Toward Fast Eventual Consistency with Performance Guarantees

Feng Yan[1], Alma Riska[2], and Evgenia Smirni[1]

[1]College of William and Mary, Williamsburg, VA, USA, fyan,esmirni@cs.wm.edu
[2]EMC Corporation, Cambridge, MA, USA, alma.riska@emc.com

ABSTRACT

As data and its processing increasingly becomes more critical to enterprises and consumers alike, systems have started to cross the physical boundaries of a single data center. It is very common nowadays, for service providers and corporations to span systems and data across multiple geographic locations, with the goal of reducing the chance that the data or its services become unavailable in case of network, power, or other outages. With such architectures, comes the need to facilitate achievement of data redundancy and integrity while autonomously and transparently handling the added network delay during ingesting or updating data in the system. Systems have adopted the notion of *eventual consistency* which means that the targeted redundancy of data in the system is reached *asynchronously*, i.e., outside of the critical path, so that performance of user traffic is impacted minimally. Here we propose a scheduling framework that makes decisions about when to schedule the asynchronous tasks associated with new or updated data such that they are completed as soon as possible without violating user traffic quality targets. At the heart of the framework lies a learning methodology that extracts the characteristics of idle periods and infers the average amount of work to be done during idle periods so that asynchronous tasks are completed transparently to the user. Extensive trace-driven evaluation shows the effectiveness and robustness of the proposed framework when compared to common practices.

1. INTRODUCTION

The majority of computer systems today are faced with the need to scale because the services they provide and the data they store is increasing at a significant pace. In order to provide uninterrupted computing services and access to the related data, it has become necessary for systems to extend their boundaries beyond a single data or computing center. In designing such scaled-out distributed systems [1, 2, 3], it is necessary to balance cost, performance, reliability, and availability. Specifically, in distributed storage systems, it is expected that data is spread across multiple nodes and geographic locations such that a wide range of network, power, and other failures do not cause data unavailability [4, 5, 6]. Yet, as new data arrives in the system, from the performance perspective, it is not as efficient for the system to propagate the data to the various locations in real time, because the impact on end user performance (now including also WAN transfers) may be significant. A solution is for the system to distribute the data across the locations asynchronously [7, 8, 4]. As a result, the data reaches its expected locations *eventually* and the systems strive to achieve *eventual data consistency* [9, 10, 11]. Systems that aim to achieve eventual data consistency often provide cloud services [11, 12], making sure that data reliability is not compromised. For example, data is protected via RAID [13] locally. However, the location failure tolerance is achieved only eventually.

Eventual data consistency is a "loose" term. It means that data can eventually reach its distributed locations, but it just does not quantify how fast. This depends largely on the supporting infrastructure, e.g., the network bandwidth, the distance between the data centers, as well as the scale of the system and its quality goals, e.g., performance, reliability and availability. It also depends on how aggressively the system schedules the asynchronous tasks [14], given that they may interfere with the normal user traffic and impact its performance. Commonly these tasks are scheduled based on the current utilization levels of each node, i.e., asynchronous tasks are scheduled mostly during periods with low node utilization.

In this paper, we focus on how to schedule these asynchronous tasks that distribute data across different locations such that the performance in the sending and receiving nodes meets predefined quality of service goals. The scheduling parameters for the asynchronous tasks are determined and updated continuously at the individual node level as they learn the characteristics of the workload they are serving. Such parameters are exchanged between the nodes in order for them to synchronize the speed of data transfer. It is expected that different pairs of nodes in a geographically distributed system have different communication speeds. As a result, it is critical to synchronize the speed of data transfer so that failed attempts are reduced and eventual consistency is achieved faster.

The learning in our scheduling policy consists of understanding the available idleness that can be used to serve the asynchronous updates. We utilize the histogram of idle periods as it is done in [15] to determine when to start and stop servicing the tasks without violating performance goals, such as the degradation in user traffic response time.

Extensive experimentation with simulations driven by traces collected in real storage systems, demonstrates the robustness of our framework, which performs orders of magnitude faster than the common practice of utilization-based scheduling and very comparably to an aggressive policy that schedules the asynchronous tasks as soon as the system becomes idle. We note that our policy provides guarantees on the performance of each node in the system

ICAC-12, September 16-20, 2012, San Jose, CA, USA.

and reduces the time to reach consistency for newly added data, something that none of the alternative policies can achieve.

This paper is organized as follows. In Section 2, we introduce the background of consistency issues and state-of-art scheduling strategies in storage systems. In Section 3, we provide a detailed analysis of a set of enterprise traces and show how the characteristics of workload can help us to develop our scheduling framework. In Section 4, we propose an analytic framework that computes the scheduling parameters based on the learned characterization of idleness and other system information. Section 5 presents an extensive set of trace-driven experiments that demonstrates the effectiveness and robustness of the framework. Section 6 discusses the related work. We conclude and discuss future directions in Section 7.

2. BACKGROUND AND MOTIVATION

In this section, we provide general background on eventual consistency in scaled-out geographically distributed systems, data redundancy schemes, and aspects of data reliability and integrity when handled asynchronously. We also summarize the state of the art in scheduling techniques, which also motivates the work presented here.

2.1 Data Redundancy and Eventual Consistency

In fast growing data centers and global services, many of today's distributed storage systems need to meet several qualities simultaneously, such as performance, reliability, availability, security, and cost effectiveness. Traditionally, data redundancy is used in such systems to enhance availability, reliability, integrity, and performance. Yet because redundancy means that data (or parts of it) needs to be written multiple times, often in different locations, then it has become common to achieve the desired redundancy for each piece of data asynchronously rather than synchronously [7, 8, 4], which means that data is acknowledged to the user before it has successfully reached all its destination nodes. As a result, data consistency is often classified as follows [14]:

- *strong consistency*, where the system acknowledges the data after it has reached all nodes that hold it,
- *weak consistency*, where the system acknowledges the data as soon as it receives and stores it locally or partially. It allows the system to complete the data distribution to its destination nodes at a later time (i.e., asynchronously). In this case, there is a temporal gap between acknowledgment of updates and distribution of updates across the system, which we call here the "inconsistency window".

Weak consistency [16, 17] favors high system performance and availability and is preferred by applications that consider liveliness more important than durability [8]. Eventual consistency [9, 1, 18, 10, 4] is a specific type of weak consistency that implies that if no new updates are made to a data object, then eventually all copies of the object data get updated. The inconsistency window reflects data reliability since data loss may occur while the targeted redundancy is not reached immediately upon the system receiving the data. Here, we denote the node in the distributed system that receives the new data as the "active" node and the nodes that would receive replicas (or parts of the data) asynchronously "inactive nodes". In our exposition, data may arrive in any node in the system, which means that any node can be an active node for some data and inactive node for other pieces of data.

