2012 32nd International Conference on Distributed Computing Systems Workshops

Fast Eventual Consistency with Performance Guarantees for Distributed Storage

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*—Data centers are nowadays ubiquitous, in a worldwide scale, and often geographically dispersed. In such environments, data reliability and availability are enhanced via data redundancy throughout the distributed storage. Because user performance is important in data centers, data updates in such distributed environments are done such that *eventual consistency* is achieved. In this paper we utilize a learning-based framework that aims at scheduling data writes during user idle times such that the impact on user performance is limited within strict predefined targets while the updates are completed as fast as possible. The effectiveness and robustness of the proposed framework are illustrated via extensive trace-driven simulations.**

I. Introduction

To accommodate the fast growth of data centers and services, data or its fragments are distributed and replicated across a subset of nodes such that the system can tolerate a wide range of network, power, and/or other type of failures that could put off-line large quantities of data [1], [2]. As data is continuously updated, keeping all replicas current and consistent is challenging due to the impact that data creation/update has on user performance as it propagates to the destination nodes. If all updates are propagated to all of their replicas in real time, then the delays experienced by users at the nodes where data replication is involved can be significant. Asynchronous propagation of updates is an attractive alternative, as long as the data updates *eventually* reach all replicas in order for the system to achieve *eventual data consistency* [3], [4], [5].

The term "eventual consistency" implies that updates of data do reach all of their distributed replicas; it just does not quantify how fast all updates are completed. Naturally, this depends largely on the supporting infrastructure, e.g., the network bandwidth between the data centers, as well as the scale of the system and its quality goals, e.g., performance, reliability, and availability. However, for any piece of data that has not reached its consistency yet, there are vulnerabilities with regard to its integrity and reliability. A robust and resilient system achieves eventual consistency quickly for each piece of data that is created or updated. Key to meeting the system quality goals is the scheduling of the asynchronous updates [6], such that they minimally interfere with normal user traffic and complete as soon as possible. Commonly these tasks are scheduled based on the current utilization levels of each node; i.e., asynchronous updates are scheduled mostly during periods of low storage node utilization.

In this paper, we focus on how to schedule the asynchronous data updates such that the performance in the sending and receiving nodes meets predefined quality of service (QoS) goals. The scheduling parameters are determined and updated continuously at the individual node level as the scheduling framework "learns" the characteristics of the workload the nodes are serving. Such parameters are exchanged between nodes in order to synchronize the speed of the transmission, given that busy or performance sensitive nodes can send/receive data at different speeds.

The learning aspect of our scheduling policy consists of understanding the available idle times that can be used to serve the asynchronous updates as in [7]. The methodology in [7] is utilized to determine when to start and stop servicing asynchronous tasks without violating performance goals, such as the degradation in user traffic response time.

Extensive experimentation with simulations driven from traces collected in real storage systems, demonstrates the robustness of our framework. Evaluation results show that our framework is orders of magnitude faster than the common practice of utilization-based scheduling and completes its updates comparably to an aggressive policy that schedules the asynchronous updates as soon as the involved nodes become idle. We note that our framework provides guarantees on the performance of each node and reduces the time to achieve data consistency, something that none of the alternative policies achieves.

II. State of the Art and Motivation

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

- *Aggressive* scheduling schedules asynchronous updates immediately 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 the utilization. If the system utilization is below a threshold, then it schedules asynchronous updates immediately. When utilization is high, it stops scheduling any asynchronous updates.

1545-0678/12 $26.00 © 2012 IEEE
DOI 10.1109/ICDCSW.2012.21

- *Utilization-guided (Conservative)* scheduling uses system utilization as guidance and schedules the asynchronous updates only when the system utilization is low. Before scheduling any asynchronous updates during a low utilization interval, the system idle waits for a certain amount of time [8] to avoid using small idle intervals, which have a higher chance to cause extra delays to user traffic.

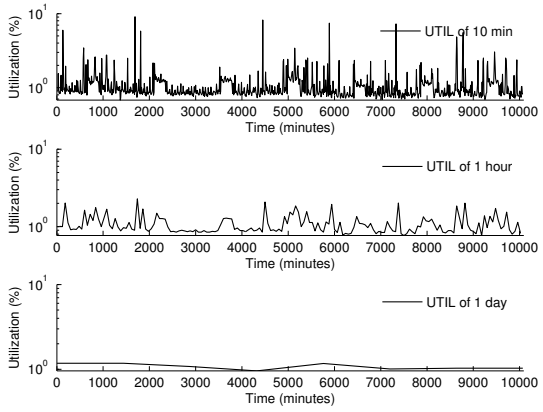

Fig. 1. Utilization over time for bin size 10 mins (top), 1 hour (middle) and 1 day (bottom). Y-axis is in log scale.

