# Restrained Utilization of Idleness for Transparent Scheduling of Background Tasks

Ningfang Mi
College of William and Mary
Williamsburg, VA, USA
ningfang@cs.wm.edu

Alma Riska
Seagate Research
Pittsburgh, PA, USA
alma.riska@seagate.com

Xin Li
University of Rochester
Rochester, NY, USA
xinli@ece.rochester.edu

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

Erik Riedel
Seagate Research
Pittsburgh, PA, USA
erik.riedel@seagate.com

## ABSTRACT

A common practice in system design is to treat features intended to enhance performance and reliability as low priority tasks by scheduling them during idle periods, with the goal to keep these features transparent to the user. In this paper, we present an algorithmic framework that determines the schedulability of non-preemptable low priority tasks in storage systems. The framework estimates *when* and *for how long* idle times can be utilized by low priority background tasks, without violating pre-defined performance targets of user foreground tasks. The estimation is based on monitored system information that includes the histogram of idle times. This histogram captures accurately important statistical characteristics of the complex demands of the foreground activity. The robustness and the effectiveness of the proposed framework is corroborated via extensive trace driven simulations under a wide range of system conditions and background activities, and via experimentation on a Linux kernel 2.6.22 prototype.

## Categories and Subject Descriptors

C.4.a [**Design studies**]: C.4 Performance of Systems; C.4.f [**Reliability, availability, and serviceability**]: C.4 Performance of Systems

## General Terms

Algorithms, Design, Management, Performance, Reliability

## Keywords

performance guarantee, low priority work, idleness, continuous data histogram

## 1. INTRODUCTION

A common practice in computer systems is to schedule maintenance tasks in *background* during idle times [7]. These background activities aim at improving system performance and availability [22, 10, 23, 1, 2, 14, 19]. If background activities[1] are instantaneously preemptable, then the performance of foreground jobs is not affected because background jobs are immediately stopped upon arrival of a new foreground request. Unfortunately, non-preemptive background activities are dominant in storage systems as tasks are commonly associated with the arm movement in a disk drive. Specific examples of non-preemptable background activities include disk scrubbing to detect media errors [19, 3], RAID rebuilds in case of disk array failures [20, 14], disk-level data mirroring [10].

Recently, in addition to these more traditional background activities, another set of background activities is emerging. These background activities are persistent and although with low priority, essential for the system operation, performance, and reliability. Commonly, such background activities help to close the gaps between the hardware characteristics and system-level requirements, therefore their on-time and transparent completion is critical. Examples include the garbage collection in emerging new storage devices such as flash-based drives to address the "WRITE amplification" problem [13]. If the garbage collection is not effective and a WRITE request arrives to find no erased memory cell to store the data, then the erasure process will need to complete in foreground and penalize foreground tasks. Similarly, to avoid silent data corruption, WRITE verification is a feature deployed in disk drives [18]. The works in [11, 13, 18] argue that background work is not something that can be always postponed but for reliability and performance reasons it is instead important to be completed early and effectively.

In this paper, we focus on the general problem of serving non-preemptable background jobs in storage systems, in general, and disk drives, in particular. Because foreground tasks are of high priority, background tasks are served only when there are no foreground jobs in the system, i.e., during system idle times. Yet, judicious scheduling of background activities during idle times is critical because their non-preemptive nature makes (potentially unfortunate) scheduling decisions non-forgiving. Therefore, effective serving of

[1]Throughout this paper we use the terms activities, tasks, jobs, and requests interchangeably.


*SIGMETRICS/Performance'09,* June 15–19, 2009, Seattle, WA, USA.

background tasks during idle intervals must meet two conflicting goals:

1. foreground performance degradation should be contained within predefined targets;
2. background work should not be starved and its throughput should be maximized.

Here, we define a general *schedulability* framework to meet these two goals by determining *when* and for *how long* the system can serve background jobs during idle times. This framework is generic yet adaptable to system dynamics and it works consistently well under a broad variety of system conditions and background demands, dealing effectively with the challenging problem of prescribed solutions that cannot possibly apply in every environment. We stress that we do not focus on the problem of *exact* background job scheduling. The specific scheduling of background jobs, i.e., their service order, is outside the scope of this work. We further assume that there are no real-time requirements that the background jobs must meet.

The core of the proposed algorithmic framework is the histogram of idle times. The effective use of this histogram contributes to the framework's robustness in a variety of environments. This histogram is obtained online via simple measurements and reflects the complex interaction of foreground arrival intensities, variability, and burstiness of foreground jobs, as well as different amounts (and variability) of background jobs. The methodology does not provide always the optimal scheduling parameters (i.e., *when* and for *how long*) background jobs should execute in idle intervals but it proves consistently robust as the obtained parameters are either among the best possible choices, or very close to those. Furthermore, monitoring a compact set of simple system metrics makes this approach pro-active rather than reactive to performance targets, allowing for quick adjustment to changes in the operational environment.

We evaluate the robustness of the algorithmic framework via an extensive set of trace-driven simulations of disk drive traces, that allow for exploring the effectiveness of the approach under a wide range of system conditions, including vastly different amount of background work. Additionally, an implementation of the framework as a Linux 2.6.22 system module further validates the simulation results and demonstrates the robustness of the new methodology. Both simulation and experimental results consistently maintain foreground performance degradation within pre-defined targets while completing as much background work as possible.

This paper is organized as follows. Section 5 presents related work. Section 2 gives an overview of the algorithmic framework and presents the detailed explanation of how the framework works. Discussion on the versatility of the framework in real world problems is given in Section 3. We validate the effectiveness of the approach in Section 4 by simulation and via implementation as a module of the IO device driver in the Linux kernel. Conclusions and future work are given in Section 6.

## 2. ALGORITHMIC FRAMEWORK

In this section, we present the new methodology that determines when and for how long to serve low priority background jobs that are not instantaneously preemptable during system idle times.

The execution of background tasks has been traditionally non-work conserving [6], i.e., when the system becomes idle, the execution of background jobs is delayed by some amount of time, which we denote by $I$ and refer to as *idle wait*. The length $I$ of the idle wait determines the balance between the service of foreground and background jobs. If the idle wait period is longer than most idle intervals, then foreground performance is affected minimally, because the background jobs are served only occasionally. On the contrary, if the idle wait elapses fast, then the effect on foreground jobs may be higher, because background jobs are served more often.

To avoid starvation of background work and any undesirable degradation on foreground performance, we complement the idle wait with the estimation of the amount of time $T$ that background work is served during an idle period. Equivalently, we determine the length of the idle wait period $I$ *and* the length of a background busy period $T$ within an idle interval. As a result, the schedulability of background work is determined from the pair of parameters $I$ and $T$, as depicted in Figure 1.

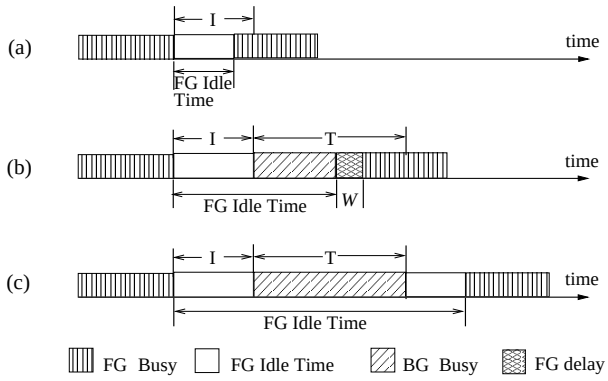



**Figure 1**: Three cases of idleness utilization: (a) no BG job are served in an idle interval shorter than $I$; (b) BG jobs are served and FG jobs are delayed in an idle interval longer than $I$ and shorter than $(I + T)$; (c) BG jobs are served without delaying FG ones for idle intervals longer than $(I + T)$.

Depending on the length of the idle intervals and the amount of background work served, foreground jobs may get delayed. We therefore classify the idle intervals into three categories:

(a) Idle intervals that do not serve any background jobs because they are shorter than $I$ (see Figure 1(a));

(b) Idle intervals that serve background jobs, but experience a foreground arrival during the execution of background work, because their length is between $I$ and $(I + T)$. In this situation, the background job that is in service continues its service to completion, but the system stops serving additional background jobs even if $T$ has not elapsed yet (see Figure 1(b));

(c) Idle intervals that serve background jobs for $T$ units of time and do not experience any foreground arrival in the meantime, because they are longer than $(I + T)$. After serving background work for $T$ units of time, the system remains idle without serving additional background work until a foreground job arrives (see Figure 1(c)).

