Hiding the overhead cost of Segment Cleaning in a Log Structured File System.

Christoph Bregler Srini Narayanan Sridhar Rajagopalan

Abstract

We study the operation of the segment cleaner in a log structured file system. When the segment cleaner is active, response time for user read requests is adversely affected. Thus, it is desirable to run the segment cleaner when the read traffic is low. We propose a simple strategy for scheduling the segment cleaner that seems to perform an order of magnitude better than a random baseline policy. We predict that implementation of our policy on the existing LFS would improve the degree of user satisfaction with the file system.

1 Introduction.

A recent innovation in file system technology was the notion of a log structured file system [RO92]. It was argued that such file systems would capitalize more effectively on the improvement in hardware technologies both in the near and not-so-near future. The basis for such an argument was two-fold: first, that memory was getting both cheaper and more plentiful, and second, that the relative speed of the CPU was increasing with respect to Disk.

The current trends in hardware technology seem to provide stronger evidence about the validity of this claim. And even though the suitability of LFS has been challenged for certain specific types of work loads, LFS remains one of the most attractive options for situations that arise in more typical shared computing environments. Thus, given the viability of LFS as a widely used file system in the near future, some effort should be invested in fine-tuning its performance. This paper presents an effort in this direction.

In an LFS, almost all data[1] is stored in a single sequential log indexed by time. Thus conceptually, the log is an infinite object. Storage devices are however, capable of storing only finite amounts of data. In an LFS, the translation between the ideal and reality is done by a *segment cleaner* process. This process periodically compresses the contents of the log and reclaims the space allocated to outdated and deleted versions of files.

The performance of an LFS depends critically on having enough free space available to support all of the write traffic. In the current implementation of LFS, segment cleaning is initiated whenever the available free space falls below a threshold value L and the cleaner continues to operate while the available free space is not more than another threshold value T. An (expected) consequence of such a policy is that that the segment cleaner tends to run at a time when the load on the file

[1]The only exceptions to this are the "super block" and the "checkpoint region". These structures are used to provide random access to data on disk.

system is large. This places an additional burden on the file system at a time when resources are already scarce. For this reason, we feel that even a random cleaning policy[2] might be superior to the one currently implemented. This motivates the following question, first asked in [RO92]: Is it possible to 'hide' the segment cleaner from the user? We mean this in the following sense: Is there a way to do segment cleaning so that the user is unable to distinguish between the situation wherein the segment cleaner is in operation and the ideal situation wherein the log is an infinite object and there is no segment cleaner competing for I/O bandwidth? In this study, we present a simple strategy for initiating the segment cleaner that tends to cause much less disruption in file system performance than the random cleaning policy. Thus, we provide all our evidence based on the assumption that a random cleaning policy is a pessimistic estimate of the real situation.

We found that in a typical file system the request sequence is very bursty. This implies that there are periods of high activity interspersed with relatively long (often more than 5 secs.) idle periods. This observation means that, at least in principle, it is possible to schedule the segment cleaner in such a way as to hide its activity from the users.

Further, recent history seems to be a good indicator of future activity. Here, we mean that if the system has been idle in the immediate past, then it is likely to persist in such a state in the immediate future. This provides a means by which one could schedule the segment cleaner. We evaluated such a policy for scheduling the segment cleaner. The measure we used in making this comparison was the average latency per read request due to segment cleaner operation. Our results are extremely encouraging and show that initiating the segment cleaner based on this policy reduces the average latency by an order of magnitude with respect to a random cleaning policy. This provides an affirmative answer to the motivating question to a substantial extent.

The ideas that motivate this paper have greater applicability than the specific issue that we address. For instance, a similar reasoning and study could justify powering off the disk in a lap top computer when the file system has been idle for some time. The annoyance that this would cause to the user who would have to wait for the disk to power up again when it is required would be offset by the power that is saved by such a scheme. Another application of the same principle would be flushing the write queue in LFS when the disk is idle.