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

III. 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 work associated with asynchronous updates so that the framework can compute correct scheduling parameters.

A. Learning-based Scheduling with Performance Guarantees

We first describe an algorithmic framework that schedules asynchronous updates with performance guarantees for user traffic. This algorithmic framework estimates the performance impact of asynchronous updates. It determines the most effective schedule by examining when and for how long to schedule asynchronous updates during idle periods in storage devices, such that the trade-off between performance degradation and timely asynchronous updates meets system quality targets.

One could argue that starting the asynchronous updates immediately after the storage device becomes idle is most efficient. However, the stochastic nature of idle periods and the non-instantaneously preemptive nature of tasks in storage devices may cause delays to user requests when it arrives in a system that is serving the asynchronous tasks. In storage systems, it is very common to idle wait for some time before starting a background task to avoid utilizing the very short idle periods for any background activities [8]. In addition, [9] suggests that limiting the amount of time that the system serves background tasks further limits the performance impact on user traffic. The framework in [7] 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). We use here this (I, T) tuple for scheduling asynchronous updates during idle periods.

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 asynchronous updates and can be computed from the (I, T) scheduling pair and other statistical information such as average response time.

B. Calculation of Scheduling Parameters

The first target is for the scheduling of asynchronous updates (e.g., replica WRITEs) to remain transparent to the user, which is measured by the performance degradation D introduced earlier. Assume that W is the average IO wait due to serving replica WRITEs. 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 user request may suffer if it arrives while the disk is still serving the replica WRITEs. The penalty can be estimated from the average IO service time because when a new user request comes, it needs to wait until the asynchronous task completes. 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 replica WRITEs 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

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

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 replica WRITEs 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 do not violate the target D. 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).

The second scheduling target is to complete all replica WRITEs. Here we define the average replication work amount *target* B_W measured in units of time as

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

where ρ_W is the average utilization contributed to WRITEs, ρ_{FG} is the average utilization of all user requests, and $E[I]$ is the average idle interval length. The term $\frac{E[I]}{1-\rho_{FG}}$ 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 period plus one following idle one.

For a pair (I,T) that guarantees the performance target D as computed above, the average amount of replica work B_{BG} measured in units of time can be estimated 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, and max is the maximum length of the idle intervals in the CDH. Intuitively, B_{BG} is composed 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 interval is of length o that falls between I and $I+T-P$. Because the replication work in this kind of interval terminates at the end of each idle interval, which is before the limiting time T, its contribution to the overall B_{BG} is only $o-I$. The second type of idle interval is of length o with a vale of at least $I+T-P$. In this case, the replication mode stays for T time units, so its 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 (I,T) scheduling pairs that can meet performance target, we choose only the one that can also meet the replication work amount target ($B_{BG} >= B_W$) so that there is never replication work starving. There might be multiple pairs that qualify for meeting both the target D and target B_W. In this case, we select the one with the smallest I. If there are multiple pairs with the smallest I, we choose the one with the largest T so that it schedules as aggressively as possible, thereby ensuring that replication work finishes as fast as possible without backlog.

IV. Experimental Evaluation

In this section, we evaluate the proposed scheduling framework via an extensive set of experiments. First, we describe the traces to drive the simulation experiments. Then we experiment with our framework and other common practices as discussed in Section II. 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 and the impact on user performance.

We use storage system traces made available through the SNIA IOTTA repository [10] collected by Microsoft from its servers at their data centers and published by the Microsoft Research Cambridge (MSR) [11]. Each trace records information about a set of attributes for each IO 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.

Table I presents an overview of various statistical measures for four traces[1]. The usr0 trace is obtained from a user files 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 behaviors. From the table, we can see that these volumes show very low utilization, which suggests that good opportunities exist for serving background work, such as WRITE synchronization. The relatively substantial Coefficient of Variation (C.V., which is a normalized measure of the dispersion defined as the ratio of the standard deviation to the mean) suggests that using idleness may be challenging. We also note these traces are WRITE dominant workloads for which the asynchronous update strategy plays a very important role.

We plot the idle time intervals across time in Figure 2. The plots clearly show that there is a daily cycle pattern which suggests that if we characterize well these idle periods within 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.

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

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)	C.V.	
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 I
GENERAL TRACE INFORMATION.

