Automated Storage Reclamation Using Temporal Importance Annotations *

Surendar Chandra, Ashish Gehani and Xuwen Yu
University of Notre Dame, Notre Dame, IN 46556, USA
{surendar,agehani,xyu2}@nd.edu

Abstract

This work focuses on scenarios that require the storage of large amounts of data. Such systems require the ability to either continuously increase the storage space or reclaim space by deleting contents. Traditionally, storage systems relegated object reclamation to applications. In this work, content creators explicitly annotate the object using a temporal *importance* function. The storage system uses this information to evict less important objects. The challenge is to design importance functions that are simple and expressive. We describe a two step temporal importance function. We introduce the notion of storage importance density to quantify the importance levels for which the storage is *full*. Using extensive simulations and observations of a university wide lecture video capture and storage application, we show that our abstraction allows the users to express the amount of persistence for each individual object.

1 Motivation

Technology trends are making it easier for end users [7] to consume vast amounts of storage. For example, compressed media objects can consume storage at the rate of gigabytes per hour of contents. However, the available storage is not always able to keep pace with the rate of content creation. We note that not all stored data are equally important. Users store different kinds of files, including important financial information, source code that they wrote, public domain source code that they downloaded for installing software, object files from these source libraries as well as cached web and media objects. Traditionally, storage systems were unaware of this *importance* difference and considered all objects to be equally important. They provided services such as persistence uniformly across all the stored contents and required applications to explicitly remove any unwanted objects. However, this reliance on end user applications causes problems, especially for systems that are expected to operate for long durations. The overhead imposed by orphaned files increases as the system matures.

*Supported by the U.S. National Science Foundation (CNS-0447671)

We describe a university lecture repository application that stores all lectures to further motivate this work. In order to understand the magnitude of the storage requirements, we video captured lectures from one course [2]. The lectures consumed over 25 GB of storage in a single semester. Our university offered 2,321 such lectures; capturing all these lectures would require about 58 TB of storage per semester. This is much larger than the managed storage available for the entire university. These storage requirements would also be significantly larger for storing higher fidelity contents. The importance of a particular object depends on the specific lecture - lectures from the current semester are important; older lectures are likely less important than newer lectures; lectures of core courses are of interest for longer durations while lectures from summer courses might be less important.

In general, our autonomic storage management mechanism is applicable in scenarios where it is easier to create large objects and where it is only feasible to store these objects for shorter durations. Our solution will allow the storage to autonomically manage its resources. Specifically, we advocate exporting the object importance information to the storage, allowing the storage system to provide variable levels of service to different objects. Many objects become less important with time. Hence our notion of object importance includes a temporally decaying component. Stored objects can be associated with the temporal importance function in a number of different ways. One approach is to designate the importance of objects statically. For example, objects stored in */tmp* as well as JPEG objects can be designated as less important. Such policies are inherently inflexible because they do not take the specific user into consideration (e.g., photo journalist and web surfers have different use for JPEG images). Another approach is for the storage system to automatically predict the object importance by using parameters such as object type and access patterns (e.g., read/write/create/delete frequencies). However, this prediction is difficult without contextual information about the intended use. For example, the importance of a downloaded video depends on the likely availability of inexpensive network connectivity to replenish the video. Besides, any false identification of importance for persistence is catastrophic; it is possible for the only copy of an object to be reclaimed. Hence, we advocate explicit annotation by end users. Operations on multimedia objects are CPU intensive. Hence, we do not consider operations such as transcoding [3] that can reduce the storage footprint of less important objects.

We introduce the notion of object *temporal importance* as a first class attribute to describe the expected longevity and importance of an object. *Lifetime* is a monotonically decreasing importance function. *Importance* is a scalar metric; objects of higher *importance* can preempt objects of lower *importance*. Hence, the storage appears full for less *important* objects.

The research challenges are to design lifetime functions that are simple and useful. Similar to systems that require user annotation, we trust that the end user is the best person to design the temporal importance function for their application. We acknowledge that the end users are not prescient and may themselves only understand the future requirements vaguely. Regardless, it is possible that users may require complex importance functions. Importance functions that require complex dependencies are harder to implement inside the storage subsystem. For example, a requirement of *video objects captured on the road is important until the user can return home and successfully create a backup copy* is harder to implement at the storage level. Such a policy might require the video upload application to annotate the objects with a high importance and the backup

application to reduce the importance on a successful backup. The goal then is to explore simple temporal functions and analyze whether these can achieve realistic user goals. The other important challenge is on providing enough hints from the storage to the user in order to help them choose the right annotation and achieve their application goals. Without feedback, an importance of say 50% might result in the object being removed immediately. Such unpredictability might force the user to conservatively create objects that are annotated with an importance of 100% always, defeating the intention of the temporal importance function. Also, on a multi-user system, the system should restrict the importance functions for fairness, lest every user request infinite (∞) lifetime, essentially reverting to the traditional *persistent until deleted model*.

