THE UNIVERSITY OF CHICAGO

SUPPORTING MILLISECOND TAIL TOLERANCE WITH FAST REJECTING SLO-AWARE OS INTERFACE

A PROPOSAL SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
MINGZHE HAO

CHICAGO, ILLINOIS
2018
# TABLE OF CONTENTS

LIST OF FIGURES ............................................................... iii
LIST OF TABLES ................................................................. iv
ABSTRACT ................................................................. v
1 THESIS STATEMENT ....................................................... 1
2 INTRODUCTION ............................................................. 2
3 MITTOS OVERVIEW ......................................................... 4
  3.1 Deployment Model and Use Case .................................. 4
  3.2 Use Case .......................................................... 4
  3.3 Goals / Principles ................................................ 5
  3.4 Design Challenges .............................................. 6
4 CASE STUDIES .......................................................... 8
  4.1 Disk CFQ Scheduler (MITT CFQ) ................................. 8
  4.2 SSD Management (MITT SSD) ....................................... 9
5 EVALUATION ............................................................. 12
  5.1 MITT CFQ Results with EC2 Noise .............................. 12
  5.2 MITT SSD Results with EC2 Noise ............................... 15
  5.3 Prediction Accuracy ............................................. 16
6 DPOS ................................................................. 17
  6.1 Previous Work on Black-Box Latency Prediction ............... 18
  6.2 DPOS Challenge .................................................... 19
  6.3 Preliminary evaluation .......................................... 20
    6.3.1 Model with surplus knowledge ............................ 21
    6.3.2 Model with some knowledge .................................. 22
  6.4 Future research work .......................................... 23
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>MITTOS Deployment Model (§3)</td>
<td>5</td>
</tr>
<tr>
<td>3.2</td>
<td>MITTOS use-case illustration (§3.2)</td>
<td>5</td>
</tr>
<tr>
<td>5.1</td>
<td>MITT CFQ results with EC2 noise. <em>The figures are explained in Section 5.1.</em></td>
<td>14</td>
</tr>
<tr>
<td>5.2</td>
<td>MITT SSD vs. Hedged. <em>The figure is explained in Section 5.2.</em></td>
<td>15</td>
</tr>
<tr>
<td>5.3</td>
<td>Prediction inaccuracy. <em>(As explained in §5.3).</em></td>
<td>16</td>
</tr>
<tr>
<td>6.1</td>
<td>IO latency training data for simulated RAID-0 style SSD.</td>
<td>19</td>
</tr>
<tr>
<td>6.2</td>
<td>Collision maps of 2 IOs on simple RAID-0 style SSD with 2 chips. <em>The collision space in this simple map is too large for regression trees, as collisions and non-collisions are interleaved. For real SSDs, this map would become more complicated for tree structures to contain.</em></td>
<td>20</td>
</tr>
<tr>
<td>6.3</td>
<td>Preliminary evaluation process based on simulated SSD.</td>
<td>21</td>
</tr>
<tr>
<td>6.4</td>
<td>Neural Network to predict EBUSY on RAID-0 style SSD. <em>Besides dense layer with regular perceptrons (wx+b), this network also contains relu activation layer as the problem is about 0/1 prediction.</em></td>
<td>22</td>
</tr>
<tr>
<td>6.5</td>
<td>Neural Network to predict EBUSY on RAID-5 style SSD. <em>The topology of this model is based on the one we design in §6.3.1 for RAID-5 style SSD prediction, but all perceptrons are with regular logics.</em></td>
<td>23</td>
</tr>
</tbody>
</table>
LIST OF TABLES

6.1 Model applied in previous works for performance prediction.  . . . . . . . 18
ABSTRACT

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. Recently this problem becomes more challenging as many applications start to generate large numbers of small requests, each expected to finish in milliseconds. In this case, performance instability that produces milliseconds of delay lead to violations of such SLOs, degrading user experience and impacting revenues negatively.

We try to tackle this problem at the level of storage devices, such as disks and SSDs, by predicting millisecond-level tail latency occurrences in real-time and mitigating their impacts by quickly issuing failovers for IOs that cannot be promptly served.