Among all idle intervals, those of the (b) category are of imminent importance, because they can cause foreground performance degradation. As our goal is to contain fore-

ground delays within targets, we especially focus on this case.

## 2.1 Data Structure and Parameters

The framework that we propose here determines the $(I, T)$ pair using the length of idle intervals that are obtained via system monitoring. The empirical distribution of idle intervals is maintained in the form of a cumulative data histogram (CDH) which consists of a compact list of $(t_j, C_j)$ pairs. The finite list of the CDH $(t_j, C_j)$ pairs indexed by the histogram bin $j$, where $t_j$ is the smallest length of idle intervals falling on the $j^{\text{th}}$ histogram bin, and $C_j$ is the corresponding empirical cumulative probability of occurrence $C_j = Pr(idle_{interval} <= t_j)$. The empirical distribution of idle times incorporates foreground workload demands into the decision making without including complex processes, such as the foreground arrival and service processes.

Additional metrics necessary to determine the $(I, T)$ pair are also obtained via system monitoring and include:

- $S^{BG}$, the average service demands of background jobs,
- $RT^{FG}$, the average foreground response time without background jobs, which is estimated by monitoring the response times of foreground jobs that are in the busy periods without any background-caused delay, i.e., the proceeding idle interval falls in the (a) and (c) categories of idle intervals (see Figure 1), and
- $W$, the average wait time that the foreground requests experience due to the execution of background jobs, which is estimated by recording the time a foreground job arrives in a system idle of foreground jobs and the time it actually gets service (see Figure 1(b)).

The only user level input in our framework is the degradation target $D$ in foreground performance. Yet, we stress that $D$ may *not* be explicitly provided as a user input. For example, the user input may be in the form of the required amount of background work to be completed. In that case, we find the $(I, T)$ pair that satisfies the user input, i.e, completes the required background work, with the smallest possible foreground degradation target $D$. We will return to this point in Section 3.

The algorithmic framework first estimates the portion of idle intervals that delay foreground requests, i.e., the idle intervals that fall into the (b) category (see Figure 1). This portion of idle intervals is denoted by $E$ and its estimation is central to our algorithmic framework, see Section 2.2. Once $E$ is estimated, the $(I, T)$ pair is derived based on the histogram of idle interval lengths, in Section 2.3.

## 2.2 Estimation of $E$

We define $E$ to be the portion of idle intervals that are utilized by background work which delays foreground jobs. Once a foreground job is delayed with the amount of time $W$, the entire set of foreground jobs belonging in the same foreground busy period will be delayed by the same amount $W$. If we assume that all foreground busy periods have the same number of foreground jobs, then $E$ approximates the probability that a foreground job experiences a background-caused delay. Hence, the average response time of foreground jobs $RT$ would be the expected foreground-only response time $RT^{FG}$, plus the average additional delay $W$ attributed to the background work, which occurs only $E$ percent of the time,

$$RT = RT^{FG} + E \cdot W. \tag{1}$$

Our goal here is to express $E$ via the monitored system metrics $RT^{FG}$, $W$, and the degradation target $D$. We relate $D$ with the expected foreground response time $RT$ and the average foreground-only response time $RT^{FG}$ as follows:

$$D = \frac{RT - RT^{FG}}{RT^{FG}}. \tag{2}$$

Combining Eq. (1) with Eq. (2), we get

$$D = \frac{RT^{FG} + E \cdot W - RT^{FG}}{RT^{FG}} = \frac{E \cdot W}{RT^{FG}}, \tag{3}$$

which can be re-written to express $E$ as

$$E = \frac{D \cdot RT^{FG}}{W}. \tag{4}$$

Because we use the degradation target $D$ in foreground performance in Eq. (4), the estimated $E$ ensures that the background-caused delay does not exceed $D$ and does not violate foreground performance targets. The estimation of $E$ is critical, because it represents the mapping of the user input $D$ on to our main data structure, i.e., the CDH of idle interval lengths, and facilitates the estimation of the $(I, T)$ pair. The accuracy of $E$ depends on the accuracy of the monitored values for $RT^{FG}$ and $W$. In our evaluation, we show that even if we use average monitored estimates, the final output is consistently satisfactory.

## 2.3 Estimation of $(I, T)$

We use the parameter $E$ estimated via Eq. (4) and the histogram (CDH) of idle times to derive the $(I, T)$ pair. For this, we scan the sorted list of the CDH $(t_j, C_j)$ pairs, for intervals of length $E$. In practice, there may not exist an interval with exact length $E$. Instead, for each $(t_j, C_j)$, we find a $(t_{j'}, C_{j'})$ such that $|C_{j'} - C_j - E| = \varepsilon$, where $C_{j'} > C_j$ and $\varepsilon$ is a small number (e.g., 0.05). Each such $(t_j, t'_j)$ pair represents one choice for $(I, T)$, which we index by $i$ and denote as $(I_i = t_j, T_i = t'_j - t_j)$. See Figure 2 for a high level depiction. The result of the entire scanning process is a set of $(I_i, T_i)$ pairs.

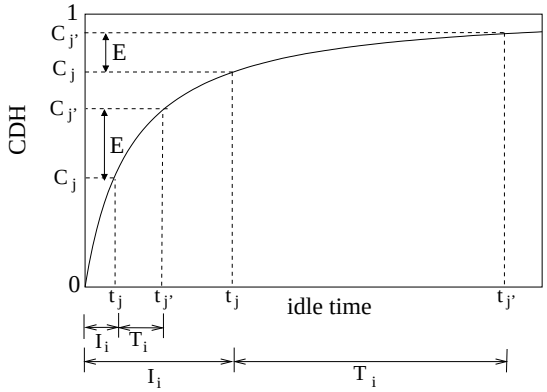



**Figure 2**: Transition from $E$ to $(I_i, T_i)$ in a cumulative data histogram. Any interval of length $E$ in the y-axis is mapped uniquely onto an interval in the x-axis described by the pair $(I_i, T_i)$. Because $E$ defines multiple intervals in the y-axis (between 0 and 1), multiple $(I_i, T_i)$ pairs exist.

### 2.3.1 Avoiding background starvation

If in the set of all $(I_i, T_i)$ pairs there is no interval $T_i$ which is at least $S^{BG}$ long, then no background job can be served

and background jobs may experience starvation. To avoid starvation, we substitute $E$ with a larger $E'$ value and estimate a new set of $(I_i, T_i)$ pairs for $E'$ such that at least one of the new $T_i \geq S^{BG}$. To prevent additional delays in foreground performance after substituting $E$ with $E'$, the background jobs are served with probability $E/E'$ in any eligible idle interval (i.e., interval longer than the idle wait $I_i$). The transition from $E$ to $E'$ is conservative with small increments (e.g., 0.05) in order to delimit foreground degradation and maintain it as close to its degradation target $D$ as possible.

### 2.3.2 Selecting among the $(I_i, T_i)$ pairs

Because every $(I_i, T_i)$ pair is chosen such that only $E$ percent of idle intervals delay foreground jobs, the foreground performance target is met by any of the $(I_i, T_i)$ pairs. The final $(I, T)$ pair is selected such that as much as possible outstanding background work is served as soon as possible.

Every $(I_i, T_i)$ can serve in average $B_i$ amount of background work measured in units of time[2]. We estimate $B_i$ as follows:

$$B_i = T_i \cdot Pr(idle > (I_i + T_i)) + \sum_{r=1}^{\lceil T_i / S^{BG} \rceil} r \cdot S^{BG} \cdot (C_r - C_{r-1}), \quad (5)$$

where $Pr(idle > (I_i + T_i))$ is the probability that an idle interval is greater than $(I_i + T_i)$ and $C_0$ is the probability that an idle interval is less than $I_i$. Idle intervals longer than $(I_i + T_i)$ can serve $T_i$ background work. Thus, the first term in Eq. (5) gives the amount of background work completed in these idle intervals. The second summation term gives the amount of background work completed in idle intervals longer than $I_i$ but shorter than $(I_i + T_i)$. In such idle intervals, less than $T_i$ background work will be served. Figure 3 demonstrates the estimation of the background work to be completed in these idle intervals. The $r^{\text{th}}$ subinterval of length $S^{BG}$ has probability $C_r - C_{r-1}$ and serves $r \cdot S^{BG}$ background work, see Figure 3. An idle interval shorter than $I_i$ does not serve any background work, thus it is not represented in Eq. (5).

