Practical Scrubbing: Getting to the bad sector at the right time

George Amvrosiadis
Dept. of Computer Science
University of Toronto
Toronto, Canada
gamvrosi@cs.toronto.edu

Alina Oprea
RSA Laboratories
Cambridge, MA
aoprea@rsa.com

Bianca Schroeder
Dept. of Computer Science
University of Toronto
Toronto, Canada
bianca@cs.toronto.edu

***Abstract*—Latent sector errors (LSEs) are a common hard disk failure mode, where disk sectors become inaccessible while the rest of the disk remains unaffected. To protect against LSEs, commercial storage systems use scrubbers: background processes verifying disk data. The efficiency of different scrubbing algorithms in detecting LSEs has been studied in depth; however, no attempts have been made to evaluate or mitigate the impact of scrubbing on application performance.**

We provide the first known evaluation of the performance impact of different scrubbing policies in implementation, including guidelines on implementing a scrubber. To lessen this impact, we present an approach giving conclusive answers to the questions: *when* should scrubbing requests be issued, and at *what size*, to minimize impact and maximize scrubbing throughput for a given workload. Our approach achieves six times more throughput, and up to three orders of magnitude less slowdown than the default Linux I/O scheduler.

***Keywords*-scrubbing; hard disk failures; latent sector errors; idleness predictors; background scheduling**

I. Introduction

It is estimated that over 90% of all new information produced in the world is being stored on magnetic media, primarily hard drives [1], making the reliability of these complex mechanical components crucial. Alas, disk drives can fail for numerous reasons, and while for many years it was assumed that disks operate in a "fail-stop" manner, Bairavasundaram et al. [2] showed that Latent Sector Errors (LSEs) are a common failure mode. In the case of LSEs, individual disk sectors become inaccessible, while the remainder of the disk is unaffected. LSEs are a particularly insidious failure mode, since they are silent and only detected when the affected disk area is accessed. If there is no redundancy in the system, the data on the affected sectors is lost. While most storage systems do provide redundancy, most commonly through the use of RAID, LSEs can still cause data loss if they are detected during RAID reconstruction after a disk failure. This scenario is becoming a major concern, particularly since the rate at which disk capacities increase suggests that by 2015, a full-disk scan will incur at least one LSE [3].

To protect against data loss due to LSEs, most commercial storage systems use a "scrubber": a background process that periodically performs full-disk scans to proactively detect and correct LSEs. The goal of a scrubber is to minimize the time between the occurrence of an LSE and its detection/correction, also referred to as the Mean Latent Error Time (MLET), since during this time the system is exposed to the risk of data loss (e.g. if another drive in the disk array fails). In addition to reducing the MLET, a scrubber must ensure to not significantly affect the performance of foreground workloads running on the system.

The importance of employing an efficient scrubber will only increase in the future, as continuously growing disk capacities will increase the overheads for a disk scan, and the rate at which LSEs will happen. Unfortunately, the scrubbers employed in today's storage systems are quite simplistic: they scan the disk sequentially in increasing order of Logical Block Numbers (LBN) at a rate determined by the system administrator. This simple approach ignores a number of design options that have the potential for reducing the MLET, as well as the impact on foreground workload.

The first design question is determining the order in which to scrub the disk's sectors. While scrubbing the disk sequentially is simple and efficient (as sequential I/O is more efficient than random accesses), recent research [4] shows that an alternative approach, called *staggered scrubbing*, provides lower MLET. Staggered scrubbing aims to exploit the fact that LSEs happen in (temporal and spatial) bursts: rather than sequentially reading the disk from beginning to end, the idea is to quickly "probe" different regions of the drive, hoping that if a region has a burst of errors the scrubber will detect it quickly and then immediately scrub the entire region. While reducing MLET, the overhead of the random I/O in staggered scrubbing can potentially reduce scrub throughput and increase the impact on foreground workloads. Unfortunately, there exists no experimental evaluation that quantifies this overhead, and staggered scrubbing is currently not used in practice.

The second design question is deciding when to issue scrub requests. The scrubbers employed in commercial storage systems today simply issue requests at a predefined rate, e.g. every r msec. This approach has two shortcomings. First, it does not attempt to minimize impact on foreground traffic, as scrub requests are issued independently of foreground activity. While research has been done on scheduling general background traffic [5], [6], [7], [8], [9], none is particularly geared towards scrubbers. Second, it is often hard for system administrators to choose the rate r that is right for their system, since it is hard to predict the impact on foreground traffic that's associated with a particular rate.

A third parameter that is not very well understood is the scrub request size, or the number of sectors scrubbed by individual scrub requests. Larger request sizes lead to more efficient use of the disk, but also have the potential of bigger impact on foreground traffic, as foreground requests that arrive while a scrub request is in progress get delayed.

Finally, there are a number of implementation choices for a scrubber that have not been well studied, including the choice of a user-level versus kernel-level implementation, or the effect of the disk interface (SCSI versus ATA).

The contributions of our work are summed up as follows:

- We developed a framework that can be used to implement scrubbing algorithms in only tens of lines of code (LoC) within the linux kernel, and made its source code publicly available[1]. We also implemented a user-level scrubber to allow for a quantitative comparison between user-level and kernel-level scrubbing.
- We provide the first implementation of a staggered scrubber and compare its performance to that of a standard sequential scrubber.
- We perform a detailed statistical study of publicly available disk I/O traces, and apply the results towards the design of an efficient approach for scheduling scrub requests.
- We provide conclusive answers to the questions: *when* should scrubbing requests be issued and at *what size*, using an approach that maximizes scrubbing throughput for a given workload, while meeting a predefined slowdown goal for foreground traffic. Our approach significantly outperforms the default linux I/O scheduler, the only one to allow I/O prioritization.
- We provide the first known evaluation of the performance impact of different scrubbing policies using our framework and our approach in conjunction with real-world workload traces.

The paper is organized as follows: In Section II we describe work related to this paper. Section III describes the framework we implemented and used to develop sequential and staggered scrubbing; a performance evaluation of the two algorithms follows in Section IV. We look at ways to increase scrubbing throughput, while avoiding impact on applications in Section V, and conclude with a discussion of our observations and future directions in Section VI.

[1]Kernel patches can be found at http://www.cs.toronto.edu/∼gamvrosi

II. Related Work

The concept of scrubbing is not new in reliability research; several papers have studied scrubbing in different contexts. While scrubbers in production systems simply scan the disk sequentially, work by Oprea et al. [4] shows that an approach based on sampling can exploit LSE locality to reduce MLET. The disk is separated into R regions, each partitioned into S segments. In each scrub interval, the scrubber begins by reading the first segment from each region ordered by LBN, then the second one, and so on. Other proposed algorithms focus on actions taken once an LSE is detected [10], [11], or scrubbing in archival systems with limited uptime [12]. All these studies focus solely on reducing the MLET, and do not evaluate the scrubber's impact on the performance of foreground traffic. Also, evaluation is done analytically, without an actual implementation. This paper provides an implementation of *sequential* and *staggered* scrubbing, and evaluates their impact on foreground traffic performance; we also examine a number of other design choices for implementing a scrubber.

A number of prior studies have also focused on ways to schedule background workloads in a storage system, while limiting the impact on foreground traffic. Unfortunately, none of this work is optimal or suitable for scheduling background scrub requests. Proposed approaches include: I/O preemption [13], merging background requests with application requests [5], [6] and piggy-backing background requests to foreground requests [14]. These approaches, however, make extensive use of prefetching, and a scrubber should ideally avoid request prefetching/merging as it pollutes the on-disk and page caches with data from scrubbed sectors. Furthermore, a scrubber needs to guarantee that the sector's contents were verified from the medium's surface (rather than a cache) at request execution time.

