Reducing Tail Latency using Duplication: A Multi-Layered Approach

Hafiz Mohsin Bashir[*], Abdullah Bin Faisal[*], M Asim Jamshed[†], Peter Vondras[*], Ali Musa Iftikhar[*], Ihsan Ayyub Qazi[+], and Fahad R. Dogar[*]

[*]Tufts University, [†]Intel Labs, [+]LUMS

ABSTRACT

Duplication can be a powerful strategy for overcoming stragglers in cloud services, but is often used conservatively because of the risk of overloading the system. We call for making duplication a first-class concept in cloud systems, and make two contributions in this regard. First, we present duplicate-aware scheduling or DAS, an aggressive duplication policy that duplicates every job, but keeps the system safe by providing suitable support (prioritization and purging) at multiple layers of the cloud system. Second, we present the D-Stage abstraction, which supports DAS and other duplication policies across diverse layers of a cloud system (e.g., network, storage, etc.). The D-Stage abstraction decouples the duplication policy from the mechanism, and facilitates working with legacy layers of a system. Using this abstraction, we evaluate the benefits of DAS for two data parallel applications (HDFS, an in-memory workload generator) and a network function (Snort-based IDS cluster). Our experiments on the public cloud and Emulab show that DAS is safe to use, and the tail latency improvement holds across a wide range of workloads.

CCS CONCEPTS

• **Networks → Cloud computing**; **Data center networks**; *Programming interfaces*;

KEYWORDS

Tail-Latency, Straggler Mitigation, Duplication, Cloning, Duplicate-Aware Scheduling, Abstraction

ACM Reference Format:

Hafiz Mohsin Bashir[*], Abdullah Bin Faisal[*], M Asim Jamshed[†], Peter Vondras[*],, Ali Musa Iftikhar[*], Ihsan Ayyub Qazi[+], and Fahad R. Dogar[*]. 2019. Reducing Tail Latency using Duplication: A Multi-Layered Approach. In *The 15th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '19), December 9–12, 2019, Orlando, FL, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3359989.3365432

1 INTRODUCTION

Meeting the performance expectations of cloud applications is challenging: typical cloud applications have workflows with high fanout, involving multiple potential bottleneck resources (e.g., network, storage, processing, etc.), so even a single slow component, or a *straggler*, ends up delaying the entire application. Studies show that stragglers can cause significant increase in tail latency [14, 17, 29, 57]. Because a slew of unpredictable factors (e.g., failures, load spikes, background processes, etc.) can cause stragglers [17, 28, 38, 75], overcoming them is difficult, especially for application workloads with small requests that only span tens to hundreds of milliseconds – for such applications, the time to detect and react to stragglers is typically too short [40].

A powerful approach for dealing with stragglers is to avoid them using *duplication*. Duplication can leverage the inherent redundancy present at different levels of a typical cloud-based system – redundant network paths [30, 34, 39, 46] or replicated application and storage servers – to overcome a broad range of straggler scenarios. For example, Dean et al. describe a collection of "tail tolerant" techniques used at Google, which duplicate get() requests for large scale data parallel applications [28]. At the *network* level, prior work has shown the benefits of duplicating flows (or specific packets of a flow) [41–43, 45, 49, 70, 74]. Similarly, other systems have shown the efficacy of duplication for *storage* (e.g., [40, 64, 68]) and *distributed job execution frameworks* [17–19, 65, 76].

Despite the potential benefits of duplication, its use is fraught with danger: the extra load caused by duplication can degrade system performance or even make the system unstable. For example, a duplicate `get()` request will not just create extra work for the application and storage servers, but it also increases the load on other resources (e.g., network, load balancers, etc). Because of this danger, existing systems typically use duplication in a conservative manner, employing techniques that *selectively* issue duplicates. For example, in hedged-request [28], a duplicate is issued only if the primary does not finish within a certain time (e.g., 95^{th} percentile expected latency). Such heuristics can turn out to be adhoc, as workload and system load changes [31], making today's multi-layered cloud systems even more brittle and complex.

We argue that these challenges stem from lack of explicit support for duplication, and call for making duplication a first-class concept in modern cloud systems: it should be easy to specify duplication requirements, and different layers of the system should have explicit support for meeting these requirements. Toward this goal, we identify three important questions that need to be answered first: i) Can we have duplication *policies* that are aggressive enough to reap the full benefits of duplication, yet are safe to use in today's multi-layered cloud systems, ii) Can we design *abstractions* that make it easy to support diverse duplication policies at different layers of the system, and iii) How can we effectively deal with *legacy layers*

CoNEXT '19, December 9–12, 2019, Orlando, FL, USA

ACM ISBN 978-1-4503-6998-5/19/12…$15.00
https://doi.org/10.1145/3359989.3365432

that cannot be modified to support duplication? Our work seeks to answer these questions and makes two contributions.

Our first contribution is a new duplication policy, duplicate-aware scheduling or DAS, a *multi-layered* duplication policy that makes *aggressive* use of duplication, but keeps the system stable through a combination of two well known primitives: *prioritization* and *purging*. Prioritization ensures that duplicates are treated at a lower priority and don't harm primaries while purging ensures that unnecessary copies in the system are removed in a timely manner so they do not overwhelm any auxiliary resource in the system. In this way, DAS is grounded in theory and informed by practice: it leverages key recent results in scheduling theory [32], which shows the benefits of prioritization when duplicating requests, with the insights from practical, large-scale systems, which show the need (and benefits) of purging [28]. Finally, DAS is multi-layered, providing suitable support at key layers (e.g., network, storage) of a typical cloud system. As we discuss in §2.1, it is the combination of aggressive duplication, using *both* prioritization and purging, and multi-layer support for duplication that makes DAS a unique duplication policy.

Our second contribution is an abstraction, duplicate-aware stage (D-Stage), that makes it easy to support DAS and other duplication policies at different layers of a cloud system. A D-Stage comprises of *queues* with suitable duplication controls that are necessary for supporting common duplication policies. The D-Stage abstraction provides three key benefits.

First, there is a *decoupling* between the duplication policy and mechanism, through a high level interface that exposes key duplication policy controls – such as the number of duplicates, their priorities, when and how they should be dispatched and purged, etc. – while hiding the details of their implementation mechanism, such as what prioritization or purging mechanism is used.

Second, a D-Stage operates on the notion of a *job*, which has an associated *metadata* that consistently identifies the duplication requirements of a job as it travels across different layers of the system – e.g., from an application request to network packets and I/O requests – enabling consistent treatment of duplicates, such as differentiating between primary and duplicates or not duplicating an already duplicate job.

Third, a Proxy D-Stage, which is a special type of D-Stage, helps in dealing with the challenge of legacy layers that may not be amenable to modification. A Proxy D-Stage is inserted in front of a legacy layer, in order to approximate the duplication functionality. It has all the functionality of a typical D-Stage but also supports *throttling* of jobs going into the legacy layer, keeping as many jobs in its own queues, in order to retain control over key duplication primitives, such as prioritization and purging.

We validate our contributions in the context of challenging workloads involving data-parallel applications and networks functions (NF). For data-parallel applications, we use the Hadoop Distributed File System (HDFS) and DP-Network [23], a research prototype used for small in-memory workloads (e.g., web search [14]). For both applications, a `read()/get()` request is duplicated and each copy is sent to one of the multiple available replicas; in this scenario, data could be served from disk or memory, which could create different bottlenecks in the system (e.g., network, storage). We implement D-Stages support for storage and network layers, leveraging existing mechanisms for prioritization and queuing that are available in commodity servers (e.g., CFQ disk scheduler [7], Priority Queues, etc). For NF evaluation, we consider a distributed IDS cluster scenario, where CPU is the main bottleneck, and D-Stages at the network and processing levels are required to get the full benefits of duplication.

We evaluate these applications using various micro and macro-benchmark experiments on Emulab [3] and the Google Cloud [5]. Our experiments on Google Cloud validate the presence of stragglers, and DAS's ability to overcome them. We also find that DAS is seamlessly able to avoid hotspots, which are common in practical workloads, by leveraging an alternate replica, thereby obviating the need for sophisticated replica selection techniques [66]. Using controlled experiments on Emulab, we show that DAS is safe to use: even under high loads, when many existing duplicate schemes make the system unstable, DAS remains stable and provides performance improvements. Our results across a wide range of scenarios show that DAS's performance is comparable to, or better than, the best performing duplication scheme for that particular scenario.

In the next sections, we describe our contributions and how we validate them. As we discuss in §8, our work also opens up several interesting directions for future work, such as implementing the D-Stage abstraction for other resources (e.g., end-host network stack) and potentially combining the work done by the primaries and duplicates to improve system throughput.

2 DUPLICATION POLICIES

A plethora of existing work focuses on the use of duplication for straggler mitigation in cloud systems [17–19, 28, 40, 45, 49, 54, 64, 65, 70, 73, 74, 76]. We first review these policies by distilling the key design choices they make (§2.1), and use this analysis to motivate the need for a new multi-layered duplication policy that is aggressive, yet safe to use (§2.2).

2.1 Design Choices for Duplication Policies

We identify three key design choices that existing duplication policies need to make.

(1) **Duplication Decision.** A fundamental decision is what to duplicate and when to duplicate – we refer to this as the duplication decision. Existing proposals fall into two broad categories: i) In *full duplication* schemes (e.g., Cloning), every job is duplicated, which simplifies the duplication decision, but dealing with the extra load of duplicates and keeping the system stable under high load is a major challenge. ii) In *selective duplication* schemes [17, 19, 28, 74, 76], only a fraction of jobs are duplicated based on some criteria (e.g. job size, duplication threshold, etc.). For example, Hedged [28] duplicates a job if it fails to finish within the 95^{th} percentile of its expected latency. Such techniques are usually safe, but can be complex to get right, typically requiring careful tuning of thresholds.