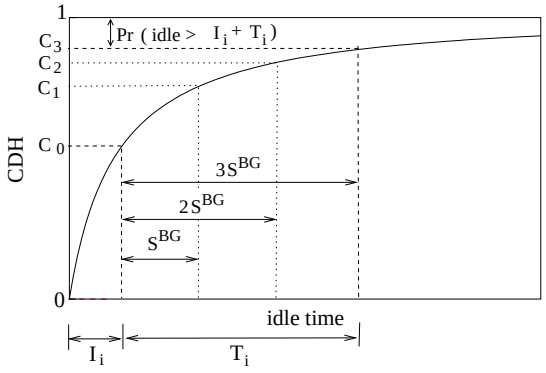



**Figure 3**: Estimation of the BG work $B_i$ that completes during idle intervals if $(I_i, T_i)$ is the schedulability pair.

Each $(I_i, T_i)$ is augmented by the corresponding $B_i$ and the selection of the final $(I, T)$ is done according to the type and amount of background work available in the system. Estimating the amount of available background work $B$ is system/feature dependent. Media scans may run continuously [3] and the amount of work associated to them is infinite, i.e., $B = \infty$.

Unlike background media scans, the work associated with the majority of background features in storage systems depends on the foreground workload. For example, WRITE verification [18] and parity updates [11], generate background work that depends linearly on the amount of WRITE foreground traffic. In these cases, the monitored foreground traffic is used to estimate the amount of background work. For example, if an average of $M$ WRITEs arrive in every foreground busy period, then the amount of background work associated with WRITE verification, where for each foreground WRITE, a background READ is generated, is $B = M \cdot S^{BG}$ in average. Also, the amount of background work associated with parity updates, where for each foreground WRITE, a background READ and a background WRITE are generated, is $B = 2M \cdot S^{BG}$.

Once $B$ is estimated, the final pair $(I, T)$ is selected such that $I$ is the smallest among all possible $(I_i, T_i)$ pairs, where $B_i > B$. The condition to select the shortest idle wait $I_i$ enables the fastest possible background response time. If $B = \infty$, then the final $(I, T)$ is the one with the maximum estimated $B_i$.

### 2.3.3 Buffered background work

Some background features generate work that may need buffer storage. An example is WRITE verification during idle times [18]. Such background activities put additional constraints on utilization of idle times, because the buffer space is limited. If the background buffer is full, then the future background work will be lost. In order to handle buffered background work, the methodology described above for the selection of $(I, T)$ is accordingly extended to account for the lost background work.

Assume that the buffer holds at most $N$ background jobs and $M_d$ is the number of background jobs generated during the $d^{\text{th}}$ foreground busy period. If $M_d > N$, then $M_d - N$ background jobs will be lost, because the buffer is full. If we construct the data histogram of the all observed values for $M_d$, then we estimate the buffered background work $B$ to be completed per idle interval as follows.

$$B = \sum_{d=1}^{\infty} min(N, M_d) \cdot P[M_d] \cdot S^{BG},$$

where $P[M_d]$ is the probability of generating $M_d$ background tasks in a busy period as computed in the data histogram of the number of background jobs generated every idle period.

## 3. FRAMEWORK'S VERSATILITY

Our framework incorporates seamlessly very different workloads and types of background work into its decision making. The flexibility and generality of our framework is associated with the set of the system metrics it uses to generate its output, i.e., the computation of the $(I, T)$ pair. Specifically, we manage to capture accurately the penalty associated with each background activity and the standalone foreground performance.

The **performance penalty of a background activity** ranges from a few milliseconds to several seconds in storage systems. For example, the penalty of background media

[2]Measuring work in units of time or number of jobs is qualitatively equivalent, because one is derived from the other using only the average background service time $S^{BG}$.

scans or parity updates is several milliseconds and is associated with positioning of the head on a location on the media. The penalty associated with spinning down a disk to conserve power is several seconds and is associated with the time it takes to spin the disk back up. The penalty associated with flushing out the cache content is several hundred milliseconds and is associated with completing several WRITE requests. In our framework, the penalty of a background task is captured by the parameter $W$ (see Eq. (4)), which is estimated via system monitoring.

The **characteristics of foreground workload**, such as the utilization of the system and the foreground service process, are captured in our framework via the foreground-*only* response time $RT^{FG}$ (see Eq.(4)). We remark that the performance effect of self-similarity (or burstiness) in the workload is captured by the measured $RT^{FG}$, therefore workload burstiness is immediately reflected in the framework. Also, the complex interaction between foreground arrival and service processes is incorporated in the histogram of idle times.

To demonstrate the flexibility of our framework, we calculate its output $(I, T)$ by emulating background tasks with penalties ranging from 0.5 to 60 times the foreground service time (using trace T1 from Table 1; this trace will be described in the following section). We show the results in Figure 4. The left y-axis gives the portion of idle intervals that will be utilized according to the calculated $(I, T)$, while the right y-axis indicates the length of $I$ and $(I + T)$. Observe that as the penalty of the background work increases the idle wait $I$ increases and correspondingly, the portion of idle intervals to be utilized decreases.

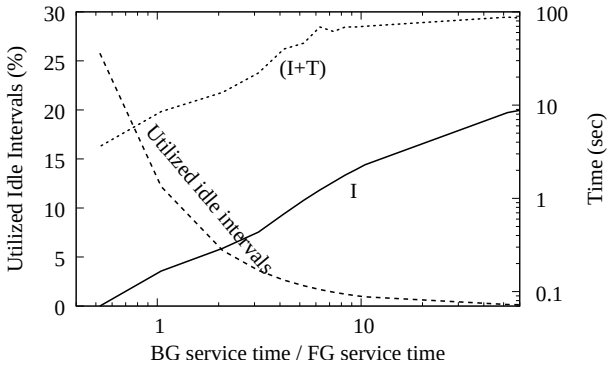



**Figure 4**: The output of our framework as a function of the background and foreground service times.

Specifically, if the foreground workload is sequential and background tasks are media scans that move the disk head away breaking the locality of the workload, then the penalty of the background tasks is twice as much as the foreground service time. While, if background activity would not move the head away, then more than twice idle intervals, e.g., 13% in Figure 4, can be utilized as the workload sequentiality is not strongly compromised and the penalty in this case is reflected on one foreground service time.

**Changing Priorities:** Although the background work (or a portion of it) represents a low-priority activity, there are cases when it *must* complete, even if it delays foreground traffic. For example, if there are no erased memory cells in a flash-based device then upon the arrival of a WRITE request the system needs to stop serving foreground requests and complete the background activity of erasing previously written cells. Similarly, if dirty parities are backlogged in a disk that deploys intra-disk parity, then the priority of updating the parities needs to increase to ensure their completion. Our methodology enables changing the priority of the background activities by increasing the input parameter $D$ in Eq. (4).

If $D$ is set to a new larger value, then the $(I, T)$ pair needs to be recalculated. A larger $D$ ensures a larger $E$, which increases the portion of idle intervals that serve background jobs, enabling more background work to complete (but also causing more delay on foreground jobs). By adjusting $D$ and recalculating $(I, T)$, the relative priority of background and foreground work may be changed throughout the lifetime of the system, as appropriate.

# 4. ANALYSIS AND EVALUATION

Here, we present an evaluation of the framework developed in the previous sections. Initially, the evaluation is based on trace-driven simulations. Later, we present results from measurements conducted in an implementation in the Linux 2.6.22 kernel.