The organization of the rest of the paper is as follows. The next few sections describe the experiment itself and then we present a detailed version of the results and some analysis of the same.

2 Trace Data

Our study is based upon the trace data of file system activity collected on a heterogeneous cluster of about 40 workstations [BHK+91]. The cluster consisted of four file servers and 40 disk less SUN and DEC workstations connected by a single network running Sprite. The data was collected at the client-server interface by instrumenting the server kernel. We assume that this data is representative of a typical workload in a typical environment.

The traces were split into 5 segments: with three of the segments covering 48 hour periods and two covering 24 hour periods. Since the data was collected on each of the four servers, there

[2] A random cleaning policy is one which initiates the segment cleaner at a randomly chosen time.

were a total of 20 traces. Each trace record contains a relative timestamp (in microseconds), and information detailing the type of request made to the server (read, write, open, etc.) and relevant data detailing each request (inode, size, userid, etc.). Thus, the traces contain sufficient data to enact a short-term simulation of a file server cache. The trace data is equipped with a library of procedures that help in accessing the data. This library has a C language interface.

3 The Method.

The data was collected at the client-server interface and thus does not reflect the data traffic to disk. In order to get a first order estimate of the traffic to disk, we implemented an extremely simple cache simulation. The request stream at the back end of the cache was then used to evaluate a number of strategies for scheduling the segment cleaner. We now detail the principle aspects of the cache simulation and the scheduling strategies.

3.1 The Cache Simulator.

The cache simulator is based on three simplifying principles. These are:

- Whole file caching.
- FIFO replacement.
- Write traffic is ignored.

The simulator does whole file caching. This is probably a pessimistic estimate of reality. A block based caching policy should deliver a better hit rate than whole file caching. Additionally, the fringe benefits of whole file caching is probably equaled by those attained by a read ahead policy.

The simulator does FIFO replacement. This too is probably a pessimistic estimate of reality: a more intelligent replacement algorithm such as LRU would probably result in a better hit rate.

Finally, the write traffic in LFS is optimized. Therefore, the disk activity due to write traffic is extremely bursty and rare and is unlikely to interfere with the segment cleaning. Thus, we choose to ignore write traffic. This assumption might result in a more optimistic outlook, but we feel that due to the reasons cited above, it is justifiable.

Given these assumptions, the cache simulation reduces to a simple data structure question. We chose to implement the cache as a hash table indexed by FileID. The data structure also maintains a queue to implement the replacement policy. A read request results in the fileID being entered into the hash table if it is not already present. At this juncture, we also generate disk traffic on the other end of the cache. The data structure may at this stage evict some fileIDs from the hash table to enforce the cache size constraint. Write requests result in a cache entry but not in disk traffic.

3.2 Segment cleaning policies.

The LFS segment cleaner is a fairly light-weight process. It generates disk traffic that occupies the disk for time in the order of a second. We evaluated two policies for segment cleaning, (in addition to the baseline 'random' policy), which we call the $1N/5D-1$ and the $5-1$ policy respectively. The first figure represents the number of seconds of idle time one has to see before initiating the segment cleaner and the second figure the amount of time the segment cleaner takes to run. $1N/5D$ means that we look for a 1 second idle interval at night and a 5 second idle interval during day. In both cases, we run the segment cleaner for 1 second at a time. The figures were chosen after a careful analysis of the characteristics of some of the data. The policies were tested on all the data sets. A more detailed discussion on what the reasons for choosing these policies were appears in section 5.