Other approaches try to limit the impact of background requests on foreground traffic by trying to predict disk idleness and scheduling background requests during those idle intervals. Golding et al. [7] provide a general taxonomy of idleness predictors, without, however, adapting their parameters to accommodate for different workloads or specific slowdown goals. Our study moves along the lines of prior work by Mi et al. [8], [9] and Schindler et al. [15], who try to determine the best start and end times to schedule background requests while limiting the number of *collisions* (situations where a foreground request arrives to find the disk busy with a background request). Our approach, instead of focusing on collisions requires more intuitive input: the average slowdown allowed per I/O request. We also find that a simple approach is sufficient, where only the start time is determined, and requests are always issued until a collision happens. Finally, we optimize a different parameter set specific to scrubbing, which provides additional degrees of freedom such as the scrub request size.

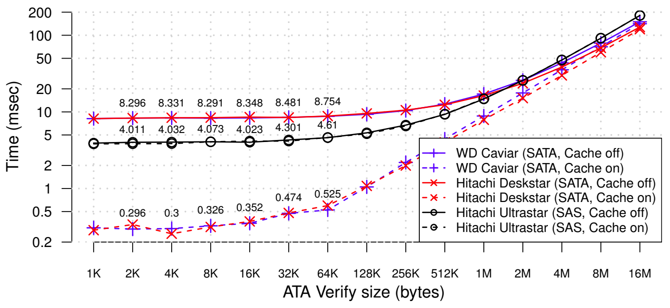


Figure 1. Response times for different ATA VERIFY sizes

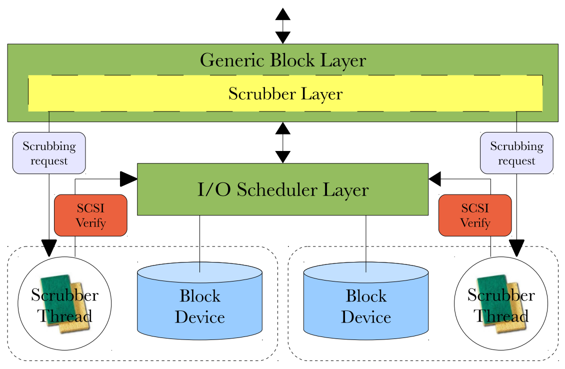


Figure 2. Architecture of our kernel scrubber.

III. IMPLEMENTING A SCRUBBER

When implementing a scrubber, three components need to be considered: an interface directing the disk to carry out the verification of specific sectors, a module that implements the scrubbing algorithm(s), and an I/O scheduler capable of issuing scrub requests in a fashion that limits the impact on foreground traffic. We experimented with the SCSI and ATA interfaces [16], and examine their ability to issue scrub requests in Section III-A. In Section III-B, we briefly present the only I/O scheduler in linux that provides I/O prioritization. Finally, we explore and evaluate different implementations of the scrubbing module in both user- and kernel-level in Section III-C.

A. *Scrub requests in SCSI/SAS vs. ATA/SATA*

Scrubbers typically rely on the VERIFY commands implemented in both the SCSI and ATA interfaces to issue scrub requests, rather than using regular disk reads. The reason is that VERIFY guarantees to verify the data from the medium's surface (rather than reading it from the on-disk cache) and prevents cache pollution by avoiding to transfer any data or use prefetching.

Interestingly, our experiments with multiple ATA drives show that ATA VERIFY is not implemented as advertised. Specifically, we observe strong evidence that ATA VERIFY reads data from the on-disk cache, rather than the medium's surface. Our evidence is summarized in Fig. 1, which shows ATA VERIFY response times for different request sizes for two popular current ATA drives (WD Caviar and Hitachi Deskstar) and one SAS drive (Hitachi Ultrastar) when accessing data sequentially from the disk. The solid lines for all models show the response times when the on-disk cache is disabled, while the dashed lines show results for when the on-disk cache is enabled. It is evident that disabling the cache affects VERIFY response times for the ATA drives but not for the SAS drive, indicating that the former do depend on the on-disk cache, rather than forcing accesses to the drive's platter. We conclude that since ATA VERIFY is implemented incorrectly in modern ATA disks, scrubbers using them would likely pollute the on-disk cache.

B. *The Completely Fair Queueing (CFQ) I/O scheduler*

In our implementation we use the linux CFQ I/O scheduler, as it is the only open source scheduler that supports I/O prioritization. CFQ provides a separate priority class (*Idle*), for scheduling of background requests. To minimize the effect of background requests on foreground ones, CFQ only issues requests from the *Idle* class after the disk has remained idle for at least $10ms$. Although this parameter is tunable, changing it in linux 2.6.35 did not seem to affect CFQ's background request scheduling.

C. *User space, or kernel space?*

While current scrubbers rely on the VERIFY commands for the reasons outlined in Section III-A, there are downsides to using them due to the way they are executed by the linux kernel. As is common with device-specific functionality in linux, a wild-card system call is used (`ioctl`) to pass a packet with the command and its arguments directly to the I/O scheduler, and then to the device driver for execution. However, since the kernel has no knowledge of the command that is about to be executed, such requests are flagged as *soft barriers*, and performance optimizations (e.g. reordering between or merging with outstanding block requests) are not applied. Since a scrubber implemented in user space has no way to avoid the performance penalty due to these scheduling decisions made in the kernel, we implement our scrubbing framework entirely in kernel space.

In Fig. 2 we present the architecture of our kernel scrubber, implemented in the block layer of the 2.6.35 kernel [17]. The scrubber is activated at bootstrapping, matching scrubber threads to every block device in the system; this matching is updated when devices are inserted/removed, e.g. due to hot swapping. The threads remain dormant by being inserted in the CPU's sleeping queue, until scrubbing for a specific device is activated. Internally, the scrubber implements SCSI VERIFY[2], since it is not natively supported by the kernel, and provides a simple API that can be

[2]Our implementation supports ATA devices through the kernel's libATA library, which performs the appropriate translation for VERIFY.

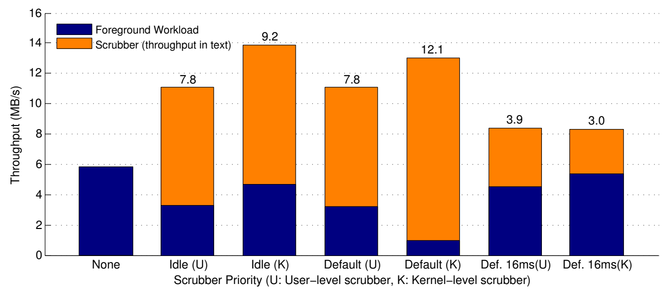


Figure 3. Comparison of our user- (U) and kernel-level (K) scrubbers.

used to code new scrubbing strategies. We implemented our framework in 2700 commented LoC, and in that we coded sequential and staggered scrubbing in approx. 50 LoC each.

To send requests to the I/O scheduler and set the I/O priority for scrubber threads (if CFQ is used), we use the Generic Block Layer interface. To enable the I/O scheduler to sort scrubbing requests among other outstanding ones, every time a scrubber thread dispatches a VERIFY request we disguise it as a regular read request bearing all relevant information, such as starting LBN and request size. This information is unavailable in the vanilla kernel, since sorting is not permitted for soft barrier commands. Once the scrubbing request has been dispatched, we put the thread back to the sleeping queue, and at request completion it is awakened by a callback function to repeat the process.

Fig. 3 shows the results from experimenting with the basic version of our kernel-level scrubber and a basic user-level scrubber. We generate a simple, highly-sequential foreground workload, with exponential think times between requests ($\mu = 100ms$) to allow for idle intervals that the scrubber can utilize. In one set of experiments we run the foreground workload and the scrubber at the same priority level (*Default*), while in a second set of experiments we use CFQ's *Idle* priority class to reduce the scrubber's priority. For both the Idle and the Default priorities we allowed the scrubber to issue requests back to back, and for the Default priority, we also experimented with a $16ms$ delay between scrub requests (*Def.* $16ms$), in order to allow the foreground workload to make progress. Results are shown for a Hitachi Ultrastar SAS drive, but we also experimented with a Fujitsu MAX3073RC SAS drive and got similar results.