## 4.1 Trace-driven Evaluation

We develop a trace-driven discrete event simulation model for the evaluation. Because the focus of the methodology is to determine when to start and stop serving background jobs, our simulation aims at correctly modeling the interaction between foreground and background busy periods rather than the specifics of scheduling each job inside a busy period.

In our evaluation, we use a set of disk-level traces measured in a number of personal and enterprise-level systems. Table 1 summarizes the main characteristics of the traces. Each trace entry corresponds to a foreground job and records its arrival and departure timestamps, the request type and length, and the request location on the disk. Since the arrival and departure times are recorded, the corresponding busy periods due to foreground jobs and the idle periods are fully defined in the traces. For the foreground request the service order is captured in the traces, while for the background jobs, we opt to schedule them in a First-Come First-Served (FCFS) order and expect results to be qualitatively similar for other scheduling policies.

An important observation in Table 1 is the low average utilization across all traces, which is expected to facilitate efficient utilization of idleness by background activities. Similarly, the length of idle intervals indicates that multiple background jobs may be able to fit into any idle interval. With the exception of the PVR trace, all other traces experience high variability (i.e., $CV > 1$) in the length of idle intervals, which is captured by the CDH of idle times. We experimented with the entire set of traces in Table 1, but to keep the presentation concise, we present here detailed results on traces $T1$ and $T2$, which we considered challenging and representative. Trace $T1$ is selected because it is the trace with the highest utilization and idle intervals with low variability. Trace $T2$ is selected because it has the highest utilization among traces with high variability in idle intervals. We also note that there is significant burstiness in the idle times of $T2$.

Figure 5 gives the empirical cumulative data histograms (CDHs) of idle interval lengths for traces $T1$ and $T2$. The tail of the distribution of the idle interval lengths for $T2$ is

| Trace | System | Length | No. Requests | Mean Arrival | Req. Length | Mean Service | Util (%) | Mean Idle Length | CV Idle |
|---|---|---|---|---|---|---|---|---|---|
| T1 | PVR | 40 min | 38,832 | 62.85 ms | 72 KB | 10.68 ms | 17.4 | 91.98 ms | 0.98 |
| T2 | E-mail | 25 hrs | 1,606,343 | 69.20 ms | 32 KB | 5.74 ms | 8.3 | 30.68 ms | 6.16 |
| T3 | Code-Dev | 12 hrs | 483,563 | 88.06 ms | 40 KB | 6.34 ms | 6.2 | 192.50 ms | 8.49 |
| T4 | User Acc. | 12 hrs | 168,148 | 246.65 ms | 48 KB | 6.10 ms | 2.0 | 632.73 ms | 2.29 |
| T5 | Desktop PC | 20 hrs | 146,248 | 509.83 ms | 20 KB | 3.08 ms | 1.1 | 2146.36 ms | 8.62 |

**Table 1: Overall characteristics of our traces.**

longer than for $T1$, which implies that trace $T2$ has many short idle intervals and some very long ones, while in trace $T1$ most idle intervals are of similar lengths.

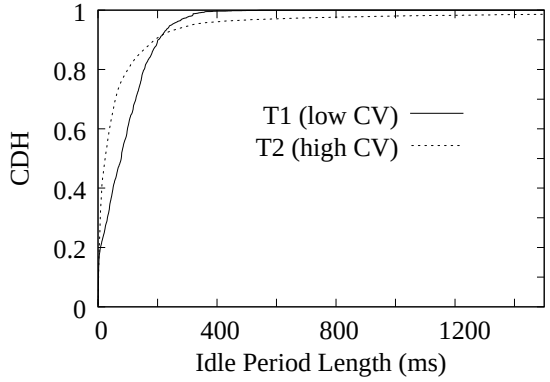



**Figure 5: CDH of idle times for traces $T1$ and $T2$.**

We evaluate the performance of our methodology under different amount of background work. Specifically, we experiment with background work that is 10%, 40%, and 90% of the foreground work, as well as the extreme case of having "infinite" background work in the system. While foreground busy and idle periods are determined by the traces, in our model we set the service time of background jobs to be exponentially distributed with a mean of 6.0 ms, which is similar to the mean service time of foreground jobs in trace $T1$.

In the scenarios evaluated here, we aim to maintain background service transparent from the user. It is common practice, to consider an additional 5%-10% slowdown in performance as small enough to not be noticed by the system user. Consequently, we set the degradation target $D$ to 7%, i.e., the middle point in the 5%-10% range. We have conducted experiments with various values of $D$ and results are qualitatively similar to those reported here. In Section 4.2, we present results for one of the most challenging cases by setting the degradation target to $D = 1\%$ for the Linux experiments. The metrics of interests are: (a) the average relative delay of foreground jobs due to background work, defined as $(RT - RT^{FG})/RT^{FG}$, and (b) the number of completed background jobs.

### 4.1.1 System performance

Table 2 shows the results for the four levels of background work. We observe that in most scenarios the relative foreground delay is well below the degradation target $D$. Under trace $T2$, the system serves significantly more background jobs than under trace $T1$, because the utilization of trace $T1$ is twice as high as the utilization of trace $T2$. If the background work is infinite, then the results in Table 2 show that trace $T1$ can accommodate background work that is twice as much as the foreground one and that trace $T2$ accommodates as much as six times more background work than foreground one.

| Trace | BG Target Work | FG Delay (Target $D$=7%) | Completed BG | |
|---|---|---|---|---|
| | | | Reqs. | Work |
| T1 | 10% | 1.4% | 3,861 | 10% |
| | 40% | 2.0% | 15,514 | 40% |
| | 90% | 3.9% | 34,953 | 90% |
| | infinite | 7.0% | 74,234 | 190% |
| T2 | 10% | 3.2% | 132,362 | 10% |
| | 40% | 6.8% | 528,287 | 40% |
| | 90% | 4.7% | 1,190,208 | 90% |
| | infinite | 3.9% | 7,862,813 | 610% |

**Table 2**: FG delay, completed BG requests, and completed BG work relative to the incoming FG work.

If the background work requires buffering, then the buffer size is the determining factor for the performance of foreground and background jobs. Here, the buffer size is set to store at most 16 background jobs. Table 3 presents the simulation results. Table 3 shows that some amount of the background work is lost because of insufficient buffer. Generally, as the amount of buffered background work increases (e.g., from 10% to 90%), the background drop rate increases. For trace $T2$, the background drop rate increases faster than for trace $T1$. Because of the high variability in idle interval lengths, trace $T2$ has many short idle intervals that either do not get utilized or serve only a few background jobs. This causes the buffer to be flushed at a slow rate. Note that we do not evaluate the case of infinite buffered background work, this is unrealistic given the limited buffer space.

| Trace | BG Work | FG Delay (Target $D$=7%) | Completed BG Num | Dropped BGs |
|---|---|---|---|---|
| T1 | 10% | 3.7% | 3,358 | 13% |
| | 40% | 6.6% | 13,035 | 15% |
| | 90% | 8.3% | 26,778 | 23% |
| T2 | 10% | 3.0% | 128,862 | 2% |
| | 40% | 5.3% | 385,573 | 27% |
| | 90% | 4.1% | 613,157 | 48% |

**Table 3**: FG and BG performance for buffered BG work. The buffer holds 16 BG jobs.

The results in Table 3 indicate that our methodology does not always meet the degradation target $D$. For example, for 90% background work and trace $T1$, our estimated $(I, T)$ yields a slowdown slightly above the target. As expected, the methodology represents a heuristic rather than an optimal solution.