4 Idle intervals and the effect of the Cache.

All following statistics are based on the output of our cache simulator. Figure 1 illustrates the difference in idle characteristics before the cache and after the cache. The cache size was set to 20MB. 20MB is a very conservative assumption: modern file servers have up to 128MB of main memory. The interesting observation here is that whereas the raw data had no idle intervals of

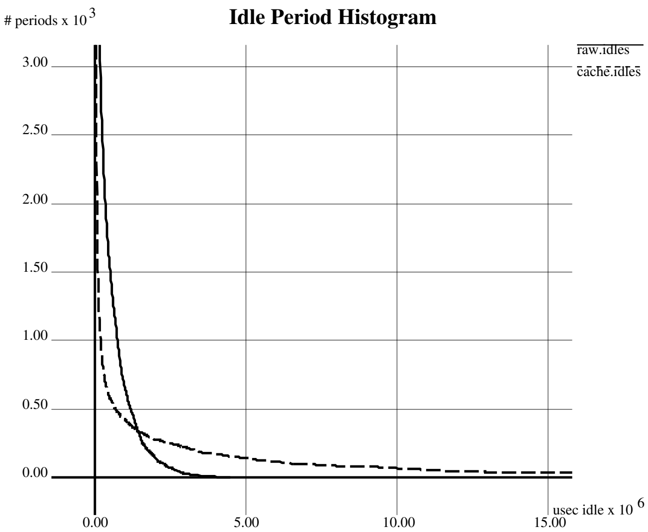

Figure 1: The figure represents the cumulative distribution of idle intervals over a moderately loaded one hour time interval on the heavily loaded *Allspice* file server. The solid line is the distribution of raw requests and the dashed one the requests that missed the cache. Thus, there were roughly 130 intervals of 5 or more seconds of idle time in the request stream that missed the cache.

size larger than 4 seconds, the after cache stream had significantly larger idle intervals (some larger than 15 secs).

The distribution of idle intervals was also related to the time of day. Figure 2 shows two histograms, one collected at 5AM and the other at noon. These indicate that the disk traffic is far more bursty and sparse at 5AM than at Noon. Further, the slopes of these curves are indicative of the fact that idle states are more likely to persist at 5AM than at Noon.

5 Selection Of a Cleaning Policy

Based on the histogram data we computed for each 1 hour period the conditional probability, that the server disk will be idle for C seconds, given it was already idle for W seconds $p(C|W)$. This probability estimates the chance that you can run the segment cleaner for C seconds without blocking any un-cached read-request, after you already discovered W seconds of disk idle. Figure 3 and figure 4 show the impact of the time of day on the conditional probability of 1 second idle intervals conditioned over 1 second and 5 seconds histories of idle periods respectively. The figure shows this conditional probability distribution for two example server disks. The conditional probabilities shown are averaged over all available traces for the specific server. Each data point in the graph is the average conditional probability value for different hours of the day, starting at midnight.

The time of day has a significant impact on the probability that the segment cleaner would run unnoticed if it were initiated after waiting for a 1 second idle period. Between the hours of 1 and 9 AM (ie. at night) we obtain an average probability value of .8 (80%). During other hours, this chance changes quite drastically. For instance, between the hours of 5 P.M. and 12 P.M., the probability that no request is made while the segment cleaner is active falls down to 62%. Indeed, more than 1/3 of the time, a server disk that is idle for 1 second, will not remain idle for another second. Thus, we conclude:

- If we wait for a one second idle period and then initiate the segment cleaner for one second at night, there is a high likelihood that the segment cleaner works uninterrupted. If we do the same during the day, the likelihood is not so high anymore that the cleaner works uninterrupted.

This suggests a simple policy for scheduling the segment cleaner. Initiate the segment cleaner only at night. Wait for a one second idle period before running the segment cleaner. Such a policy would work well if cleaning at night alone would suffice to meet the segment cleaning demands for the rest of the day. While some of the server disks (*Anise* for example) support relatively less traffic and therefore have a large number of long idle intervals at night, other server disks (*Allspice* for example) have a significantly smaller number of long idle intervals and may not be able to meet their cleaning requirements this way. The same policy would impose a significant performance penalty during the day. During the day, it is relatively unlikely that a 1 second idle interval will persist for another second. So while the policy enunciated above would work in many cases, it depends too much on the assumption of being able to complete all the cleaning at night. For heavily loaded servers, this is an unreasonable assumption.

