Busy Bee: How to Use Traffic Information for Better Scheduling of Background Tasks

Feng Yan
College of William and Mary
Williamsburg, VA, USA
fyan@cs.wm.edu

Alma Riska
EMC Corporation
Cambridge, MA, USA
alma.riska@emc.com

Evgenia Smirni
College of William and Mary
Williamsburg, VA, USA
esmirni@cs.wm.edu

ABSTRACT

Computer systems, in general, and storage systems, in particular, rely on meeting their performance, reliability, and availability targets via scheduling of management and maintenance activities as background tasks. Such tasks may cause significant delays to user workload if scheduled extemporaneously. Here, we propose a scheduling policy for background tasks that is based on the statistical characteristics of the system's busy periods and that aims at completing background work expediently. Extensive trace-driven simulations show that the scheduling policy is robust and that it succeeds in completing background work faster than common practices while impacting user performance minimally.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of Systems

General Terms

Performance, Algorithms

Keywords

performance, workload characterization, busy periods, background tasks, asynchronous tasks, user traffic, storage systems

1. INTRODUCTION

Systems that support emerging computing paradigms such as cloud computing are growing distinctively larger and more complex. In order to meet the ever increasing user needs for high availability, reliability, performance, and cost-effectiveness [13, 20, 12, 3], systems are built by integrating off-the-shelf components that are managed and maintained *asynchronously*, i.e., outside the critical path of user requests. While the amount and criticality of asynchronous management is commensurate with system complexity, the expectation for such work is to remain transparent from the system users. Examples of tasks that complete asynchronously in the system, i.e., in the background, include logging of monitored resources, garbage collection, data synchronization, and data verification. Within the storage component, a significant amount of work is completed asynchronously in the background, especially because storage tasks are not instantaneously preemptable [14, 19].

While background work in storage systems may be associated with performance improvement, e.g., moving data from the low performing tier of SATA drives to the high performing tier of SSD drives [8], it mostly targets enhancement of data availability and reliability, e.g., verification of data consistency for protection against bit-rots and replication of data in multiple storage devices or systems for added redundancy. The goal is to strike a balance between meeting user service level objectives while completing the background work as fast as possible. This goal is particularly important for background tasks that are time sensitive. Examples of time sensitive background tasks include geographically distributed data centers where data consistency is achieved only *eventually* by distributing the redundant new data asynchronously, in the background [22].

Judicious selection of scheduling *asynchronous work* vs. *user traffic* is not an easy task. The challenge lies in the fact that future user workload characteristics are seldom known a priori. If the background tasks are scheduled without consideration for user traffic, the impact on user performance may be severe.

Common practices use simplistic measures, such as average utilization, to guide background task scheduling. Such metrics cannot describe accurately current system conditions and often yield unstable solutions because the workload, particularly in storage systems, can be fairly dynamic over short time scales. To limit the impact of background work on user performance, there exist elaborate techniques that focus on idle waiting before starting background work [5, 7]. There are techniques that even provide guarantees on the performance impact caused to user performance [15]. While some techniques that are used to schedule background work operate on fixed parameters that restrict their adaptivity to a changing workload [5, 7], others rely on monitoring a variety of complex processes, such as system idleness, delays caused by the background tasks, and user performance [15].

In this paper, we present a simple yet adaptive solution to the problem of scheduling tasks in the background by proposing a quantitative framework that aims at monitoring, learning, and making scheduling decisions based on a few, easy to monitor metrics. The monitored metrics cap-

ICPE'12, April 22-25, 2012, Boston, Massachusetts, USA

ture sufficient details on the current foreground workload and the resulting available idle capacity that allow the proposed scheduling policy to complete the background work as fast as possible but with minimal impact on user performance. All scheduling decisions are based *only* on the stochastic characteristics of the length of *user busy periods* in the system. The goal is to schedule as much as possible background work when the impact on performance of user traffic is anticipated to be small (because upcoming busy periods are short) and limit delays on foreground traffic when busy periods are anticipated to be long.

Results from extensive experimentation via trace-driven simulations show that the proposed scheduling policy can maintain the same foreground performance while completing the asynchronous work up to 50% faster. The benefits of the proposed scheduling policy are particularly high when it matters most, i.e., when foreground performance imposes stringent limitations on the tolerance toward additional delays due to background work. The proposed scheduling policy enables the system to sustain its performance in the presence of background tasks, even where there are changes in the user traffic characteristics, by adapting the background scheduling parameters to current foreground characteristics. The robustness and resilience of the scheduling policy is evident especially under swift changes in user workload.

This paper is organized as follows. In Section 2, we give an overview of related work. In Section 3, we provide a detailed characterization of a set of enterprise traces and show how this characterization can be used to develop the new scheduling strategy. In Section 4, we propose a dynamic scheduling framework aiming at improving the performance of background work while maintaining foreground performance. Section 5 presents an extensive set of trace-driven experiments that demonstrates the effectiveness and robustness of the proposed scheduling technique. We conclude and discuss future directions in Section 6.

2. STATE OF THE ART AND MOTIVATION

Today's systems complete most of their resource management and maintenance tasks in the background. In storage systems there is a plethora of activities that are executed asynchronously as background tasks [1, 21] aiming at improving performance, reliability, and availability [11, 2, 24, 12]. In addition, a large body of literature points out the existence of idle periods that are interleaved with periods of high utilization [7, 18, 5]. These idle periods offer an opportunity to serve tasks of low priority, such as data synchronization, but may lead to performance degradation if a foreground task arrives while a background task is in service. This is the case especially in storage systems, because tasks are not instantaneously preemptable. As a result, the foreground requests could be unavoidably delayed when the system executes background tasks.

Conventionally, scheduling of background tasks is done using a non-work-conserving approach by delaying the execution of an outstanding background job with a fixed time when the system becomes idle of foreground workload [5]. This technique avoids using short idle intervals to serve long background jobs and averts severe degradation in foreground performance. Approaches for adaptively determining the amount of time that the system should stay idle, while there is background work to be completed, are proposed for power saving in mobile devices by spinning-down their disks [4, 10]. pClock is a framework that allows multiple workloads to share storage while achieves performance isolation via scheduling [9]. This approach may be also used to allocate spare system capacity to background jobs. Storage performance insulation has been achieved by co-scheduling time slices for each workload type [23].

In [15, 16], the authors propose a framework to estimate when and for how long to utilize idle periods in a system for processing low priority background tasks without violating pre-defined foreground performance targets. This is achieved by extending the non-work-conserving nature of background scheduling as first suggested in [7, 5]. The histogram of past idle intervals can be used to determine: (1) the amount of idle wait till a background task can start and (2) the amount of the expected idle time to be used for scheduling background tasks. The consequence is that the system may remain idle while background tasks are still outstanding after the estimated time to utilize an idle interval for background scheduling elapses. Key to the methodology developed in [15, 16] are the statistical characteristics of idle times which are used for effective background task scheduling.

Systems today have to support a wide range of background tasks. These tasks should be served transparently from foreground tasks but should not starve. Avoiding starvation is the primary target to be met. In addition, if the background tasks are time-sensitive, as it is often the case in storage systems, then they should complete as soon as possible. There is an ever increasing number of time-sensitive asynchronous tasks in storage systems that are served in the background. Examples of such tasks include the asynchronous data updates in geographically distributed data centers. The data in such systems resides in multiple devices, nodes, and locations for purposes of availability and performance. New data is committed asynchronously to all designated nodes in order to avoid network and other delays that may severely impact user perceived performance. As a result the consistency of data across the distributed system is achieved *eventually* as data is committed to its destinations as a background process [22]. Note that for as long as the data is not consistent across *all* of its assigned nodes, data integrity is compromised. This is a clear case where completion of background tasks is time sensitive.

