevelDB+Riak (§5). For this experiment, we primarily evaluate MITTCFQ with disk-based IOs and use the same EC2 disk noise distribution. Figure 13a shows the resulting latency CDF (similar to earlier experiments), showing that MITTCFQ can also help LevelDB+Riak to cut the latency tail. Figure 13b depicts the situation over time from the perspective of a single node. In this node, when the noise is high (the high number of outstanding IOs make the deadline cannot be met), MITTOS in this node will return `EBUSY` (the triangle points). But when the noise does not break the deadline, `EBUSY` is not returned.

7.8.5 All in One. Finally, we enable MITTCFQ, MITTSSD, and MITTCACHE in one MongoDB deployment with 3 users whose data are mostly in the disk, SSD (flash cache),

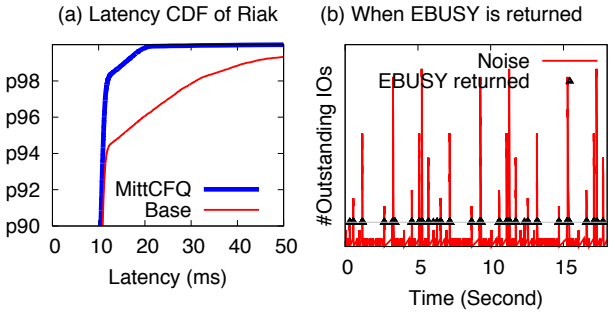

Figure 13: MITTOS-powered Riak+LevelDB. *The figure is explained in Section 7.8.4.*

and OS cache, with three different deadlines, 20 ms, 2 ms, and 0.1 ms for the three users, respectively. The SSD is mounted as a flash cache (with Linux `bcache`) between the OS cache and the disk, thus our MongoDB still runs on one partition. On one replica node, we simultaneously injected three different noises, disk contentions, SSD background writes, and page swapouts. Put simply, we combine the earlier microbenchmarks (§7.1) into a single deployment. We obtained results similar to Figure 4, showing that all MITTOS resource managements can co-exist.

7.8.6 Writes Latencies. Our work only addresses read tail latencies for the following reasons. In many storage frameworks (MongoDB, Cassandra, etc.), writes are first buffered to memory and flushed in the background, thus user-facing write latencies are not directly affected by drive-level contention. Even if the application flushes writes, most modern drives employ (capacitor-backed) NVRAM to absorb writes quickly and persistently. We ran YCSB write-only workloads with disk noise and found that the `Base` and `NoNoise` latency lines are very close to each other.

8 DISCUSSIONS

8.1 MITTOS Limitations

We identify two fundamental MITTOS limitations: **(1)** First, besides hardware/resource-level queueing delays, the software stack can also induce tail latencies. For example, an IO path can traverse a rare code path that triggers a long lock contention or inefficient loops. Such corner-case paths are hard to foresee. **(2)** Second, while rare, hardware performance can degrade over time due to many factors [21, 26, 27], or the other way around, performance can improve as device wears out (*e.g.*, faster SLC programming time as gate oxide weakens [24]). This suggests that latency profiles must be recollected over time; a sampling runtime method can be used to catch a significant deviation.

In terms of design and implementation, our current version has the following limitations (which can be extended in the future): **(1)** Currently, applications only pass deadline latencies to MITTOS. Other forms of SLO information such as throughput [53] or statistical latency distribution [37] can be included as input to MITTOS. Furthermore, applications must set precise deadline values, which could be a major burden. Automating the setup of deadline/SLO values in general is an open research problem [34]. For example, too many `EBUSY`s imply that the deadline is too strict, but rare `EBUSY`s and longer tail latencies imply that the deadline is too relaxed. The open challenge is to find a "sweet spot" in between, which we leave for future work. **(2)** MITTOS essentially returns a binary information (`EBUSY` or success). However, applications can benefit from richer responses, for example, predicted wait time (§7.8.1) or certainty/confidence of how close to or far from the deadline the prediction is. **(3)** Our performance models require white-box knowledge of the devices and resources queueing policies. However, many commodity disks and SSDs do not expose their complex firmware logic. Simpler and more generic device models can be explored, albeit with higher prediction errors (§7.7).