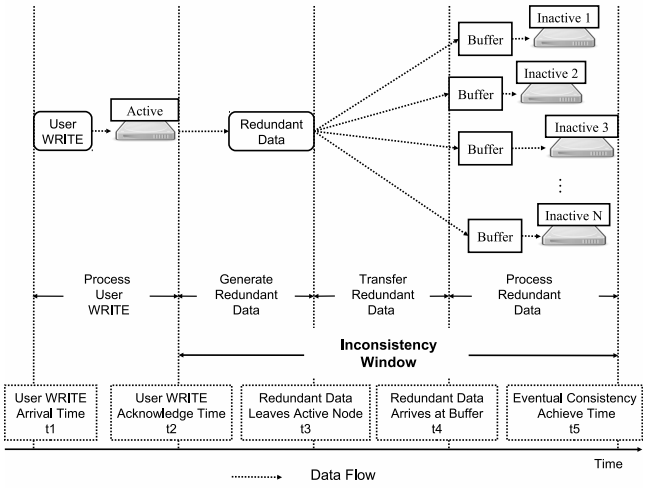

Figure 1: Schematic view of the system with asynchronous replication and eventual consistency.

A schematic view of how data is stored redundantly and asynchronously is shown in Figure 1. When new data arrives, it is acknowledged and processed by its active node and then spreads across the system (i.e., to the nodes that should receive it). In large scale storage systems today, data either is replicated in multiple locations or it is stripped, coded, and distributed in different locations (erasure coding). Independent of the specifics on how the data is redundantly stored, the fact is that the targeted redundancy is achieved asynchronously as background tasks (BG), which is outside the critical path of serving the user traffic.

When the redundant data is sent out from the active node, over the network, it can be delayed depending on the distance between the nodes and the amount of data being transferred. The inactive node that receives it, buffers it in cache before committing it to a storage device. The buffer is required because the inactive node may be serving its own user traffic, and the goal is not to impact its performance according to system quality targets. If the inactive node process such data upon arrival then its user performance impact may be severe. It is clear that the inconsistency window has three parts: the time it takes to send out the data from the active node, the time to transfer the data over the network, and the time to commit the data on the storage devices of the inactive node. The eventual consistency for each piece of data is achieved when all inactive nodes that should have received a copy or fragment of it have done so successfully. This means that from the perspective of modeling the duration of the inconsistency window, the problem can be simplified to having one active node and one inactive (slowest) node, without loss of generality.

The issue with the asynchronous traffic is that it impacts system performance regardless of how carefully it is scheduled because often IO tasks are not *instantaneously* preemptable [19, 20, 21]. Judicious scheduling of asynchronous tasks that is done as quickly as possible so that data durability is high, while user traffic is not affected, is challenging.

2.2 State of the Art in Scheduling of Background Jobs

In this section we quickly review three scheduling methods that are widely used to schedule background work in storage systems:

- *Aggressive* scheduling schedules replication work immedi-

ately and without any consideration of foreground user traffic. Such scheduling reduces the inconsistency window but may result in very high and unpredictable user performance degradation.

- *Utilization-guided (Aggressive)* scheduling takes the user traffic into consideration by monitoring utilization. If the system utilization is below a threshold, then it schedules replications immediately. When utilization is high, it stops scheduling any replication work.
- *Utilization-guided (Conservative)* scheduling uses system utilization as guidance and schedules the replication work only when the system utilization is low. Before scheduling any replication job during a low utilization interval, the system idle waits for certain amount of time [22] to avoid using small idle intervals, which have a higher chance to cause extra delays to user traffic.

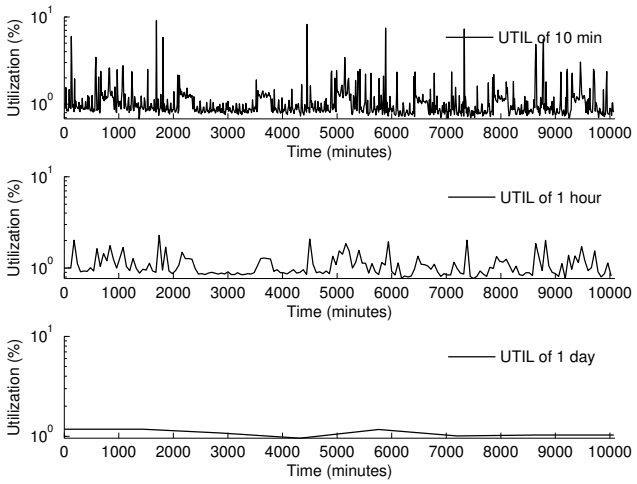

Figure 2: The utilization over time plots, the bin size for the top plot is 10 mins, for the middle one is 1 hour and for the bottom one is 1 day. Note y-axis is in log scale.

From the above policies, only the third one strives to reduce the performance impact of the inactive node traffic, although still without performance guarantees. Note that utilization-based policies depend on the characteristics of system utilization that may be very different across different time scales (e.g., minutes versus days). To illustrate this, we plot in Figure 2 the average utilization of a representative trace from Microsoft Research, and this trace is described in detail in the following section. The plot shows a large variance in utilization when looking in 10 minute, 1 hour, and 1 day windows and suggests that utilization, as a steady-state metric, is not suitable for scheduling purposes here. If utilization is monitored in a too long interval, then it cannot capture well the unpredictability of user traffic. If it is monitored in a too short interval, it may not be able to predict the near future correctly based on current and past information because utilization changes swiftly in such scale. This observation motivates us to devise a more sophisticated yet simple learning-based scheduling framework to overcome the above shortcomings.

3. WORKLOAD CHARACTERIZATION

In this section, we analyze the set of traces used in our evaluation. First, we give some general information about these traces. Then we further characterize the idle periods length in more details and give some intuitions on how we take advantage of such characterization for the purpose of running fast and with performance guarantees the asynchronous tasks that aim at achieving eventual consistency in a distributed storage system.

3.1 Overview of Traces

We use storage system traces made available through the SNIA IOTTA repository [23] collected by Microsoft from its servers in data centers and published by the Microsoft Research Cambridge (MSR) [24]. Each trace records information about a set of attributes for each I/O request. Specifically, for each IO, we have the arrival time stamp, request type (write/read), offset from the start of logical disk, request size, and response time. In addition, other storage features such as simultaneous IO requests are reflected in these traces.

Table 1 presents an overview of various statistical measures for four traces[1]. The usr0 trace is obtained from a user file server, the mds0 trace comes from a media server, the ts0 trace is collected from a terminal server, and the web0 trace is captured in the Web/SQL server. Each trace has a duration of one week (168 hours) and represents a wide range of common traffic patterns. From the table, we can see that these systems show very low utilization, which suggests that good opportunities exist for serving background work, such as asynchronous tasks. The relatively substantial Coefficient of Variation (C.V., which is a normalized measure of dispersion, defined as the ratio of the standard deviation to the mean) suggests that using idleness may be challenging because scheduling too much background work during small idle periods may cause performance degradation while during large idle periods, scheduling too little background work may waste idleness and slow down the synchronization speed. We also note these traces are WRITE dominant workloads for which the asynchronous tasks of propagating the data through the system nodes play a very important role.