In this paper, we focus on designing the importance functions. We assume that the users are not malicious and want to fairly choose the best lifetimes. Still, many different users who use the same application might interfere and we study such behavior. We leave the study of simultaneous and different applications vying for storage to follow up work. We show that lifetimes provide the necessary information for the storage system to reclaim the object. We introduce the notion of storage importance density to quantify the importance levels for which this storage is *full* while still allowing storage at higher importance levels. We show that these importance boundaries are dynamic and depend on the current storage pressure. As more storage is added, the system is able to prolong less important objects.

2 Related work

Lately, there has been considerable interest in self-managing (autonomic) storage abstractions. Wilkes et. al. [13] argued that users should specify the storage need for a particular file using a *user-specified data property* and the system should then choose the appropriate underlying storage technology. Self management is one of our design goals; our objects are independent and manage their own lifetimes without relying on applications or services.

In their position paper, Douglis et. al. [4] introduced the notion of retention value of objects. New data were assigned a current retention value that described the initial *importance*. They also described another function which defined how this retention value decayed with time, eventually aging out of the system. They described several retention decay functions such as *infinite retention*, *unlimited retention*, *fixed and immutable expiration*, *fixed but mutable expiration*, *approximate expiration*, *fixed-priority expiration*, *variable-priority expiration* and *arbitrary priority expiration*. Infinite and unlimited retention policies represent traditional storage. Fixed and immutable/mutable expiration allowed for explicit deletion as well as automatic expiration. Approximate expiration defines an approximate interval for object expiration. Variable priority expiration is similar to our temporal importance function, though they envision more general decay functions. Variable priority expiration defines a priority among the objects that are to be deleted. In general, our notions of temporal functions represent a mechanism for objects to compete for storage while the system is operating under storage pressure. In our system, objects can be preempted even before expiration. Our work primarily focuses on developing simple and expressive temporal importance functions and designing mechanisms that can help application developers choose the right functions. Their work is primarily focused on improving the disk layout for deletion opera-

tions by grouping objects that expire together. We incorporate their ideas into our own attempts at developing a temporal lifetime function. Bhagwan et. al. [1] extend their work [4] and defined time varying specifications for storage system performance.

Palimpsest [9] is also closely related to our system. Palimpsest provides an abstraction where files are reclaimed based on the rate at which new files are created. The object creator monitors the various storage units to identify current reclamation rates (time constant) and continuously rejuvenate important objects. Unless the application can predict this rejuvenation duration accurately, objects might be irreparably lost. We show that these time constant values are harder to predict by solely analyzing the current object eviction rates.

The Elephant file system [11, 10] protects users by managing the storage reclamation. The file system provides a versioned view where the user can rollback in time by appending the time for the view. Elephant defines policies such as 'keep one', 'keep all', 'keep safe' and 'keep landmark'. An important difference between our work and Elephant is that unless users delete a file, the file in Elephant file system is not reclaimed. In our system, objects can be automatically reclaimed depending on the storage pressure and the importance requirements of incoming objects.

Modeling the lifetime behavior of objects in order to predict their future behavior is used for process scheduling. For example, process age was used [5] to decide when to preemptively migrate processes. The objects and programming languages community have looked at object lifetimes [12] in order to reduce the garbage collection overhead. Previous research efforts [8] have analyzed the lifetime behavior of web objects for proxy caching decisions.