### 4.1.2 Optimality of the $(I, T)$ Pair

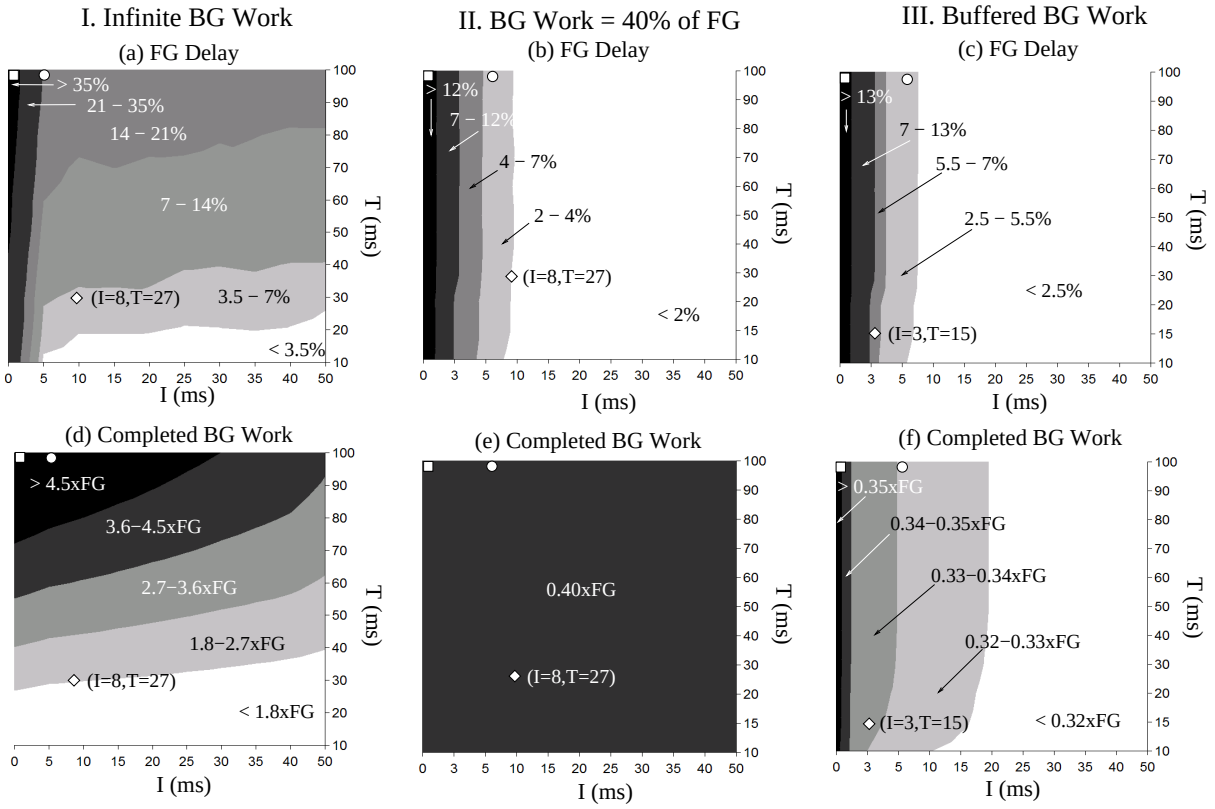



**Figure 6**: Trace $T1$. FG delay and completed BG work for *any* $(I, T)$ pair. Diamond shapes mark our solution. Square and circle shapes mark common practices.

To evaluate the effectiveness of our methodology in utilizing idleness, we perform a state space exploration, i.e., estimating the foreground and background performance for *any* $(I, T)$ pairs. Figures 6 and 7 give the results of the state space exploration analysis for traces $T1$ and $T2$, respectively. We evaluate the cases of infinite background work in the first column, background work that is 40% of the arriving foreground work in the second column, and buffered background work that is 40% of the foreground work in the third column. The first row in Figures 6 and 7 shows the background-caused delay on foreground performance and the second row presents the completed background work. In each plot, we mark the pair generated by our approach with a diamond. For comparison with common practices [6], we also mark with a square the results for the pair $(I = 0, T = \infty)$, i.e., no idle wait, and with a circle the results for the pair $(I = 6, T = \infty)$, i.e., fixed idle wait equal to the average background service demand.

Figures 6 and 7 clearly indicate that the pairs representing common practices provide a fixed solution independent of the effect they have on foreground or background performance (see the fixed position of the circle and square shapes in all plots). The pair $(I = 0, T = \infty)$ significantly degrades foreground performance, by more than 10% for both traces, and confirms that idle wait is necessary in scheduling background work. With an idle wait equal to the average background service demand, the pair $(I = 6, T = \infty)$ keeps the background-caused delays low for several scenarios, in particular for trace $T1$ with low variability in idle periods. However, it fails to meet performance targets if the amount of background work is large (e.g., the infinite case) or if idle intervals are variable (e.g., trace $T2$).

Figures 6 and 7 indicate that there is a set of pairs $(I, T)$ that would satisfy the degradation target $D = 7\%$. For example, plots (a) and (d) in Figure 6, indicate that the idle wait $I$ should be at least 5 ms and the length of background busy period $T$ should be at most 40 ms. However, having $T$ shorter than 25-30 ms or $I$ larger than 20 ms would result in reduced levels of completed background work.

The pair $(I, T)$ estimated using our methodology is *consistently* among the ideal choices that strike a good balance between the completed background work and foreground performance. Our results confirm that it is necessary not only to idle wait but also to limit the amount of background work completed in every idle interval (i.e., have $T < \infty$) to sustain foreground performance at desired levels. Furthermore, controlling foreground performance by only changing the idle wait length $I$ (as in common practices) would result in background work starvation.

### 4.1.3 Foreground response time distribution

One of the concerns associated with background work is that it adds to the variability of foreground response time. For the majority of background activities, the preemption level is related to the disk head position. As a result, the effect on the foreground response time tail is not expected to be severe. Figure 8 captures the tail of foreground response time distribution, by plotting the complimentary cumulative distribution function (CCDF), for traces $T1$ and $T2$ under

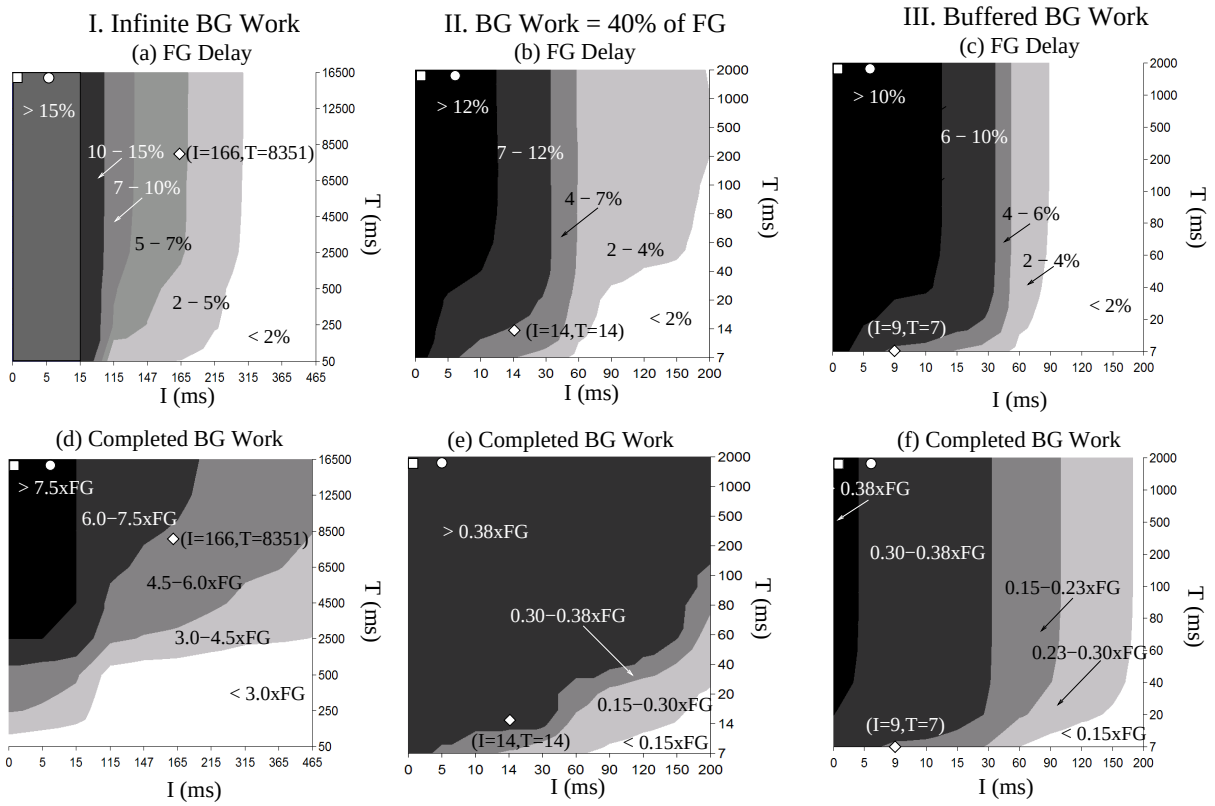



**Figure 7**: Trace $T2$. FG delay and completed BG work for *any* $(I, T)$ pair. Diamond shapes mark our solution. Square and circle shapes mark common practices.

the three types of background work.