3.2 Characteristics of Idle Periods

We further evaluate the characteristics of the idle periods because asynchronous tasks are to be scheduled during these intervals. Figure 3 shows the Cumulative Distribution Histogram (CDH) of the idle period lengths. The figure indicates that more than half of the idle periods are very small (note the log scale in the x-axis), which means that if we schedule the asynchronous tasks during these short intervals then it is highly possible that user requests may be delayed as a result of arriving to a system that is serving the asynchronous tasks, which cannot be preempted *instantly*. The goal is to incorporate the learning of characteristics of idle periods in a policy that schedules the asynchronous tasks such that they are served as fast as possible while the user requests are impacted in minimum.

Figure 4 plots the idle time intervals across time. The plots clearly show that there is a daily cycle pattern which suggests that if we characterize well these idle periods within such a cycle, then we may be able to accurately predict the next cycle. Comparing to the utilization, idleness depicts more of a cyclic behavior, making it more reliable as a metric to guide the scheduling policy. In addition, we expect that using the information from the CDH of idle intervals rather than a simple average value of idle interval lengths would result in more reliable predictions and robust scheduling.

[1]The Microsoft IOTTA repository has a larger number of traces than what we show here. We have selected only these four traces as representatives.

Trace	Duration (hour)	Utilization (%)	Average Arrival Rate (1/ms)	Average Service Rate (1/ms)	Average Response Time (ms)	Idle Length		R/W ratio
						Average (ms)	CV	
usr0	168	1.07	0.0012	0.1203	8.94	805.36	1.74	0.11
mds0	168	0.52	0.0007	0.1412	7.21	1404.16	1.93	0.03
ts0	168	0.61	0.0008	0.1455	7.06	1150.20	1.74	0.04
web0	168	0.72	0.0010	0.1468	7.12	959.72	2.11	0.13

Table 1: General trace information. ms stands for millisecond.

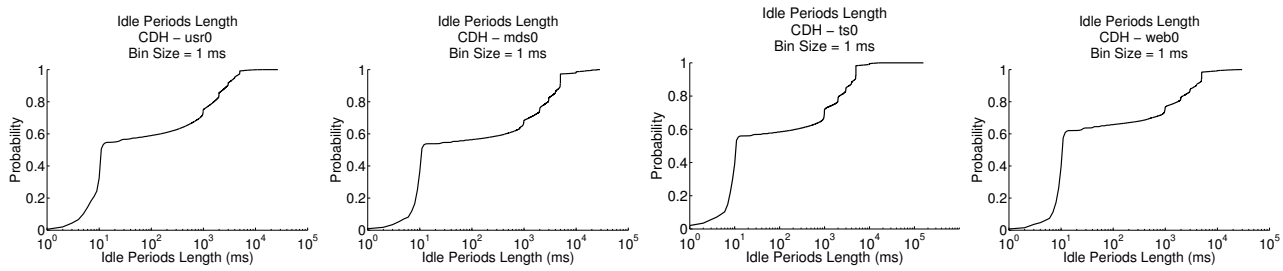

Figure 3: The CDH of idle period lengths measured in ms. Note that the x-axis is in log scale.

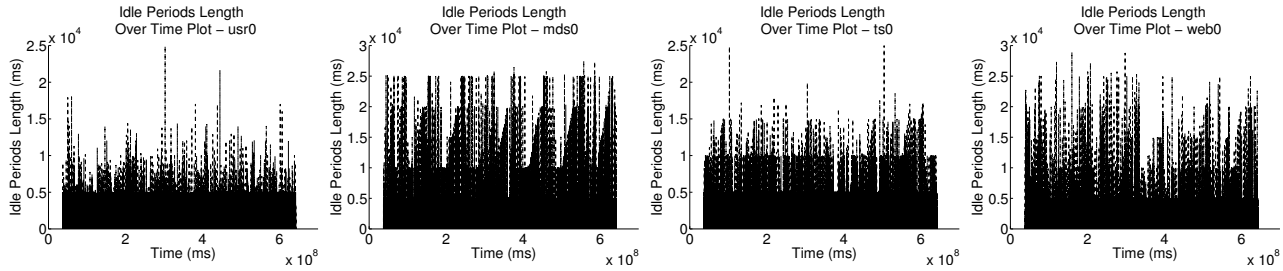

Figure 4: The idle periods length overtime plots.

4. ASYNCHRONOUS UPDATE SCHEDULING FRAMEWORK

In this section, we propose a learning-based framework for scheduling asynchronous updates. We first introduce the basic premise of the learning-based scheduling of background work. Then we explain in more details how to estimate the amount of replication work so that the framework can compute correct scheduling parameters.

4.1 Learning-based Scheduling with Performance Guarantees

We first describe an algorithmic framework that schedules background work, e.g. asynchronous tasks, with performance guarantees for the foreground traffic. This algorithmic framework is used to estimate the performance impact of background work and determine the most effective schedule for it by determining when and for how long to schedule background tasks in storage devices, such that the trade-off between performance degradation and how fast background tasks can be scheduled meets system performance targets.

One could argue that starting a background task immediately after the storage subsystem becomes idle would be most efficient. However, because of the stochastic nature of idle periods and the *non-instantaneously* preemptive nature of tasks in storage devices (e.g. disk drives), user performance may suffer significantly. In storage systems, it is very common to idle wait for some time before starting a background task, as to avoid utilizing the very short idle periods for any background activities [22]. In addition to that, [25] suggests that limiting the amount of time that the system serves background tasks further limits the performance impact on foreground jobs. The framework in [15] computes both the *idle wait* I and the duration T of the time to serve background jobs as a function of past workload (i.e., the stochastic characteristics of past idle periods). Note the T is introduced here so that the disk can be proactively ready to serve user traffic and therefore avoid user performance degradation in all the idle periods used for scheduling background work. We use here this (I, T) tuple to compute the schedules of the asynchronous updates in distributed storage systems, while meeting predefined performance targets.

Central to the calculation of I and T is the CDH of idle intervals. In addition to the CDH, the framework also uses the user-provided average performance (degradation) target D, which is defined as the allowed average relative delay of an IO operation due to the background tasks and can be computed from the (I, T) scheduling pair and other statistical information such as average response time.

Let's assume that W is the average IO waiting due to serving background tasks. Without loss of generality, we measure the idle interval length as well as the wait within the 1 ms granularity. Because a disk is activated upon an IO arrival, W can be at most P, which is the time penalty that a foreground IO request may suffer, if it arrives while the disk is still serving the asynchronous tasks of propagating new data throughout the distributed system. We assume that the data to be redundantly stored in distributed nodes is already stored in the local storage. As a result, the penalty can be estimated from the average service time of an IO request done to

the local storage, because when a new user request comes, it needs to wait until the asynchronous task completes. With "local storage" we mean any storage device that can be used, from memory to SSD to local disk. Consequently the penalty is different for different storage devices and we reflect it in our computations. However, we argue that in the scaled-out systems, where efficiency is key, utilizing memory or SSDs to store the data that is asynchronously distributed across the nodes, may not be the most cost-effective choice because these background work should be off the critical path for better user traffic performance. In our evaluations later-on we assume it is the disk IO and the penalty about several ms long depends on the specific characteristics of disk IO workload.