Consider now figure 5. This shows that longer idle intervals are consistently fairly likely to persist for a few seconds more. Specifically, notice that waiting for 5 seconds before initiating cleaning seems to provide a fairly consistent guarantee of cleaning uninterrupted. This hypothesis was validated by experiment. We noticed significant improvements in the probability of an additional second of idle time when we had a 5 second idle time. For example, the expected chance of finding an additional 1 second idle interval improved to around 90% even during heavy load periods. Clearly, this indicates that waiting for 5 second intervals before initiating cleaning is a good idea.

There is a tradeoff between waiting longer (say 5 seconds or more) before initiating cleaning and thereby having better odds of cleaning segments uninterrupted, and throwing away valuable idle time by waiting longer. The correct answer to this problem depends on the resolving this tradeoff to match the cleaning requirement of the disk. This introduces a voodoo constant into the picture and we do not immediately see a simple way to avoid this.

5.1 Summary

Our observations on the disk traffic patterns provided us with a policy selection rationale. Before proceeding to describe the policies, we summarize our observations.

- Disk traffic patterns are significantly different depending on the time of day (at least at the granularity: night versus day).
- Disk traffic distribution patterns are different on different servers. This is not surprising, as it is a normal result of different workloads on different servers. However, it is relevant in the selection of a suitable cleaning policy.
- The recent history of idle intervals gives valuable information regarding the expected the future length of idle time. This information can hopefully be effectively used to initiate and control the operation of a segment cleaner.

6 Policies.

Based on the observations discussed in the last section, we identified the following policies as candidates for further analysis. In all the simulations we chose a cache size of 20MB and a cleaning capacity of 750 segments. The cleaning capacity is the sum of the number of dirty segments read into memory and clean ones written back to disk by the segment cleaner in a 24 hour period. This figure (750) was arrived at by examining the log maintained by the segment cleaner on *Allspice*.

- The $1N/5D-1$ policy: The segment cleaner waits for a 1 second idle interval at night and a 5 second idle interval during the day. In each case it runs the segment cleaner for one second. If at the end of the second there is no pending request the cleaner is reactivated for an extra second. This continues until the required number of segments are cleaned.
- The $5-1$ policy: The segment cleaner is initiated after a 5 second idle interval at all times. Otherwise it is identical to the first policy.
- The random policy: The segment cleaner is initiated at random times throughout the day.

7 Evaluation Technique

7.1 What Should We Look For?

Segment cleaning affects LFS in two main ways.

- First the cleaner generates additional *write* traffic to disk. This affects the efficiency with which the file system uses the disk to write data. The measurements done by Rosenbloom [RO92] suggests that on average the amount of write traffic generated by cleaning is fairly small (on the order of $3 - 8\%$, and scheduling policies such as ours are unlikely to affect this efficiency.

- The segment cleaner increases the latency for read requests while in operation. This has two main impacts. It increases the queue length of read requests going to disk, and also affects read response time at the user end. We expected the main benefit of intelligent scheduling of the segment cleaner to come from decreased latencies for read requests. Our analysis process was therefore tuned to evaluate this hypothesis.

Using our cache simulator and the trace data, we simulated the cleaning policies outlined earlier. Our evaluation function looked at the latencies on read requests. In defining the evaluation functions, we wanted to estimate the following parameters for each of our policies.

- The expected latency for read requests that go to disk (at the back end of the cache) and conflict with cleaning intervals divided by the total number of disk requests. This gives us an idea of the change in queuing times for disk traffic with and without cleaning for the different policies.

- The latency generated by read requests that go to disk and conflict with the segment cleaner operation divided by all requests (before cache). This gives us an idea on the impact of different policies on the response time to user level requests.

- The ratio of read requests that conflict with some cleaning interval compared to the total number of requests to disk. This gives us an idea of the number of conflicting requests going to disk.