To achieve that we develop MittOS, which provides operating system support to cut millisecond-level tail latencies for data-parallel applications. MittOS exposes a fast rejecting SLO-aware interface wherein applications can provide their SLOs (e.g., IO deadlines) and will promptly return EBUSY signal if these SLOs cannot be met, allowing the application to failover (retry) to another less-busy node without waiting. MittOS no-wait approach helps reduce IO completion time up to 35% compared to wait-then-speculate approaches.

Based on MittOS, we are now proposing DPOS (Deep Learning Operating System). In this project, we plan to utilize customized DNN to support performance prediction for general black-box SSDs, which are now widely applied in high-performance storage systems.
CHAPTER 1

THESIS STATEMENT

Notorious tail latency problem in storage devices are now entering millisecond-level. Tail-tolerant techniques should utilize information about underlying resource busyness from OS to better predict these tail latencies and mitigate their negative impacts in real-time.
CHAPTER 2

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) [15], 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 [9, 38, 47]. 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 [9, 43]; 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”) [14]; to achieve this, IO queueing and revocation management must be built in the application layer [11]. A more conservative alternative is “hedged requests” [14], where a duplicate request is sent after the first request is outstanding for more than, for example, the 95th-percentile expected latency; but the slow requests (5%) must wait before being retried. Finally, “snitching” [1, 39] – 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 MTTOS (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 [11, 17, 26, 31, 37, 40, 45]. In a nutshell, MTTOS provides an SLO-aware read interface, “\texttt{read(..., slo)},” such that applications can attach SLOs to their IO operations (e.g., “\texttt{read()} should not take more than 20ms”). If the SLO cannot be satisfied (e.g., long disk queue), MTTOS immediately rejects the IOs and returns \texttt{EBUSY (i.e., no wait)}, hence allowing the application to quickly failover (retry) to another node.

We evaluate our MTTOS-powered MongoDB in a 20-node cluster with YCSB workloads and the EC2 noise distribution. We compare MTTOS with three other standard practices (basic timeout, cloning, and hedged requests). Compared to hedged requests (the most effective among the three), MTTOS reduces the completion time of individual IO requests by 23-26% at p95\(^1\) and 6-10% on average.

Based on this success, we propose DPOS (Deep Learning Operating System) as an extension of MTTOS. DPOS utilizes customized DNN, which incorporates domain knowledge about underlying storage devices, to do prediction for general black-box SSDs, which are widely deployed in high-performance storage systems. Preliminary evaluation from simulated data has shown that the customized DNN is able to achieve much higher accuracy than peer models.

\footnote{1. We use “p\(Y\)” to denote the \(Y^{th}\)-percentile; for example, p90 implies the 90\(^{th}\)-percentile (\(y=0.9\) in CDF graphs).}
CHAPTER 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 3.1. 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.

3.2 Use Case

Figure 3.2 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. We use the 95\(^{th}\)-percentile latency as the deadline value, which we will discuss more in Sections 5.1, to what value a deadline should be set. ② The application then tags \texttt{read()} calls with the deadline SLO. To support this, we create a new \texttt{read()} system call that can accept application SLO (essentially one extra argument to the existing \texttt{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.

**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 3.2). Failing over to another machine (5 in Figure 3.2) only involves one more network
hop (e.g., 0.3ms in EC2 and our testbed or even 10µs with Infiniband [33]).

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 contentsions. 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 3.2); 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 3.2).

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 [22], 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).
CHAPTER 4

CASE STUDIES

The goal of this section is to demonstrate that MITTOS principles can be integrated to many resource managements such as the CFQ (§4.1) IO schedulers and SSD (§4.2) 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 CFQ Scheduler (MITT-CFQ)

We build MITT-CFQ 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, MITT-CFQ 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, MITT-CFQ 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 <5μs per IO prediction.

**Accuracy:** With the method above, MITT-CFQ 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, MITT-CFQ 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.

### 4.2 SSD Management (MITT-SSD)

Latency variability in SSD is an ongoing problem [5, 6, 14]. 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 [18, 44] or isolation [21, 26]), none of them cover all possible cases. For example, under write bursts or no idle period, read requests can still be delayed significantly [44, §6.6]. Even with isolation, occasional wear-leveling page movements will introduce a significant noise [21, §4.3].