It is evident from Fig. 3 that when allowing the scrubber to issue requests back-to-back, both the scrubber and the foreground workload achieve significantly higher throughput in the kernel-level implementation. Also, priorities have no effect on the user-level scrubber whose requests are soft barriers, as opposed to the kernel scrubber, which is benefiting by the workload's think time and starving it under the Default priority. When the scrubber is delayed, however, the maximum scrubbing throughput (3.9MB/s, or 64KB/$16ms$) is reached only by the user-level scrubber; proper prioritization limits the kernel scrubber's throughput at 3MB/s. These results clearly motivate the use of sophisticated scheduling with I/O prioritization, if the scrubber's impact is to be mitigated while retaining high throughput.

IV. GETTING THERE QUICKLY

The goal of this section is to compare the performance of staggered and sequential scrubbing. While sequential scrubbing is the approach currently employed in practice, research indicates that staggered scrubbing can significantly reduce the MLET. The reason practitioners are shying away from using staggered scrubbing, is the fear that moving from sequential to more random scrubbing I/O patterns might affect the performance of the I/O system, both in terms of the scrubber's throughout, and its impact on foreground workloads. To quantify these effects, we implemented both a sequential and a staggered scrubber within the kernel-level framework described in Section III-C. We begin our experimental evaluation in Section IV-A by comparing the scrubbing throughput that can be achieved by a staggered versus a sequential scrubber, and then evaluate their impact on foreground traffic, using simple synthetic foreground workloads in Section IV-B, and more realistic trace-based workloads in Section IV-C.

A. *Staggered versus sequential scrubbing*

The performance of both a sequential and a staggered scrubber will naturally depend on the choice of their parameters. Therefore, we begin by evaluating the effect of the scrubbers' parameters on their performance to identify the optimal set of parameters for each scrubber.

The only tunable parameter of a sequential scrubber is the *request size* S of each scrubbing request, a parameter shared with the staggered scrubber. Our first goal in this section is to distinguish the range of request sizes that can achieve high scrubbing throughput. We measured the response time of SCSI VERIFY for different request sizes for two SAS drives (Hitachi Ultrastar 15K450 300GB and Fujitsu MAX3073RC 73GB) and one SCSI disk (Fujitsu MAP3367NP 36GB), and we found that for requests $\leq$ 64KB, response times remain almost constant for all models (Fig. 4). As a consequence, we henceforth experiment only with request sizes starting at 64KB, and report the throughput achieved by each scrubber. The results for the two SAS drives are shown in the two solid lines in Fig. 5a, suggesting that the largest scrubbing request size should be preferred, from 64KB to 4MB.

In the case of the staggered scrubber there is an additional parameter: the number of regions that the disk is divided into. Recall that a staggered scrubber separates the disk into R regions, based on a predefined *region size*, each of which gets scrubbed in $\lceil \frac{R}{S} \rceil$ rounds. In the first round, the first S bytes are verified from each region, then the S after those, and so on. To experiment with the region size, we ran the staggered scrubber on our two SAS disks using 64KB

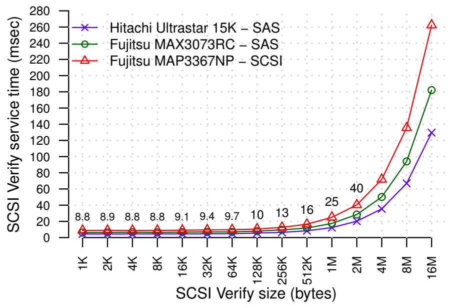
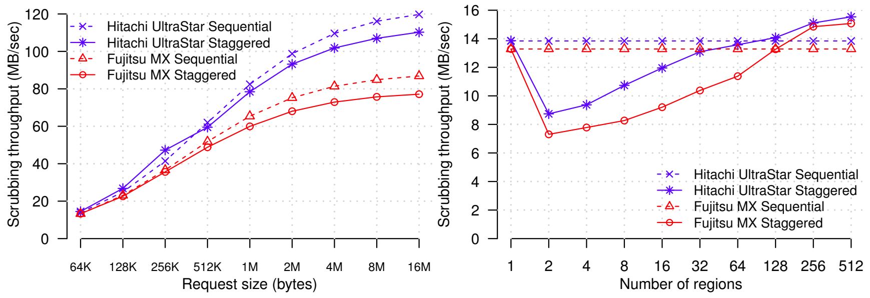


Figure 4. Service times for different SCSI VERIFY sizes



(a) Request size (128 regions)

(b) Number of regions (64KB requests)

Figure 5. Impact of scrubbing parameters on sequential and staggered scrubbing performance.

requests (the presented trend holds regardless of request size), dividing the disks in up to 512 regions. The solid lines in Fig. 5b represent the staggered scrubber's throughput for the two drives as a function of the number of regions. We observe that the throughput of the scrubber continuously increases as the number of regions increases from two to 512 (in the case of one region, the staggered scrubber's actions are identical to a sequential scrubber). To answer our original question, which is how the performance of a staggered scrubber compares to that of a sequential scrubber, the dashed lines in Fig. 5b represent the throughput achieved by a sequential scrubber (using 64KB requests). Interestingly, for more than 128 regions we find that the staggered scrubber performs equally well or better than the sequential one. We have observed this trend for drives of varying capacities, which leads us to assume that it is independent of the disk's capacity. To verify that this trend holds for larger request sizes, we have also plotted the throughput of a staggered scrubber as a function of the request size (while fixing the number of regions to 128) in Fig. 5a. Since work on the impact of staggered scrubbing on the MLET shows that the number of regions has a relatively small impact on MLET [4], we recommend using small region sizes, compared to the disk's capacity. To obtain conservative results, though, we fix the number of regions to 128 in subsequent experiments.

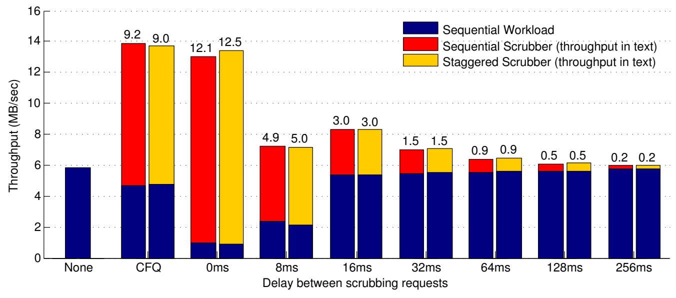
The fact that staggered can outperform sequential scrubbing may seem counter-intuitive. However, since VERIFY avoids transferring data to the controller or on-disk cache, when a sequential scrubbing request is completed, the next one will have to be serviced from the medium's surface. At the same time, the head has moved further along the track, while the VERIFY result was propagated to the controller. Hence, when the next VERIFY is initiated, the head needs to wait for a full rotation of the platter until it reaches the correct sector. For the staggered scrubber, this describes only the *worst case*. However, when the regions are too large (less than 64 in total), the overhead of jumping between regions dominates the overhead caused by rotational latency. We have validated this hypothesis experimentally by increasing the delay between subsequent scrubbing requests, by intervals smaller than the rotational latency. As expected, only the staggered scrubber was harmed by such delays.