In this paper, we strive to achieve two goals: first to complete the background work while avoiding starvation at all costs and second to reduce its response time as much as possible to better serve time sensitive tasks. Our aim is to maintain the performance of foreground tasks at the same level as common practices, e.g., the approach in [5], while serving background tasks faster. Background tasks in storage systems have similar service demands as foreground requests. This means that if a foreground request arrives to find the system serving a background task, then the delay expectation is approximately two times the average service demand of a foreground request (i.e., accounting for the background work to complete and the storage system to get ready - positioning - to serve the next request). We consider such a delay to be "tolerable". This means that controlling foreground delay due to background work is effectively done by delaying *only* the start of a background busy period.

Deploying any "wait period" before background tasks start execution [5, 15] would result in non-work-conserving scheduling of the background tasks with low degradation on fore-

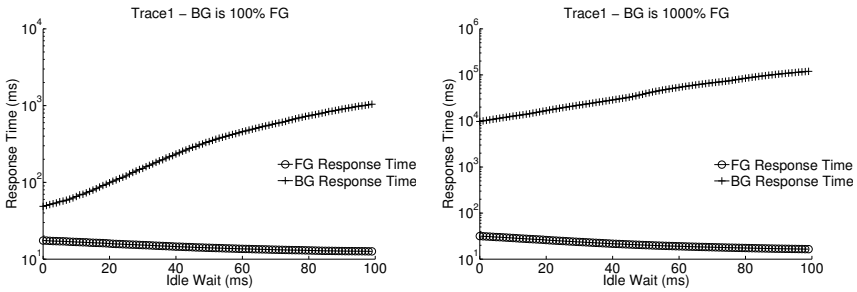

Figure 1: Performance comparison in terms of mean response time between the foreground and background tasks for a disk-level trace under conservative scheduling with fixed waiting ranging from 1ms to 100ms. The results of aggressive scheduling are also shown in the graph, i.e., the point corresponding to idle wait = 0. The response times are in log scale.

ground performance. Such non-work-conserving scheduling we denote as "conservative". A zero "wait period" would result in work-conserving scheduling of the background tasks and better utilization of the available idleness. We denote this policy as "aggressive". To gain intuition on the simultaneous effect on the performance of both foreground and background jobs, we evaluate the aggressive and the conservative scheduling policies via a set of trace-driven simulations. We consider constant idle wait times as in [5] ranging from 0 to 100 ms. Details on the disk drive traces that are used are provided in Section 3. Here we simply want to highlight the advantages and disadvantages of aggressive versus conservative scheduling.

As already discussed, background tasks in a system are commonly a function of the current workload (e.g., data synchronization). Therefore, it is reasonable to assume that multiple asynchronous features generate background work out of the incoming user workload. We explore here two scenarios where the background work (BG) is 100% and 1000% of the foreground work (FG). Results are shown in Figure 1. From the graphs, we can see that aggressive scheduling gives the worst foreground performance while achieving the best background performance. With conservative scheduling, the foreground performance improves as the fixed idle wait (see x-axis) increases, which confirms the need to protect foreground performance via idle waiting. However, we also observe that background performance decreases much faster when compared with the degradation caused to foreground performance. For large periods of idle waiting, foreground response time improves slightly while the performance of the background tasks degrades by orders of magnitude when compared to shorter or zero idle waiting. Since these two scheduling policies are complementary to each other, we are motivated to design a new scheduling algorithm to improve the response time of background work while preserving foreground performance.

Recall that performance degradation of foreground work comes from the fact that the system needs some time to switch from serving background work before it can serve foreground requests. The *entire set* of foreground requests in the following busy period is delayed. Our key observation here is that the impact of background tasks on foreground performance is larger if the delayed foreground busy periods are long (i.e., measured in number of requests) than if they are short.

We stress that in prior work, all efforts focused on incorporating characteristics of the arrival process, service process, or idleness of the system into the scheduling of background tasks. In this paper, we design an intelligent scheduling mechanism by exploring and taking advantage of the stochastic characteristics of busy periods *only*.

3. WORKLOAD CHARACTERIZATION

In this section, we analyze the enterprise disk-level traces used in the evaluation of the scheduling policy that is devised in this paper. We give general information about the traces but also focus on the stochastic characteristics of their busy periods.

3.1 Overview of Traces

We use three enterprise traces measured at the disk level from servers running enterprise-grade applications [18]. Although the storage subsystem of the servers consists of multiple RAID groups, we use here the user traffic seen by three individual disks located in different RAID groups. The traces are twelve hours long. Each trace contains the following information for each request: the arrival time, the departure time, the type of request (i.e., read or write), the request length in bytes, and its location on the disk.

In Table 1 we show a set of metrics that provide some general information on the availability of idle time at the disk level and the characteristics of foreground busy periods. The data in Table 1 shows that the disks are clearly underutilized and they have good potential to serve background work. The large coefficient of variation (C.V.), which is a normalized measure of the dispersion defined as the ratio of the standard deviation to the mean, and the large maximum length of idle intervals imply significant variability in the length of idle periods. This concurs with the discussion in the previous section: if the purpose is to serve background work timely, then limiting the time where background work can be served is not a good strategy. For busy period lengths, the moderate C.V. values coupled with the large value of the maximum length, suggest that there is also variability in the length of busy periods, albeit at a less degree than in idle periods. The impact on foreground performance due to interleaving foreground with background work may be quite different from one busy period to the next.

3.2 Characteristics of Busy Periods

Because the impact of the background tasks is strongly related to the length of the upcoming foreground busy period,

Trace	Util	Idle Periods in *ms*			Busy Periods in *IOs*		
	(%)	Mean	Maximum	C.V.	Mean	Maximum	C.V.
Trace1	5.6	192.6	325589	8.4	2.16	240	2.1
Trace2	1.7	767.5	186817	2.3	2.84	110	1.3
Trace3	0.7	2000.2	364876	3.8	2.39	190	2.4

Table 1: General busy period and idle period characteristics of our traces.

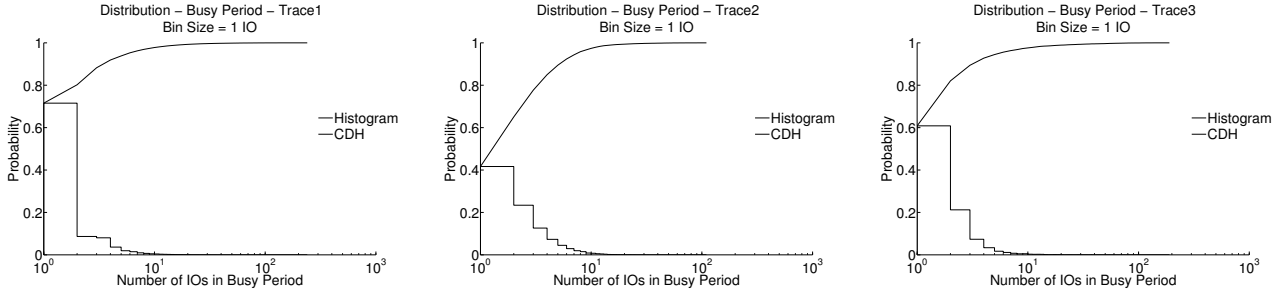

Figure 2: The distribution of the busy periods measured by number of requests.