Fortunately, not all SSDs are busy at the same time, a situation that empowers MITT-SSD. 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 [8, 28], 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 [34] is gaining popularity and publicly available (e.g., Linux LightNVM [10] 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: $T_{\text{chipNextFree}} += 100\mu s$ per new page read. That is, a page (16KB) read takes $100\mu s$ (chip read
and channel transfer); >16KB multi-page read to a chip is automatically chopped to individual page reads. Thus, \( T_{\text{wait}} = T_{\text{now}} - T_{\text{chipNextFree}} + (60\mu s \times \#IO_{\text{SameChannel}}) \). 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 60\( \mu s \) is the channel queueing delay (consistent with the 280 MBps channel bandwidth in the vendor specification). If there is an erase, \( T_{\text{chipNextFree}} += 6ms \).

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.
CHAPTER 5
EVALUATION

We now evaluate MITTCFQ and MITTSSD (with the data set up in the disk and SSD respectively). We use YCSB [13] to generate 1KB key-value \texttt{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 \texttt{get()} request has three choices. For MITTCFQ, each node runs on an Emulab d430 machine (two 2.4GHz 8-core E5-2630 Haswell with 64GB DRAM and 1TB SATA disk). For MITT-SSD, 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 show the latencies obtained from the client \texttt{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).

5.1 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 to our testbed. Later, 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 \texttt{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. 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 5.1a shows the comparisons of MITT/CFQ 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 \( 13 \text{ms} \), the \( p95 \) latency (Figure 5.1a) 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” [14]. More specifically, if the first try does not return in \( 13 \text{ms} \), 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 \( 13 \text{ms} \), MongoDB will cancel the first try and make a second try, and so on. With MITT/CFQ and application timeout, the third try (rare) disables the timeout; otherwise, users can undesirably get IO errors.

**Results:** We discuss Figure 5.1a 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 \( 13 \text{ms} \) delay before reacting. The 2nd try will take at least another few ms for a disk read, hence AppTO still exhibits around \( >20 \text{ms} \)
Figure 5.1: **MittCFQ results with EC2 noise.** The figures are explained in Section 5.1.

tail latencies above p95.

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 5.1b shows (at specific percentiles) the % of latency reduction that MittCFQ achieved compared to the other techniques. 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,

---

1. % of Latency Reduction = ($T_{Other}$ - $T_{MittCFQ}$) / $T_{Other}$
Figure 5.2: **MITTSSD vs. Hedged.** *The figure is explained in Section 5.2.*

MITTCFQ’s latency reductions are more significant.

### 5.2 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, 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 EC2 nodes with ephemeral SSDs, 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 5.2), 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 an 8-thread machine) cause the long tail.
Figure 5.3: Prediction inaccuracy. (As explained in §5.3).

5.3 Prediction Accuracy

Figure 5.3 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 [24, §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 $T_{\text{processActual}} \leq T_{\text{deadline}}$) and false negatives (EBUSY is not returned, but $T_{\text{processActual}} > T_{\text{deadline}}$). 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 5.3 shows the % of false positives and negatives over all IOs. In total, MITTCFQ inaccuracy is only 0.5-0.9%. Without our precision improvements (§4.1), its inaccuracy can be as high as 47%. MITTSSD inaccuracy is also only up to 0.8%. Without the improvements (§4.2), 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 $<3\text{ms}$ and $<1\text{ms}$ on average, for disk and SSD respectively. We leave further optimizations as future work.
CHAPTER 6

DPOS

One major drawback of MITTOS is that it requires in-depth knowledge about underlying schedulers and hardware. While it is feasible to make accurate prediction for disk drives with simple seeking algorithm plus open-source scheduler, and software-defined SSDs, MITTOS cannot work with black-box SSDs, which hide their internal information and cover most commerical SSDs.

There are many works focusing on extracting performance-related characteristics inside black-box SSDs [12, 20, 25, 27] and further improving their overall performance [30, 41]. Though successful at pulling up aggregate metrics like average latency and throughput, these works only make use of static properties (#parallelism, chunk size etc.), while most of tail-latency occurences in SSDs are caused by dynamic behaviors such as GCs and IO contentions. There are also works on predicting the performance of black-box devices with statistical and ML mechanisms [16, 19, 32, 42, 46], but they likewise target prediction of high-level metrics and cannot achieve good accuracy for individual IOs.