To validate the above observation, we conduct a simple experiment on a small scale HDFS cluster where a client retrieves objects of 10MB size under different policies. Details about the experiment setup, including the noise model used to create stragglers, are described in §6.2. Figure 1 shows that duplicating every request (e.g. Cloning) only works at low load and overloads the system at medium and high loads. *Selective duplication* (e.g. Hedged), on

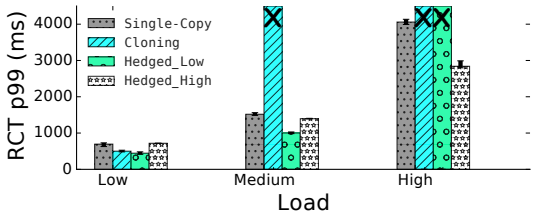


Figure 1: A simple experiment to validate limitations of existing proposals. Single-Copy is the base case. Hedged is selective duplication with duplication threshold optimized for specific load (e.g. Hedged-Low is optimized for low load). Max-sized bar with black cross indicates that the corresponding scheme becomes unstable.

the other hand, requires careful tuning of duplication threshold. Hedged-Low (optimized for low load) provides benefits at low load while Hedged-High (optimized for high load) performs well at high load. However, at high load Hedged-Low becomes unstable as it starts duplicating too many requests.

In summary, *full duplication* is simple but not *safe*[1], while selective schemes like Hedged are (usually) *safe* but are difficult to get right.

(2) **Duplication Overhead.** Another choice that many duplication policies need to make is how to minimize the overhead of duplicates. Existing approaches typically use one of the following two primitives:

- Purging. Purging is used to remove unnecessary copy(ies) of a job from the system in a timely fashion. While the use of purging is quite common, schemes differ in terms of *when* to purge, some purging extra copies when at-least one copy is finished [17, 28, 74], while others purging when at-least one copy starts being served [28, 54]).
- Prioritization. Prioritizing primaries over duplicates provides two benefits: i) it allows duplicates to make use of spare system resources (i.e., work conservation), and ii) it shields primaries from the overhead of duplicates (i.e., isolation) [28, 32, 70].

(3) **Cross-Layer Support for Duplication.** Duplication can cause extra load for other layers of the system, so another important consideration for duplication policies is whether to provide support at multiple layers of the system or not. Most existing proposals don't provide cross-layer support; the few proposals that provide such support, do so in a highly application specific manner. For example, Dolly [17] employs a technique called *delay assignment* to avoid contention for intermediate data access – a technique highly specific to job execution frameworks and targeting only specific layers.

Synthesis. Table 1 summarizes the positioning of existing schemes with regards to the design choices discussed above. We make three observations. First, few schemes make use of full-duplication even though it's simple to use: the main challenge is that its aggressive nature makes it challenging to keep the system stable. Second, most existing schemes use either prioritization or purging, but not both. Third, most schemes either do not provide cross-layer support or only support it in a limited context. These observations motivate the need for our proposed duplication policy, which we present next.

[1]We define safety as system stability, where the queues are bounded.

Scheme	Duplication Decision	Prioritization	Purging	Cross Layer Support
Hedged Request [28], AppTO [28], MittOS [40] RepFlow [74]	Selective	No	Yes	No
Mantri [19], LATE [76], Dolly [17]	Selective	No	Yes	Limited
Sparrow [54], Tied Request [28]	Full	No	Yes	No
Primary First [32]	Full	Yes	No	No
DAS(§2.2)	Full	Yes	Yes	Yes

Table 1: Comparison of various duplication schemes.

2.2 Duplicate-Aware Scheduling (DAS)

DAS is a *multi-layered*, *aggressive*, and *safe* duplication policy. It proactively duplicates every job once, sending the primary and duplicate to two different servers. Yet it is safe because it employs *both* prioritization and purging: duplicates are given strictly lower priority compared to primaries, and once a job finishes, its corresponding duplicate (or primary) is immediately purged. This support needs to be provided at every potential bottleneck layer of the system, making DAS a first of its kind *multi-layered* duplication scheme.

The aggressive approach of DAS obviates the need to have a sophisticated selection mechanism (e.g., speculation, using duplication thresholds, etc.) and thus allows dealing with challenging workloads where jobs may last milliseconds or even less [40].

The choice of using both prioritization and purging is motivated by recent scheduling theory results as well as practical considerations: while prioritization helps DAS shield primaries from duplicates, purging ensures that unnecessary copies in the system are removed in a timely fashion. Duplicates provide benefits even though they are serviced at low priority because the likelihood of finding both the primary and secondary to be overloaded is low under typical system loads. Specifically, recent results from scheduling theory [32] show that prioritizing primaries over duplicates is sufficient to deal with the duplication overhead while providing the benefits of duplication, in a single bottleneck system. However, real systems have multiple layers, and purging is required because it ensures that any *auxiliary* system resource does not become a bottleneck. For example, duplicate jobs waiting in a lower priority queue may cause exhaustion of transmission control blocks (TCBs), limit buffer space for primary jobs, or increase contention overhead of thread scheduling. Not surprisingly, for this reason, purging is considered an important component of many duplication schemes [28].

Finally, since in practical systems any layer can become a performance bottleneck, the *multi-layered* approach of DAS enables duplication support at every such layer, albeit the full DAS functionality is typically not used at every layer. For example, a job is duplicated only once at a specific layer (depending on the use case), while other layers may just provide prioritization and purging support for that job.

In summary, DAS is the right duplication policy because: i) it is simple, ii) it allows duplicates to make use of spare system resources (e.g., network bandwidth) while shielding primaries from the overhead of duplicates, and iii) its multi-layered approach makes

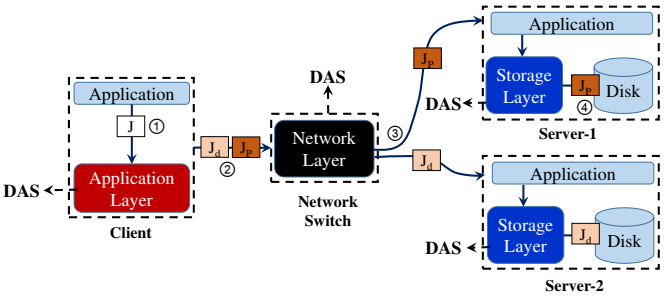


Figure 2: Example showing how DAS support at multiple layers can be combined to serve a get() request from a client application. DAS at application layer duplicates the job whereas, network and storage layers provide differential treatment to duplicate.

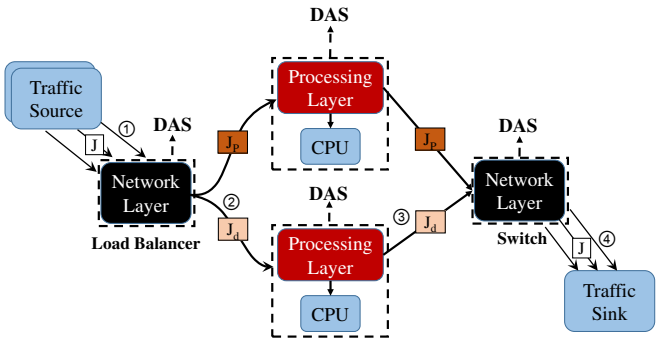


Figure 3: Example showing how DAS enables duplication for NFs (e.g., an IDS cluster). Job is duplicated at an intermediate network layer (load balancer).

it robust under practical cloud scenarios, where the bottlenecked resource (e.g., network, storage) is unknown.

To illustrate the richness of DAS, we now describe two use cases, which both involve multiple diverse layers.

1- Data Parallel Applications. Figure 2 shows an example of a *get()* operation in data parallel applications (e.g., MongoDB, HDFS). On the client side, at the application layer, DAS duplicates a *get()* request at lower priority and sends both the primary and the duplicate to a subsequent layer, scheduling them based on their priority. Other layers in this example – like the network and storage – don't create any more duplicates: they just provide differential treatment (prioritization) to primaries and duplicates. Once the primary (or duplicate) finishes, DAS purges the other copy, with all subsequent layers removing any corresponding job(s) from their queues.

2- Network Function Virtualization (NFV). Cluster deployments of NFs are common in today's cloud environments. These clusters are often CPU bound, e.g., an IDS cluster which process incoming packets to detect malicious traffic. Unlike the previous case, here an intermediate D-Stage is responsible for job duplication in an application agnostic manner. As shown in Figure 3, a network layer (load-balancer) duplicates packets on the critical path

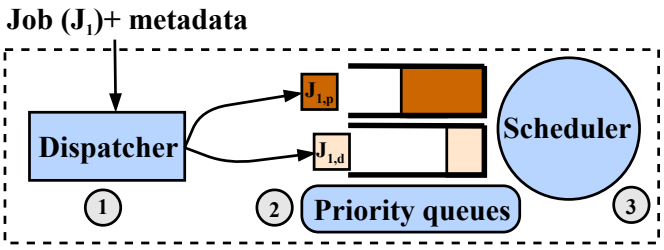


Figure 4: The D-Stage abstraction has three components. Job + metadata enters the stage, the dispatcher (1) creates a duplicate copy (if required) and puts the job(s) in their respective priority queues (2). A scheduler (3) schedules them in a strict priority fashion. Jobs can also be purged (not shown).

```
dispatch (job, metadata)
schedule()
purge (job-id(s), CASCADE_FLAG)
```

Table 2: D-Stage Interface

of flows (in-network duplication) and forwards the primary and duplicate flows to different nodes in the NFV cluster. A processing layer running at each NF node processes packets in a priority-aware fashion: duplicates are only processed if there is spare capacity on that node. Yet another network layer is responsible for taking processed packets, and filtering out duplicates before forwarding them to the next layer, which could be a cluster of web-servers running a data-parallel application.