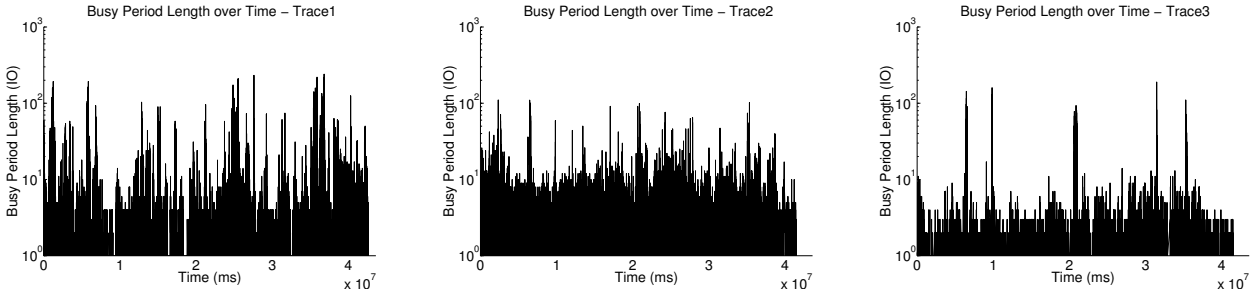

Figure 3: The across time plots of busy periods length measured by number of requests.

we now focus on the statistical features of foreground busy periods. In Figure 2 we plot the CDH (Cumulative Distribution Histogram) and relative frequencies using a bin size of one request. Note the log scale for the x-axis. The shape of the plots implies long tails for busy periods across all workloads, i.e., most of the busy periods are short while a few of them are quite long. One can see that across all workloads, 90% of busy periods are less or equal to 4 requests per busy period. This implies that if the background work delays a busy period, then it is with high probability that there are up to four requests to be delayed. Yet, there is also a sizable percentage of the workload with long busy periods (i.e., more than 4 requests) where the performance degradation of foreground work is going to be noticeable. The argument here is that *if* we can anticipate when these long busy periods arrive, then the performance can be improved significantly by avoiding to serve background jobs during those time intervals.

Next, we plot the length of every busy period across time, measured in number of requests, see Figure 3. The plots show a clear repetitive cluster behavior in the sequence of long busy periods (i.e., greater than 4 requests) for Trace1 and Trace2. The graphs show that the majority of busy periods are 4 to 6 requests. We conclude that this number can be used as a threshold that distinguishes busy periods as short or long.

In addition, the "clustering" of long busy periods shown in Figure 3 suggests that there is a consistent behavior across time. If we understand better how such clustering occurs, then we can use it to detect the upcoming clusters of busy periods. Once such a cluster is detected, then it would be beneficial to foreground performance if the background work is scheduled "conservatively" (i.e., the system idle waits before starting the background tasks). Once the system predicts that the upcoming busy periods are not expected to be long, then it resumes a more "aggressive" scheduling of background tasks (i.e., schedule them immediately after the system becomes idle of foreground requests). These observations are the basic premises for the design of a scheduling policy that dynamically adapts to a changing workload.

4. DYNAMIC SCHEDULING POLICY

In this section, we propose a dynamic scheduling policy that interleaves background tasks with foreground tasks efficiently. The goal here is to improve the performance of background work, measured via its response time, while preserving foreground performance, also measured via its response time. The dynamic policy that we propose alternates between scheduling background tasks aggressively or conservatively, based on the statistical characteristics of foreground busy periods and their recent history. As future busy periods are not known a priori, the algorithm cannot always make the best decision, but it can reach to a well-informed decision based on statistics of recent workload history. The

policy parameters are extracted from the most recent history of foreground busy periods.

1. if in *characterization state* do
  a. update busy period length trace
  b. calculate the *Threshold* of long busy period based on 90th percentile
  c. calculate the Cluster Window Size (CWS) based on Eqs. 1
2. if system in *decision making state* do
  a. initialize:
    i. system state (sys_state) = *idle*
    ii. busy period state (BP_state) = *short*
    iii. cluster count ($cluster\_count$) = 0
    iv. busy period length (BP_length) = 0
    v. queue length (QL) = 0
  b. **if** sys_state = *idle*
    i. if BP_state = *long* and $cluster\_count > 0$
      for no FG IO arrive do
        use *aggressive scheduling* to schedule BG work
        $cluster\_count - -$
    ii. **else**
      for no FG IO arrive do
        use *conservative scheduling* to schedule BG work
  c. upon FG IO arrive
    i. $sys\_state = busy$;
    ii. QL ++
    iii. $BP\_length++$
    iv. **if** $BP\_length >= Threshold$ and (BP_state) = *short*
      BP_state = *long*
      $cluster\_count = CWS$
    v. go to Step **2.b**
  d. upon FG IO depart
    i. QL ++
    ii. **if** $QL == 0$
      sys_state = *idle*
      $BP\_length = 0$
    iii. go to Step **2.b**

Figure 4: Algorithm of dynamic scheduling.

Aggressive scheduling may result in foreground performance degradation, because if short idle periods are utilized for background work, then with high probability, it delays all requests in the upcoming foreground busy period. The idle wait ensures that only long idle periods are used for background work. For a thorough discussion on the impact of idle wait on the performance of both foreground and background work, we direct the reader to [5, 15]. If there is a large amount of background work that is time critical, then the background tasks would have to continue to run as long as the system is idle, endangering the performance of upcoming foreground tasks. We argue that rather than limiting the amount of background work during idle periods as in [15], we limit the potential degradation on foreground performance:

- by selecting a fixed large idle wait for the periods when the system is experiencing a sequence of long foreground busy periods, with the expectation that such idle waiting would forbear the system from serving background work, and
- by canceling idle waiting if it is detected that the system is experiencing short foreground busy periods. This action would give the system the opportunity to serve a large amount of background work while delaying only a small portion of the foreground requests.

The algorithm first categorizes busy periods as long or short. Within a predefined time window, we log the information of busy period lengths and update their histogram, a process that is inexpensive, both computationally and space-wise. At the end of the time window, the count of requests that corresponds to the 90th percentile of the busy period histogram defines a *Threshold* whose value distinguishes busy periods as long or short. A new histogram is build for the next time window, which allows the algorithm to adapt well the *Threshold* parameter to changing workloads.

After categorizing the busy periods, the next step is to predict the incoming busy period length. To this end and according to the analysis in Section 3, we explore the clustered pattern of busy periods within each time window. This suggests that after an elapsed long busy period and based on recent history, we may be able to predict with accuracy whether the upcoming busy periods are long or short. To achieve this, we observe the conditional probability that two subsequent busy periods are long, i.e., if they are separated by one idle period with lag equal to one, as well as the conditional probabilities of busy periods that are separated by two or more idle periods, i.e., with lags equal to two or more. We define the *Cluster Window Size* (CWS) as the average number of consecutive long busy periods occurring with a given high probability value. Let P_{lag} be the conditional probability that the lagth busy period is long given that the current busy period is long (we could use any sufficiently large number here instead of twenty so that we capture enough probability mass). We define CWS as the smallest lag such that the sum of P_{lag} is equal or over 0.8:

$$CWS = min\{lag| \sum_{lag=1}^{20} P_{lag} >= 0.8\} \quad (1)$$

After a long busy period is detected, then CWS gives the number of upcoming busy periods that are expected to be long. During the intermittent idle intervals within those periods (which may be long or short), background work is served conservatively, i.e., deploying an idle wait period. After this number expires, background tasks are served aggressively, i.e., without any idle waiting, till the next long busy period is detected and conservative scheduling gets activated again. Note that the calculation of CWS is done once for every time window, in order to reflect well changes in the process of the foreground busy periods.