If the trace has high variability in its idle intervals, e.g., $T2$, then the tail of the foreground response time is almost not affected by the execution of background work. If the trace has low variability in its idle intervals, e.g., $T1$, then the effect in the tail is more noticeable but only for about 1% of foreground requests.

Spinning up and down a disk to conserve power represents a background activity that has preemptability penalty around a few seconds and it is expected to affect the foreground response time tail more than the background activities with low preemptability penalty that we evaluated here. In this case, our methodology can be enhanced by exploiting burstiness in the idle interval lengths to reduce the tail. In depth evaluation of these scenarios is part of our future work.

### 4.1.4 Prediction accuracy

Our methodology relies on monitoring of system previous history to devise the current schedulability pair $(I, T)$. For traces with idle times of low variability like $T1$, predicting the immediate future using data from the immediate past results in undoubtedly good $(I, T)$ prediction. The results for $T1$ are not reported here due to the limited space, but due to its low variability, results are better than for $T2$. Trace $T2$ is more challenging because of its high variability. In such cases, it is important to have a large monitoring window to capture better the stochastic behavior of idle times and its distribution tail.

Here, we split trace $T2$ into two parts (each corresponding to about 12 hours of operation) and characterize each subtrace individually as shown in Table 4. We use the monitored parameters in the first part to determine the $(I, T)$ pair used in the second part, these results are presented in the columns marked "training" in Table 5. The columns marked "exact" in the table represent the results obtained by estimating the $(I, T)$ pair for the second part using the data from the second part (i.e., perfect knowledge of the present and immediate future). Observe that the prediction error, i.e., the difference between the "training" and "exact" columns, is small and increases as the amount of background work in the system increases (the highest error is for infinite background work). We conclude that the metrics we select for our estimation of the $(I, T)$ pair are robust to changes in the system.

| Idle | Part1 | Part2 |
|---|---|---|
| Mean | 159.0 ms | 103.7 ms |
| CV | 6.7 | 3.7 |

**Table 4**: Idle times mean and CV for two parts of trace $T2$.

## 4.2 Linux Implementation

We prototyped our algorithmic framework as a module of the Linux kernel 2.6.22 by modifying the Linux generic block layer and the IDE device driver. Note that the implementation can be extended easily to work with the SCSI device drivers as well. Table 6 gives the machine configuration where we performed our measurements.

In our experiments, all user and system IO requests are

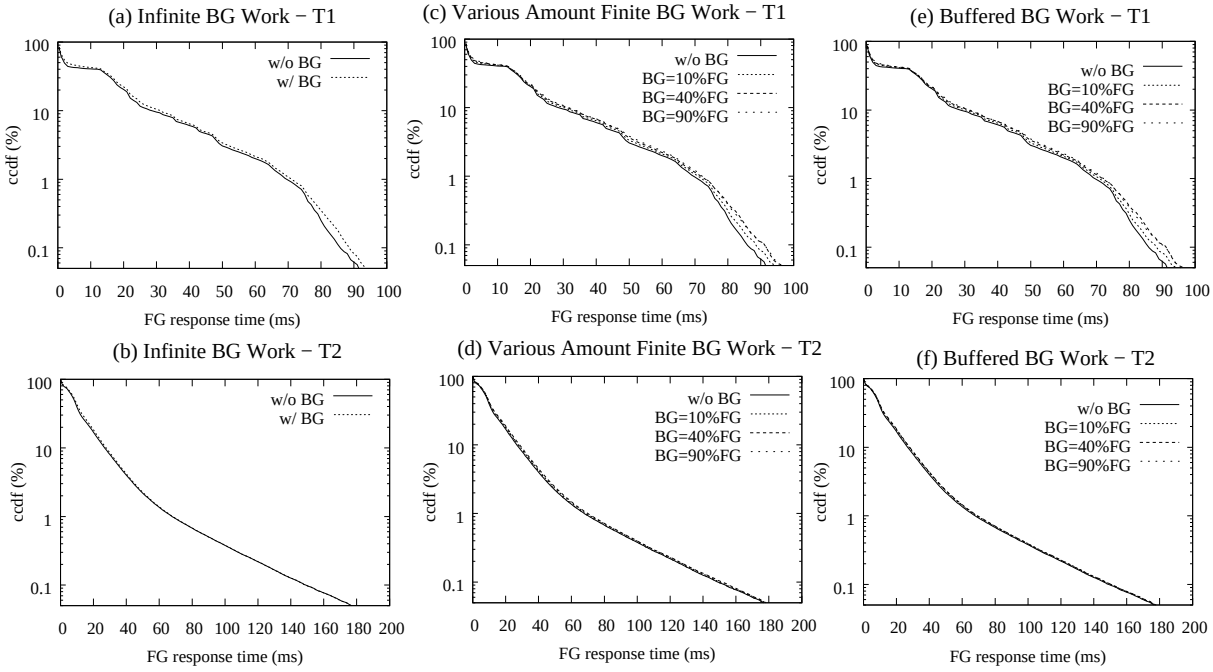



**Figure 8: CCDFs of the FG response time with and without BG work for traces $T1$ and $T2$.**

| BG Work | FG Delay | | Completed BG Num | |
|---|---|---|---|---|
| | training | exact | training | exact |
| 10% of FGs | 1.4% | 1.3 % | 60,513 | 60,513 |
| 40% of FGs | 4.8% | 4.3% | 242,747 | 242,747 |
| 90% of FGs | 7.5% | 5.8% | 545,264 | 545,263 |
| infinite | 8.4% | 5.9% | 2,756,096 | 2,220,666 |

**Table 5**: FG and BG performance for different amounts of BG work for trace $T2$.

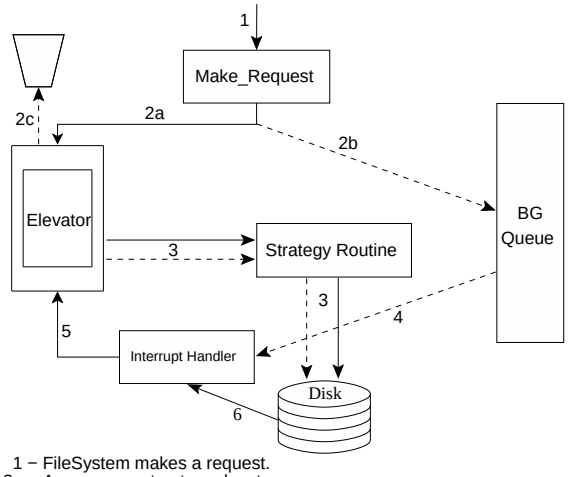
| | |
|---|---|
| System | Dual Intel Pentium 4 CPU 2.80GHz<br>1GB memory |
| OS | Linux kernel 2.6.22 |
| File System | Ext3 |
| IO scheduler | Anticipatory |
| Disk Drive | Seagate Barracuda 7200 rpm, 80 GB |

**Table 6: Specifications of the measurement system.**

considered as foreground traffic. Background work is injected in the queue of outstanding block-level requests at the IDE device driver. Figure 9 gives a high-level schematic description. The implementation emulates the case of infinite background work.

The IDE device driver maintains the list of outstanding foreground requests in the elevator queue and in the dispatch queue. Because the elevator queue feeds the dispatch queue, we view them both as one big queue of outstanding foreground requests, which we call the *elevator queue* (see Figure 9). Background requests are queued separately in the *BG queue*. For simplicity, we choose to have background jobs that are copies of foreground ones. The results would be qualitatively the same for other types of background activities. We remark here that caching is disabled in our experiments to ensure that the background requests are served by the disk and not by the system cache.