B. *Scrubbing impact on synthetic workloads*

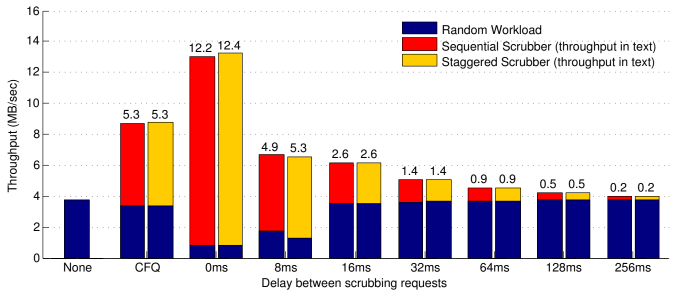
Next, we evaluate the performance impact of scrubbing on two simple synthetic foreground workloads. The first is a workload with a high degree of sequentiality: it picks a random sector and reads the following 8MB using 64KB requests. Once the chunk's last request is serviced, it proceeds to pick another random sector and start again. The second is a random workload, which reads random 64KB chunks of data from the disk. For both workloads we insert an exponentially distributed think time between requests, with a mean of $100ms$, to allow for idle intervals that the scrubber can utilize. In all cases, we send requests directly to the disk, bypassing the OS cache.

We experiment with both the staggered and the sequential scrubber using 64KB scrub requests, which represent the best case in terms of collision impact by imposing minimal slowdown (recall Fig. 4). We schedule scrub requests in two commonly used ways: in one case we issue scrub requests back-to-back through CFQ, using the *Idle* priority to limit their impact on the foreground workload; in the second case we use the *Default* priority and limit the scrubber's rate by introducing delays between scrub requests, ranging from 0-$256ms$ (anything larger scrubs less than 320GB bi-weekly).

The results for the sequential workload are shown in Fig. 6a. We observe that the highest combined throughput for the workload and the scrubber is achieved with CFQ (where the scrubber submits requests back-to-back); however, this comes at a significant cost for the foreground application: a drop of 20.6% in throughput, compared to the case when the foreground workload runs in isolation. When inserting delays between scrub requests, rather than using CFQ to limit the impact of the scrubber, we see that for large enough delays ($\geq 16ms$) the throughput of the foreground workload is comparable to that without a scrubber. However, in those cases the throughput of the scrubber is greatly reduced, from 9MB/s under CFQ to less than 3MB/s with a delay of more than $16ms$. As a secondary result, we note that again there



(a) Sequential foreground workload results



(b) Random foreground workload results

Figure 6. Throughput comparison of sequential and staggered scrubbing, when run alongside a synthetic workload (64KB segment size, 128 regions).

is no perceivable difference between the staggered and the sequential scrubber for sufficiently small regions (here: 128).

We observe similar results when we scrub against the random workload in Fig. 6b: to achieve application throughput comparable to the case without a scrubber in the system, large delays are required that cripple the scrubber's throughput. Note that random workloads induce additional seeking, decreasing the scrubber's throughput.

C. Scrubbing impact on real workloads

To experiment with more realistic workloads, we replayed a number of real I/O traces from the *HP Cello* and *MSR Cambridge* collections, available publicly from SNIA [18]. We have worked with a set of 77 disk traces in total, spanning from one week to one year (we make use of only one week in our experiments) and being used in almost all possible scenarios: home and project directories, web and print servers, proxies, backups, etc. Although we ran the experiments in the rest of the paper for almost all disks, we have chosen to focus mainly on four disks from each trace collection when presenting our results. These disks contain the largest number of requests per week, and represent diverse workloads. The characteristics of the chosen traces are summarized in Table I [3].

Trace	Disk	Requests	Description
MSR Cambridge (2008)	src11	45,746,222	Source Control
	usr1	45,283,980	Home dirs
	proj2	29,266,482	Project dirs
	prn1	11,233,411	Print server
HP Cello (1999)	c6t8d0	9,529,855	News Disk
	c6t5d1	4,588,778	Project files
	c6t5d0	3,365,078	Home dirs
	c3t3d0	2,742,326	Root & Swap
MS TPC-C (2009)	disk66	513,038	TPC-C run
	disk88	513,844	TPC-C run

Table I
SNIA Block I/O traces used in the paper

[3]We originally added TPC-C [18] for completeness (database workload), but later found it to produce an unrealistic distribution of inter-arrivals.

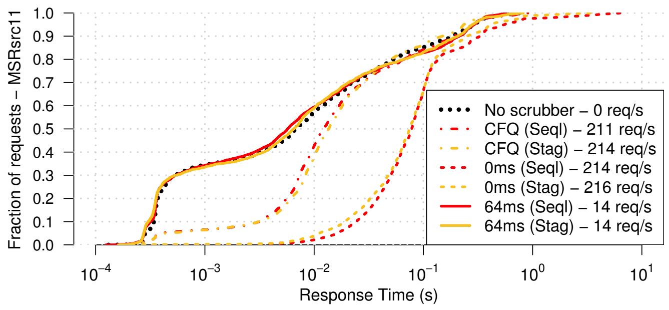


Figure 7. Response Times CDFs during the replay of a real I/O trace.

Fig. 7 shows the impact of our scrubbers on one of these (more realistic) workloads by plotting the cumulative distribution function of the response times of application requests. For better readability we only include results for four different cases: no scrubber; back-to-back scrub requests scheduled through CFQ's Idle priority class; and scrub requests with delays of $0ms$ and $64ms$ between them.

Again, we observe results very similar to those for the synthetic workloads. Using back-to-back scrub requests, even when lowering their priority through CFQ, greatly affects the response times of foreground requests. On the other hand, when making delays between scrub requests large enough to limit the effect on the foreground workload ($64ms$), the throughput of the scrubber drops by more than an order of magnitude (the scrubber's throughput for each experiment is included in the legend of Fig. 7). As before, the results are identical for the staggered and the sequential scrubber. These results motivate us to look at more sophisticated ways to schedule scrub requests.

V. Knowing when to scrub

Our experiments in Section IV showed that simply relying on CFQ to schedule scrub requests, or issuing them at a fixed rate is suboptimal both for minimizing impact on foreground traffic, and for maximizing the scrubber's throughput. This motivates us to consider more sophisticated methods for scheduling scrub requests in this section. Using statistical analysis of I/O traces, we identify some statistical properties

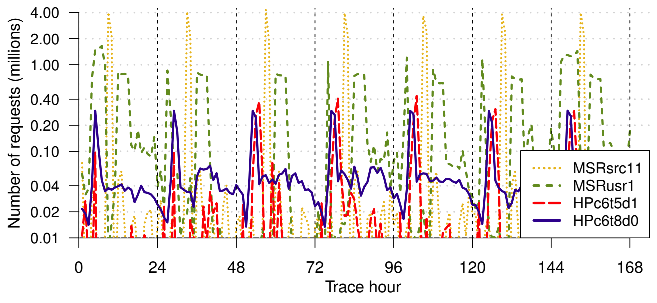


Figure 8. Request activity for four disks from both the *HP Cello* and the *MSR Cambridge* traces.

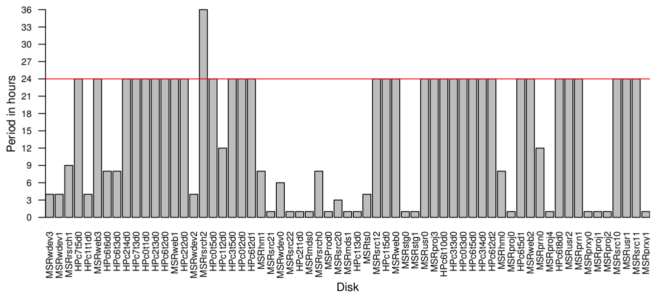


Figure 9. Periods detected for the busiest 63 disks using ANOVA.

of I/O workloads, to aid us in deriving better techniques for scheduling scrub requests in Section V-A. We define and experimentally evaluate those techniques in the remaining sections.

A. *Insights from the statistical analysis of traces*

While previous work [19], [20] has provided some general analysis of I/O workloads, in this section we look at I/O traffic characteristics that are relevant to scheduling background scrub requests. The main goal is to schedule scrub requests with minimal impact on foreground traffic by making better use of the idle intervals that hard drives experience.