Note that, according to Equation (1), the stronger the clustering in the foreground busy period lengths, the shorter the CWS, and the longer the system serves background tasks aggressively. If the long foreground busy periods in the system are distributed randomly, i.e., there is no clustering, then CWS is long and the system schedules background tasks conservatively. Hence, the dynamic scheduling policy we propose here extracts the stochastic characteristics of foreground busy period lengths and reduces to the common practice of conservatively scheduling depending on the predicted foreground arrivals. Figure 4 gives the pseudo-code of the dynamic background scheduling policy.

In Figure 5 we give an example of how the aggressive, conservative, and dynamic algorithms work. We assume that there are several background tasks outstanding and the system currently operates under short foreground busy periods (the first three user busy periods marked with "S" in the fig-

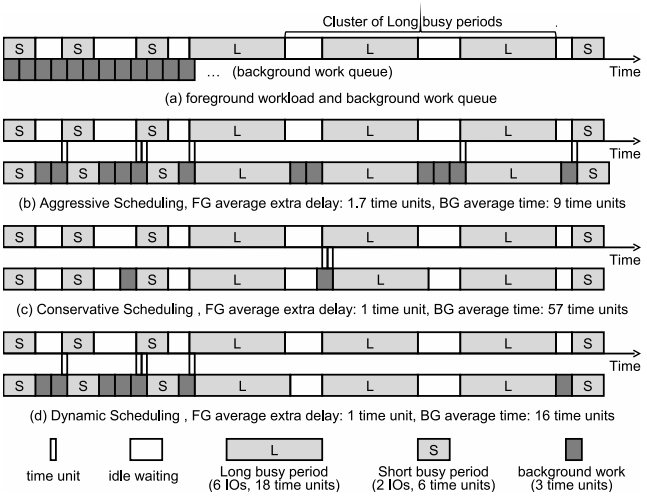

Figure 5: Example on the behavior of the three different background scheduling algorithm, aggressive, conservative, and dynamic.

ure) that are then followed by long user busy periods. After detecting the first long busy period (the fourth busy period), the next busy period (the fifth busy period) is marked as part of the next cluster of long busy periods. We assume that the estimations from previous observations have converged on a cluster size of 2 (i.e., the value of CWS) and that the threshold to differentiate busy period lengths is 4 requests. We assume in the example that the short busy periods are 2 requests long and that the long busy periods are 6 requests long. We assume that each user request is 3 time units, with one time unit being 2 ms. The six idle intervals in the depicted scenario are 5, 8, 4, 7, 8 and 3 time units long, respectively.

Based on the discussion in this section:

- Idle waiting for the dynamic scheduling is larger than the value selected by common practices for the conservative scheduling (i.e., two times of user service demands). In the example we assume that idle wait for dynamic scheduling is 1.5 times longer than the idle time for conservative scheduling.

- Aggressive scheduling does not idle wait and serves the background tasks the fastest (i.e., uses 9 time units on the average) with the largest extra delay (e.g., 1.7 time units) per user request.

- Conservative scheduling serves the background work the slowest (i.e., 57 time units on the average) with an average extra delay per user request of 1 time unit.

- Dynamic scheduling works best because it strikes a good balance between the performance of background tasks (i.e., 16 time units on the average) and an average added delay per user request of 1 time unit only.

This high-level example shows that dynamic scheduling is expected to behave like conservative scheduling with regard to foreground performance, and like aggressive scheduling with regard to background work performance. In the following section we evaluate these scheduling polices in detail.

5. EXPERIMENTAL EVALUATION

In this section, we evaluate the dynamic algorithm illustrated in Figure 4. The goal is to demonstrate that our algorithm can (1) effectively use the learned foreground busy period characteristics to schedule background tasks and (2) swiftly adapt its background scheduling to changing foreground traffic patterns such that both foreground and background tasks sustain the best possible performance. We evaluate two scenarios. In the first scenario, the system operates under a "stable" workload, while in the second one, the system operates under a workload that changes swiftly half-way through the experiment.

5.1 Experimental Setting

Our experimental evaluation is trace driven. The traces described in Section 3, are used as our foreground traffic. We use Trace1, Trace2, and Trace3 as representative of a stable operating environment. The scenario with the "swiftly changing" workload is achieved by concatenating Trace2 and Trace1, in this order.

As discussed in previous sections, our framework can be applied for scheduling of asynchronous tasks that when new data arrives into a geographically distributed storage system and need to be replicated across nodes for redundancy. In such systems, the redundancy is in the form of replication (e.g., the Google File System [6] replicates data 3 times) or erasure coding (e.g., the data is split into N fragments, encoded into N+M fragments, and distributed into N+M different disks/nodes) [17]. The asynchronous tasks in such scenarios consist of reading the recently updated data, computing the codes for the case of erasure coding, and sending them to their destination via the network. Consistent with this behavior, in our evaluation the background tasks have similar demands as the foreground ones and their intensity is a function of the WRITE foreground traffic, which varies by system. The results hold across a wide range of amount of background work but here we show only two representative cases: (1) the background work is equal to the amount of foreground work (i.e., common scenario, 100% of foreground work) and (2) the background work is 10 times the amount of foreground work (i.e., an extreme scenario, 1000% of foreground work).

Switching from serving background tasks to serving user requests is not instantaneous. Upon arrival of a new user IO which finds the system serving a background task, the system must first complete the background work before repositioning the disk head back to the location of the new request. In our evaluation, we assume that the penalty experienced by foreground requests due to background tasks is about two times the average service time of foreground requests. Note that because both foreground and background tasks have service and response times at the millisecond (ms) level, all our metrics of interest are measured in ms. Although replicating a large file or set of files may take overall more time, they are considered tasks that are generally split into multiple smaller tasks. Serving the smaller tasks faster is the goal of our framework.

Our dynamic algorithm uses short-term history (i.e., observations during a time window to calculate the *Threshold* and CWS. During each time window, we build the histogram of busy periods and based on this histogram we calculate the *Threshold* and CWS parameters, which are used to schedule the background tasks during the next time win-

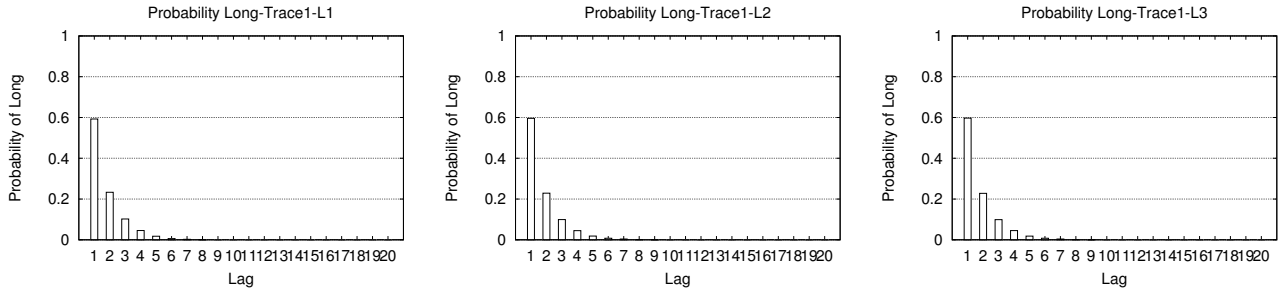

Figure 6: Probability plots that a long busy period is followed by a similar long one for lag 1 to lag 20 for different portions of Trace1. Three windows are considered: $Start = 0.5 \quad hour$ (left graph), $Start = 1 \quad hour$ (center graph), and $Start = 1.5hour$ (right graph).