By denoting a possible delay by w and its respective probability by $Prob(w)$ then

$$W = \sum_{w=1}^{P} w \cdot Prob(w). \quad (1)$$

where the delay w caused to the IOs of the busy period following the scheduling of background tasks may be any value between 1 and P. Using the probabilities in the CDH of idle periods length, the probability of any delay w caused to the IOs of the following busy period is given by the equation below

$$Prob(w) = \begin{cases} \text{CDH}(I+T-w+1) - \text{CDH}(I+T-w), \\ \qquad\qquad \text{for } 1 \leq w < P \\ \text{CDH}(I+T-P) - \text{CDH}(I), \text{for } w = P, \end{cases} \quad (2)$$

where CDH(.) indicates the cumulative probability value of an idle interval in the monitored histogram. The intuition behind this equation is that for a scheduling pair (I, T), the delay to the busy period following the scheduling of background tasks is w ($1 <= w < P$) if the idle interval length is larger than $I + T - P$ and the probability is given as $CDH(I + T - w + 1) - CDH(I + T - w)$. And the delay is P for all idle intervals whose length falls between I and $I + T - P$, where the probability of this event is given as $CDH(I + T - P) - CDH(I)$.

To find the qualified scheduling pair (I, T), we scan the CDH of idle periods length for (I, T) pairs that would not violate the target D. Note that I and $I + T$ correspond to successive histogram bins. A pair (I, T) guarantees the performance target D if

$$D \geq \frac{W_{(I,T)}}{RT_{w/o\ BG}}, \quad (3)$$

where $RT_{w/o\ BG}$ is monitored and $W_{(I,T)}$ is computed using Eq. (1). Usually, larger D ensures faster background work completion.

4.2 Calculation of Scheduling Parameters

The replication work should be transparent to user performance. We measure transparency in terms of the performance degradation D as introduced earlier. The first scheduling target is to complete all replication work without violating any performance target. The algorithmic framework in 4.1 can be used to schedule asynchronous updates (e.g. replica WRITEs in disk IOs) during appropriate idle periods at both active and inactive nodes. The framework uses the histogram of idle periods length to generate a "schedule" for replication work and estimate the amount of completed work for each idle interval so that it is higher than the average amount of replica WRITEs. This is necessary to prevent uncontrolled replica backlogs. Here we estimate the average WRITE work amount B_W measured in units of time as

$$B_W = \frac{\rho_W * E[idle]}{1 - \rho_{FG}} \quad (4)$$

where ρ_W is the average utilization contributed to WRITE requests, ρ_{FG} is the average utilization of all user requests, and $E[idle]$ is the average idle interval length. The term $\frac{E[idle]}{1-\rho_{FG}}$ corresponds to the average length of one busy period plus one following idle period, and if multiplied by ρ_W, it represents the average amount of time WRITE requests need to be served during one busy plus one following idle periods.

As a second step, we use the framework introduced in 4.1 to compute all valid scheduling pairs (I, T) given the performance target D. Each scheduling pair schedules in average B_{BG} amount of background task measured in units of time in idle intervals at the storage nodes. We calculate B_{BG} as follows:

$$B_{BG} = \sum_{o=I}^{I+T-P} p(o) \cdot (o - I) + \sum_{o=I+T-P}^{max} p(o) \cdot (T - P) \quad (5)$$

where $p(o)$ is the probability that an idle interval is of length o, max is the maximum length of the idle intervals in the CDH. Intuitively, B_{BG} is comprised of two kinds of idle intervals that are larger than idle wait time I (intervals smaller than I are not used for replication work). The first type of idle intervals are of length o that falls between I and $I + T - P$. Because the replication work in this kind of intervals terminates at the end of each idle interval, which is before the limiting time T, their contribution to the overall B_{BG} is only $o - I$. The second type of idle intervals are of length o that at least $I + T - P$. In this case, the replication mode stays for T time units, so their contribution to the overall B_{BG} is $T - P$. Then we multiply them by the probability of each used interval and sum them together to get the average amount of replication work B_{BG}. Among all the valid scheduling pairs (I, T), we only choose the one with $B_{BG} >= B_W$ so that there is never replication work that is never served (i.e., there is no starvation). There might be multiple pairs that qualify for meeting both the target D and B_W. From those, we select the one with smallest I. If still multiple pairs qualify, we select the one with largest T so that the scheduling can schedule as aggressively as possible to ensure that replication work also finishes as fast as possible and there is no backlog.

4.3 Learning-based+ Scheduling

We also provide a more aggressive variation of the scheduling mechanism described above. The standard approach above only schedules for a T period of time for each idle interval longer than I. If there are still asynchronous tasks to complete upon T elapsing, the system does not schedule them even if it is still idle. For this reason we consider the above scheduling policy as being strictly work non-conserving, guided by both I and T.

Here we are proposing a more aggressive and less non-work-conserving policy by relaxing the condition on T. Specifically, after scheduling asynchronous tasks for T time units and the system remains idle with additional asynchronous work outstanding, then the policy is changed to wait another I in idle and re-start scheduling for another T time units. This is done repeatedly until there is no more asynchronous work to be served or the system becomes busy.

This extension to our framework, ensures that the very long idle intervals are utilized more if there are asynchronous tasks waiting for completion. It does not change the behavior for the short idle intervals, where the potential for delays to user traffic is higher. However, since the goal is to serve as fast as possible all the asynchronous tasks, then by allowing the long idle intervals (that are only a few) to be utilized more if there is work to be done, then we achieve a faster response time for asynchronous tasks without the additional delay on user performance.

Pairs	1	2	3	4
Active	usr0	mds0	ts0	web0
Inactiv	mds0	ts0	web0	usr0

Table 2: The traces used for pairing active and inactive nodes during experiments.

5. EXPERIMENTAL EVALUATION

In this section, we evaluate the proposed scheduling framework via an extensive set of experiments. We use the traces described in Section 3 to drive a set of simulations that experiment with our framework and other common practices as discussed in Section 2.2. The experiments that we present in this section validate the robustness and efficiency of our framework with regard to

- the time it takes to achieve the eventual consistency,
- the impact on user performance, and
- the amount of buffer space required to store all incoming data updates at the destination nodes before committing them on persistent storage.

5.1 Experiment Scenarios

The set of simulations that we developed to evaluate the framework proposed in Section 4 as well as the other baseline alternatives are driven by the Microsoft Research traces. Recall that the node that receives the new data is the "active node" and the node that does the same updates in the background (asynchronously) as the "inactive" one. The inconsistency window is composed of three parts: active node delay, network delay and inactive node delay as introduced in Section 2.1.