8.2 Beyond the Storage Stack

We believe that MITTOS principles are powerful and can be applied to many other resource managements such as CPU, runtime memory, and SMR drive managements.

In EC2, CPU-intensive VMs can contend with each other. The VMM by default sets a VM's CPU timeslice to 30ms, thus user requests to a frozen VM will be parked in the VMM for tens of ms [56]. With MITTOS, the user can pass a deadline through the network stack, and when the message is received by the VMM, it can reject the message with `EBUSY` if the target VM must still sleep more than the deadline time.

In Java, a simple "`x = new Request()`" can stall for *seconds* if it triggers GC. Worse, *all* threads on the same runtime must stall. There are ongoing efforts to reduce the delay [40, 43], but we find that the stall cannot be completely eliminated; in the last 3 months, we study the implementations of many Java GC algorithms and find that `EBUSY` exception cannot be easily thrown for the GC-triggering thread. MITTOS has the potential to transform future runtime memory management.

Similar to GC activities in SSDs, SMR disk drives must perform "band cleaning" operations [23], which can easily induce tail latencies to applications such as SMR-backed key-value stores [41, 46]. MITTOS can be applied naturally in this context, also empowered by the development of SMR-aware OS/file systems [9].

8.3 Other Discussions

With MittOS*, should other tail-tolerant approaches be used?* We believe MITTOS handles a major source of storage tail latencies (*i.e.*, storage device contention). Ideally, MITTOS

is applied to all major resources as discussed above. If MITTOS is only applied to a subset of the resources (*e.g.*, storage stack only), then other approaches such as hedged/tied requests are still needed and can co-exist with MITTOS.

Can MittOS*' fast replica switching cause inconsistencies?* MITTOS encourages fast failover, however many NoSQL systems support eventually consistency and generally attempt to minimize replica switching to ensure monotonic reads. MITTOS-powered NoSQL can be made more conservative about switching replicas that may lead to inconsistencies (*e.g.*, do not failover until the other replicas are no longer stale).

Can MittOS *expose side channels?* Some recent works [33, 63] show that attackers can use information exposed by the OS (*e.g.*, CPU scheduling [33], cache behavior [62]). MITTOS can potentially make IO side channels more effective because an attacker can obtain a less noisy signal about the I/O activity of other tenants. We believe that even without MITTOS, attackers can deconstruct the noises by probing IO performance periodically, however more research can be conducted in this context.

9 RELATED WORK

Storage tails: A growing number of work has investigated many root causes of storage latency tail, including multi tenancy [31, 36, 42, 53], maintenance jobs [10, 15, 39], inefficient policies [30, 38, 58], device cleaning/garbage collection [8, 19, 31, 36, 57], and hardware variability [28]. MITTOS does not eliminate these root causes but rather expose the implied busyness to applications.

Storage tail tolerance: Throughout the paper, we discussed solutions such as snitching/adaptivity [1, 52], cloning at various different levels [11, 55], hedged and tied requests [19].

Performance isolation (QoS): A key to reduce performance variability is performance isolation, such as isolation of CPU [61], IO throughput [25, 50, 53], buffer cache [42], and end-to-end resources [12, 17]. QoS-enforcements do not return busy errors when SLOs are not met; they provide "best-effort fairness." MITTOS is *orthogonal* to this class of work (§3.3).

OS transparency: MITTOS in spirit is similar to other works that advocate more information exposure to applications [13] and first-class supports for interactive applications [59]. MITTOS provides busyness transparency by slightly modifying the user-kernel interfaces (mainly for passing deadlines and returning `EBUSY`).

10 CONCLUSION

Existing application-level tail-tolerant solutions can only guess at resource busyness. We propose a new philosophy: OS-level SLO awareness and transparency of resource busyness, which eases applications' tail and other performance management. In a world where consolidation and sharing are a fundamental reality, busyness transparency, as embodied in the MITTOS principles, should only grow in importance.

11 ACKNOWLEDGMENTS

We thank Ramakrishna Kotla, our shepherd, and the anonymous reviewers for their tremendous feedback. This material was supported by funding from NSF (grant Nos. CCF-1336580, CNS-1350499, CNS-1526304, CNS-1405959, and CNS-1563956) as well as generous donations from Huawei, Dell EMC, Google Faculty Research Award, NetApp Faculty Fellowship, and CERES Center for Unstoppable Computing.