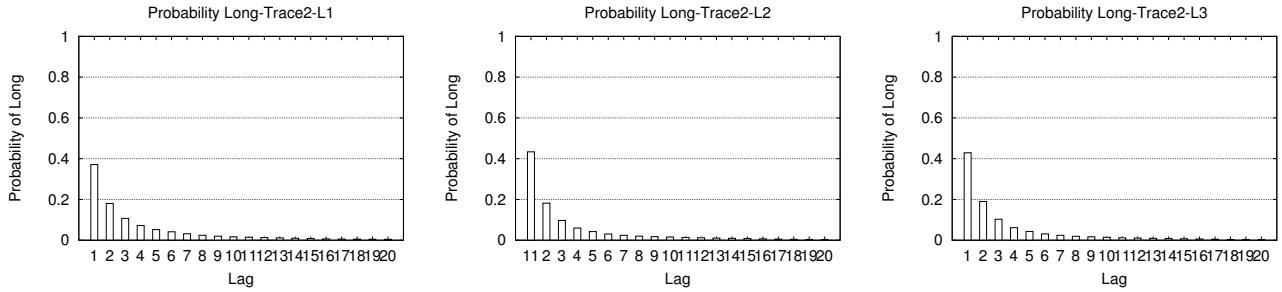

Figure 7: Probability plots that a long busy period is followed by a similar long one for lag 1 to lag 20 for different portions of Trace2. Three windows are considered: $Start = 0.5$ $hour$ (left graph), $Start = 1$ $hour$ (center graph), and $Start = 1.5$ $hour$ (right graph).

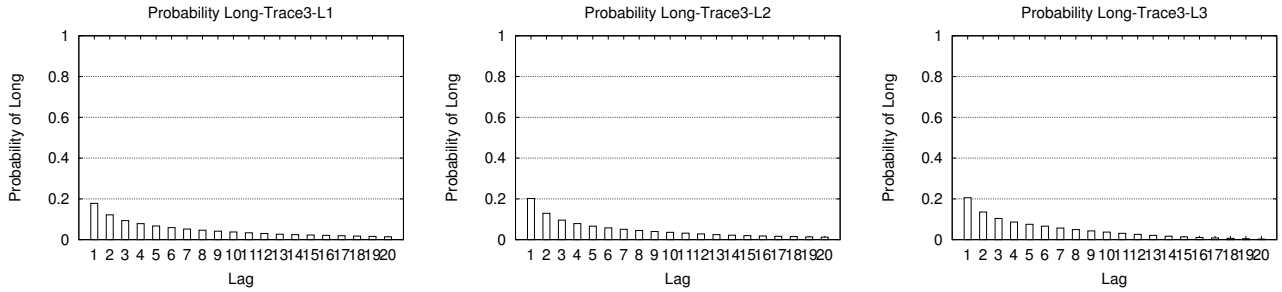

Figure 8: Probability plots that a long busy period is followed by a similar long one for lag 1 to lag 20 for different portions of Trace3. Three windows are considered: $Start = 0.5$ $hour$ (left graph), $Start = 1$ $hour$ (center graph), and $Start = 1.5$ $hour$ (right graph).

dow. The moment the *Threshold* and CWS parameters are computed, the histogram is discarded. During the next window where background scheduling is enabled, we collect data to construct a new histogram which is then used to calculate the *Threshold* and CWS parameters for the next scheduling window. Note that for Trace1, Trace2, and Trace3, we specifically focus on a 5-hour window, i.e., we collect the histogram during a window defined by $[Start,\ Start + 5)$ and apply the policy during $[Start + 5, Start + 10)$. To show the robustness of the policy irrespective of the *Start* value, we show results for three different sequences of 10-hour periods.

In our experiments, the amount of idle wait before starting the asynchronous tasks determines the aggressiveness of background scheduling. As idle wait increases, the impact on the response time of foreground requests decreases and the response time of background tasks increases. Here, we evaluate the entire range of idle wait values from 0 to 100 ms. Zero idle wait corresponds to the most aggressive background scheduling.

5.2 Evaluation Scenario One: Stable Workload

We drive our simulation using the three traces described in Table 1. Each trace has characteristics that change gradually over the course of its 12 hours span. Since changes are not dramatic, we consider such traces to represent stable operating environments, where our framework is expected to capture gradual changes effectively.

During the first time window, our scheduling framework

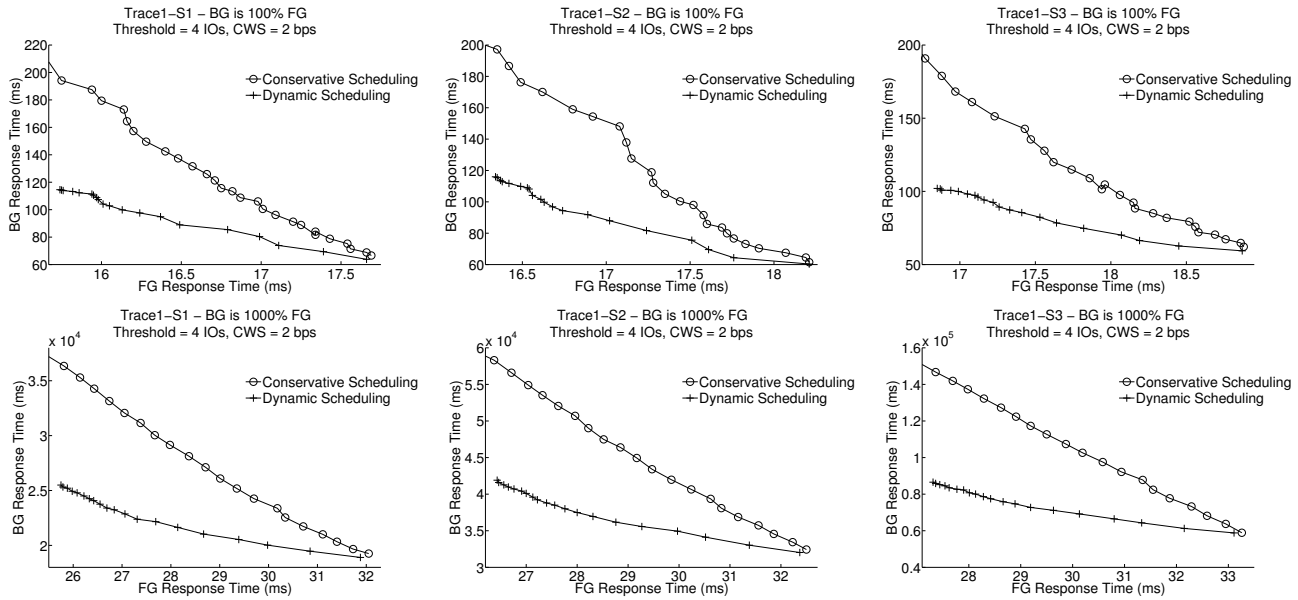

Figure 10: Scheduling comparison between dynamic and conservative scheduling for Trace1, scheduling results use the three respective periods given in Figure 6 (left, center, right columns) to schedule in the next 5 hours.