We apply our scheduling framework for both active and inactive nodes and focus on minimizing the delays experienced at these nodes. We do not limit the buffer space, contending that the faster we complete the synchronization of data the less buffer is needed. We also assume that there is no packet loss in the network and that the network delay is exponentially distributed with an average of 100 ms (i.e., the average delay for intercontinental round trip communication).

In our experiments, we use four different pairs of traces to evaluate our framework and the alternatives under 4 different workload combinations. These pairings are given in Table 2. For each workload combination, we divide the available traces into seven portions or time windows, each corresponding to a full day workload (i.e., recall that the traces are 7 days long). Recall that during learning we update the histogram of idle periods length, the average arrival and service rate of WRITE, the average arrival and service rate of all IO. Our framework uses these monitored parameters to compute the scheduling parameters, i.e., when and for how long during the idle interval, the asynchronous tasks are executed. The learning procedure of our framework occurs during one full time window and the learning results apply on the next time window. This means that we run our framework once a day and update the scheduling parameters accordingly. We run the experiments across all six time windows (the first day/time window is used only for learning), but due to the limited space, we only show results only for a subset of time windows.

In our experiments we evaluate the following solutions for achieving eventual consistency: the fully work-conservative approach (we label it as "Aggressive") that starts to serve the asynchronous tasks as soon as the node becomes idle. The "Utilization-based" policy monitors the utilization of the system for the past 10 minutes, and if it increases above a threshold (the threshold is chosen as the average utilization during a long period, e.g., one day), then no asynchronous tasks are scheduled. If utilization drops below the threshold, then asynchronous tasks are scheduled aggressively, i.e., as soon as the node becomes idle. Note we use 10 minutes as the measurement window for utilization-based approach because the utilization is a statistical parameter and if set too small (e.g. 1 min), it is statistically meaningless and such swift change is difficult to be used for predicting near future; if set too large (e.g. 1 hour), the synchronization speed is too slow and there is always backlog. The above two policies are evaluated as baseline versions to compare with the two scheduling versions of our framework, the basic "Learning-based" non-work-conserving version and the "Leaning-Based+" work-conserving variant introduced in Section 4.1.

Note that the "Utilization-based" approach is not work-conserving but is widely used in systems today, in an effort to limit the unpredictable performance impact that an "Aggressive" approach would have during periods of high utilization. Our experiments show that the impact of *all* alternative methodologies have an unpredictable impact on node performance and that only our "Learning-based" methods provide a solution that can maintain user-performance guarantees.

5.2 Delay on Achieving Eventual Consistency

Our initial experiments evaluate the total time that it takes, on the average, to propagate the new data or updates (e.g. WRITE in disk IOs) from the active node to the inactive node. Obviously, the faster the propagation of WRITEs, i.e., the smaller the inconsistency window, the more robust and resilient the system is because during the inconsistency window, the system has staled data in inactive nodes, which may cause various problems, e.g., impact the back-order rate in TPC-W system [7] or break the application's contract with the user as the classic example discussed in [8]. We provide the results of the experiments on the duration of the inconsistency window in Figure 5, each row of plots in the figure corresponding to the node pairs described in Table 2. Since our framework relies on the knowledge of various scheduling parameters including the CDH of idle intervals, we compute the (I, T) scheduling pair based on system measurements in the previous time interval (an entire day). The columns of Figure 5 correspond to results for three different days. Results are plotted for different user performance targets (in $\%$) (captured in the x-axis). For different performance targets (captured in the x-axis) there are different scheduling parameters for our framework and consequently, different results. However the results for the baseline approaches are independent of such goals and their corresponding results do not change across the x-axis.

The Aggressive approach performs best with regard to how fast the WRITEs propagate through the distributed system, because it represents the *only* work-conserving policy that we are evaluating here. However, as we show in the next subsection, it also causes the largest, possibly unbounded (e.g. the delay can propagate and accumulate) delays in user performance because. As a result, in systems today, it is rarely used, but we include it here to use its performance with regard to the length of the inconsistency window as a baseline of the possible minimum. The closer other policies come to this approach without sacrificing performance, the more resilient they are.

On the other hand the Utilization-based policy makes scheduling decisions based on the monitored utilization levels in the immediate past. Because of the strong oscillations in the short-term utilization, it behaves as a very conservative policy that does not take into consideration the available idleness in the system. Observe that the inconsistency window is orders of magnitude higher than the other alternative policies. Similar policies are common practices in systems today.