A APPENDIX: DETAILS

This section describes how we compute $T_{nextFree}$ (`freeTime`) as discussed in§4.1-4.2. For the noop scheduler (§4.1), we model the disk queue `Q` as a list of outstanding IOs {`ABCD...`}. Thus, the `Q`'s wait time (`qTime`) is the total *seek cost* of every consecutive IO pairs in the queue, which will be added to `freeTime`:

```
qTime += (seekCost(A,B) + seekCost(B,C) + ...);
freeTime += qTime;
```

We then model the `seekCost` from IO `X` to `Y` as follows:

```
seekCost(X,Y) =
  seekCostPerGB * (Y.offsetInGB - X.offsetInGB) +
  transferCostPerKB * X.sizeInKB;
```

The `transferCostPerKB` is currently simplified as a constant parameter. The `seekCostPerGB` parameter is more challenging to set due to the complexity of disk geometry. To profile a target disk, we measure the latency (seek cost) of all pairs of random IOs per GB distance. For example, for a 1000 GB disk, we fill a matrix `seekCostPerGB[X][Y]`, where `X` and `Y` range from 1 to 1000, a total of 1 million items in the matrix. We profile the disk with 10 tries and use linear regression for more accuracy.

The next challenge is to model the queueing policy. For noop (FIFO) scheduler, the order of IOs do not change. However, as the IOs are submitted to the disk, the disk has its own disk queue and will reorder the IOs. Existing works already describe how to characterize disk policies [48, 49]. For example, we found that our target disk exhibits SSTF policy. Thus, to make `freeTime` more accurate, `qTime` should be modeled into an SSTF ordering (`sstfTime`), for example if the disk head position (known from the last IO completed) is currently near `D`, then the `freeTime` should be:

```
sstfTime += (seekCost(D,C) + seekCost(C,B) + ...);
freeTime += sstfTime;
```

For CFQ, the modeling is more complex as there are two-level of queues: the CFQ queues (based on priority and fairness) and the disk queue (SSTF). Thus, we predict two queueing wait times `cfqTime` and `sstfTime`. Predicting `cfqTime` is explained in Section 4.2.

REFERENCES

[1] Cassandra Snitches. http://docs.datastax.com/en/archived/cassandra/2.0/cassandra/architecture/architectureSnitchesAbout_c.html.

[2] CFQ (Complete Fairness Queueing). https://www.kernel.org/doc/Documentation/block/cfq-iosched.txt.

[3] SNIA IOTTA: Storage Networking Industry Association's Input/Output Traces, Tools, and Analysis Trace Repository. http://iotta.snia.org.

[4] Tuning Linux I/O Scheduler for SSDs. https://www.nuodb.com/techblog/tuning-linux-io-scheduler-ssds.

[5] Google: Taming The Long Latency Tail - When More Machines Equals Worse Results. http://highscalability.com/blog/2012/3/12/google-taming-the-long-latency-tail-when-more-machines-equal.html, 2012.

[6] The case against SSDs. http://www.zdnet.com/article/the-case-against-ssds/, 2015.

[7] Open-Channel Solid State Drives. http://lightnvm.io/, 2016.

[8] Abutalib Aghayev and Peter Desnoyers. Skylight-A Window on Shingled Disk Operation. In *Proceedings of the 13th USENIX Symposium on File and Storage Technologies (FAST)*, 2015.

[9] Abutalib Aghayev, Theodore Ts'o, Garth Gibson, and Peter Desnoyers. Evolving Ext4 for Shingled Disks. In *Proceedings of the 15th USENIX Symposium on File and Storage Technologies (FAST)*, 2017.

[10] George Amvrosiadis, Angela Demke Brown, and Ashvin Goel. Opportunistic Storage Maintenance. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[11] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Effective Straggler Mitigation: Attack of the Clones. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[12] Sebastian Angel, Hitesh Ballani, Thomas Karagiannis, Greg O'Shea, and Eno Thereska. End-to-end Performance Isolation Through Virtual Datacenters. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[13] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Nathan C. Burnett, Timothy E. Denehy, Thomas J. Engle, Haryadi S. Gunawi, James A. Nugent, and Florentina I. Popovici. Transforming Policies into Mechanisms with Infokernel. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[14] Matias Bjørling, Javier González, and Philippe Bonnet. LightNVM: The Linux Open-Channel SSD Subsystem. In *Proceedings of the 15th USENIX Symposium on File and Storage Technologies (FAST)*, 2017.