While these use cases illustrate the diversity of layers and their use, they also point to the main challenge: how to simplify supporting a range of duplication functionality at such diverse layers of typical cloud systems.

3 THE D-STAGE ABSTRACTION

We propose D-Stage, an abstraction that simplifies supporting rich duplication policies (including DAS) at multiple, diverse layers of a cloud system. A D-Stage comprises of *queues*, which are inserted at potential bottleneck layers of a system, providing the necessary *control* for duplication, such as when to create a duplicate, its priority, and support for purging. Figure 4 zooms into a D-Stage: each D-Stage operates on a job with associated metadata (e.g., id, priority) that consistently identifies the job across different layers of the stack, and supports key duplication controls through a high level interface (Table 2). The D-Stage abstraction provides three key benefits:

1. Making duplication visible across layers. While the notion of a job changes across layers – such as a flow for a transport D-Stage and a packet for a network D-Stage – the associated metadata with a job (e.g., id, priority) consistently identifies that job and its duplication requirements. This ensures that a job can be tracked across layers (e.g., for purging), gets differential treatment (e.g., lower priority for duplicate jobs), and layers don't end up making unnecessary duplicates of jobs that have already been duplicated. This simple requirement of associating the right metadata with

jobs is crucial for making duplicates a first class concept in cloud systems. (§3.1)

2. Decoupling of policy and mechanism. By providing control over key duplication primitives through a high level interface, a D-Stage decouples the duplication policy from the mechanism, enabling support for rich policies at the top and diverse implementations at the bottom. As a concrete example, the policy may specify that duplicates should be given strictly lower priority, but the mechanism will decide the best way to support such prioritization. The benefits of this separation of concerns are well known in systems design. It is even more crucial for duplication in today's cloud systems which involve many diverse layers: it will be impractical for system administrators, who want to specify policies, to implement duplication mechanisms that are optimized for specific resources, such as storage, network, etc. Similarly, this decoupling allows specifying different policies without worrying about the underlying implementation details. (§3.2)

3. Support for legacy layers. Introducing D-Stage into existing layers could be challenging: modifying a layer may be infeasible, and some layers may not be amenable to purging or queuing of jobs (e.g., MongoDB [40]). A D-Stage abstraction can also support a Proxy D-Stage, which can approximate the behavior of a legacy D-Stage. The key idea is to throttle jobs going into the legacy layer to retain control over prioritization and purging in a preceding Proxy D-Stage. (§3.3)

```
job-id //identifies the job
priority //priority of the job
start-time //arrival time in the system
sched-time //wait time before scheduling
duplication() //creating duplicate copies
atStart() //processing when its turn comes
atEnd() //processing when done
```

Table 3: Job Metadata

3.1 Job and Metadata

As noted earlier, each layer has its own notion of a job. For example, at the network D-Stage, the job could be a packet while an application layer D-Stage may operate on *read* requests for files/objects. Each job should have an associated *metadata* that contains duplication specific information:

Table 3 lists the metadata required by a D-Stage. A `job-id` is used to identify a job – it should be unique and consistent across D-Stages, so a D-Stage can purge a job which is enqueued inside another D-Stage using the *job-id*. To work with legacy layers, a D-Stage can also maintain A mapping of its own job-ids to the job-ids used by the legacy layer. For example, an application layer D-Stage can maintain the mapping of application-level request to its corresponding flow or socket identifier that is used at the transport layer.

The `priority` of a job is used to decide how this job will be scheduled. Many jobs already have a notion of priority, even if it is rarely used. For example, the ToS bit in the packet header can be used to determine the priority of a packet. The `start-time` and `sched-time` are useful in making scheduling and purging decisions. For example, a scheme may decide to purge jobs which have been outstanding for a certain time; similarly, a scheme like hedged-request may want to delay the scheduling of a duplicate until a certain time. The `dup-stage-id`[2] is used to specify the D-Stage responsible to duplicate the job. For example, in the case of NFVs (§2.2), the stage-id of the intermediate network D-Stage will be used here.

Finally, the metadata includes three *callbacks*: i) `duplication()`, which implements the duplication logic, such as whether the job should be duplicated, and if yes, the number of duplicates to create, and any metadata that is necessary for each duplicate copy, such as its priority, new name, etc, ii) `atStart()`, which implements any logic that needs to be executed when a job is *scheduled* (e.g., purge corresponding copy, as in Tied-Request) and iii) `atEnd()`, which provides similar support for any action that needs to be taken once the job is *finished* executing; again this can be useful for purging the corresponding copy of the job. These functions are implemented by domain experts in a layer specific fashion.

Passing Metadata Across Layers. In an end to end system, a job may traverse different D-Stages. This requires passing metadata across different layers of the system. In DAS, a prior D-Stage translates metadata for the next D-Stage. However, for some D-Stages this translation may not be trivial. Under such conditions, the dispatcher can have pre-configured rules for job processing. For example, to duplicate a job at a network D-Stage, its dispatcher may use the hash of the packet header fields in a match-action fashion to create and dispatch duplicate copies.

3.2 Interface

We identify key duplication primitives that can be combined to support a wide range of duplication policies. These primitives are used through a high-level interface (Table 2), hiding the implementation details from other system layers. We describe the interface, and comment on its use for common types of D-Stages, such as network, storage, and processing.

`dispatch(job, metadata)`. The dispatcher controls how jobs are placed inside the D-Stage queues. The dispatcher interprets and acts on the *metadata*: it creates the necessary duplicates using the *duplication()* callback and puts both the primary and duplicate jobs in their respective queues. For example, in Figure 4, the job J_1 is duplicated (following DAS) as job $J_{1,p}$ with high priority and $J_{1,d}$ with low priority – $J_{1,p}$ goes into the high priority queue and $J_{1,d}$ is enqueued in the low priority queue. Jobs with a later *start-time* are put in a special delay queue (with an associated timer) where they wait until the timer expires or are purged from the queue. In general, we do not expect the dispatcher to become a bottleneck for a D-Stage. However, under high load it can stop duplicating jobs, following the DAS principle that duplicates are strictly lower priority.

`schedule()`. A D-Stage requires a *priority scheduler* to provide differential treatment to primaries and duplicates based on duplication policy. A job's turn should be determined by its priority, and once it is scheduled, the scheduler should first execute `atStart()`, then process the job (just like it would do normally), and finally call the `atEnd()` function. An ideal priority scheduler should ensure strict

[2]Each D-Stage is identified by a *stage-id*.

Scheme	Duplication()	atStart()	atEnd()
Cloning	Duplicates job at same priority.	None	Purges other copy
Hedged	▸ Duplicate job at the same priority. ▸ Put duplicate in special wait queue. ▸ Set timer to 95^{th}% of expected latency	None	Purges other copy.
Tied	Duplicates job at same prioirty	Purges other copy.	None
DAS	Duplicates job at lower prioirty	None	Purges other copy

Table 4: Supporting different duplication policies using D-Stage

priority – with *preemption* and *work conservation* – while incurring minimal overhead. A desirable feature in this context is to have the ability to break a large job into smaller parts so that the scheduler can efficiently preempt a lower priority job and then later resume working on it in a work conserving manner.

Fortunately, many layers already support some form of priority scheduling. For example, for storage applications, Linux supports I/O prioritization using *completely fair queuing* (CFQ) scheduler. Similarly, for CPU prioritization, we can use support for thread prioritization on modern operating systems – for example, our implementation uses the POSIX call *pthread_setschedprio* [9], which can prioritize threads at the CPU level. Finally, modern switches and end-host network stacks already provide support for prioritization, where some header fields (e.g., ToS bit) can be used to decide the priority of flows and packets.

Supporting Different Policies. Table 4 shows how the D-Stage interface, specifically the three key functions, can support different duplication policies. For example, Hedged requires the `duplication()` function to implement a special wait queue for duplicate requests: duplicates wait in this queue for a certain time (e.g,. 95^{th} percentile of expected latency) before being issued. Once a copy of the job (primary or duplicate) finishes, `atEnd()` purges the other copy. Similarly, for Tied, duplicates are created at the same priority and when one of the two starts being served, it purges the other copy.

`purge(job(s), CASCADE_FLAG)`. This interface supports specifying one or more jobs that should be purged by this D-Stage. An ideal purging implementation would allow purging of jobs from both the queues as well as the underlying system while they are being processed. The job(s) can be specified based on a predicate on any metadata information, such as matching a *job-id* or those jobs that started before a certain *start-time*. The CASCADE_FLAG specifies whether the purge message should propagate to a subsequent stage, if the current stage is already done with processing the job and can no longer purge it. For example, an application may call *purge()* on the transport flow, which would result in the transport D-Stage purging its corresponding data from the end-host buffer, and if the flag is set, it will also call *purge* on the subsequent network D-Stage, so the packets that have left the end-host (and are inside the network) can be purged. Adding support for CASCADE_FLAG requires a D-Stage to have the ability to invoke the `purge` mechanism of the subsequent layer.