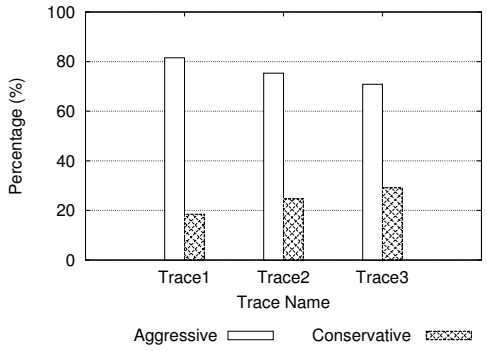

Figure 9: The percentage of time in aggressive mode and conservative mode under dynamic scheduling.

monitors the system busy periods, builds their histogram, and once the time window elapses, computes the *Threshold* and CWS values. Recall that *Threshold* corresponds to the value of the 90^{th} percentile of busy periods, while the CWS is computed based on the values of the conditional probability P_{lag} that two busy periods separated by lag idle intervals are both long. As different histograms are collected over different windows, the changes in the workload are captured by $Threshold$ and CWS. Figures 6, 7, and 8 show the values of the conditional probabilities of long busy periods over three different 5-hour windows.

The dynamic algorithm strives to exploit any relationship that exists in the sequence of foreground busy periods. If the clustering across time is weak, as in Trace3 (see Figure 8), then the expectation is for the dynamic algorithm to operate more often in the conservative mode (i.e., applying some idle wait). If the clustering is non-existent, then the proposed algorithm should *always* operate in the conservative mode.

Figures 6, 7, and 8 clearly show a stable behavior across time within each trace. Across traces, we notice that Trace1 has long busy periods clustered together because its conditional probability values are highest among the three traces. Clustering reduces for Trace2, while Trace3 depicts the least clustering. This means that the dependence structure weakens from Trace1 to Trace3. Therefore the computed CWS values increase as the dependence of long busy periods reduces from Trace1 to Trace3.

Figure 9 shows how long (in percentage of time) the dynamic algorithm operates in the conservative mode and how long in the aggressive mode. As expected from the discussion on the results in Figures 6, 7, and 8, Trace1 spends the most time in the aggressive mode because the long busy periods in this trace are well clustered, allowing the algorithm to predict well their occurrence.

Because of the overhead to switch from a background task to a foreground task, the more background work served, the higher the impact on foreground performance. The goal is to serve faster the outstanding background work, while sustaining foreground performance. Here we evaluate the effectiveness and robustness of the proposed dynamic scheduling by comparing the background mean response time for the *same* foreground mean response time under both the dynamic and conservative scheduling policies.

Figures 10, 11, and 12 show the average performance of background work for Trace1, Trace2 and Trace3, respectively, as a function of the achieved foreground response time. The figures are organized in a 2 by 3 grid, where each column corresponds to the performance achieved in a given window (the same ones depicted in Figures 6 through 8), and each row corresponds to the amount of background work generated in the system (i.e., 100% and 1000% of foreground work).

Recall that foreground response time is generally increased by the execution of asynchronous tasks because they arrive stochastically and the switch between asynchronous and

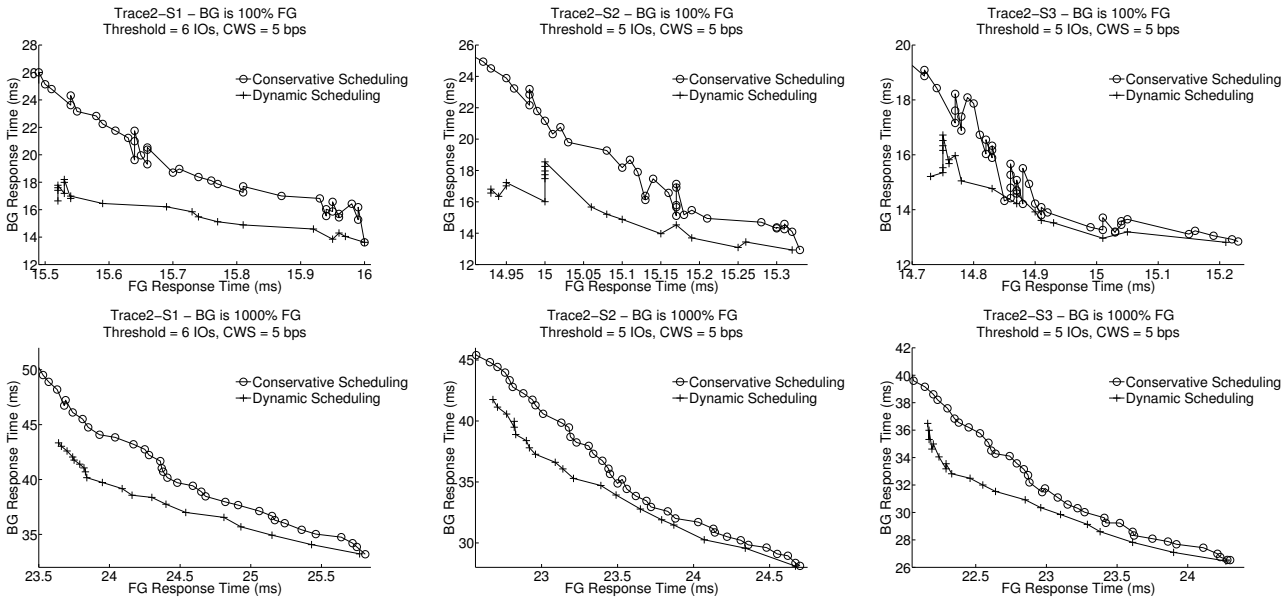

Figure 11: Scheduling comparison between dynamic and conservative scheduling for Trace2, scheduling results use the three respective periods given in Figure 7 (left, center, right columns) to schedule in the next 5 hours.

foreground tasks is not instantaneous. This means that as long as the idle wait value is smaller than the maximum idle interval length, there may be degradation in foreground performance. Idle wait is a way to control and limit performance degradation but not avoid it [16]. Our goal is to make sure we do not violate any foreground performance targets in the system. As expected, the foreground response time increases as the value of the idle wait decreases. For an idle wait of zero (i.e., corresponding to the aggressive scheduling policy), there is almost no distinction between the foreground and background work, because the background work starts executing as soon as the system becomes idle. Our scheduling always converges to this case in all plots (see the rightmost points in Figures 10, 11, and 12). Across all graphs, the more the background work (see the differences between the rows of plots), the higher the foreground degradation and background response time. Results can be summarized as follows:

- **Trace1:** Figure 10 clearly indicates that there are consistent gains across all time periods and for all amounts of background work. The dynamic scheduling can often speed up background work by as much as 50 percent.

- **Trace2:** Figure 11 shows that the gains of dynamic scheduling reduce when compared with the results of Trace1 because the probabilities of a long busy period being followed by another long busy period within a certain lag reduce (compare Figure 6 with Figure 7). However, dynamic scheduling consistently outperforms conservative scheduling, particularly for large idle waits that are captured by the leftmost part of the plots.

- **Trace3:** Figure 12 shows that Trace3 behaves similarly to Trace2. Note that the dynamic scheduling is more robust than the conservative one, which causes fluctuation on performance of background work. This is a result of variability in both idle and busy periods.

One of the most important observations is that for longer idle wait times (left portion of each plot) where foreground performance is degraded less, the dynamic scheduling consistently outperforms the conservative one. As a result, in cases when there are stringent performance targets for foreground requests, the performance advantage of dynamic scheduling is clear. If the foreground work is less sensitive to delays, then conservative scheduling with short idle waits results to a simple and good solution. In general, aggressive scheduling is not a good practical choice because it may cause severe or unbounded delays to foreground performance.