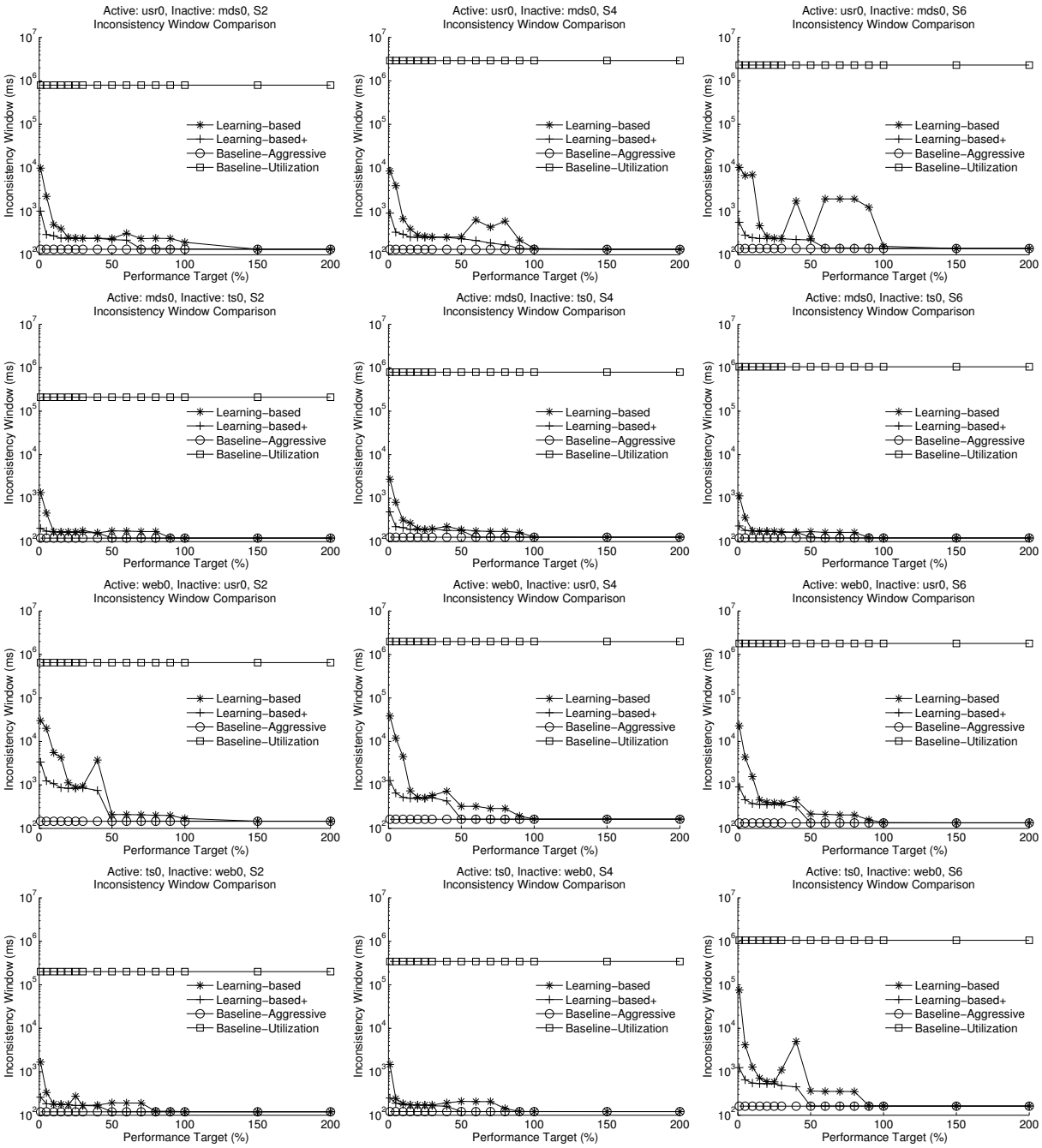

Figure 5: Inconsistency Window comparison between different scheduling for various active-inactive pairs (first row: usr0 - mds0, second row: mds0 - ts0, third row: web0 - usr0, fourth row: ts0 - web0. Three learning windows are considered: $Start = first\ day$ **(left column),** $Start = third\ day$ **(center column), and** $Start = fifth\ day$ **(right column).**

The curves corresponding to our framework, dynamically change as the target performance goal changes. As expected, for systems that are more sensitive to performance and where the target is low, the eventual consistency is achieved at a slower pace than when the performance target is less stringent. Our scheduling converges to the Aggressive scheduling as the performance target increases to the performance degradation caused by the Aggressive approach. Note that the higher the performance target, the smaller the value of I, which indicates how non-work-conserving the policy is (i.e., $I = 0$ and large T corresponds to a work-conserving policy). As expected Learning-based+ achieves eventual consistency faster than the basic Learning-based approach and converges faster to the Aggressive scheduling. The few fluctuations in our scheduling results is due to the fact that we use the learning of a previous day, which obviously can result in some errors on the predicted workload characteristics.

The main observation from Figure 5 is that our framework (both its versions) performs comparable to the Aggressive policy for any

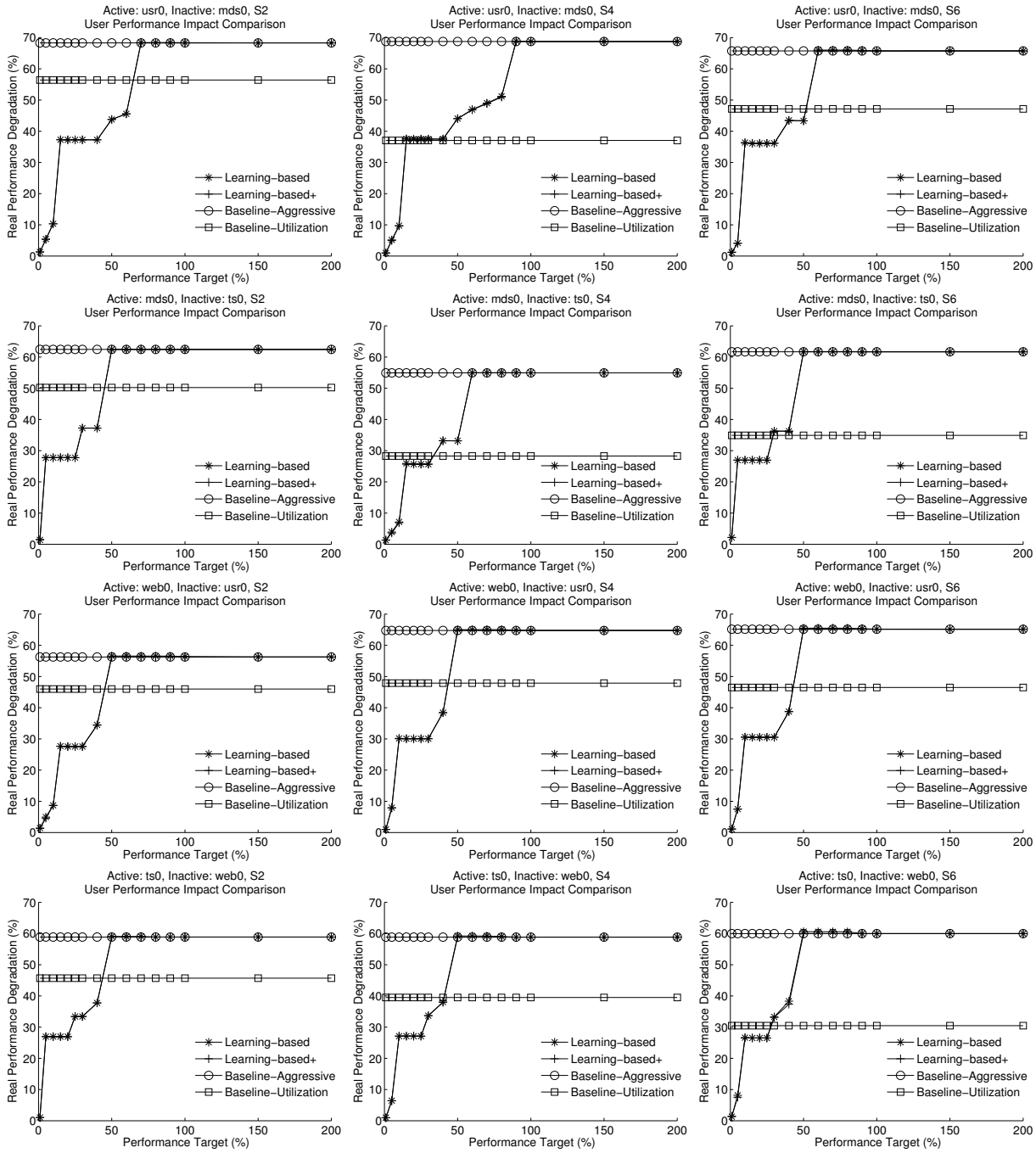

Figure 6: User performance impact comparison between different scheduling for various active-inactive pairs (first row: usr0 - mds0, second row: mds0 - ts0, third row: web0 - usr0, fourth row: ts0 - web0). Three learning windows are considered: $Start = first\ day$ **(left column),** $Start = third\ day$ **(center column), and** $Start = fifth\ day$ **(right column).**

performance target (excluding the very small and impractical ones 1-5%). The Utilization-based approach is orders of magnitude worse, as we show later, for several times higher performance degradation.