3.3 Proxy D-Stage

To support legacy layers in a system, we design a Proxy D-Stage, which is a specific implementation of a D-Stage and sits in front of a legacy (unmodified) stage; its goal is to retain maximal control over jobs going into the legacy stage by keeping them in its *own* queues. This allows the Proxy D-Stage to *approximate* duplication support

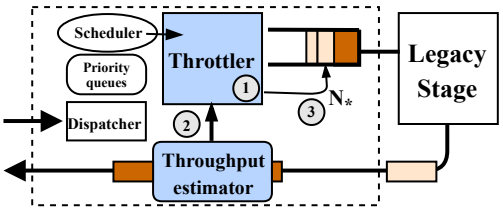


Figure 5: Proxy D-Stage sitting in front of a legacy stage. In addition to the components of traditional D-Stages, it has a throttler (1) which gets feedback from a throughput estimator (2) monitoring the output rate of the legacy stage. Based on this estimate, it discovers the multiplexing level (3).

of an unmodified stage, using its own mechanisms for prioritization and purging.

Figure 5 shows the high-level working of a Proxy D-Stage. Unlike a traditional D-Stage, it uses a special scheduler that *throttles* outgoing jobs based on the output rate of the legacy stage. Thus, the proxy D-Stage needs to be placed at a location where it can control and observe the input and output of the legacy stage.

Objective. Choosing a suitable multiplexing level is important; a small value can lead to under-utilization of the unmodified stage whereas a large value would reduce queuing inside the proxy, thereby lowering its control (e.g., prioritizing, purging). This trade-off crystallizes into the following objective: *maximize* utilization (of the unmodified stage) while *minimizing* the level of multiplexing (i.e., dispatching as few jobs as possible).

Throttling algorithm. Our desired objective closely matches with prior work on finding the right number of transactions to dispatch to a database [60]. In their work, Bianca et al. use offline queuing analysis to initialize the multiplexing level, which is then adjusted using feedback control in an online setting.

We take a similar approach but, for simplicity the throttler is initialized with a multiplexing level of one. We use the throughput of the legacy stage observed by the proxy to modulate the multiplexing level in the following way:

(1) **Probing phase.** The throttler increases the multiplexing level as long as it observes commensurate increase in the throughput. Otherwise, it stops and enters the *exploitation* phase.
(2) **Exploitation phase.** Having discovered the "best policy", the throttler continues dispatching requests as per the optimal target[3] (N_*).

One of our primary use cases for the Proxy D-Stage is its use with an unmodified HDFS server (data-node), which has no notion of duplicates (§6). An HDFS cluster in a cloud setting could be bottle-necked anywhere. Thus, we empirically verified that our throttling algorithm was able to determine the right multiplexing level for: i) SSD and HDD, if disk is the bottleneck, and ii) network links, if the workload is being served from cache.

4 CASE STUDIES

We now present two case studies that show how D-Stages are used to support DAS in real applications.

[3]This is the minimum number of requests needed to maximize utilization.

4.1 Data Parallel Applications

We now describe how multiple D-Stages can be combined to implement DAS for two data parallel applications: i) HDFS [1] and ii) an in-memory research prototype, which we refer to as DP-Network [23]. DP-Network comprises of client and server applications. The client application requests the server to transfer a specified flow size (i.e., server sends the response) using a pool of persistent TCP connections between the client and the servers. DP-Network has been used extensively in prior works (e.g., ClickNP [48], MQ-ECN [23], FUSO [25], etc) to evaluate performance of network protocols over small-scale (40 nodes), 10G clusters.

While both applications have similarities, there are also key differences in terms of how the D-Stages are supported within these applications (e.g., the purging mechanism and the use of Proxy D-Stage). We first describe the common workflow of these (and other data parallel applications) and how we insert D-Stages at different layers. We then describe the specific support that we added for both of these applications.

We divide the steps required to enable DAS support into two parts: the request phase, and the response phase.

Request Phase. The first D-Stage is the *request handler* stage, which is a processing D-Stage responsible for duplicating *get()* requests. The API between this stage and the application is the typical *get()* API, which is enhanced to carry information about replicas and their priorities. As a processing D-Stage, it has support for thread prioritization, so a lower priority thread can work on sending the lower priority *get()* requests, which ensures that the request generation part for duplicate requests does not hurt the primary requests. This is the *only* stage in the *get()* pipeline where duplication is involved. For all other stages, the dispatcher only places the job in its respective priority queue.

Requests are sent over the network D-Stage as per their priority until they reach their respective server node. On the server side, there is a request handler processing D-Stage, which is similar to the request handler stage of the client, except that it will not duplicate the job, as indicated by the metadata for the request. The request handler stage will then pass on the request to a storage D-Stage, which will follow the functionality described earlier.

Response Phase. The responses sent by the primary and secondary replicas traverse the network, using their appropriate priorities – high priority for responses from the primary replica and low priority for responses from the secondary replica. On the client side, there is another processing D-Stage called the *response handler* stage. Like typical processing D-Stages, it uses multiple threads, with different priorities, to process responses from the primary and secondary replicas. Once the entire *get()* operation is complete, the object/file is delivered to the application, and ongoing jobs at primary or secondary replicas corresponding to this *get()* operation are purged.

Support for HDFS. HDFS does not queue requests natively. Instead of adding queues and other D-Stage primitives, we decided to test our Proxy D-Stage with unmodified HDFS datanodes. On the client side, we added support for dispatching – creating and sending multiple requests to different replicas. Because HDFS uses a separate TCP connection for each request, we used the closure of a TCP connection as a purge signal as well.

Support for DP-Network. We modified this application to introduce D-Stages at both the client and server. The bottleneck for the target workloads is the network, and the server already has support for adding prioritization to different network flows. The original server did not use queues, so we added support for queuing and purging. Given that typical target workloads for this application include small requests (e.g., a few KBs), the system uses persistent TCP connections, so we explicitly sent purge messages from client to server. These purge messages were sent using a separate TCP connection. Given the small request sizes, we only purged requests from the server queues as purging in-flight requests would not provide much savings. We also added support for pipelining of requests on the same TCP connection and used a fixed number of pre-established connections – these optimizations improved our baseline results (Single-Copy) compared to the vanilla implementation.

4.2 Network Function Virtualization

The case of NFV cluster is different from the data parallel applications in two ways: i) jobs are duplicated by an intermediate network D-Stage instead of the application D-Stage, ii) purging is even more challenging because of the extremely short timescale of (i.e., per packet) operations.

Similar to Figure 3, we consider a Snort [12] based IDS cluster deployment as a use case for DAS. We have a network D-Stage, built on top of Packet Bricks [11], which is responsible for duplicating and load balancing incoming packets to IDS nodes. On each node, a processing D-Stage runs primary and duplicate Snort instances on different threads pinned to the same core at high and low priority respectively. It uses the POSIX call *pthread_setschedprio*() to enable thread level CPU prioritization. Finally, another network D-Stage sits after the IDS cluster and acts as a "response handler"; it performs de-duplication and forwards only unique packets to an interested application (e.g., a web server). We approximate purging at processing D-Stages on IDS nodes by limiting the duplicate Snort's queue size.

5 IMPLEMENTATION

In addition to the above applications, we have also implemented a Proxy D-Stage in C++; it implements the basic D-Stage interface (with support for purging, prioritization) as well as the throttling based scheduler. The proxy is multi-threaded, supports TCP-based applications, with customized modules for the HDFS application, in order to understand its requests. The proxy has been evaluated with multiple applications for benchmarking purposes: it adds minimal overhead for applications with small workloads (e.g., DP-Network) and for the applications where we actually use the Proxy (i.e., HDFS), the performance matches that of the baseline (without the proxy). The proxy also supports a module to directly interact with the storage and a client interface that can be used to generate *get()* requests (similar to the HDFS and DP-Network applications). For prioritization we use the different primitives available for each resource (as described earlier). Specifically, for the network, we used native support for priority queuing, including Linux HTB

queues at end-points. For storage, Linux provides *ioprio_set*() [8] system calls to explicitly set I/O prioritization of a request.

To add DAS support for HDFS we have implemented an HDFS proxy D-Stage in ~1500 lines of C and ~300 lines of python code. Similarly, to add DAS support for DP-Network, we added ~750 new lines of C code.

In general, the implementation complexity of DAS varies across layers and each layer has its own challenges. We discuss some of these in §8.

6 EVALUATION

Our evaluation covers a broad spectrum, in terms of environments (public cloud and controlled testbed), policies, applications (HDFS, DP-Network, Snort IDS), system load (low, medium, high[4]), workloads (with mean RCT from μs to ms and seconds), schemes against which DAS is evaluated (e.g., Hedged, Cloning, etc), and micro-benchmarking various aspects of our system (e.g., prioritization and duplication overhead, etc). Our key insights are:

- **DAS effectively avoids different types of stragglers observed in the "wild".** In our experiments on the public cloud, we encounter different types of stragglers (e.g. caused by storage and network bottlenecks). DAS is able to avoid most of these stragglers resulting in up to a 4.6× reduction in tail latency (p99 RCT) compared to the baseline. (§6.1)
- **DAS is safe to use.** Using controlled experiments on Emulab, we show that DAS remains stable at high loads, and performs as well as the best performing duplication scheme under various scenarios – all these benefits come without requiring any fine tuning of thresholds. (§6.2 and §6.3)
- **DAS can effectively deal with system and workload heterogeneity.** A (somewhat) surprising finding of our study is that the use of replicas through DAS can also shield small flows from large flows (workload heterogeneity) without requiring flow-size information (or approximation), as is required by most datacenter transports [22, 52], also obviating the need for any intelligent replica selection mechanism [27, 66]. (§6.3)
- **DAS is feasible for an IDS cluster.** We show that dealing with lower priority duplicate traffic does not affect the throughput of Snort nodes, and DAS is able to effectively deal with stragglers. (§6.4)

6.1 HDFS Evaluation in Public Cloud Settings

The goal of this experiment is to evaluate the robustness of our system in the "wild" – this environment has natural stragglers and also involves multiple potential bottleneck resources (e.g., network, storage).