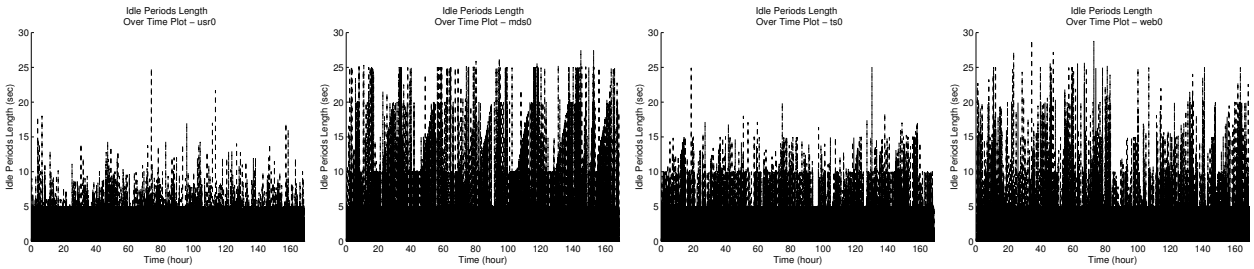

Fig. 2. The idle periods length overtime plots.

A. Experiment Scenarios

The set of simulations that we developed to evaluate the framework proposed in Section III as well as the other baseline alternatives are driven by the Microsoft Research traces. We call the node that receives the WRITE traffic (i.e. all updates/creates) the "active node" and the node that receives the asynchronous updates the "inactive" one (replica). The inconsistency window consists of 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.

We focus on minimizing the delays experienced at the active and inactive 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 two different pairs of traces to evaluate our framework, i.e., (msd-active, ts0-inactive) and (web0-active, usr0-inactive). For each pair, we divide the traces into seven portions or time windows, each corresponding to a full day workload. 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. Learning in 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. We also do experiments with other learning window and the results are not as good as one day, which verifies the choice according to the daily cycle as analyzed earlier.

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. The above two policies are evaluated as baseline versions to compare with our scheduling framework (we label it as "Learning-based").

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" method provides a solution that can maintain user-performance guarantees.

B. Delay on Achieving Eventual Consistency

Our initial experiments evaluate the total time that it takes, on the average, to propagate the WRITE from the active node to the inactive node. Obviously, the faster the propagation of WRITEs, i.e., the smaller the inconsistency window, and the more robust and resilient the system is. We provide the results of the experiments on the duration of the inconsistency window in Figure 3, each row of plots in the figure corresponding to the node pairs described in Section IV-A. Since our framework relies on the knowledge of various scheduling

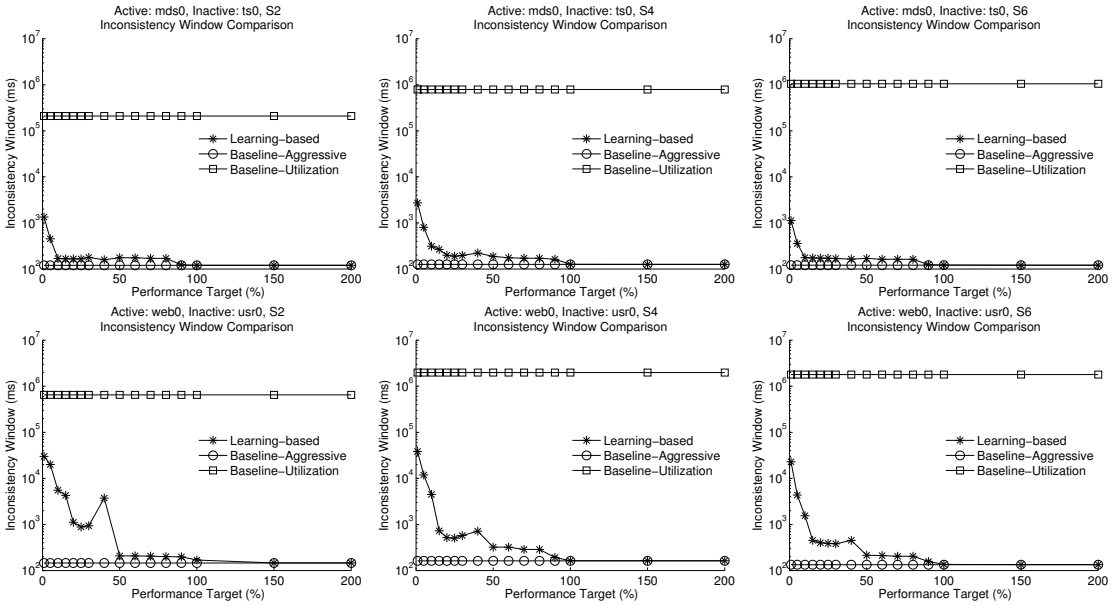

Fig. 3. Inconsistency Window comparison between different scheduling for various active-inactive pairs (first row: mds0 - ts0, second row: web0 - usr0. Three learning windows are considered: $Start = first\ day$ (left column), $Start = third\ day$ (center column), and $Start = fifth\ day$ (right column).