We modified the service process at the Linux generic block




1 − FileSystem makes a request.
2a − A new request enters elevator queue.
2b − A copy of the request enters BG queue.
2c − BG requests removed from elevator queue.
3 − Strategy routine schedules an FG/BG request from elevator queue.
4 − If the elevator is empty, then a request is drawn from BG queue.
5 − The interrupt handler sends the notification of the completion of a request.
6 − Data is ready. Disk drive raises an interrupt.

**Figure 9**: Schematic description of our prototype. The solid lines capture the flow of FG jobs and the dashed lines capture the flow of BG jobs.

layer and the IDE device driver to handle two modes, i.e., foreground and background. We also modified the kernel to keep track of the current operation mode in the system and the time a transition between the two modes occurs. The $(I, T)$ pair is made available to the entire set of modules in the Linux generic block layer and stored at the */proc/sys* directory for user space accessibility.

Any enqueued request (either foreground or background) is dequeued via the "strategy routine", which for the IDE device driver is *ide_do_request*. The block layer module pro-

vides to the "strategy routine" a mechanism, which is supported by a set of interrupt handlers, e.g., *ide_dma_intr*, *task_in_intr*, or *task_out_intr*, that enables serving background jobs after the idle wait period $I$ has elapsed, with only little additional modification.

If the elevator queue is empty upon the notification from the interrupt handler of a foreground request completion, then a background request is taken from the BG queue and placed in the elevator queue and released to the strategy routine from the interrupt handlers after $I$ elapses. The BG queue feeds the elevator queue until a foreground request arrives or $T$ elapses. If during the service of a background request, a foreground request arrives, then the background job in service completes. After that, for simplicity, the elevator queue is flushed of any remaining background requests.

For each IO request that arrives at the elevator queue and for each request served from the background queue, statistics are logged using the *klog* utility. We compute the foreground response time with and without background jobs, background jobs service demands, and the empirical distribution of idle intervals. We stress that the overhead to keep these statistics is small and does not interfere with the system performance. For example, the CDH of idle times, the largest data structure we maintain, is at most a list of 1000 pairs of floating point numbers.

We focus our evaluation on two scenarios: video streaming and Linux kernel compilation. We consider these two scenarios representative, since they generate idle intervals with very different characteristics. Specifically, video streaming yields idle times that are almost deterministic and kernel compilation yields idle times that are highly variable.

**Video streaming:** This workload consists of playing back a 10 minute movie. The foreground IO workload is dominated by read requests. The average request service time is 25 ms and the average response time of 35 ms. The mean idle time of a foreground-only workload is 135 ms and its coefficient of variation is 0.48, which indicates that idle intervals have low variability (see Figure 10(a) for the empirical CDH). The injected background work consists of read requests with an average service time of 9 ms. For these experiments, we consider the challenging case of setting a very small degradation target $D = 1\%$.

We assess the quality of our $(I, T)$ pair by measuring system performance under any $(I, T)$ pair. Plots (b) and (c) in Figure 10 present the foreground delay and the number of completed background jobs, respectively. Foreground performance degradation is severe (dark region in Figure 10(b)), if $I$ is set at a value between 50-165 ms or is set to zero, because such $I$ values utilize ineffectively many idle intervals, represented by the abrupt jumps in the CDH of Figure 10(a). The amount of completed background work decreases as $I$ increases and $T$ decreases, with the desired region being the upper left corner in Figure 10(c). Our estimated pair of $I = 1$ ms and $T = 90$ ms shows the robustness of our methodology by being located in the best region of both surface plots in Figure 10.

**Compilation:** In this workload, compiling the Linux kernel takes in average 10 to 15 minutes. The foreground IO workload consists of a mix of short requests with an average service time of 9 ms and average response time of 11.2 ms. The average idle time (if only foreground traffic is present) is 130 ms and the coefficient of variation 2.68, which means that the idle intervals have high variability (see the CDH in Figure 11(a)). The injected background work consists of similar requests as the foreground traffic with the same average service time. For our estimation, we set the degradation target $D = 1\%$.

The CDH figure of idle intervals in Figure 11(a) indicates that there is a large portion (almost 60%) of small idle intervals in the range of 10 ms to 20 ms, which if utilized would degrade severely foreground performance. The rest of idle intervals (approximately 40%) are long and can be used for serving background work.

Similarly to the video streaming workload, we measure the performance of any pair $(I, T)$ and present the foreground delay and the number of completed background jobs, respectively, in Figures 11(b) and (c). For this workload, our approach estimated $I = 76$ ms and $T = 28$ ms and we indicate our estimation with a diamond in Figures 11(b) and (c).

We observe that for this workload, the degradation target is reached only in the middle range of $I$ and bottom range of $T$ (see Figure 11(b)). The pair $(I = 76, T = 28)$ that we estimate is right at the area where the degradation target is satisfied with the maximum possible background work completed.

We conclude that our framework proves to be robust in identifying accurately the schedulability pair $(I, T)$ in systems with different characteristics, e.g., video streaming with low variability and kernel compilation with high variability in idle intervals. Although the methodology requires to monitor various metrics in the system, they are obtained easily by utilizing counters already in the Linux generic block layer and by minor additions in it. More importantly, they do not incur any overhead in either memory or computing requirements for the system.

# 5. RELATED WORK

Systems that serve both foreground and background tasks can be viewed as systems that serve tasks of multiple priorities, see [21, 16] and references within for analytic results and see [8] and references within for priority scheduling aiming at performance virtualization in storage systems. As it becomes more common for systems to operate 24/7, idle times may offer the only time window to complete maintenance work [7, 10, 22, 1, 2]. Storage systems depend heavily on background activities to deploy features that enhance performance and availability. For example, detection of latent sector errors, a possible cause for data loss [3], is routinely done via background media scans. Recently, several features that enhance the data integrity of disk drives such as WRITE verification [18] and intra-disk parity [11] are also designed to use idle times. Also garbage collection in solid state drives to address the hardware idiosyncrasies of handling the WRITE traffic [13, 12] is most efficient when completed in background. This makes the problem of effective scheduling of background jobs during idle times in storage systems of imminent importance [7, 6], distinguishing it from the more general problem of priority scheduling.

Various studies in the literature have shown that in systems, periods of high utilization may be interleaved with idle times [7, 17]. Golding et. al. in their seminal paper [7] identified categories of useful idle-time operations in storage systems (e.g., required work that can be delayed such as RAID rebuilds, work that will be probably requested later such as disk read-ahead, shifting work within resources to

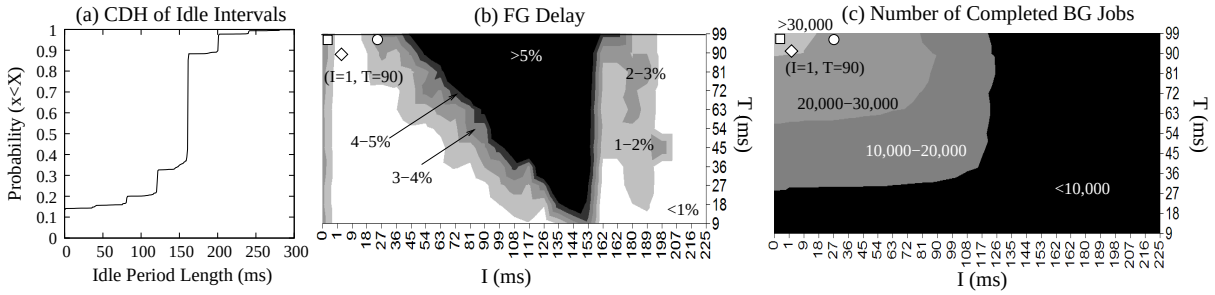



**Figure 10**: Video streaming workload. Plots (b) and (c) capture system performance for *any* pair $(I, T)$. Diamond shapes mark our solution. Square and circle shapes mark common practices, i.e., $(I = 0, T = \infty)$ and $(I = 25, T = \infty)$, respectively.