Periodicity: Periodic behavior in the length of idle intervals is helpful for scheduling scrub requests, since it provides a predictable pattern that a scheduler can exploit. While we are not aware of prior work that has specifically focused on periodic behavior of I/O workloads, one might expect I/O workloads to exhibit periodic patterns, such as diurnal trends. We begin our analysis of periodicity by a visual inspection of how the request arrival rate varies as a function of time. Fig. 8 plots the number of requests per hour as a function of time for four representative traces from our trace collection. We observe that all four traces exhibit repeating patterns, often with spikes at 24 hour intervals, but in some cases also at other time intervals. For *Cello*, these consistent spikes could be attributed to daily backups [21], while for *MSR* activity peaks on different hours for different disks, with some days seeing smaller or no peaks. We believe that such activity spikes can be common in the field, with

Trace	Disk	Mean (s)	Variance	CoV
MSR Cambridge (2008)	src11	0.4640	101.31	21.693
	usr1	0.0997	0.7448	8.6516
	proj2	0.1384	772.18	200.75
	prn1	0.2280	8.3073	12.641
HP Cello (1999)	c6t8d0	0.1502	4.3243	13.845
	c6t5d1	0.4503	180.13	29.807
	c6t5d0	0.4345	15.545	9.0731
	c3t3d0	0.4555	14.051	8.2301
MS TPC-C (2009)	disk66	0.0014	1.5e-6	0.8608
	disk88	0.0015	1.6e-6	0.8785

Table II
SNIA TRACE IDLE INTERVAL DURATION ANALYSIS RESULTS

varying intensity based on their causes that could be any: from scheduled tasks to applications and/or human activity.

For a more statistically rigorous approach to periods, we used analysis of variance (ANOVA) to identify the time interval with the strongest periodic behavior for each trace in our data set. The results are shown in Fig. 9. We observe that for most traces ANOVA does identify periods, most commonly at intervals of 24 hours (our analysis was done at the granularity of hours, so periods of one hour in Fig. 9 mean there was no periodicity identified).

Autocorrelation: Autocorrelation is an interesting statistical property, as it means that the length of previous (recent) idle intervals is predictive of the lengths of future idle intervals; information that a scheduler could use to identify long idle intervals, in which to schedule scrub requests. Previous work [19] has reported evidence of autocorrelation for some (not publicly available) disk traces in the form of Hurst parameter values larger than 0.5. We studied the autocorrelation function for all our traces and found that 44 out of the busiest 63 disk traces exhibit strong autocorrelation.

Decreasing hazard rates and long tails: Previous work [19] has reported that the distribution of I/O request inter-arrival times exhibits high variability. We verified that this is also the case for our traces. We observe Coefficients of Variation[4] typically in the 10–30 range (Table II), and in one case as high as 200. These numbers are even higher than those reported in [19], who observed a CoV of 19 for their most variable trace. To put those numbers in perspective, we remind the reader that an exponential distribution has a CoV of 1. The exponential distribution is a memoryless distribution, i.e. if idle times were exponentially distributed, then the expected time until the next request arrival would always be the same, independent of the time since the last arrival. We believe this is unrepresentative of real workloads, and found it to be the case only for the TPC-C traces.

The high CoV values signify long tails in the distribution of idle intervals. In the context of our work, a long tail implies that a large fraction of the system's total idle time is concentrated in a small fraction of very long idle intervals.

[4]Recall that the Coefficient of Variation (CoV) is defined as the standard deviation divided by the mean.

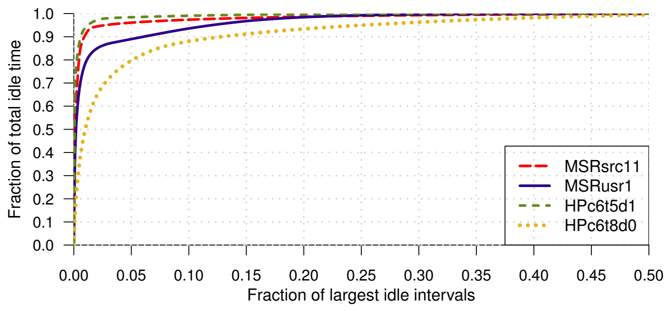


Figure 10. What fraction of a disk's idle time do the largest idle periods make up? Note that the x axis is cut off at the 50^{th} percentile.

For a closer study of this characteristic, Fig. 10 plots the fraction of the total idle time in the system that is made up by the longest idle intervals, i.e. each data point (x, y) shows that the $x\%$ largest idle intervals in the system account for $y\%$ of the total idle time in the system. We observed that for all traces a very large fraction of idle time is concentrated in the tail of the idle time distribution: typically more than 80% of the idle time is included in less than 15% of the idle intervals, in most cases the skew is even stronger.

The strong weight in the tail of the distributions is good news for scheduling background workloads. It means that if we can identify the 15% largest intervals, we can make use of more than 80% of the total idle time in the system. Scheduling background work for only a small percentage of the idle intervals is advantageous for limiting the number of possible collisions, where a foreground request arrives while a background request is in progress.

An obvious problem that occurs at this point is identifying at the beginning of an idle interval, whether it is going to be one of the few very large ones (and hence, scrubbing should be initiated). We can derive two ideas for identifying long idle intervals from our previous two observations: since there is periodicity in the data, we could schedule background work only during those times of the day that tend to be lightly loaded; alternatively, since there is strong autocorrelation in the data, we could use auto-regression to predict the length of upcoming idle intervals based on the lengths of previous ones.

The high CoVs we observed in our traces suggest a third option for predicting idle times: it is possible that the CoVs in our traces are so much higher than that for an exponential distribution, because their empirical idle time distributions have *decreasing hazard rates*, i.e. the longer the system has been idle, the longer it is expected to stay idle. In this case, an approach based on waiting might work well, where we wait until the system has been idle for a certain amount of time before we issue a scrub request.

To check for decreasing hazard rates we plotted the expected remaining idle time, as a function of how long the disk has been idle (Fig. 11). A data point (x, y) in the graph means that after being idle for x seconds, the system will

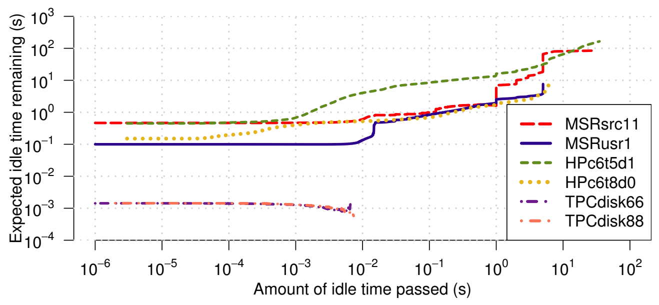


Figure 11. Expected idle time remaining for the traces in Table I

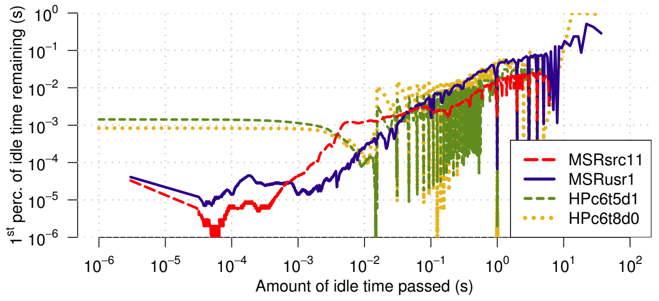


Figure 12. **1^{st} percentile** of idle time remaining for the traces in Table I

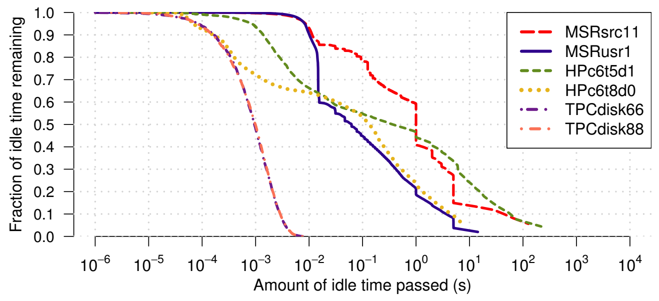


Figure 13. Fraction of idle time remaining for the traces in Table I

be idle in expectation for an *additional* y seconds before the next request arrives. We observe that the lines for all Cello and MSR traces are continuously increasing. In fact, having been idle for a long time increases the expected remaining idle time by several orders of magnitude (note the log scale on the y-axis). Since the expected remaining idle time is just an average (on average after waiting for x seconds it will take another y seconds until the next foreground request arrives) which might be biased by outliers, we also plotted the first percentile of remaining time in Fig. 12. A data point (x, y) in this graph means that in 99% of the cases, after waiting for x seconds we have at least another y seconds before the next foreground request arrives. We again note strongly increasing trends.