Techniques in these works do not help in our case as they either (1) only take advantage of aggregate metrics collected from high-level (application/system-level) or (2) focus on designing algorithms based on static properties of SSDs, neglecting dynamic detail contained at lower-level (OS/device-level), which help MITTOS achieve accurate prediction for individual IOs. Though key information like GC occurences will not be directly available in the case of black-box SSDs, the remaining subordinate pieces can still be valuable. For example, by tracking the IO events inside the schedulers, OS can know about current residents in SSD IO queues, as SSDs mostly serve IO with FIFO. As a result, OS can learn about potential contentions between IOs by looking at their latencies and try figuring out the mapping of underlying SSDs.

Given that, we propose DPOS, an enhanced version of MITTOS aiming at predicting latency for black-box SSDs, with the help of DL techniques. In the next few sections we will introduce the major challenges, the preminilary models we have built, and future research plan.
<table>
<thead>
<tr>
<th>Model Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear regression</td>
<td>Linear relation between average IO latency and #outstanding IOs</td>
</tr>
<tr>
<td>Linear regression + designated algorithm</td>
<td>Linear model for CPU, network usage and sequential IOs; Designated algorithm (non-ML) to predict occurrences of data flush</td>
</tr>
<tr>
<td>Two-level model</td>
<td>Ridge for filtering; Lasso for modeling</td>
</tr>
<tr>
<td>CART (classification and regression tree)</td>
<td>Regression tree using request-level and workload-level metrics to predict average and 90th percentile response latency</td>
</tr>
</tbody>
</table>

Table 6.1: Model applied in previous works for performance prediction.

6.1 Previous Work on Black-Box Latency Prediction

Before digging into the challenge, we first recap the models in previous works for predictions on black-box devices and double-check if they can be applied to solve our issue. The goal is to know once we substitute the high-level metrics they used with low-level information collected with OS, would these models still keep their successes.

Table 6.1 summarizes the models we have gone through in previous works[16, 19, 32, 42, 46]. To evaluate how they work in our case, we simulate a simple SSD with 8 independent chips, RAID-0 style FTL mapping (PBN = LBA % #chips), and queue-depth of 16. We assume that chips in this SSD require constant time (40us) to serve each IO. The SSD always batches incoming IOs and do not serve the next batch until the previous is finished. Training data includes the binary-format LBAs, together with the latency of last IO in each batch as target label, as illustrated in Figure 6.1.

The result is unfortunately disappointing: none of these models is able to return fair accuracy (error \( \leq 30\% \)). This is somehow expected as SSDs have more complicated internal structure. First, SSDs handle IOs in parallel, so applying regression on whole device certainly would not work. Second, RAID-0 style FTL maps LBA to PPN in a round-robin manner (in real SSDs the mapping would be more complicated), which makes the hypothesis space too large for decision trees/random forests to cover, as shown in Figure 6.2.

However, these models can still bring us insights. For example, evaluation in DBSeer[32]
Figure 6.1: **IO latency training data for simulated RAID-0 style SSD.**

shows that employing knowledge about storage systems/devices parameters and workload metrics can help improve accuracy. Inside-out[19] indicates the power of model ensemble at different levels. Exploring the feasibility of applying these insights will become an essential part of DPOS project.

### 6.2 DPOS Challenge

To achieve accurate prediction, DPOS needs to provide two answers: (1) How many operations (including background GC operations) are in contention with the one we want to predict? and (2) How long will these operations take? The first answer requires a fair understanding of the FTL, while the second one is impacted by properties of operations (IO sizes) and SSDs (performance specification).

A good understanding of FTL can lead to break down of other problems, as knowing the mapping can allow us to profile the performance specficiation of each chip, and track the IOs on it. Combining with the mechanisms to learn static properties such as buffer type/size, we can further estimate the occurrences of GCs and predict IO latency.