- The ratio of read requests that conflict with some cleaning interval compared to the total number of requests (before cache). This gives us an idea from the user's perspective of the total amount of read traffic that conflicts some cleaning interval.

7.2 The Analysis Process

In order to perform the analysis outlined earlier, we simulated the cache and our policies on the trace data. Our simulation initiated the cleaner at the time and for a duration specified by the various policies. During the simulation we kept track of the total number of requests (both cached and un-cached), as well as the read requests that went to disk during any cleaning interval. For those read requests that went to disk during any cleaning interval we also kept track of the time that a particular request came in. This information was used to compute the increased latency due to conflicts.

Our simulator also allowed us to vary several parameters. We could vary the cache size as well as cleaning capacity. Durations for which the cleaner operated, as well as the rate at which the cleaner generated clean segments could also be varied easily. Results described in this report will assume a cache size to 20 megabytes, a cleaning capacity of 750 segments per day (includes segments read + segments written) and a cleaning rate of 1 segment/second. The cleaning capacity number was chosen from recent data from the segment cleaning log on the *Allspice* server. In general, fixing these parameters helped us create a baseline for policy comparison.

Our simulations were performed using data from all traces. The simulations were also separated by server disks. In some cases due to lack of trace data, we had no information about activity in specific servers. In such cases, our numbers are averaged our the traces that did have data for these disks.

To estimate the latencies incurred during cleaning, we measured the expected latency per read request using the following formula

The expected latency/read request =

$\sum_i (T_{ej} - T_{ij}/N)$

- $i's$ are read requests
- $j's$ are cleaning intervals
- N = Total Number of requests.
- T_{ej} = Time at the end of the cleaning interval j.
- T_{ij} = Time at which the i_{th} read request arrives during the j_{th} cleaning interval.

Basically, we count the interval between the time of arrival to the end of the cleaning interval for any conflicting read request. The denominator N can either be the total number of requests at the front end of the cache, in which case it counts the number of user level requests (cached or un-cached), or it could be the total number of requests at the back end of the cache, in which case it counts only the disk traffic.

We also count the number of requests to read requests to disk in during any cleaning interval j. N_j is the number of read requests in interval j. $\sum_j N_j$ is then the total number of conflicting requests. This allows us to compute the ratio of conflicting requests to total number of requests both at the user level and to disk.

8 Results

Tables 1 to 3 pertain to the latency of simulating our policies for different servers. Tables 1a, 2a, and 3a show the latencies divided by the total number of requests that go to disk. Tables 1b, 2b, and 3b show the expected latency results divided by the total number of client requests. The two sets of results differ only in that the denominator of the latency expression N as explained earlier.

In computing the results in Tables 1 through 3, we note the following.

1. All numbers in both tables are in milliseconds/request.

2. The maximum (minimum) values are computed as follows. First the maximum (minimum) latency values for any cleaning interval are collected per server per trace. These values are then averaged over all traces. The values shown in the table for any server are the average values of of the maximum (minimum) cleaning latencies over all traces for that server.

3. The average values shown in both tables refer to the average expected latency per request again averaged over all traces (or as many as there was data for).

The latency results in Tables 1 through 3 enable several observations. Table 1 pertains to data collected for the *Allspice* server. Tables 2 and 3 refer to data collected for the servers *Assault* and *Anise* respectively. The following inferences can be drawn from our results

- We note that in every case, there are idle intervals that span six seconds or more. This is true for all traces and for all policies. This can be seen in the fact that the minimum cleaning latency number always has a 0 entry for the $5-1$ policy.

- There is a significant variance in the penalties for conflicts at different intervals. We see this in the difference between the average and the maximum and minimum values for all servers. This is only to be expected due to the burstiness of the file traffic.