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 3 correspond to results for three different days. Results are plotted for different user performance degradation targets (in %) (captured in the x-axis). For different performance degradation 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, delays in user performance. 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.

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 degradation target is less stringent. Our scheduling converges to the Aggressive scheduling as the performance degradation target increases to the performance degradation caused by the Aggressive approach. Note that the higher the performance degradation target, the smaller the value of I, which indicate how (non)work conserving the policy is (i.e., $I = 0$ and large T corresponds to a work-conserving policy). 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 3 is that our framework (both its versions) performs comparable to the Aggressive policy for any performance degradation target (excluding the very small and impractical ones 1-5%). The Utilization-based approach is orders of magnitude worse, and as we show next, it also suffers from high performance degradation.

C. Impact on User Performance

As discussed above, the time it takes to propagate the asynchronous traffic and achieve eventual consistency is highly dependent on how much the user performance is degraded. Recall that serving the asynchronous updates as background work delays foreground user requests that arrive while the system serves asynchronous 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 4, each row corresponding to

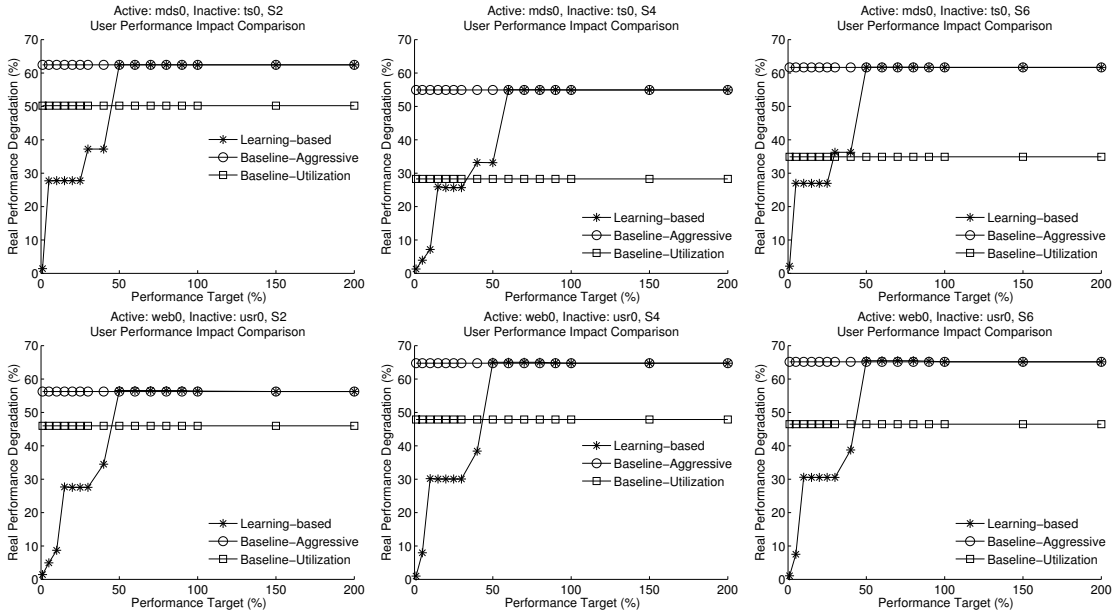

Fig. 4. User performance impact comparison between different scheduling for various active-inactive pairs (first row: mds0 - ts0, second row: web0 - usr0). Three learning windows are considered: $Start = first\ day$ (left column), $Start = third\ day$ (center column), and $Start = fifth\ day$ (right column).

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 (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 beyond 50%, despite the fact that the the work associated with asynchronous updates 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 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. The results in Figure 4 confirm the robustness of periods of long learning as being more robust and effective than shorter learning periods as used in the Utilization-based policy.

V. Conclusions

In this paper, we utilized a framework that learns the idleness characteristics in a storage node dynamically. It determines how fast the newly written data can be asynchronously sent or received to/from nodes in a distributed storage environment (like geographically distributed data centers) without violating performance goals so that eventual data consistency is achieved quickly. Our simulation results indicate that the framework performs orders of magnitude better than the common practices in terms of achieving consistency speed and maintaining performance.

Acknowledgments

This work is supported by NSF grants CCF-0811417 and CCF-0937925.

References

[1] 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.

[2] 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.

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

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

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

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

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