Experimental Setup. We set up an HDFS storage cluster on 70 VMs on Google Cloud [10]. Our cluster has 60 data-nodes and 10 clients. For HDFS data-nodes, we use `n1-standard-2` type machines (2 vCPUs, 7.5 GB RAM), while clients run on `n1-standard-4` machines (4 vCPUs, 15GB Memory). The HDFS replication factor is set to 3. We provision a total of 6TB of persistent storage[6] backed by standard hard disk drives. Persistent disk storage is not locally attached to the systems so the bottleneck could be anywhere in the system. On each data-node, we have two types of D-Stages that are used: a Proxy D-Stage for handling the HDFS data-node's unmodified storage D-Stage, and a network D-Stage, which makes the end-host support DAS while using the cloud network[5].

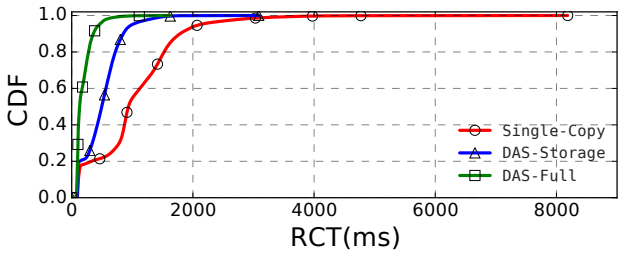


Figure 6: DAS's performance in public cloud for HDFS cluster. DAS reduces latency at p99 by 4.6× and mean latency by 5.4×.

Workloads and Metric. For our experiments, the client generates `get()` requests for a fixed size file (10MB), according to a Poisson process. We run this experiment at medium load (estimated from the disk throughput guaranteed by the cloud provider which translates to more than 2000 requests per minute) for several hours. Our selection of file size corroborates with the small file sizes[6] observed in the HDFS deployment at Facebook for their messages stack [44]. The size of our data set is ~1.5TB. Our evaluation metric in all of our experiments is the request completion time (RCT) unless specified otherwise. Note that each job comprises of a single request – with a higher scale out factor (multiple requests in a single job), we can expect even more stragglers to show up [40].

Schemes. For this experiment, we compare the performance of Single-Copy (base-case) with DAS under two different settings: i) DAS-Storage, which only uses the storage D-Stage, ii) DAS-Full which uses both storage and network D-Stages.

Avoiding Stragglers at Multiple Layers. Figure 6 shows the CDF of RCTs for the three schemes. We observe a long tail for the baseline: at p99.9, the latency is 5× higher than the median latency (3.5× and 2.3× at p99 and p95 respectively). The results show that DAS reduces the tail by 2.26× at p99 with only storage D-Stage enabled. However, with both the storage and the network D-Stages (DAS-Full) yields higher gains; RCTs are reduced at all percentiles with the benefits more pronounced at higher percentiles (p99 latency is reduced by 4.6×). Our analysis of the results revealed that the long tail was primarily caused by poorly performing datanodes – these straggers can be avoided even if we just enable the storage D-Stage (DAS-Storage). However, there were also scenarios where network was the bottleneck (e.g., data being served from the cache) and by enabling the network D-Stage, we reduced the network interference caused by the duplicate jobs. This highlights the ability of DAS in dealing with different sources of stragglers, which may appear at different layers of the system.

[4]By our definition, low load is 10-20%, medium load is 40-50%, high load is 70-80%

[5]Cloud networks present the big switch abstraction, with network bottlenecks only appearing at the edges.

[6]Harter et al.[44] found that 90% of files are smaller than 15MB and disk I/O is highly random.

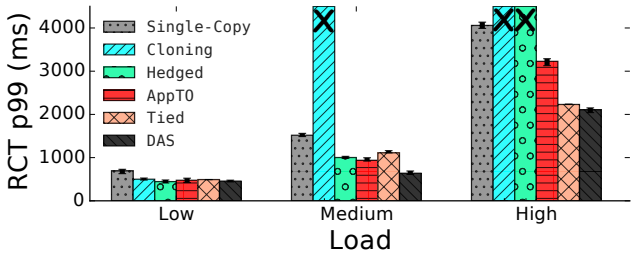


Figure 7: DAS's comparison with other duplication based schemes. DAS performs equally well at low and medium loads while keeping the system stable at high load. Max-sized bar with black cross indicates that the corresponding scheme becomes unstable.

6.2 HDFS Evaluation in Controlled Settings

We now move to the controlled environment of Emulab, to evaluate DAS against other duplication schemes under various scenarios. This environment, combined with our use of a straggler/noise model derived from our previous experiment, allow us to perform repeatable, yet realistic experiments.

Schemes. We compare the performance of DAS against Single-Copy (base case), and several duplication schemes proposed by Dean et al [28]: Cloning, Hedged, AppTO, and Tied [28]. For Cloning, the client proactively duplicates every request at the same priority. For Hedged, if the first request fails to finish before p95 deadline, the client makes a duplicate request to a different replica. Upon completion of any of the two copies, the other is purged. In AppTO, requests have a timeout value. If a request fails to finish before timeout, the client purges the first request and issues its duplicate request to another replica (request restart).

In our experiments we use 480ms as the threshold for triggering a duplicate request for Hedged and AppTO schemes. The Tied scheme duplicates every request and ties the identity of the other replica with the request. When a request's turn arrives, it sends a purge message to its counterpart. The corresponding request, if not finished, gets purged.

Experimental Setup. For this experiment, we setup a 10 node HDFS cluster on Emulab [3]. We provision one client for this experiment. We use d430 type machines (2x2.4 GHz 8-core with 64GB RAM, 1TB 7200 RPM 6 Gbps SATA Disks). HDFS datanodes are connected by 1Gbps network links; whereas, the client is connected by a 10Gbps network link. For random read I/O requests, we benchmarked the disk throughput to be less than the network link capacity, making disks the bottleneck resource. This experiment uses the same workload and metrics as used in the previous experiment.

Noise Model. To model stragglers, we derive a noise model from our experiment on the Google Cloud (§6.1). We use the median latency experienced by Single-Copy experiment as our baseline and calculate the percentage of additional latency experienced by the top 10 percentiles (p91 to p100). We then model this noise in our experiment by randomly selecting 10% of the requests and delaying the application response by the calculated factor. We use sleep() call to delay the response.

Results. Figure 7 compares the performance of all the schemes at the p99 latency across different loads. The load corresponds to three regimes: low, medium, and high. This is the offered load, calculated with respect to the bottleneck resource (i.e., disk)[7]. We make four observations:

(1) At low loads, all the duplication based schemes perform well compared to Single-Copy. They reduce the p99 latency by *at least* ~1.45×.
(2) At medium and high loads, the duplication overheads of aggressive schemes (Cloning, Hedged) make the system unstable. DAS, despite duplicating every request, remains stable and continues to reduce tail latency (p99 by 2.37× and 1.9× at medium and high loads respectively).
(3) DAS is most effective at medium load. This is because at medium load, transient load imbalance on any one node is common despite the overall load being moderate. This enables DAS to leverage the spare capacity on other nodes. In contrast, stragglers are less common at low load, while there are little opportunities to exploit an alternate replica at high load.
(4) Tied is useful in terms of system stability (does not cause system overload at high load). However, it fails to cope with noise encountered once the request has started being served, which is evident from low gains at low and medium loads. DAS successfully handles such scenarios by continuing to work on both requests until one finishes. Further, as we show in §6.3, the lack of prioritization in Tied is catastrophic for workloads with small requests, where Tied becomes unstable at high loads (see Fig. 8b).

6.3 DP-Network in Public Cloud Settings

The goal of these experiments is to evaluate the performance under scenarios where the *network* is the bottleneck. Compared to HDFS, the workload is also different (smaller requests) which creates new challenges and opportunities.

Setup. We use a 10 VM setup on Google Cloud with one client and nine servers. On the client side, we use `n1-highcpu-16` type VM, while servers use the `n1-stand-4` VM types. The server VMs are rate limited to 1Gbps, while the client VM has a rate limit of 16Gbps. This experiment predominantly focuses on small flows of size 50KB, the average flow size of short flows in the web search workload [14]. We also consider the full web search workload, which includes mix of large and small flows, and highlight how DAS is able to effectively deal with flow size heterogeneity. Our client generates requests based on a Poisson process, randomly choosing the primary and secondary servers.