[15] Eric Brewer. Spinning Disks and Their Cloudy Future (Keynote). In *Proceedings of the 14th USENIX Symposium on File and Storage Technologies (FAST)*, 2016.

[16] Yanpei Chen, Sara Alspaugh, and Randy H. Katz. Interactive Analytical Processing in Big Data Systems: A Cross-Industry Study of MapReduce Workloads. In *Proceedings of the 38th International Conference on Very Large Data Bases (VLDB)*, 2012.

[17] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F. Wenisch. The Mystery Machine: End-to-end Performance Analysis of Large-scale Internet Services. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[18] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*, 2010.

[19] Jeffrey Dean and Luiz Andre Barroso. The Tail at Scale. *Communications of the ACM*, 56(2), February 2013.

[20] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[21] Thanh Do, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, and Haryadi S. Gunawi. Limplock: Understanding the Impact of Limpware on Scale-Out Cloud Systems. In *Proceedings of the 4th ACM Symposium on Cloud Computing (SoCC)*, 2013.

[22] Benjamin Farley, Venkatanathan Varadarajan, Kevin Bowers, Ari Juels, Thomas Ristenpart, and Michael Swift. More for Your Money: Exploiting Performance Heterogeneity in Public Clouds. In *Proceedings of the 3rd ACM Symposium on Cloud Computing (SoCC)*, 2012.

[23] Tim Feldman and Garth Gibson. Shingled Magnetic Recording: Areal Density Increase Requires New Data Management. *;login:*, 38(3), 2013.

[24] Laura M. Grupp, Adrian M. Caulfield, Joel Coburn, Steven Swanson, Eitan Yaakobi, Paul H. Siegel, and Jack K. Wolf. Characterizing Flash Memory: Anomalies, Observations, and Applications. In *42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42)*, 2009.

[25] Ajay Gulati, Arif Merchant, and Peter J. Varman. mClock: Handling Throughput Variability for Hypervisor IO Scheduling. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[26] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC)*, 2014.

[27] Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, and Kurnia J. Eliazar. Why Does the Cloud Stop Computing? Lessons from Hundreds of Service Outages. In *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC)*, 2016.

[28] Mingzhe Hao, Gokul Soundararajan, Deepak Kenchammana-Hosekote, Andrew A. Chien, and Haryadi S. Gunawi. The Tail at Store: A Revelation from Millions of Hours of Disk and SSD Deployments. In *Proceedings of the 14th USENIX Symposium on File and Storage Technologies (FAST)*, 2016.

[29] Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The Unwritten Contract of Solid State Drives. In *Proceedings of the 2017 EuroSys Conference (EuroSys)*, 2017.

[30] Jun He, Duy Nguyen, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Reducing File System Tail Latencies with Chopper. In *Proceedings of the 13th USENIX Symposium on File and Storage Technologies (FAST)*, 2015.

[31] Jian Huang, Anirudh Badam, Laura Caulfield, Suman Nath, Sudipta Sengupta, Bikash Sharma, and Moinuddin K. Qureshi. FlashBlox: Achieving Both Performance Isolation and Uniform Lifetime for Virtualized SSDs. In *Proceedings of the 15th USENIX Symposium on File and Storage Technologies (FAST)*, 2017.

[32] Sitaram Iyer and Peter Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[33] Suman Jana and Vitaly Shmatikov. Memento: Learning Secrets from Process Footprints. In *IEEE Symposium on Security and Privacy (SP)*, 2012.

[34] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Inigo Goiri, Subru Krishnan, Janardhan Kulkarni,

and Sriram Rao. Morpheus: Towards Automated SLOs for Enterprise Clusters. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[35] Swaroop Kavalanekar, Bruce Worthington, Qi Zhang, and Vishal Sharda. Characterization of Storage Workload Traces from Production Windows Servers. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2008.

[36] Jaeho Kim, Donghee Lee, and Sam H. Noh. Towards SLO Complying SSDs Through OPS Isolation. In *Proceedings of the 13th USENIX Symposium on File and Storage Technologies (FAST)*, 2015.