One potential problem with the wait-based approach is that we miss out on using the idle time that passes while we wait. Fig. 13 plots the fraction of the total idle time in the system that we can exploit if we only schedule scrub requests after waiting for x seconds. The figure shows that even after waiting for some time before issuing scrub requests, we can still make use of a significant fraction of the total idle time. For example, for a wait time on the order of $100msec$ we can still make use of more than 60-90$\%$ of the total idle time in the system, depending on the trace. At the same time, the number of possible collisions between arriving foreground and background requests is limited, since less than 10% of all idle intervals in our traces are larger than $100msec$ and will be picked for scrubbing.

Finally, our observation of decreasing hazard rates has another implication for designing a scheduler for scrub requests. Existing approaches for scheduling background requests [7], [8] consist of a method for identifying a starting criterion (when to start issuing background requests), and a stopping criterion (when to stop issuing background requests). Fig. 11 tells us that we need not worry about a stopping criterion. The goal of background scheduling policies is to schedule background requests when the chance of a foreground request arriving is low. Decreasing hazard rates, however, imply that the chance of a foreground request arriving at any given moment diminishes with time, past the beginning of the idle interval. Therefore, once scrubbing is initiated, if the system is still idle upon the completion of a scrub request, the chance of a foreground request arriving is even lower than before issuing the scrub request. This means that once an idle interval is identified as long, the policy that makes most sense statistically is to keep sending background requests, until the next foreground request arrives.

B. *Profiting from idleness*

In this section we define and evaluate three different policy classes for scheduling scrub requests, which have all been directly motivated by our results in Section V-A.

1) ***Autoregression (AR)*** *– Predicting the future:* The strong periods and autocorrelation we observed in our analysis in Section V-A motivated us to look into approaches that capture repetitive patterns. We examined autoregressive (AR) models, which use successive observations of an event to express relationships between a dependent and one or more independent variables. In our experiments we used the simple AR(p) model, which regresses a request inter-arrival interval of length X_t against past intervals $X_{t-1}, ..., X_{t-p}$:

$$X_t = \mu + \sum_{i=1}^{p} a_i(X_{t-i} - \mu) + \epsilon_t, \qquad (1)$$

where $a_i, ..., a_p$ are parameters of the model, ϵ_t is white noise, and μ expresses a mean calculated over past intervals. We estimate the order p using Akaike's Information Criterion [22], that optimizes the ratio between the number of parameters and the accuracy of the resulting model. Since AR models can only be applied to *regular time series*, i.e. sequences of events recorded at regular intervals, we model the *durations* of request inter-arrival intervals [23]. This also implies that AR predictions are estimations of the amount of time until the arrival of the next request. We attempted to fit several AR models to our data, including ACD [24] and ARIMA [25], and found that AR(p) is the only model that can be fitted quickly and efficiently to the millions of samples that need to be factored at the I/O level.

Our AR policy works by predicting the length of the current idle interval X_t based on previous intervals $X_{t-1}, ..., X_{t-p}$ using the AR(p) model. The policy makes the prediction for X_t at the beginning of the current idle interval and starts firing scrub requests if the prediction X_t is larger than some specified time threshold c, which is a parameter of the policy. Once it starts issuing scrub requests it continues until a foreground request arrives.

2) ***Waiting*** *– Playing the waiting game:* The decreasing hazard rates in our traces imply that after the system has been idle for a while, it will likely remain idle. This property is exploited by the *Waiting* policy, which dictates that no requests are to be issued, unless the system has remained idle for some time t (t is a parameter of the policy). Requests stop being issued only upon the arrival of a foreground request.

3) ***AR+Waiting*** *– Combining auto-regression and waiting:* This policy combines the *Auto-regression* and *Waiting* approaches. It waits for a time threshold t and if the system has been idle for that long it starts firing, provided that the auto-regression prediction for the length of this idle interval is larger than some time threshold c.

Comparison of policies: Naturally, for all policies there is a trade-off in the throughput that the scrubber achieves, and the resulting impact on the performance of the foreground traffic. The trade-off is governed by the parameters of the policies, so choosing larger values for parameters c and t of the AR and Waiting policies, respectively, will lead to lower impact on the foreground traffic at the cost of reduced scrub throughput. For a fair comparison of the different policies, we need to find which one achieves highest scrub throughput for a given fixed penalty to the foreground traffic.

Therefore, we compare policies by varying their corresponding parameters, and plotting the resulting amount of idle time that can be utilized for scrubbing versus the number of resulting collisions (the fraction of foreground requests that are delayed due to a scrub request in progress). The results are shown in Fig. 14: MSRusr2 (right) is representative of most disks in our trace collections, while HPc6t8d0 (left) is characterized by multiple short idle intervals, representing a worst case scenario with respect to collisions. The numbers on top of the data points provide the parameter setting used to derive that data point: the wait time threshold t for the Waiting approach and the threshold c for the AR approach.

In addition to AR and Waiting, we also plot results for the combined approach. For that, we experiment with four different values of the c parameter for the AR component. These four values were chosen to be the 20^{th}, 40^{th}, 60^{th}, and 80^{th} percentile of all observed AR values for a trace.

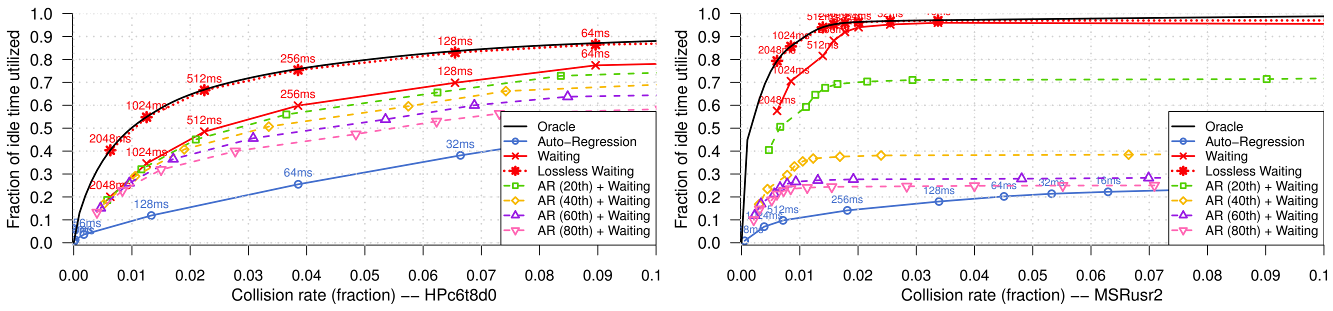


Figure 14. Comparison of the Auto-regression and Waiting approaches with the optimal (Oracle), for two disks: HPc6t8d0 (left) and MSRusr2 (right).