- The values are significantly different for different servers. This is also expected since load distribution across servers is not balanced. *Allspice* which has the highest amount of traffic seems to be the greatest beneficiary of any intelligent scheduling policy. However, we note that the trend suggesting the relative performance of the policies does not change with server.

- Our results show that either of the intelligent scheduling policies perform significantly better than the random policy. This occurs both in the average expected latency as well as in the maximum latency. Note in particular the value for the *Allspice* server, which shows an order of magnitude change in the average expected value, and more than an order of magnitude change in the maximum value. This shows that policies that initiate cleaning during high traffic are significantly worse than those that use recent idle history information.

- The results for total user level requests suggests that the overhead for average cleaning latencies may not be very high as far as the user is concerned. One reason for this may be the specific assumptions about the duration of cleaning being limited to 1 second. Increased cleaning intervals will impact this number adversely. Another reason might be that none of our policies were based on the state of the log on disk. For a policy that was solely based on the number of free segments on disk, we expect that the average latency during cleaning would be greater, since the load is also likely to be high at that time.

- For the specific parameters we chose (cache size = 20 MB., cleaning capacity = 750 segments, cleaning rate 1seg/sec.), the wait five seconds then clean for one second policy ($5-1$ policy) uniformly performed better. We suspect that this is due to the good availability of long idle intervals at night. We believe that this policy may have to be reevaluated under different workloads to be shown as demonstrably better in all circumstances.

Table 4 shows the results of the ratio of conflicting requests to total number of requests both at the front and at the back of the cache for different servers. Again all numbers are averaged over all traces. The percentage of conflicts at the user level seem to suggest that the cleaning overhead will not be felt at this level. However, as expected from the other results, we notice a significant decrease in the number of conflicts for the more intelligent policies.

8.1 Impact Of Larger Cleaning Intervals

One issue the earlier policies did not test was the potential for conflict if the segment cleaning interval was changed from 1 second to 3 seconds, or 5 seconds. While we expected that latencies would increase, we also felt that the decrease in the total number of cleaner initiations would help skew our policies towards doing more of the cleaning at night.

Our initial results of this experiment are shown in table 5. This table shows the average latency values for the three policies with different cleaning intervals for the *Allspice* server. Each policy was simulated with the cleaner operating for 1 second (the result already seen) for 3 seconds, and for 5 seconds. The latency calculations for the 3 and 5 second cleaning intervals were done in the same manner as described for the 1 second interval. The numbers shown in the table represent the latencies and are measured in milliseconds/per-user request.

The results of this analysis show that extending cleaning intervals from 1 to 3 or 5 seconds varies in its impact based on the server. The general trend suggests however, that the longer you wait before you initiate cleaning, the more likely you are to get a large chunk of cleaning time without interruption. Also, the trends for the ranking of the three simulated policies remains largely unchanged. We noticed a comparatively sharp increase in latency with increase in cleaning interval from 1 to 5 seconds for *Allspice*. Clearly, in this case, cleaning overhead is beginning to imply noticeable increases in average read latency at the user level. If our initial results are a good indicator, we feel that more interrupt driven cleaning policies may be the only way to handle very heavily loaded disks.

Our initial results from other servers seems less dramatic but we have not completed all the analysis for other servers yet.

9 Discussion

The numbers in Tables 1 through 5 are simplistic in two ways.

- First, we expect that the latency numbers may be too pessimistic since in reality, the cleaner gets scheduled in a round-robin fashion, so when read requests come in during cleaning, they may have to wait less than the end of the current cleaning interval before being serviced. Our numbers would be correct only in cases where the cleaner had to work for the entire cleaning interval before other read requests could be serviced.

- On the other hand, the numbers may be overly optimistic since in computing the latencies, we assume that the read requests that go to disk during cleaning intervals are all serviced instantaneously at the end of that interval. In a realistic case, both read queue size restrictions, and dequeuing latencies at the end of the cleaning interval will tend to increase the total latency values considerably.