Performance Under Workload Heterogeneity. Figure 8a shows the performance of DAS with the DCTCP web search workload (containing a mix of short and long flows) under medium load. DAS significantly improves the RCT for short flows without hurting the long ones. This happens because the co-existence of long and short flows significantly affects the transfer times of short flows [14]. DAS is a natural fit for such scenarios. If a short flow and a long flow get mapped to a critical resource together, namely a link or a queue,

[7]Note that this is the load induced by the primary requests only. Duplication may increase the load depending on the particular scheme

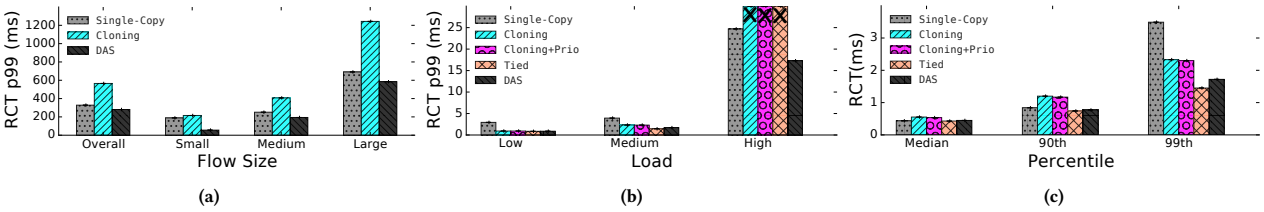


Figure 8: DAS under network bottleneck scenarios. (a) RCTs at p99 for web search workload [14] at medium load for different flow sizes – Small (<100KB), Medium (100KB to 10MB), Large (>10MB) (b) RCTs at p99 across varying load when the flow sizes are fixed at 50KB. At high load, DAS keeps the system stable while other duplication based schemes fail (c) performance for fixed 50KB transfers at medium load at important percentiles. DAS is within 10% of Tied Request while considerably better than other strategies at the tail (≥ p90). Max-sized bar with black cross indicates that the corresponding scheme becomes unstable.

DAS provides opportunity to the duplicate requests of short flows to get served from another less loaded replica. This highlights that if we have replicas and have a scheme like DAS, we can shield small flows from long flows without requiring flow size information (or estimation) as is required by most datacenter transports [52].

Performance with Small Requests. We now focus on fixed small (50KB) request sizes. We introduce noise through background flows: the client randomly picks a server and starts a 10MB transfer, emulating a hotspot in the network. Overall, the noise accounts for only 1% of the total link capacity.

Figure 8b shows the performance of DAS at the p99 across variable network loads – low, medium, and high. Similar to the HDFS experiments, we observe that DAS remains stable at high loads while providing duplication gains at low loads. Note that no other duplication scheme remains stable at high loads; this is unlike the HDFS experiment (where schemes like Tied worked well), and is because of the challenging nature of this workload which involves small (sub-millisecond) requests. In such scenarios, purging is less effective at low loads (in fact, it has overhead) and prioritization becomes critical, as highlighted by the slight difference in the performance of DAS and the Cloning+Prioritization scheme. However, our analysis shows that at high loads we need *both* prioritization and purging in order to keep the system stable. Figure 8c zooms into the medium load and highlights gains achieved over Cloning (unstable at high load) and Single-copy (stable at high load) at different percentiles. It is also within 10% of the Tied scheme which becomes unstable under high load.

6.4 DAS with IDS cluster

We evaluate the feasibility of using DAS when CPU is the bottleneck in the context of an IDS cluster. This can be useful in scenarios where some nodes are stragglers while others have spare CPU capacity to process traffic. Unlike the previous evaluation scenarios, which were focused on latency improvement, our measure of performance for the IDS scenario is the system goodput i.e. the number of packets processed by the IDS cluster[8].

[8]Note that the IDS throughput can also affect the application latency (e.g., an overloaded IDS instance can cause packet drops which may result in increased latency for applications [69])

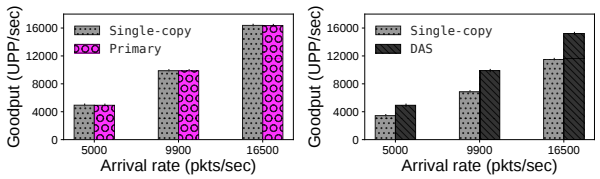
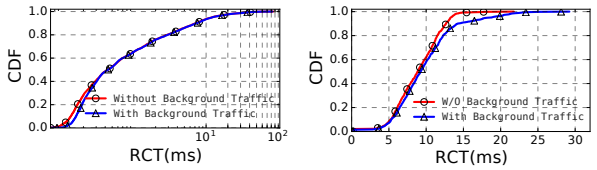


Figure 9: Feasibility of using duplication when CPU is the bottleneck in an IDS cluster. (a) shows that at varying input load, the introduction of a duplicate snort does not hurt primary snort throughput. (b) shows that with a straggler node, DAS outperforms Single-Copy as duplicate snort instances running at other nodes can process the duplicates of the packets that are backlogged at the straggler.

We consider an IDS cluster receiving packets at varying load (defined as the arrival rate of traffic). For each load, we determine the performance of having single snort (Single-Copy) instances versus having two snort (Primary and Duplicate) instances. This comparison highlights that duplication of work does not hurt performance. Secondly, for each load, we consider the effect of having a straggler node. This comparison highlights that duplication can alleviate the problem of stragglers.

Experimental Setup. For this experiment we provision 5 d430 type machines on Emulab. Our setup reflects the layout given in Figure 3 (§2.2). The Snort IDS runs on two nodes; each node runs a primary and a duplicate instance of Snort at high and low priority respectively. On each node, Snort instances are pinned to the same core, and we use a single thread for the Snort instance. The other three nodes are configured as a traffic source, an in-network packet duplicator and an in-network de-duplicator.

A straggler in this setting could occur due to system overload or some failure [36, 37]. To emulate this behaviour, we run a CPU-intensive background task lasting 60% of the experiment duration on one of the IDS nodes. This effectively results in a drop in Snort's throughput. We focus on the goodput – unique packets processed per second (UPP/sec) – achieved by the IDS cluster under varying system load.



(a) Network Prioritization (b) I/O Prioritization

Figure 10: Prioritization benchmarks showing RCTs of high priority requests with and without low priority background traffic.

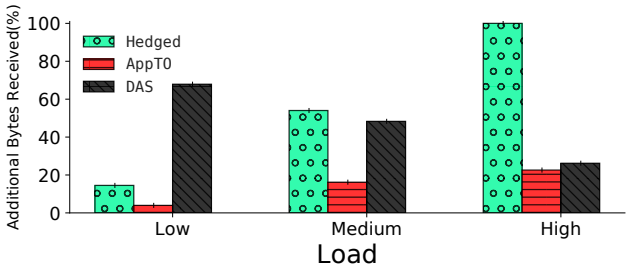


Figure 11: At low load, DAS services a large number of duplicate requests to make full use of spare system resources which results in high overhead. At medium and high load, request queues start to build up, causing many duplicate requests to get purged from queues which results in decreased overhead. For other duplication schemes (e.g. Hedged, and AppTO), the overhead increases with increase in system load.

Prioritization Works. Figure 9a shows the goodput for varying system loads without any stragglers. We observe that with thread prioritization the impact of duplicate Snort on the goodput of primary Snort is negligible – even when the traffic arrival rate is high.

DAS Alleviates Stragglers. Figure 9b shows the performance of Snort for varying system load when one Snort node becomes a straggler. We observe that goodput in the case of DAS is higher than Single-Copy (no duplication). This is because packets dropped by straggler node in the case of Single-Copy are processed by the duplicate Snort instance in the case of DAS.

6.5 Microbenchmark Results

In this section we scrutinize the: i) overhead of disk I/O prioritization, and ii) overhead of network prioritization.

Efficiency of network prioritization. We investigate the overhead of the Linux network prioritization mechanism, incurred by high priority requests in the presence of a low priority network flow. We set up a client connected to a server on a 10Gbps link. The client runs the DP-Network application and generates requests according to the web search workload ([14]) at medium load. The ToS bit for all these requests are marked as "high" while a single long TCP flow runs in the background at low priority. Figure 10a shows the CDF of the RCTs of the high priority requests. Ideally, adding background traffic shouldn't have an impact on the RCTs of the high priority requests due to the enforced prioritization. We observe an overhead of $\leq$ 20% at p10, at p20 it is $\leq$ 15% and at p40 and beyond it is $\leq$ 3%. This shows that network prioritization has some overhead for small requests (under 20%), but the gains due to duplication outweigh this overhead, as shown earlier in our macrobenchmark experiments.

Efficiency of disk I/O prioritization. We evaluate the efficiency of Linux's CFQ [7] scheduler which we used for disk I/O prioritization. For this experiment, we deploy a 100GB data set on a Linux server and run variable number of high priority and low priority workers. Each worker continuously requests random 4KB blocks in a closed loop. Figure 10b shows the CDF of RCTs of high priority requests with and without low priority background requests. We observe that roughly 20% of these requests are hurt by the presence of low priority requests. While for realistic workloads the gains of duplication outweigh this penalty, we have also verified that this overhead can be further reduced by tuning CFQ configuration parameters (e.g., *slice_sync, slice_idle_time*, etc).

DAS Overhead. We now evaluate the redundant (extra) work (in terms of bytes processed) done by DAS in comparison with other duplication schemes. Note that our previous experiments show that doing this redundant work has no adverse effect on application performance. However, the cost of doing redundant work may translate in other forms, such as additional cost of using the cloud resources (e.g., network bandwidth) or additional energy consumption, aspects that require further investigation as part of future work. For our analysis of redundant work, we use the same setup as used in §6.2.

Figure 11 shows duplication overhead, as a percentage of additional number of bytes received on client, compared to the baseline (Single-Copy). At low load, request queues don't build up i.e., a large number of duplicate requests get scheduled as soon as they arrive into the system and finish processing before `purge()` is initiated. This results in high percentage of redundant work done by DAS. As the load increases, request queues start to build up, which results in purging becoming more effective in eliminating the redundant work done by DAS. Specifically, many low priority requests are either purged from the request queue or purged before they complete which results in reduced overhead at medium and high loads. The figure also shows that Hedged and AppTO have low overheads at low load because they duplicate only a fraction of jobs. However, increase in system load affects duplication decision of these scheme (e.g., 95^{th} percentile of expected latency). This leads to duplication of more and more requests and results in high overheads and even leads to system instability (e.g., Hedged at high load).