For each of these values we plot one line in the graph, which results from varying the wait time threshold.

Finally, for reference we also plot the best possible results that could be achieved by a clairvoyant *Oracle* that can accurately predict the $x\%$ longest idle intervals and only utilize those, maximizing the amount of idle time one can utilize for a collision rate of $x\%$. This Oracle provides an upper bound on the results one can hope to achieve.

The results in Fig. 14 are quite interesting. We find that the simple Waiting approach clearly outperforms AR and the combined approaches, as it consistently manages to utilize more idle time for a given collision rate. On the other hand, the pure AR policy shows by far the worst performance, which we attribute to its inability to capture enough request history to make successful decisions. While outperforming the other policies, the Waiting approach is weaker than the Oracle. One might wonder whether this is due to the fact that Waiting fails at predicting all the long idle intervals, or because it wastes idle time while sitting idle waiting for t time before firing. To answer this question we also plotted results for a hypothetical policy, *Lossless Waiting*, which utilizes the same intervals as Waiting, while assuming that we can magically also make use of the time that passes while waiting. We see that this hypothetical policy performs very closely to the best possible policy (the Oracle). This means that the Waiting approach is extremely good at identifying long idle intervals, and only falls short from achieving optimal possible performance due to the time spent waiting.

C. *Sizing up throughput opportunities*

One important remaining parameter is the size of each scrub request. While larger scrub requests will increase scrub throughput (recall Fig. 4), they also increase the impact on foreground workloads, as collisions become more expensive (increasing delays for the foreground request arriving while a scrub request is in progress, and often also for the ones following that). Our goal is to take as input from a system administrator the *average* and *maximum tolerable slowdown* per foreground application request, and within these limits find the parameters that maximize scrub throughput.

Fig. 15 shows that the throughput a scrubber can achieve while limiting the slowdown of the foreground application

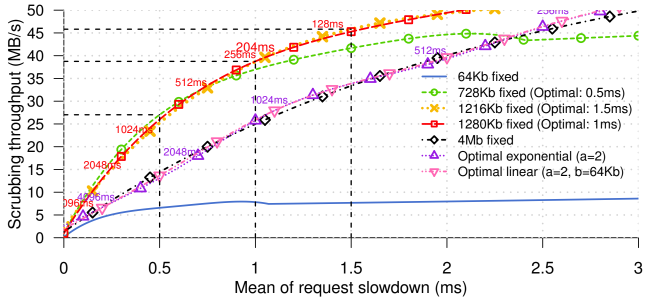


Figure 15. Comparison of different *Waiting* variants. *Fixed* variant prevails.

to some acceptable threshold, can vary dramatically as a function of the request size. The *fixed* lines in Fig. 15 were obtained by simulating the Waiting policy while keeping the request size fixed at 64KB, 768KB, 1216KB, 1280KB and 4MB, and varying the wait time threshold. For each threshold value, we plot the resulting average slowdown on foreground requests versus the throughput achieved by the scrubber (we experimented with request sizes ranging from 64KB to 4MB and a maximum allowed request slowdown of $50ms$, but plot only results for 5 sizes for readability). We observe that for the two extreme examples of 64KB and 4MB, the scrubber using 4MB requests has a consistently higher scrub throughput for the same foreground application slowdown.

This motivated us to try and determine the optimal scrub request size for a specific request slowdown, i.e. the request size that will lead to a slowdown within the prescribed limit, while maximizing scrub throughput (we focus only on the Waiting approach, since we showed how it outperforms the rest). We accomplished this using simulation to find the optimal request size in the range from 64KB to 4MB (bounded by the maximum tolerable slowdown), and the wait time threshold for which it yields the maximum throughput, satisfying the given average slowdown goal per I/O request. For each size, the optimal threshold can be found efficiently through binary search, since for a given request size, larger thresholds will always lead to smaller slowdowns. Using the threshold we can then estimate the maximum throughput per request size, and use that as a comparison metric to find the optimal request size. To summarize, our policy chooses for each slowdown the request size (and corresp. wait time threshold) that maximizes the scrubber's throughput. Results for our policy are shown in Fig. 15, where we find, for example, that some optimal (*slowdown, request size*) pairs are: ($0.5ms$, 768KB), ($1.0ms$, 1280KB) and ($1.5ms$, 1216KB). We observe that this approach performs significantly better than either the 64KB or the 4MB approach.

In the experiment above, we limit ourselves to picking the best request size (and wait time) to maximize throughput for a given acceptable slowdown, and have the scrubber send back-to-back scrub requests of that fixed size until a collision occurs. Based on our observations of decreasing hazard rates in Section V-A, one might think of doing even better by varying the request size over time, as the scrubber fires. When the wait time is just over and the scrubber starts sending requests, the probability of a foreground request arriving during that scrub request is much higher than later, when the scrubber has already been firing requests for a while (due to the decreasing hazard rates, the probability of collision decreases as the system remains idle – recall Fig. 11, 12). One could, therefore, start with smaller request sizes at the beginning of a scrub interval (when chances of a foreground request arriving are still high) and then gradually increase the request size over time.

We have experimented with three adaptive approaches, all of which wait for some time t before firing scrub requests of a start size s; this size is then increased with time. The *exponential* approach multiplies the request size by a factor a every time a scrub request is completed without a collision occurring. The *linear* approach uses the factor a calculated by the exponential approach and affixes a constant b to each increase, so each new request is multiplied by a and further increased by b. Using both, we can find the optimal rate of increase for our request size (we apply the exponential approach first, since it affects that rate more than the linear). We used simulations to find the constants a, b that provide the best trade-off between slowdown and scrub throughput for each trace. We have also experimented with a simpler *swapping* policy, that considers only two different request sizes: when the initial wait time t is over it starts firing with the optimal request size s that achieves the average given slowdown, and then at some later point t' it switches over to the maximum request size, whose service time does not exceed the maximum allowed slowdown.

The results for our adaptive approaches are given by the dashed lines in Fig. 15 (omitting *swapping*, for which we found $t'_{opt} = \infty$). Surprisingly, we notice that none of these adaptive approaches outperforms the fixed approach where one optimal request size is picked for a given slowdown goal. Through detailed statistical analysis, we managed to identify the reason behind the poor performance of the adaptive approaches. The technique of increasing the request size only works in the presence of strongly decreasing hazard rates, i.e. over time the instantaneous probability of collision decreases. While we found this to be the case for idle time distributions in their entirety, we also found that upon "cutting off" the initial wait time, the resulting truncated distribution shows a much weaker decrease in hazard rates. In other words, the long intervals captured by the Waiting approach, are far longer than the time it takes the slowest of our adaptive approaches to reach the maximum allowed request size. Since this size will be larger than the optimal and will be reached on every captured interval, each collision will incur more slowdown than it would with the optimal size. As a result, the extra throughput comes at a cost of extra slowdown, and vice versa: when the predefined slowdown goal is considered, the corresponding throughput is lower than that for the optimal fixed approach[5].

D. Putting it all together

In summary, we have made some interesting observations for optimizing the background scheduling of scrubbing requests in I/O schedulers. First, we found that a simple approach based on waiting outperforms more complex ones based on auto-regression. Second, we found that picking one fixed request size for the scrubber, rather than adapting it within an idle interval, is sufficient. This allows for a simple scrub policy with only two tunable parameters: the *scrub request size* and the *wait time threshold*. We further found that for a given slowdown target these two parameters can be determined relatively easily, based on a short I/O trace capturing the workload's periodicity, and simulations guided by binary search. The simulations can be repeated to adapt the parameter values if the workload changes substantially.