Despite these simplifying assumptions, the uniformity of the trends observed in our results suggests that in more realistic situations the actual benefits of the policies may vary in the magnitude, but the relative trends should hold true.

10 Summary

In this study we looked at the feasibility of improving the performance of a log structured file system by a more intelligent segment cleaner. In order to validate this study, one needs to implement the segment cleaning policy in a real system. Two other questions arise as immediate consequences of the observations that we make.

Is there a further benefit to be had by integrating the existing hot-cold policy and the segment cleaning policy that we propose? The hot-cold policy tells us which segment to clean and this policy tells us when to clean. It is entirely conceivable that an optimal strategy would clean only hot segments during the day time and the hot and cold segments at night. Thus, work done when there is a resource crunch would have a more immediate impact. Work done at night would have a long term impact.

It would also be interesting to study the other situations where it is possible to hide the overhead costs of housekeeping or where resources could be better utilized by invoking prediction mechanisms based on recent history.

References

[RO92] Mendel Rosenblum, John K. Ousterhout, The design and implementation of a Log-Structured File System. *ACM Trans. on Comp. Sys.* **10.1**, Feb, 1992. pp26-52.

[BHK+91] Baker, M.G., Hartman, J.H., Kupfer, M.D., Shirriff, K.W., Ousterhout, J.K. Measurements of a distributed file system. In *Proceedings of the 13th Symposium on Operating System Principles*, Pacific Grove, Calif. Oct 1991, ACM, pp198-212.

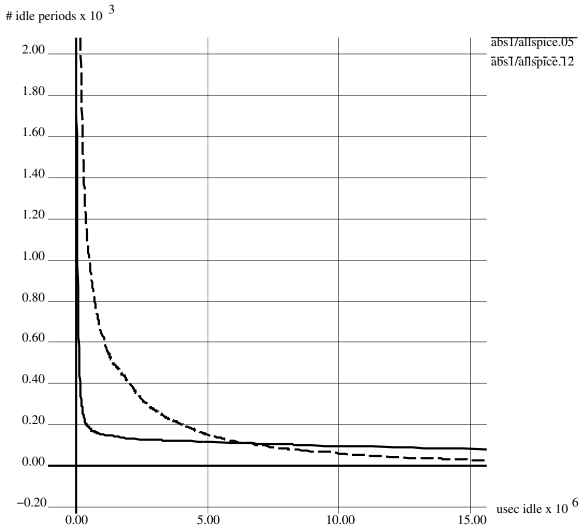

Figure 2: Idle period statistics at two different times of the day. The bold line shows the cumulative distribution at 5AM and the dashed one at Noon. The readings are taken on the *Allspice* file server and represent the stream at the back end of the cache.

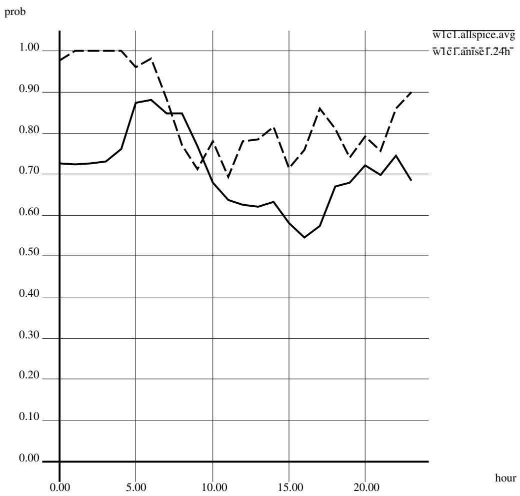

Figure 3: This figure shows the conditional probability that you can clean uninterrupted for 1 second if you start the segment cleaner after a 1 sec idle period. The measurements are made on two file servers. The bold line shows the characteristics on *Allspice* and the dashed line on *Anise*.