5.3 Impact on User Performance

As discussed above, the time it takes to propagate the WRITE traffic and achieve eventual consistency is highly dependent on how much the user performance is degraded. Recall that serving the IO replicas as background work delays foreground user requests that arrive while the system serves replica updates because IO tasks are not *instantaneously* preemptable. Here, we focus on how the various approaches perform with respect to foreground task degradation, measured as the percentage of the average user response time increase in presence of asynchronous tasks. We show the results in Figure 6, each row corresponding to different active-inactive pairs, and each column corresponding to different days in the trace. We still use the performance target (in %) as index of the x-axis and plot the *actual* performance degradation measured in simulations

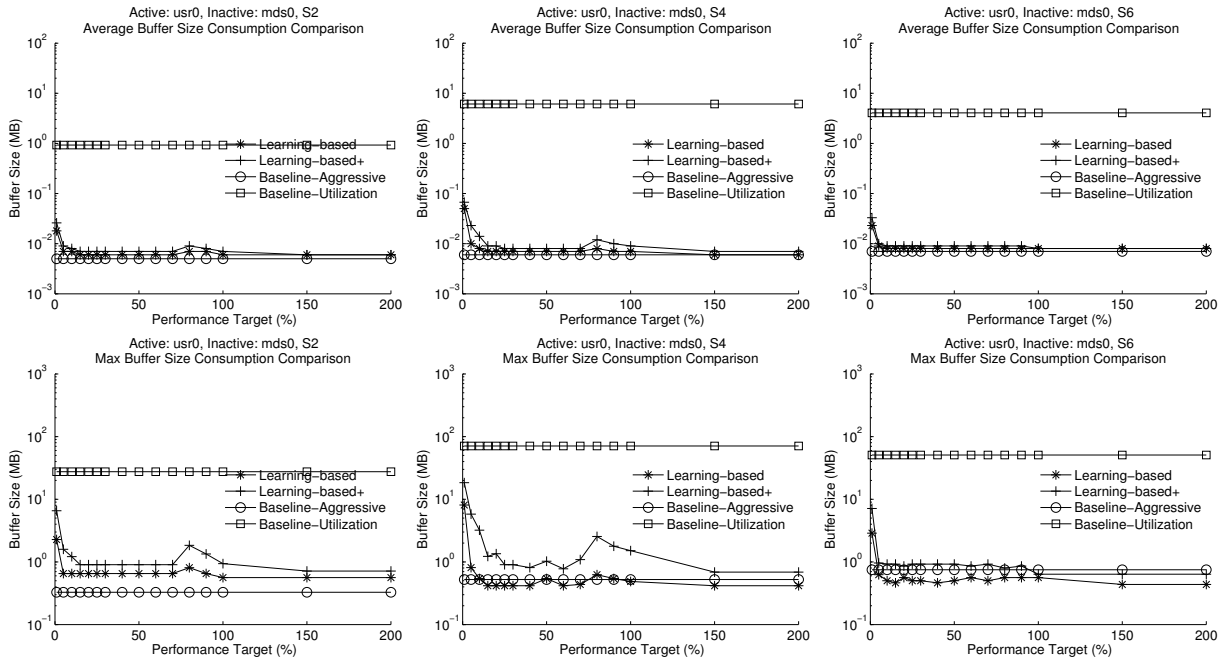

Figure 7: Buffer consumption comparison between different scheduling for usr0 - mds0 pair, both Standard and Aggressive Version of our framework are provided and also both Mean (first row) and Max (second row) Buffer consumption are provided. Three learning windows are considered: $Start = first\ day$ **(left graph),** $Start = third\ day$ **(center graph), and** $Start = fifth\ day$ **(right graph).**

(in %) in the y-axis.

As expected, the Aggressive policy performs very poor with regard to the actual user degradation in the system. The average user response time increases well beyond 50%, despite the fact that the asynchronous replica work is modest. The Utilization-based policy proves to be really ineffective, because although it results in very slow eventual consistency, it still penalizes user performance significantly, which attests to the inefficiency of making decisions based on short-term learning. We believe that not only the short-term learning is ineffective, but also the metric of utilization itself and a guide to scheduling asynchronous tasks, despite the fact that it is widely used in practice.

Our framework, on the other hand, adapts its decisions to the system quality targets striking a good balance between system user performance and replica completion speed with the goal of achieving eventual consistency quickly without significant performance loss. The results in Figure 6 confirm the robustness of periods of long learning (we update our learning once a day, see results per column) as being more robust and effective than shorter learning periods as used in the Utilization-based policy.

5.4 Buffer Space Requirements

Since there cannot be a perfect synchronization between the speed that the active node sends its updates with the speed that the inactive node processes them, there is a clear need for buffering at the inactive node to temporarily store the incoming replica WRITEs. Although we do not limit buffer availability here, as to be able to assess the maximum buffer requirement for each of the evaluated approaches, in real systems the buffer space is limited. Therefore, buffer size is preferred to be as small as possible. We show the required buffer size for the various policies in Figure 7 for the usr0 - mds0 pair. Results for the other three active-inactive pairs are not shown here for the interest of space but we remark that they are qualitatively the same as those reported in Figure 7. The x-axis in the graphs of Figure 7 is the performance degradation target (%) and y-axis is the required buffer space (in MB).

The Utilization-based policy demands the largest buffer space since under that policy the replica WRITEs accumulate for a long time before being served. The Aggressive policy requires the least buffer space because it serves the incoming asynchronous tasks the fastest. The buffer space under our scheduling policies depends on the performance target. The smaller the performance target the larger the buffer space. Yet, as expected it converges to the Aggressive policy buffer requirements for higher performance targets. Note, that there are cases when our framework consumes less maximum buffer space than the Aggressive policy. This is because the Aggressive policy causes often the WRITEs to arrive in large batches at the inactive node, while our framework smooths out this bursty behavior for sending out *almost* equal number of WRITEs every idle interval.

In conclusion, the results presented here support our claim that learning the characteristics of the right process, here the length of idle periods, is crucial to the effectiveness of the two learning-based approaches. Also, the workload in systems, as seen via captured traces from live systems, does not change drastically. As a result, learning over long periods of time, as we do in our framework (a whole day) results not only on a more resilient approach, but is also computationally inexpensive. Our framework, introduces only a small overhead on the system for monitoring and storing the results. System gains are nevertheless of orders of magnitude favorable regarding eventual consistency and user performance impact, which is critical for the availability, reliability, and performance of scaled-out systems.

6. RELATED WORK

In today's storage systems, there are various activities maintained in the background [26], aiming to increase the performance and reliability features. There is a large body of work in the literature suggests, in systems, periods of high utilization may be interleaved with idle times [25, 27, 22]. These idle times offer an opportunity to serve low priority tasks, such as synchronization and replication, but this may lead to performance degradation if a foreground task arrives and cannot be executed instantaneously.