However, so far this task seems impossible and there is little previous work on it. On the
Figure 6.2: **Collision maps of 2 IOs on simple RAID-0 style SSD with 2 chips.** The collision space in this simple map is too large for regression trees, as collisions and non-collisions are interleaved. For real SSDs, this map would become more complicated for tree structures to contain.

contrary, some works [30, 41] start to ignore the FTL and directly map the LBA to chips. Though these works claim that this mechanism can predict GCs (with no accuracy evaluation shown) and bring improvement, theoretically it is supposed to work only when workload is read intensive and mapping is stable.

But there is still clue on interpreting FTL, modern FTLs policies usually have belong to certain categories to achieve features such as load balance, performance parallelism and wear-leveling [23, 29, 35, 36]. We believe that a combination of ML/DL models, integrated with these domain knowledge about FTL, can reach a good view of the FTL, thereby facilitating accurate latency prediction.

### 6.3 Premilinary evaluation

To have a premilinary verification of our hypothesis, we generate some IO traces, input them to a simulated SSD, and collect metrics including IO latency and EBUSY flag as target labels. Then these IO traces, metrics, together with domain knowledge about the simulated SSD, will be input to ML models. The goal is to see whether these models can achieve fair accuracy, and how their
Figure 6.3: Premilinary evaluation process based on simulated SSD.

performance varies as they get more/less hints about SSDs. The whole process is illustrated in Figure 6.3. So far we have tried simulated SSDs with RAID-0/4/5 style FTL. For these simulated SSDs we generate train/test data like the one shown in Figure 6.1. For all experiments in following subsections, we use neural network as a basis.

6.3.1 Model with surplus knowledge

We first evaluate models integrated with excess information from simulated SSDs, to see the upper bound of our models’ accuracy. To be specific, for RAID-0 style SSD, we hint the model that there is a mod operation mapping the LBA to chips, and for RAID-5 style SSD, the model further explicitly knows about the existence of rotating parity. All these hints are given by specifying operators (mod operation etc.) of certain layers in the model. Both models do not know about #chips, how long each IO takes and EBUSY threshold. We train both models with 1k - 1m random generated entries, and use separate test set for accuracy measurement. The result is not surprising. Even with only 1k training entries, both models can achieve 100% accuracy, as key operators are already provided.

The major drawback of building models in this way is quite obvious: the models are limited by
their knowledge from respective SSDs and cannot generalize even a little bit. Though it is possible to use model selection algorithms to try picking out the correct model, the space of possible models is huge. Besides, in reality there is no way to know such low-level detail (RAID-0/5 style FTL) from black-box SSDs and apply it to the model.

### 6.3.2 Model with some knowledge

Our next exploration starts from opposite side of §6.3.1: what if the model completely knows nothing about the SSDs. To find that out, we create a NN with dense layers composed of regular perceptrons (wx+b), and activation layers, as shown in Figure 6.4.

This model is able to achieve 100 lower digits of each LBA to chips. We further try this model on RAID-5 style SSD, which incorporates more complicated logic, but only reach 78% accuracy. To enhance the model to cover more complicated logics, this time instead of injecting operators to some layers, we start manipulating the topology of the model. We design separate components to deal with possible striping and rotating parity, and combine them together. This model, shown in Figure 6.5, can achieve >98% accuracy for RAID-5 style SSD.
Future research work

Based on previous experiments, we set following research goals as our future work:

(1) We need to find out a systematic methodology to design NN for various mapping logics, as eventually we target at building a NN inside OS to make prediction for different types of SSDs automatically. Evaluation in §6.3 has shown that have a understanding of FTL features can help build a good model, as the model in Figure 6.5 is from our knowledge on the existence of possible striping and parity rotation. However, we would need experience on building more models for white/gray/black-boxes to learn a systematic way to design topology, which may or may not contain specific logics in perceptrons. For example, next we can get input block-level traces to FEMU, a decent SSD emulator which supports various mapping designs, to collect output and try designing models for them.

(2) Though current RAID-5 model (striping + parity rotation) can also work with RAID-0 (striping only), it is still not verified that if models designed to handle complex mappings are also compatible with easier ones (as subsets). If that is the case, then a model which covers all potential FTL designs would be able to handle the mapping prediction problem.

(3) So far all evaluations are on simulated white-box SSDs and only involve simple FTL mappings. Later things will further move to emulations, and real black-box SSDs, which have more features affecting IO latencies such as GCs, buffer and wear-leveling etc..
REFERENCES