[37] Ning Li, Hong Jiang, Dan Feng, and Zhan Shi. PSLO: Enforcing the Xth Percentile Latency and Throughput SLOs for Consolidated VM Storage. In *Proceedings of the 2016 EuroSys Conference (EuroSys)*, 2016.

[38] Lanyue Lu, Yupu Zhang, Thanh Do, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Physical Disentanglement in a Container-Based File System. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[39] Christopher R. Lumb, Jiri Schindler, Gregory R. Ganger, David Nagle, and Erik Riedel. Towards Higher Disk Head Utilization: Extracting Free Bandwidth from Busy Disk Drives. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI)*, 2000.

[40] Martin Maas, Krste Asanovic, Tim Harris, and John Kubiatowicz. Taurus: A Holistic Language Runtime System for Coordinating Distributed Managed-Language Applications. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[41] Adam Manzanares, Noah Watkins, Cyril Guyot, Damien LeMoal, Carlos Maltzahn, and Zvonimr Bandic. ZEA, A Data Management Approach for SMR. In *the 8th Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2016.

[42] Michael Mesnier, Feng Chen, Tian Luo, and Jason B. Akers. Differentiated Storage Services. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[43] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. Yak: A High-Performance Big-Data-Friendly Garbage Collector. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[44] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast Crash Recovery in RAMCloud. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[45] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. SDF: Software-Defined Flash for Web-Scale Internet Storage System. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[46] Rekha Pitchumani, James Hughes, and Ethan L. Miller. SMRDB: Key-Value Data Store for Shingled Magnetic Recording Disks. In *The 8th Annual International Systems and Storage Conference (SYSTOR)*, 2015.

[47] Bryan Reinero. Selecting AWS Storage for MongoDB Deployments: Ephemeral vs. EBS. https://www.mongodb.com/blog/post/selecting-aws-storage-for-mongodb-deployments-ephemeral-vs-ebs.

[48] Chris Ruemmler and John Wilkes. An Introduction to Disk Drive Modeling. *IEEE Computer*, 27(3):17–28, March 1994.

[49] Jiri Schindler and Gregory R. Ganger. Automated Disk Drive Characterization. In *Proceedings of the 2000 ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2000.

[50] David Shue, Michael J. Freedman, and Anees Shaikh. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[51] Riza O. Suminto, Cesar A. Stuardo, Alexandra Clark, Huan Ke, Tanakorn Leesatapornwongsa, Bo Fu, Daniar H. Kurniawan, Vincentius Martin, Uma Maheswara Rao G., and Haryadi S. Gunawi. PBSE: A Robust Path-Based Speculative Execution for Degraded-Network Tail Tolerance in Data-Parallel Frameworks. In *Proceedings of the 8th ACM Symposium on Cloud Computing (SoCC)*, 2017.

[52] P. Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection. In *Proceedings of the 12th Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[53] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. IOFlow: A Software-Defined Storage Architecture. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[54] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[55] Zhe Wu, Curtis Yu, and Harsha V. Madhyastha. CosTLO: Cost-Effective Redundancy for Lower Latency Variance on Cloud Storage Services. In *Proceedings of the 12th Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[56] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. Bobtail: Avoiding Long Tails in the Cloud. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[57] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs. In *Proceedings of the 15th USENIX Symposium on File and Storage Technologies (FAST)*, 2017.

[58] Suli Yang, Tyler Harter, Nishant Agrawal, Salini Selvaraj Kowsalya, Anand Krishnamurthy, Samer Al-Kiswany, Rini T. Kaushik, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Split-Level I/O Scheduling. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[59] Ting Yang, Tongping Liu, Emery D. Berger, Scott F. Kaplan, and J. Eliot B. Moss. Redline: First Class Support for Interactivity in Commodity Operating Systems. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[60] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[61] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, vrigo Gokhale, and John Wilkes. CPI2: CPU performance isolation for shared compute clusters. In *Proceedings of the 2013 EuroSys Conference (EuroSys)*, 2013.

[62] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM Side Channels and Their Use to Extract Private Keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS)*, 2012.

[63] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *Proceedings of the 2014 ACM Conference on Computer and Communications Security (CCS)*, 2014.