7 RELATED WORK

Other Duplication Policies. To supplement the earlier discussion in §2.1, we comment on the most relevant duplication policies. RepFlow [74] replicates certain flows (but at the *same* priority) while other duplication schemes have highlighted the benefits of prioritization through analysis and simulations [32, 33, 45, 70]. We

show that for practical systems, in addition to prioritization, purging *must* be used. Further, none of the above schemes focus on an abstraction for supporting duplicates. A recent work, MittOS [40] introduced a fast rejecting SLO-aware interface to support millisecond tail tolerance but only consider storage stacks [40] – its technique to reject a request based on expected latency needs is highly customized for storage.

Elastic Data Replication. Elastic data replication [13, 16, 26, 71] is considered an effective strategy to mitigate stragglers caused by skewed data demands (e.g. *hot* and *cold* objects [26]) in distributed processing frameworks (e.g., Hadoop). While these proposals look at how to determine the right number of replicas for different data objects, they are orthogonal to DAS. DAS focuses on retrieving data from existing replicas: it randomly selects two out of any number of available replicas to service *read* requests.

Stage Abstraction. Welsh et al. proposed the staged event-driven architecture (SEDA) for designing scalable Internet services [72]. In SEDA, applications consist of a network of event-driven stages connected by explicit queues. In contrast, we consider *duplicate-aware* stages (D-Stages), where each stage may correspond to a different resources (e.g., network, compute, and storage). IOFlow [67] introduces the abstraction of a data plane stage for storage systems but does not consider duplication. We show that duplication brings its own unique challenges in designing stages for each bottleneck resource and structuring their interactions.

Metadata Propagation. A plethora of recent work [50, 51, 59, 62] focuses on context propagation (e.g., request IDs) along with individual requests to record the work done on behalf of each request, within and across the nodes of a distributed software system. However, propagating context across layers other than the application (e.g., kernel) is still an open problem [21]. DAS makes use of layer specific methods to deal with this challenge (§3.1).

Resource Capacity Estimation. Our Proxy D-Stage's use of capacity estimation in its throttling mechanism is similar to resource capacity estimation in systems like PARDA [35] and VDC [20]. Similar to us, these systems use capacity estimation techniques that are inspired by TCP. However, these systems focus on dealing with increased latency or SLO violations whereas our goal is to maintain maximal control at the Proxy D-Stage while avoiding under utilization.

8 DISCUSSION AND FUTURE WORK

Implementation Complexity. The implementation complexity of DAS is dependent on the layer; in general, higher system layers (e.g., application) are easier to modify compared to lower layers (e.g., a system driver). Similarly, supporting different features of DAS (e.g., prioritization, purging, etc) pose different challenges at each layer. For example, purging packets from inside network switches is challenging, whereas prioritization is readily supported in today's network switches [15, 22]. These challenges stem from the brittle nature of legacy systems [53, 63]. However, emerging technological trends (e.g., programmable network switches [24] and programmable storage stacks [61]) can facilitate supporting DAS at deeper layers.

Duplicates with User-level Network Stacks. The support for duplicates can be effectively introduced in today's high performance user-level network stacks [4, 47] that use kernel-bypass and leverage network I/O libraries such as DPDK [2] or Netmap [58]. Existing NICs provide support for multiple queues and by applying appropriate filters (e.g., by using Intel's Ethernet Flow Director), duplicate traffic can be pinned to separate queues. These queues may then be served by a (strictly) low priority thread.

Work Aggregation. Making duplicates safe to use opens up another opportunity: work aggregation. This requires the scheduler to do *fine-grained work*, allowing *aggregation* of the fine-grained work done by the the primary and duplicate copies of the job. For example, consider a job – having two sub-parts (A and B) – that is being processed at two different stages. The primary copy can first process part A while the duplicate copy processes part B – aggregating these parts can allow the job to finish even though both the primary and duplicate copies are not individually finished.

Redundancy-based Storage Systems. Redundancy-based and quorum-based storage systems can also naturally deal with stragglers [55, 56], but they incur high reconstruction cost because they need to make more requests than required. Such systems can potentially use D-Stages to reduce overhead. For example, K out of N requests can be high priority while the other $N - K$ can be lower priority.

Energy Considerations. While making duplicates may increase energy consumption, there are two factors which can limit this overhead or may even reduce the overall energy consumption: i) we use purging so the overall work done by the system (in terms of executing a request) may not necessarily be significant, and ii) the overall reduction in response time implies that requests stay in the system for less time, possibly consuming fewer system resources and thus lowering the energy consumption.

9 CONCLUSION

Tail latency is a major problem for cloud applications. This paper showed that by making duplicate requests a first-class concept, we can proactively use replicas without worrying about overloading the system. To this end, we proposed a new scheduling technique (DAS), an accompanying abstraction (D-Stage) that helps realize DAS for different resources, and end-to-end case studies and evaluation that highlight the feasibility and benefits of DAS for different bottleneck resources.

REPRODUCIBLITY

We have published the source code used for our main experiments at https://github.com/hmmohsin/DAS. We direct the readers to README.md file in the repository for detail about how to setup and run the experiment.

ACKNOWLEDGEMENTS

We thank our shepherd Robert Beverly, the anonymous CoNEXT reviewers, and Raja Sambasivan for their constructive feedback on this work. This work was supported by NSF CNS under award number 1618321.