Another characteristic of the dynamic scheduling policy that sets it apart from the conservative one, is its resilience with regard to changes in the workload and scheduling parameters. In all evaluated scenarios in Figures 10, 11, and 12 the results from the dynamic scheduling are gradually reflecting the change, i.e., there are no oscillations on performance as it is often the case for the conservative scheduling. This is a direct outcome of the fact that our dynamic scheduling adapts its parameters to the changes in workload characteristics while the conservative or aggressive policies are oblivious to the workload characteristics. Such gradual changing behavior as characteristics change is desirable in systems because it allows applications to run smoothly.

Of particular importance is the sensitivity of the scheduling policies toward the chosen idle wait value. Figures 10, 11, and 12 show that the performance of both foreground and background work under the proposed dynamic scheduling policy varies but in a significantly narrower range than under the conservative scheduling policy. This implies that for the dynamic scheduling policy, identifying the optimal idle wait value is not critical. Applying the common practices that suggest to select an idle wait as a function of foreground service demands would yield satisfactory results.

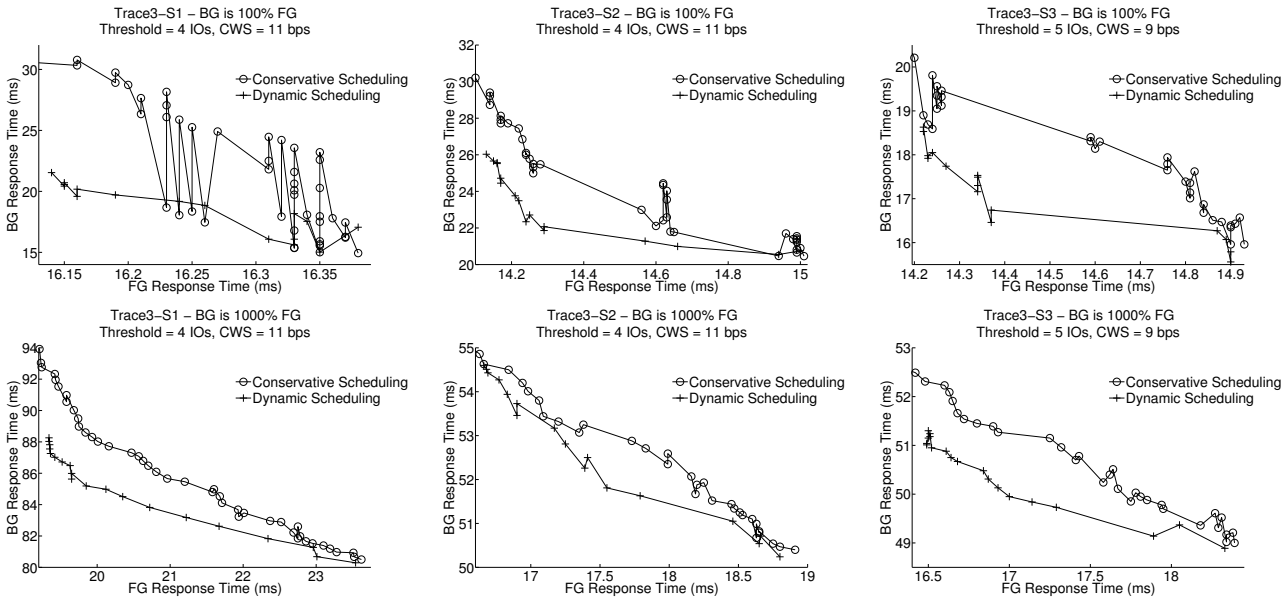

Figure 12: Scheduling comparison between dynamic and conservative scheduling for Trace3, scheduling results use the three respective periods given in Figure 8 (left, center, right columns) to schedule in the next 5 hours.

Overall, we conclude that the dynamic scheduling policy is robust and consistently achieves fast service of asynchronous tasks while sustaining foreground performance.

5.3 Evaluation Scenario Two: Swiftly Changing Workload

We concatenate Trace2 and Trace1 to obtain a new trace which we name Trace4. Trace4 is used to evaluate the adaptivity of the proposed scheduling policy as the workload changes swiftly. The new trace has a 24-hour span. Because Trace2 and Trace1 have different characteristics (e.g., *Threshold* and CWS), we expect a significant change around the 12th hour in the characteristics of Trace4. In order to capture the behavior of the dynamic scheduling policy, we chose to show here the following three learning windows from the 24-hour duration of Trace4.

- Period 1: learning window from the beginning up to the 8th hour; scheduling decisions apply from the start of the 9th hour through the 16th hour (i.e., learning happens before the workload change and applies during the workload change).
- Period 2: learning window from the 6th hour to the 14th hour; scheduling decisions apply from the start of the 15th hour through the 22nd hour (i.e., learning includes only a small portion of changed workload and applies over the period after the workload change).
- Period 3: learning window from the 8th hour to the 16th hour; scheduling decisions apply from the start of the 17th hour through the 24th hour (i.e., learning has equal portion before and after the workload change and applies over the period after the workload change).

Figure 13 shows the conditional probabilities for the three time windows and reflects the workload changes. We note also that *Threshold* changes gradually from 6 in the leftmost plot to 4 in the rightmost plot as the observed amount of Trace1 increases.

We present the scheduling results in Figure 14. We observe that the dynamic scheduling policy is robust and consistently performs well, even during the workload transition periods. Performance improves as the learning window includes more of Trace1 (e.g., note the differences in the foreground and the background performance in the center and rightmost columns). Overall, we conclude that the learning process incorporated in the dynamic scheduling algorithm, enables the scheduling policy to adapt well even to swift changes in workload characteristics.

6. CONCLUSIONS

In this paper, we propose a dynamic framework for scheduling background tasks, often associated with eventual consistency in geographically distributed storage systems. The framework ensures that the performance of foreground traffic is sustained while data consistency is achieved as fast as possible. We define a metric that measures the likelihood that busy periods of similar length arrive in a clustered way. This metric allows us to identify patterns in the length of busy periods and their probabilistic arrival. The reasoning behind the proposed scheduling framework is that if there is a cluster of short busy periods, then the system schedules aggressively the background work without much impact on foreground performance. If the cluster of long busy periods is detected, then scheduling of background tasks is done conservatively during the anticipated duration of long busy periods, i.e., only long idle intervals are used for serving background work. Extensive trace-driven experimentation shows that the framework is effective and robust. It achieves better response time for the background work without degrading performance of foreground traffic.

In the future, we plan to extend the work presented here

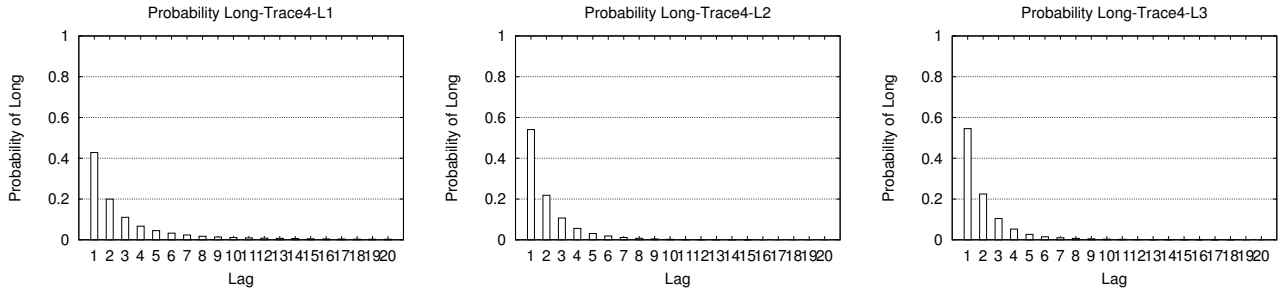

Figure 13: Probability plots that a long busy period is followed by a similar long one for lag 1 to lag 20 for different portions of Trace4. Three windows are considered: $Start = 0$, i.e., starting at the beginning of the trace (left graph), $Start = 6$ *hour* (center graph), and $Start = 8$ *hour* (right graph).