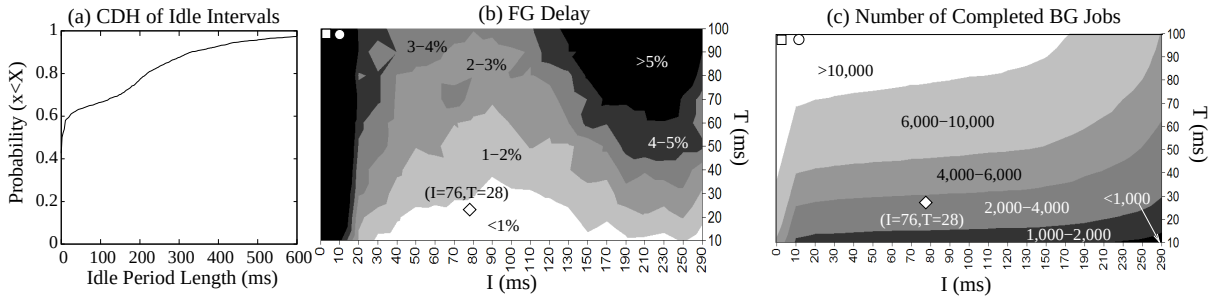



**Figure 11**: Linux compilation workload. Plots (b) and (c) capture system performance for *any* pair $(I, T)$. Diamond shapes mark our solution. Square and circle shapes mark common practices, i.e., $(I = 0, T = \infty)$ and $(I = 9, T = \infty)$, respectively.

reduce disk traffic) and outlined features of an effective idle detector focusing on predicting the start time of an idle interval as well as the idle period duration.While idle times offer an opportunity to serve background tasks, the main performance pitfall relates to cases where background jobs cannot be preempted instantaneously. In these cases, foreground performance may be significantly affected and accurate prediction of the idle period start time and its duration becomes critical.

In [7] the authors identify several policies for predicting the idle start time and its duration, and perform an exhaustive combinatorial search to determine the best (and worst) set of policies for a specific set of traces. The results show that the task of combining the best start time and duration policies is a challenging one, especially when adaptive predictors are employed.In [6] the authors conclude that effective scheduling of non-preemptive background tasks is done using a non-work-conserving approach by using a simple fixed delay in the execution of a background job during an idle interval. This technique avoids using short idle intervals to serve long background jobs, thus limits potentially severe degradation in foreground performance. In [15] the authors take advantage of burstiness in idle periods to improve prediction of the duration of successive idle intervals.Adaptively determining idle times has been also proposed in mobile devices for power-saving by spinning-down their disks [5, 9].

In [8], an arrival curve based approach is used to provide QoS guarantees. This approach may be also used to allocate spare system capacity to background jobs. In [4], the authors present a reactive, closed-loop approach to schedule low priority jobs based on continuous monitoring of the performance of low priority processes, whose progress is used to manage the degradation of performance of high-priority processes. The proposed progress-based regulation could be also employed to interleave background and foreground jobs in storage systems.

The algorithmic framework that is presented in this paper is an open-loop approach and is most closely related to the techniques presented in [7, 6, 15]. Here we present an algorithmic framework that rigorously quantifies the schedulability of background activities by determining two important parameters, i.e., *when* and for *how long* the system should serve non-instantaneously preemptable background jobs with strictly lower priority than foreground ones. While [7, 6] discuss general issues related to utilization of idle intervals and present guidelines and tools that can be effective in the process, they do not provide a generally applicable algorithm that can work effectively with different workloads. Our framework bases it decision mainly on the histogram of idle times avoiding any need to monitor the complex arrival and service processes [7] or multiple performance metrics that feed to computationally expensive statistical prediction techniques [4]. Unlike the methodology laid out in [7], which uses different techniques to predict when the idle busy period starts and when it ends, our framework determines simultaneously both parameters, increasing the overall accuracy and efficiency.

## 6. CONCLUSIONS

This paper proposes a methodology that decides *when* and for *how long* during idle times in storage systems, non-preemptive background jobs can be served such that two conflicting goals are met: (1) degrade foreground performance by no more than a predefined target, and (2) avoid background work starvation. The only input parameter in the framework is in the form of a foreground performance

degradation target. The rest of the necessary information is obtained by on-line monitoring of a compact set of system metrics, with the most important one being the empirical distribution of idle times.

The choice to use the empirical distribution of idle times proves to be an important one that allows our methodology to incorporate accurately the complex interaction between the arrival and service processes of foreground traffic. An extensive set of trace-driven simulation experiments and measurements in a prototype on the Linux 2.6.22 kernel, show that our approach meets the performance targets by finding a solution that is among the best while balancing the two conflicting goals.

In the future, we intend to deploy our framework to evaluate in-depth specific system architectures and features that rely on effective service of background work. An example is to power-off disks in a storage system with the goal of reducing power consumption.

## Acknowledgements

This work has been done during the internships of Ningfang Mi and Xin Li at Seagate Research. Evgenia Smirni has been partially supported by NSF grants CSR-0720699 and CCF-0811417 and by Seagate Research.

## 7. REFERENCES

[1] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Lazy verification in fault-tolerant distributed storage systems. In *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 179–190, Oct. 2005.

[2] E. Bachmat and J. Schindler. Analysis of methods for scheduling low priority disk drive tasks. In *Proceedings of the ACM SIGMETRICS Conference*, pages 55–65, June 2002.

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

[4] J. R. Douceur and W. J. Bolosky. Progress-based regulation of low-importance processes. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, pages 247–260, Dec. 1999.

[5] F. Douglis, P. Krishnan, and B. N. Bershad. Adaptive disk spin-down policies for mobile computers. In *Proceedings of the 2nd USENIX Symposium on Mobile and Location-Independent Computing*, pages 121–137, 1995.

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

[7] R. Golding, P. Bosch, C. Staelin, T. Sullivan, and J. Wilkes. Idleness is not sloth. In *Proceedings of the Winter'95 USENIX Conference*, pages 201–222, 1995.

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

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

[10] H. Huang, W. Hung, and K. G. Shin. Fs2: dynamic data replication in free disk space for improving disk performance and energy consumption. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, pages 263–276, 2005.

[11] I. Iliadis, R. Haas, X.-Y. Hu, and E. Eleftheriou. Disk scrubbing versus intra-disk redundancy for high-reliability raid storage systems. In *Proceedings of the 2008 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 241–252, 2008.

[12] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim. A case for flash memory SSD in enterprise database applications. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 1075–1086, 2008.

[13] A. Leventhal. Flash storage memory. *Communications of ACM*, 51(7):47–51, 2008.

[14] A. Merchant and P. S. Yu. An analytic model of reconstruction time in mirrored disks. *Performance Evaluation Journal*, 20(1-3):115–129, 1994.

[15] N. Mi, A. Riska, Q. Zhang, E. Smirni, and E. Riedel. Efficient management of idleness in systems. *ACM Transactions on Storage (to appear)*, 2009.

[16] Z. Niu, T. Shu, and Y. Takahashi. A vacation queue with setup and close-down times and batch markovian arrival processes. *Perform. Evaluation*, 54(3):225–248, 2003.

[17] A. Riska and E. Riedel. Disk drive level workload characterization. In *Proceedings of the USENIX Annual Technical Conference*, pages 97–103, May 2006.

[18] A. Riska and E. Riedel. Idle Read After Write - IRAW. In *Proceeding of the USENIX Annual Technical Conference*, pages 43–56, 2008.

[19] T. J. E. Schwarz, Q. Xin, E. L. Miller, D. D. E. Long, A. Hospodor, and S. Ng. Disk scrubbing in large archival storage systems. In *Proceedings of the International Symposium on Modeling and Simulation of Computer and Communications Systems (MASCOTS)*, pages 409–418, 2004.

[20] M. Sivathanu, V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving Storage System Availability with D-GRAID. In *Proceedings of the Third USENIX Symposium on File and Storage Technologies (FAST '04)*, pages 15–30, March 2004.

[21] H. Takagi. *Queuing Analysis Volume 1: Vacations and Priority Systems*. North-Holland, New York, 1991.

[22] E. Thereska, J. Schindler, J. Bucy, B. Salmon, C. R. Lumb, and G. R. Ganger. A framework for building unobtrusive disk maintenance applications. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST)*, pages 213–226, 2004.

[23] A. Venkataramani, R. Kokku, and M. Dahlin. TCP Nice: a mechanism for background transfers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 329–343, 2002.