Efforts have been placed to effective scheduling that can guarantee the foreground task. Conventionally, scheduling of non-preemptive background tasks is done using a non-work-conserving approach by delaying the execution of a background job during an idle interval [22]. This technique avoids using short idle intervals to serve long background jobs and to avoid severe degradation in foreground performance. Storage performance insulation has been achieved by co-scheduling timeslices for each striped workload in [28].

Our work significantly differs from the above in that instead of focusing on predicting the idle period size, we concentrate on the best way to coordinate scheduling between the active and inactive nodes to achieve quick eventual consistency without further degrading the foreground traffic in the system.

The work that is most related to the work in this paper is [15], where the authors propose a framework to estimate when and for how long idle periods can be used for processing low priority background tasks without violating pre-defined foreground performance target. In this paper, we generalized the algorithm in a current use case by introducing various analytic formulas and constrains. Though the algorithm part is partially based on [15], we are investigating a totally different problem. For example, in [15], the focus is trying to finish as much as possible the background work (measured by the amount of background work) in a single node, here we focus on how to serve background work as fast as possible (measured by the response time and buffer requirement) in a distributed scenario. The distinctively difference can also be identified from the totally different experiment environment setting, which driven by a new set of traces collected in a distributed scenario. In addition, in [15], it assumes the perfect information of future workload is known prior. In this paper, we discussed and experimented with specific learning strategy, e.g. predict the information of future workload by monitoring the past information.

7. CONCLUSIONS

In this paper, we presented a framework that facilitates the efficient synchronization of data distribution in the background for quick eventual data consistency, common in distributed storage systems, with user performance guarantees. The framework learns the idleness characteristics dynamically and determines how fast the data can be sent or received without violating performance goals. Once such capabilities are shared among the nodes in the distributed system, each pair can synchronize the speed of sending and receiving. The result is orders of magnitude faster than the common practices without performance loss or large buffer requirements on the receiving end for eventually consistency applications. Extensive experimentation via trace-driven simulation indicates that the learning process is robust and that the near past predicts reasonably the near future, with regard to idleness characteristics.

8. REFERENCES

[1] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *SOSP*, 2007, pp. 205–220.

[2] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *SOSP*, 2003, pp. 29–43.

[3] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber, "Bigtable: A distributed storage system for structured data (awarded best paper!)," in *OSDI*, 2006, pp. 205–218.

[4] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis, "Zeno: Eventually consistent byzantine-fault tolerance," in *NSDI*, 2009, pp. 169–184.

[5] J. Kubiatowicz, D. Bindel, Y. Chen, S. E. Czerwinski, P. R. Eaton, D. Geels, R. Gummadi, S. C. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Y. Zhao, "Oceanstore: An architecture for global-scale persistent storage," in *ASPLOS*, 2000, pp. 190–201.

[6] O. Tatebe, Y. Morita, S. Matsuoka, N. Soda, and S. Sekiguchi, "Grid datafarm architecture for petascale data intensive computing," in *CCGRID*, 2002, pp. 102–110.

[7] L. Gao, M. Dahlin, A. Nayate, J. Zheng, and A. Iyengar, "Application specific data replication for edge services," in *WWW*, 2003, pp. 449–460.

[8] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "Pnuts: Yahoo!'s hosted data serving platform," *PVLDB*, vol. 1, no. 2, pp. 1277–1288, 2008.

[9] W. Vogels, "Eventually consistent," *ACM Queue*, vol. 6, no. 6, pp. 14–19, 2008.

[10] E. Anderson, X. Li, A. Merchant, M. A. Shah, K. Smathers, J. Tucek, M. Uysal, and J. J. Wylie, "Efficient eventual consistency in pahoehoe, an erasure-coded key-blob archive," in *DSN*, 2010, pp. 181–190.

[11] H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu, "Data consistency properties and the trade-offs in commercial cloud storage: the consumers' perspective," in *CIDR*, 2011, pp. 134–143.

[12] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann, "Consistency rationing in the cloud: Pay only when it matters," *PVLDB*, vol. 2, no. 1, pp. 253–264, 2009.

[13] P. M. Chen, E. L. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson, "Raid: High-performance, reliable secondary storage," *ACM Comput. Surv.*, vol. 26, no. 2, pp. 145–185, 1994.

[14] A. D. Fekete and K. Ramamritham, "Consistency models for replicated data," in *Replication*, 2010, pp. 1–17.

[15] N. Mi, A. Riska, X. Li, E. Smirni, and E. Riedel, "Restrained utilization of idleness for transparent scheduling of background tasks," in *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems, SIGMETRICS/Performance*, 2009, pp. 205–216.

[16] K. Daudjee and K. Salem, "Lazy database replication with snapshot isolation," in *VLDB*, 2006, pp. 715–726.

[17] E. Pacitti, P. Minet, and E. Simon, "Fast algorithms for maintaining replica consistency in lazy master replicated databases," in *VLDB*, 1999, pp. 126–137.

[18] K. Petersen, M. Spreitzer, D. B. Terry, M. Theimer, and A. J. Demers, "Flexible update propagation for weakly consistent replication," in *SOSP*, 1997, pp. 288–301.

[19] M. McKusick and G. Ganger, "Soft updates: A technique for eliminating most synchronous writes in the fast filesystem," in *Proceedings of the annual conference on USENIX Annual Technical Conference*. USENIX Association, 1999, pp. 24–24.

[20] M. I. Seltzer, G. R. Ganger, M. K. McKusick, K. A. Smith, C. A. N. Soules, and C. A. Stein, "Journaling versus soft updates: Asynchronous meta-data protection in file systems," in *USENIX Annual Technical Conference, General Track*, 2000, pp. 71–84.

[21] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Analysis and evolution of journaling file systems," in *USENIX Annual Technical Conference, General Track*, 2005, pp. 105–120.

[22] L. Eggert and J. Touch, "Idletime scheduling with preemption intervals," in *In Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, 2005, pp. 249–262.

[23] "Snia iotta repository." [Online]. Available: http://iotta.snia.org/traces

[24] D. Narayanan, A. Donnelly, and A. I. T. Rowstron, "Write off-loading: Practical power management for enterprise storage," in *FAST*, 2008, pp. 253–267.

[25] R. A. Golding, P. B. II, C. Staelin, T. Sullivan, and J. Wilkes, "Idleness is not sloth," in *USENIX Winter*, 1995, pp. 201–212.

[26] E. Bachmat and J. Schindler, "Analysis of methods for scheduling low priority disk drive tasks," in *ACM Conference on Measurements and Modeling of Computer Systems (SIGMETRICS)*. ACM Press, 2002, pp. 55–65.

[27] A. Riska and E. Riedel, "Disk drive level workload characterization," in *Proceedings of the USENIX Annual Technical Conference*, May 2006, pp. 97–103.

[28] M. Wachs and G. R. Ganger, "Co-scheduling of disk head time in cluster-based storage," in *SRDS*, 2009, pp. 278–287.