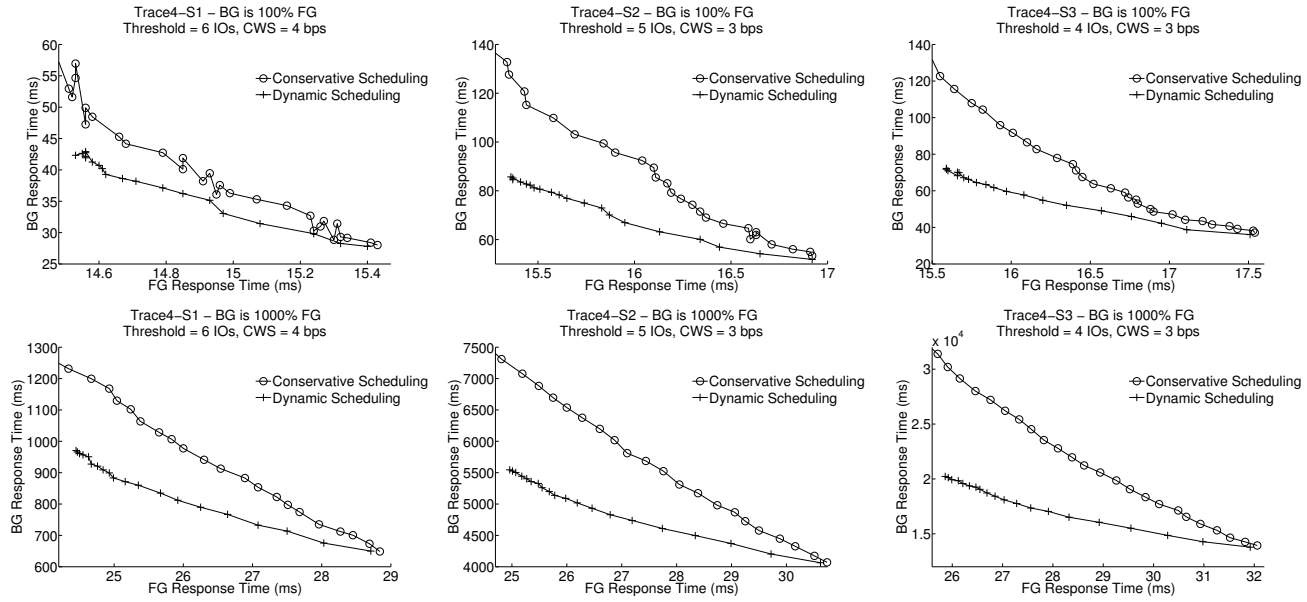

Figure 14: Scheduling comparison between dynamic and conservative scheduling for Trace4, scheduling results use the three respective windows given in Figure 13 (left, center, right columns) to schedule in the next 8 hours.

to learn and detect the length of the cluster of both busy and idle periods, aiming for the best outcome on scheduling time sensitive background work. We are also planning to use this framework to schedule work with different but close priorities, where foreground work can be delayed more than background work, at least for some periods of time.

7. ACKNOWLEDGMENTS

This work is supported by NSF grants CCF-0811417 and CCF-0937925. The authors thank Seagate Technology for providing the enterprise traces used for this work. We thank our shepherd J. Nelson Amaral for his assistance in improving the presentation of this paper.

8. REFERENCES

[1] E. Bachmat and J. Schindler. Analysis of methods for scheduling low priority disk drive tasks. In *SIGMETRICS*, pages 55–65, 2002.

[2] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler. An analysis of latent sector errors in disk drives. In *SIGMETRICS*, pages 289–300, 2007.

[3] J. L. Bruno, J. C. Brustoloni, E. Gabber, B. Özden, and A. Silberschatz. Disk scheduling with quality of service guarantees. In *ICMCS, Vol. 2*, pages 400–405, 1999.

[4] F. Douglis and P. Krishnan. Adaptive disk spin-down policies for mobile computers. *Computing Systems*, 8(4):381–413, 1995.

[5] L. Eggert and J. D. Touch. Idletime scheduling with preemption intervals. In *SOSP*, pages 249–262, 2005.

[6] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *SOSP*, pages 29–43, 2003.

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

[8] J. Guerra, H. Pucha, J. S. Glider, W. Belluomini, and

R. Rangaswami. Cost effective storage using extent based dynamic tiering. In *FAST*, pages 273–286, 2011.

[9] A. Gulati, A. Merchant, and P. J. Varman. pclock: an arrival curve based approach for qos guarantees in shared storage systems. In *SIGMETRICS*, pages 13–24, 2007.

[10] D. P. Helmbold, D. D. E. Long, T. L. Sconyers, and B. Sherrod. Adaptive disk spin-down for mobile computers. *MONET*, 5(4):285–297, 2000.

[11] H. Huang, W. Hung, and K. G. Shin. Fs2: dynamic data replication in free disk space for improving disk performance and energy consumption. In *SOSP*, pages 263–276, 2005.

[12] I. Iliadis, R. Haas, X.-Y. Hu, and E. Eleftheriou. Disk scrubbing versus intra-disk redundancy for high-reliability raid storage systems. In *SIGMETRICS*, pages 241–252, 2008.

[13] C. R. Lumb, A. Merchant, and G. A. Alvarez. Façade: Virtual storage devices with performance guarantees. In *FAST*, pages 131–144, 2003.

[14] M. K. McKusick and G. R. Ganger. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. In *USENIX Annual Technical Conference, FREENIX Track*, pages 1–17, 1999.

[15] N. Mi, A. Riska, X. Li, E. Smirni, and E. Riedel. Restrained utilization of idleness for transparent scheduling of background tasks. In *SIGMETRICS/Performance*, pages 205–216, 2009.

[16] N. Mi, A. Riska, Q. Zhang, E. Smirni, and E. Riedel. Efficient management of idleness in storage systems. *TOS*, 5(2), 2009.

[17] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, and Z. Wilcox-O'Hearn. A performance evaluation and examination of open-source erasure coding libraries for storage. In *FAST*, pages 253–265, 2009.

[18] A. Riska and E. Riedel. Disk drive level workload characterization. In *USENIX Annual Technical Conference, General Track*, pages 97–102, 2006.

[19] 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*, pages 71–84, 2000.

[20] M. Sivathanu, V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving storage system availability with d-graid. *TOS*, 1(2):133–170, 2005.

[21] E. Thereska, J. Schindler, J. S. Bucy, B. Salmon, C. R. Lumb, and G. R. Ganger. A framework for building unobtrusive disk maintenance applications. In *FAST*, pages 213–226, 2004.

[22] W. Vogels. Eventually consistent. *ACM Queue*, 6(6):14–19, 2008.

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

[24] F. Yan, X. Mountrouidou, A. Riska, and E. Smirni. Copy rate synchronization with performance guarantees for work consolidation in storage clusters. In *GreenMetrics 2011 Workshop*, San Jose, CA, USA, 2011.