Disk	Avg. Sldn	Throughput	Threshold	Req. Size
HPc6t8d0 (Waiting)	1.0 ms	38.75 MB/s	205 ms	1280KB
	2.0 ms	50.53 MB/s	88.1 ms	1536KB
	4.0 ms	61.51 MB/s	32.4 ms	2048KB
CFQ	938 ms	6.25 MB/s	10 ms	64KB
HPc6t5d1 (Waiting)	1.0 ms	68.12 MB/s	632 ms	3072KB
	2.0 ms	72.83 MB/s	340 ms	4096KB[6]
	4.0 ms	73.97 MB/s	247 ms	4096KB[6]
CFQ	4.4 ms	13.75 MB/s	10 ms	64KB
MSRsrc11 (Waiting)	1.0 ms	73.54 MB/s	22.3 ms	3072KB
	2.0 ms	75.77 MB/s	15.3 ms	4096KB[6]
	4.0 ms	76.40 MB/s	12.6 ms	4096KB[6]
CFQ	7.7 ms	13.19 MB/s	10 ms	64KB
MSRusr1 (Waiting)	1.0 ms	62.49 MB/s	10.8 ms	1472KB
	2.0 ms	71.24 MB/s	39.8 ms	3072KB
	4.0 ms	71.58 MB/s	18.5 ms	4096KB[6]
CFQ	106 ms	13.44 MB/s	10 ms	64KB

Table III
FIXED WAITING APPROACH RESULTS FOR DIFFERENT DISK TRACES

Table III summarizes the throughput a scrubber can accomplish for four of our traces and three *average slowdown* goals of one, two and four $msec$, respectively. The table

[5]This is why the adaptive and 4MB *Fixed* approaches overlap in Fig.15.

[6]The maximum slowdown allowed ($50.4ms$) limits the request size at 4MB. If that restriction is relaxed, this run can achieve higher throughput.

also provides the wait time threshold and the request size that was used to achieve those throughputs. To put these results into perspective, we also include numbers for CFQ, albeit for 64KB requests. We find that our scrubber achieves significantly less slowdown (up to 3 orders of magnitude for busier traces) for up to 64x larger requests, yielding significantly more throughput per ms of slowdown. CFQ comes (somewhat) close to our approach only when its fixed threshold ($10ms$) happens to align with the workload.

VI. Conclusions and future work

With this work, we have taken a broad look at ways to issue background scrub requests in storage systems, in order to maximize the rate at which the system is being scrubbed. At the same time, we limit the impact on foreground applications running on the system to a predefined threshold. We have also developed an experimental framework within the Linux kernel, which can be used to implement scrubbing algorithms in only tens of LoC and we have made its source code publicly available[1]. Using our framework, we performed the first experimental comparison of sequential and staggered scrubbing. While sequential scrubbing is the approach that is currently used in production systems, we find that staggered scrubbing implemented with the right parameters can achieve the same (or better) scrub throughput as a sequential scrubber, without additional penalty to foreground applications. In addition, we presented a detailed statistical analysis of publicly available I/O traces, and used the results to define policies for deciding when to issue scrub requests, while keeping foreground request slowdown at a user-defined threshold. We find that the simplest approach, based on idle waiting and using a fixed scrub request size outperforms more complex statistical approaches and approaches using variable request sizes.

While we have focused on issuing scrub requests, we believe that our approach and observations can be applied for other uses of idle time. Examples include: contributing to power savings in data centers (e.g. by spinning disks down), guaranteeing availability (e.g. checkpointing, backups), performance (e.g. prefetching), reliability (e.g. lazy parity updates), or profit in the cloud by encouraging sharing a disk among more users while retaining QoS.

Acknowledgments

We thank Sotirios Damouras for his help on matters of statistics, Ari Juels and Alma Riska for insightful conversations on scrubbing, Vladislav Sekulic for assisting us patiently as systems administrator of our group, Austin Donnelly from Microsoft Research for clarifications on the MSR trace, and Jens Axboe for addressing our questions on CFQ. We also thank the four anonymous reviewers for their kind reviews and their suggestions on improving the presentation of the paper. This work was funded by EMC, an NSERC Discovery grant and a NetApp Faculty Fellowship.

References

[1] M. Hilbert and P. López, "The world's technological capacity to store, communicate, compute information," *Science*, 2011.
[2] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler, "An analysis of latent sector errors in disk drives," in *Proc. of ACM SIGMETRICS*, 2007.
[3] S. Hetzler, "System impacts of storage trends: Hard errors and testability," *USENIX ;login:*, vol. 36, June 2011.
[4] A. Oprea and A. Juels, "A clean-slate look at disk scrubbing," in *Proc. of USENIX FAST*, 2010.
[5] E. Thereska, J. Schindler, J. Bucy, B. Salmon, C. R. Lumb, and G. R. Ganger, "A framework for building unobtrusive disk maintenance applications," in *Proc.of USENIX FAST*, 2004.
[6] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger, "Argon: performance insulation for shared storage servers," in *Proc. of USENIX FAST*, 2007.
[7] R. Golding, P. Bosch, C. Staelin, T. Sullivan, and J. Wilkes, "Idleness is not sloth," in *Proc of USENIX ATC*, 1995.
[8] N. Mi, A. Riska, X. Li, E. Smirni, and E. Riedel, "Restrained utilization of idleness for transparent scheduling of background tasks," in *Proc. ACM SIGMETRICS*, 2009.
[9] N. Mi, A. Riska, Q. Zhang, E. Smirni, and E. Riedel, "Efficient management of idleness in storage systems," *ACM Trans. Storage*, vol. 5, pp. 4:1–4:25, June 2009.
[10] B. Schroeder, S. Damouras, and P. Gill, "Understanding latent sector errors and how to protect against them," in *Proc. of USENIX FAST*, 2010.
[11] J.-F. Pâris, S. Thomas Schwarz, A. Amer, and D. Long, "Improving disk array reliability through expedited scrubbing," in *Proc. of IEEE NAS*, 2010.
[12] 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 *Proc. of IEEE MASCOTS*, 2004.
[13] Z. Dimitrijevic, R. Rangaswami, and E. Y. Chang, "Systems support for preemptive disk scheduling," *IEEE Trans. Comput.*, vol. 54, pp. 1314–1326, October 2005.
[14] C. R. Lumb, J. Schindler, and G. R. Ganger, "Freeblock scheduling outside of disk firmware," in *Proc. of FAST*, 2002.
[15] E. Bachmat and J. Schindler, "Analysis of methods for scheduling low priority disk drive tasks," in *Proc. of ACM SIGMETRICS*, 2002.
[16] D. Anderson, J. Dykes, and E. Riedel, "More Than an Interface—SCSI vs. ATA," in *Proc. of USENIX FAST*, 2003.
[17] J. Axboe, "Linux Block IO – present and future," in *Proc. of the Ottawa Linux Symposium*, July 2004.
[18] Storage Networking Industry Association, "I/O traces, tools, and analysis repository," `http://iotta.snia.org/`.
[19] A. Riska and E. Riedel, "Disk drive level workload characterization," in *Proc. of USENIX ATC*, 2006.
[20] ——, "Evaluation of disk-level workloads at different time-scales," in *Proc. of IISWC*, 2009.
[21] C. Ruemmler and J. Wilkes, "Unix disk access patterns," in *Proc. of USENIX Winter Technical Conference*, 1993.
[22] Akaike, H., "A new look at the statistical model identification," *IEEE Trans. on Automatic Control*, vol. 19, no. 6, 1974.
[23] N. Hautsch, *Modelling Irregularly Spaced Financial Data: Theory and Practice of Dynamic Duration Models*, ser. *Lecture Notes in Econ. and Mathem. Systems*. Springer, 2004.
[24] Engle, Robert F. and Russell, Jeffrey R., "Autoregressive Conditional Duration: A New Model for Irregularly Spaced Transaction Data," *Econometrica*, vol. 66, no. 5, Sept. 1998.
[25] J. Maindonald and J. Braun, *Data Analysis and Graphics Using R*, 3rd ed. Cambridge Univ. Press, 2010.