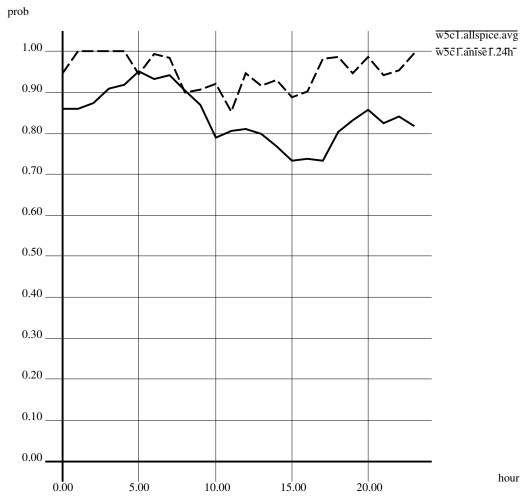

Figure 4: This figure shows the conditional probability that you can clean uninterrupted for 1 second if you start the segment cleaner after a 5 sec idle period. The measurements are made on two file servers. The bold line shows the characteristics on *Allspice* and the dashed line on *Anise*.

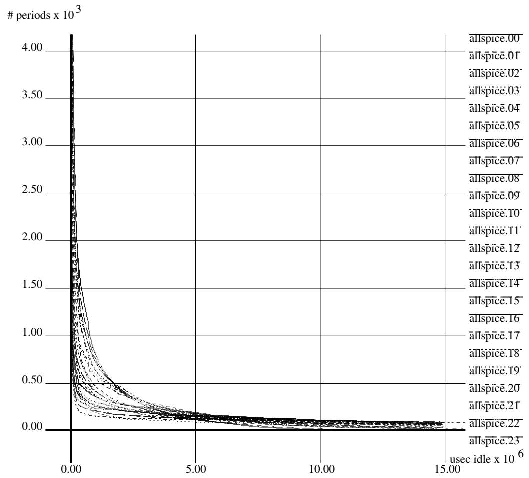

Figure 5: This figure shows the aggregate of idle histograms over a 24 hour period. The data used was the trace record on the *Allspice* file server. The interesting observation concerns the nexus that seems to occur at around 5 seconds.

Policy	1N/5D-1	5-1	Random
Min.	0	0	0
Average	11.5	8.6	56.74
Max.	21.34	12.1	763.30

a: before cache

Policy	1N/5D-1	5-1	Random
Min.	0	0	0
Average	2.67	1.98	13.09
Max.	4.91	2.77	175.12

b: after cache

Table 1: Latency/requests for Allspice

Policy	1N/5D-1	5-1	Random
Min.	0	0	0
Average	3.4	2.5	7.74
Max.	7.90	7.1	12.99

a: before cache

Policy	1N/5D-1	5-1	Random
Min.	0	0	0
Average	0.715	0.506	1.58
Max.	1.6	1.452	2.64

b: after cache

Table 2: Latency/requests for Assault

Policy	1N/5D-1	5-1	Random
Min.	0	0	0
Average	0.1	0.09	1.78
Max.	1.34	1.1	5.0

a: before cache

Policy	1N/5D-1	5-1	Random
Min.	0	0	0
Average	0.024	0.021	0.43
Max.	0.34	0.26	1.21

b: after cache

Table 3: Latency/requests for Anise

Server	Allspice	Anise	Assault
1N/5D-1	0.8%	0.1%	0.43 %
5-1	0.61 %	0.09 %	0.31 %
Random	3.03 %	1.77 %	1.11 %

a: before cache

Server	Allspice	Anise	Assault
1N/5D-1	.19%	0.24%	0.1%
5-1	0.14 %	0.025 %	0.07%
Random	.69 %	.42 %	.22 %

b: after cache

Table 4: Conflicts/requests for Different Policies

Cleaning Time	1 second	3 seconds	5 seconds
1N/5D-1	2.67	6.6	16.65
5-1	1.98	4.4	10.8
Random	13.09	29.42	67.54

Table 5: Impact of Cleaning Interval On Our Policies for Allspice