REFERENCES

[1] 2017. Apache Hadoop. https://hadoop.apache.org/.
[2] 2017. DPDK: Data Plane Development Kit. http://dpdk.org/.
[3] 2017. Emulab. http://www.emulab.net.
[4] 2017. F-Stack: High Performance Network Framework Based On DPDK. http://www.f-stack.org/.
[5] 2017. Google Cloud. https://cloud.google.com/.
[6] 2017. Google Cloud Persistent Disk. https://cloud.google.com/compute/docs/disks/#pdspecs.
[7] 2017. Kernel Document. https://www.kernel.org/doc/Documentation/block/cfq-iosched.txt.
[8] 2017. Linux Manpage. http://man7.org/linux/man-pages/man2/ioprio_set.2.html.
[9] 2017. Linux Pthread Manpage. http://man7.org/linux/man-pages/man3/pthread_setschedprio.3.html.
[10] 2018. Google Cloud. https://cloud.google.com/.
[11] 2018. Packet Bricks. https://github.com/bro/packet-bricks.
[12] 2018. Snort3. https://www.snort.org/snort3.
[13] C. L. Abad, Y. Lu, and R. H. Campbell. 2011. DARE: Adaptive Data Replication for Efficient Cluster Scheduling. In *Proc. IEEE International Conference on Cluster Computing*.
[14] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data center TCP (DCTCP). In *Proc. ACM SIGCOMM*.
[15] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. 2013. pFabric: Minimal Near-optimal Datacenter Transport. In *Proc. ACM SIGCOMM*.
[16] Ganesh Ananthanarayanan, Sameer Agarwal, Srikanth Kandula, Albert Greenberg, Ion Stoica, Duke Harlan, and Ed Harris. 2011. Scarlett: Coping with skewed content popularity in MapReduce clusters. In *Proc. EuroSys*.
[17] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. 2013. Effective Straggler Mitigation: Attack of the Clones.. In *Proc. Usenix NSDI*.
[18] Ganesh Ananthanarayanan, Michael Chien-Chun Hung, Xiaoqi Ren, Ion Stoica, Adam Wierman, and Minlan Yu. 2014. GRASS: trimming stragglers in approximation analytics. In *Proc. Usenix NSDI*.
[19] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. 2010. Reining in the Outliers in Map-reduce Clusters Using Mantri. In *Proc. USENIX OSDI*.
[20] Sebastian Angel, Hitesh Ballani, Thomas Karagiannis, Greg O'Shea, and Eno Thereska. 2014. End-to-end Performance Isolation Through Virtual Datacenters. In *Proc. USENIX OSDI*.
[21] Dan Ardelean, Amer Diwan, and Chandra Erdman. 2018. Performance Analysis of Cloud Applications. In *Proc. USNIX NSDI*.
[22] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. 2015. Information-Agnostic Flow Scheduling for Commodity Data Centers. In *Proc. Usenix NSDI*.
[23] Wei Bai, Li Chen, Kai Chen, and Haitao Wu. [n. d.]. Enabling ECN in Multi-Service Multi-Queue Data Centers. In *Proc. Usenix NSDI*.
[24] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.* 44, 3 (July 2014), 87–95.
[25] Guo Chen, Yuanwei Lu, Yuan Meng, Bojie Li, Kun Tan, Dan Pei, Peng Cheng, Layong Luo, Yongqiang Xiong, Xiaoliang Wang, et al. 2016. Fast and Cautious: Leveraging Multi-path Diversity for Transport Loss Recovery in Data Centers.. In *Proc. USENIX ATC*.
[26] Z. Cheng, Z. Luan, Y. Meng, Y. Xu, D. Qian, A. Roy, N. Zhang, and G. Guan. 2012. ERMS: An Elastic Replication Management System for HDFS. In *Proc. IEEE Cluster Computing Workshops*.
[27] Mosharaf Chowdhury, Srikanth Kandula, and Ion Stoica. 2013. Leveraging endpoint flexibility in data-intensive clusters. In *Proc. ACM SIGCOMM*.
[28] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* (Feb. 2013), 74–80.
[29] Fahad R Dogar, Thomas Karagiannis, Hitesh Ballani, and Antony Rowstron. 2014. Decentralized Task-aware Scheduling for Data Center Networks. In *Proc. ACM SIGCOMM*.
[30] Fahad R Dogar and Peter Steenkiste. 2012. Architecting for Edge Diversity: Supporting Rich Services Over an Unbundled Transport. In *Proc. ACM CoNext*.
[31] Abdullah Bin Faisal, Hafiz Mohsin Bashir, Ihsan Ayyub Qazi, Zartash Uzmi, and Fahad R. Dogar. 2018. Workload Adaptive Flow Scheduling. In *Proc. ACM CoNEXT*.
[32] Kristen Gardner. 2017. Modeling and Analyzing Systems with Redundancy. PhD thesis. http://www.cs.cmu.edu/~harchol/gardner_thesis.pdf.
[33] Kristen Gardner, Mor Harchol-Balter, Esa Hyytiä, and Rhonda Righter. 2017. Scheduling for efficiency and fairness in systems with redundancy. *Performance Evaluation* (2017).
[34] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. 2009. VL2: a scalable and flexible data center network. In *Proc. ACM SIGCOMM*. ACM.
[35] Ajay Gulati, Irfan Ahmad, and Carl A. Waldspurger. 2009. PARDA: Proportional Allocation of Resources for Distributed Storage Access. In *Proc. USENIX FAST*.
[36] 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. 2014. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proc. ACM SoCC*.
[37] Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, and Kurnia J. Eliazar. 2016. Why Does the Cloud Stop Computing?: Lessons from Hundreds of Service Outages. In *Proc. ACM SoCC*.
[38] Haryadi S. Gunawi, Riza O. Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, Gary Grider, Parks M. Fields, Kevin Harms, Robert B. Ross, Andree Jacobson, Robert Ricci, Kirk Webb, Peter Alvaro, H. Birali Runesha, Mingzhe Hao, and Huaicheng Li. 2018. Fail-Slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems. In *Proc. USENIX FAST*.
[39] Dongsu Han, Ashok Anand, Fahad Dogar, Boyan Li, Hyeontaek Lim, Michel Machado, Arvind Mukundan, Wenfei Wu, Aditya Akella, David G. Andersen, John W. Byers, Srinivasan Seshan, and Peter Steenkiste. 2012. XIA: Efficient Support for Evolvable Internetworking. In *Proc. USENIX NSDI*. San Jose, CA.
[40] 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 *Proc. ACM SOSP*.
[41] Osama Haq and Fahad R. Dogar. 2015. Leveraging the Power of the Cloud for Reliable Wide Area Communication. In *Proc. ACM Hotnets*.
[42] Osama Haq, Cody Doucette, John W Byers, and Fahad R Dogar. 2019. Judicious QoS using Cloud Overlays. *arXiv preprint arXiv:1906.02562* (2019).
[43] Osama Haq, Mamoon Raja, and Fahad R. Dogar. 2017. Measuring and Improving the Reliability of Wide-Area Cloud Paths. In *Proc. WWW*.
[44] Tyler Harter, Dhruba Borthakur, Siying Dong, Amitanand Aiyer, Liyin Tang, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2014. Analysis of HDFS Under HBase: A Facebook Messages Case Study. In *Proc. USENIX FAST*.
[45] Ali Musa Iftikhar, Fahad Dogar, and Ihsan Ayyub Qazi. 2016. Towards a Redundancy-Aware Network Stack for Data Centers. In *Proc. HotNets*.
[46] Syed Mohammad Irteza, Hafiz Mohsin Bashir, Talal Anwar, Ihsan Ayyub Qazi, and Fahad Rafique Dogar. 2017. Load balancing over symmetric virtual topologies. In *Proc. IEEE INFOCOM*.
[47] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *Proc. NSDI*.
[48] Bojie Li, Kun Tan, Layong Larry Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. 2016. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *Proc. ACM SIGCOMM*.
[49] S. Liu, H. Xu, L. Liu, W. Bai, K. Chen, and Z. Cai. 2018. RepNet: Cutting Latency with Flow Replication in Data Center Networks. *IEEE Transactions on Services Computing* (2018).
[50] Jonathan Mace, Peter Bodik, Rodrigo Fonseca, and Madanlal Musuvathi. 2015. Retro: Targeted Resource Management in Multi-tenant Distributed Systems. In *Proc. USENIX NSDI*.
[51] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2015. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *Proc. SOSP*.
[52] Ali Munir, Ghufran Baig, Syed M Irteza, Ihsan A Qazi, Alex X Liu, and Fahad R Dogar. 2014. Friends, not Foes: Synthesizing Existing Transport Strategies for Data Center Networks. In *Proc. ACM SIGCOMM*.
[53] Akshay Narayan, Frank Cangialosi, Deepti Raghavan, Prateesh Goyal, Srinivas Narayana, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. 2018. Restructuring Endpoint Congestion Control. In *Proc. ACM SIGCOMM*.
[54] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: Distributed, Low Latency Scheduling. In *Proc. SOSP*.
[55] George Parisis, Toby Moncaster, Anil Madhavapeddy, and Jon Crowcroft. 2013. Trevi: Watering Down Storage Hotspots with Cool Fountain Codes. In *Proc. ACM HotNets*.
[56] K.V. Rashmi, Nihar B. Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. 2014. A "Hitchhiker's" Guide to Fast and Efficient Data Reconstruction in Erasure-coded Data Centers. In *Proc. ACM SIGCOMM*.
[57] Xiaoqi Ren, Ganesh Ananthanarayanan, Adam Wierman, and Minlan Yu. 2015. Hopper: Decentralized Speculation-aware Cluster Scheduling at Scale. In *Proc. ACM SIGCOMM*.
[58] Luigi Rizzo. 2012. Netmap: A Novel Framework for Fast Packet I/O. In *Proc. USENIX ATC*.
[59] Raja R. Sambasivan, Ilari Shafer, Jonathan Mace, Benjamin H. Sigelman, Rodrigo Fonseca, and Gregory R. Ganger. 2016. Principled Workflow-centric Tracing of Distributed Systems. In *Proc. SoCC*.
[60] Bianca Schroeder, Mor Harchol-Balter, Arun Iyengar, Erich M. Nahum, and Adam Wierman. 2006. How to Determine a Good Multi-Programming Level for External

Scheduling. In *Proc. IEEE ICDE.*
[61] Michael A. Sevilla, Noah Watkins, Ivo Jimenez, Peter Alvaro, Shel Finkelstein, Jeff LeFevre, and Carlos Maltzahn. 2017. Malacology: A Programmable Storage System. In *Proc. EuroSys.*
[62] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure.* Technical Report. https://research.google.com/archive/papers/dapper-2010-1.pdf
[63] Ioan Stefanovici, Bianca Schroeder, Greg O'Shea, and Eno Thereska. 2016. sRoute: Treating the Storage Stack Like a Network. In *Proc. USENIX FAST.*
[64] Christopher Stewart, Aniket Chakrabarti, and Rean Griffith. 2013. Zoolander: Efficiently Meeting Very Strict, Low-Latency SLOs. In *Proc. USENIX ICAC.*
[65] Riza O. Suminto, Cesar A. Stuardo, Alexandra Clark, Huan Ke, Tanakorn Leesatapornwongsa, Bo Fu, Daniar H. Kurniawan, Vincentius Martin, Maheswara Rao G. Uma, and Haryadi S. Gunawi. 2017. PBSE: A Robust Path-based Speculative Execution for Degraded-network Tail Tolerance in Data-parallel Frameworks. In *Proc. ACM SoCC.*
[66] Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. 2015. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *Proc. Usenix NSDI.*
[67] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. 2013. Ioflow: A software-defined storage architecture. In *Proc. ACM SOSP.*
[68] Beth Trushkowsky, Peter Bodík, Armando Fox, Michael J. Franklin, Michael I. Jordan, and David A. Patterson. 2011. The SCADS Director: Scaling a Distributed Storage System Under Stringent Performance Requirements. In *Proc. USENIX FAST.*
[69] Balajee Vamanan, Jahangir Hasan, and T.N. Vijaykumar. 2012. Deadline-aware Datacenter TCP (D2TCP). In *Proc. ACM SIGCOMM.*
[70] Ashish Vulimiri, Philip Brighten Godfrey, Radhika Mittal, Justine Sherry, Sylvia Ratnasamy, and Scott Shenker. 2013. Low latency via redundancy. In *Proc. ACM CoNext.*
[71] Qingsong Wei, Bharadwaj Veeravalli, Bozhao Gong, Lingfang Zeng, and Dan Feng. 2010. CDRM: A Cost-Effective Dynamic Replication Management Scheme for Cloud Storage Cluster. In *Proc. IEEE CLUSTER.*
[72] Matt Welsh, David Culler, and Eric Brewer. 2001. SEDA: an architecture for well-conditioned, scalable internet services. *ACM SIGOPS Operating Systems Review* 35, 5 (2001), 230–243.
[73] Zhe Wu, Curtis Yu, and Harsha V. Madhyastha. 2015. CosTLO: Cost-Effective Redundancy for Lower Latency Variance on Cloud Storage Services. In *Proc. USENIX NSDI.*
[74] Hong Xu and Baochun Li. 2014. RepFlow: Minimizing flow completion times with replicated flows in data centers. In *IEEE INFOCOM.*
[75] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. 2013. Bobtail: Avoiding Long Tails in the Cloud.. In *Proc. Usenix NSDI.*
[76] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. 2008. Improving MapReduce Performance in Heterogeneous Environments. In *Proc. USENIX OSDI.*