3 Temporal Importance abstraction

The ability of users to express the object lifetimes and for the storage system to use them is fundamental to our design. Assume a storage of size N bytes. Objects stored in the system have attributes of size (s) and time of arrival (t_a). The entire storage would be fully used up when N equals $\sum s_j$. We introduce lifetimes as a mechanism that allows the producers to quantify the duration beyond which an object is not important and can leave the system. Lifetimes (L) are attached as a first class attribute of all objects; objects (O) are described by the tuple (s, t_a, L). However, objects need not be equally valuable during the entire lifetime duration, typically becoming less valuable over time. We express this aspect using the temporal lifetime function. Temporal lifetime is a monotonically decreasing importance function (L(t)). The current importance of an object describes the probability of eviction. Note that we require monotonically decreasing function because increasing the importance value cannot be represented using a static longevity function. Consider a lifetime requirement that *"linearly decreases in importance from the sixth to the tenth month. However, at the beginning of the ninth month, the object will be fully rejuventated for another six months"*. The probability of rejuvenating the object fully at the beginning of the ninth month depends on the conditional probability that the object escape eviction prior to the ninth month. We disallow such conditional probabilities and require an active intervention by the user to increase an existing importance in the future. The overall longevity duration of the object is t_{expire} (i.e., $L(t_{expire}) = 0$). Importance of an object is a scalar comparison metric in the range $[0..1]$. Objects that have higher importance at a given time instant can preempt an object with lower importance.

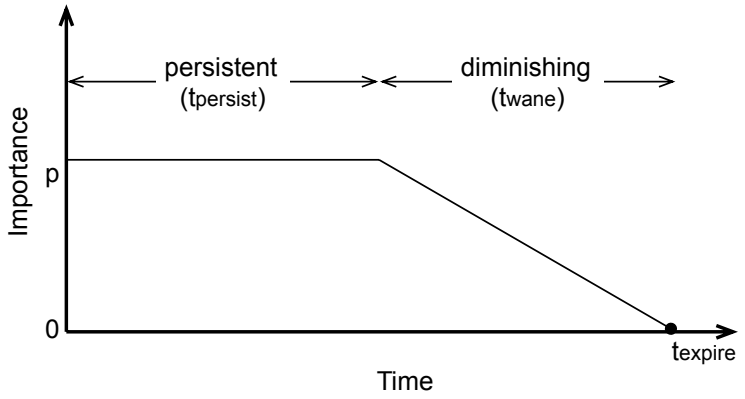


Figure 1: Two piece temporal importance

Thus, objects of importance *one* are not preemptible, while objects of importance *zero* may be freely replaced by any other object. Objects need not be deleted at the end of t_{expire}; rather, the system makes no guarantees on object availability after this duration. If there was no storage pressure, the object may exist in storage beyond this interval.

3.1 Nature of temporal importance

For each object, assume that users intuitively understand the importance and their expected degradation. On the other hand, we do not assume that the user can comprehend the interplay of their requirements and future storage pressure. On a hypothetical one petabyte storage, a user who stores one TB need not know precisely when their objects will be reclaimed. We investigate the long term behavior of user choices which are affected by competition from other users. Some reasonable lifetime functions are:

- *no object expiration* - The lifetime expressed as ($L(t) = 1, t_{expire} = \infty$) describes traditional persistent storage. Though not our focus, a majority of applications will continue to require this level of management. Our hypothesis is that this lifetime is not necessary for some objects.

- *Palimpsest or cache degradation* - Systems like Palimpsest [9] treat all data as ephemeral, implementing a FIFO policy. Applications renew objects using explicit *refresh*. Similarly, web caches retain objects until the expiration times specified by the web servers. Ignoring the server specified intervals, the lifetime functions used by these systems are of the form ($L(t) = \delta, t_{expire} = 0$), where δ is the Dirac Delta function. Incorporating the web server provided lifetimes would require us to relax our requirement that only objects of higher importance can preempt another object (web caches are allowed to discard any objects, whether they have expired or not. This requires that all objects have an *importance* of 0, even though non-expired objects have some *importance*).

- *no temporal degradation* - i.e., ($L(t) = 1$). This policy is similar to the *fixed-priority expiration* policy described by Douglis et. al. [4]. These policies differ from storage retention requirements imposed by the Sarbanes-Oxley Act in the USA; organizations following the law may want the objects to be completely deleted after the required retention interval.

- *two step function* - We express the lifetimes as a two piece function (Figure 1), wherein objects have a constant importance (p) for a duration $t_{persist}$ and have linearly diminishing importance for another duration t_{wane}.

  $$L(t) = \begin{cases} p, t \leq t_{persist} \\ f(t), t_{persist} < t < t_{expire} \end{cases}$$

  This function captures the policy *"objects exhibit a constant importance for some time ($t_{persist}$) and diminishing importance for some more time $t_{expire} - t_{persist} = t_{wane}$"*. The diminishing component could be linear, exponential or some other function. For simplicity, we chose a linear function. The two step function is flexible and can represent the no temporal degradation policy if ($t_{expire} = t_c$). It can also represent the cache like degradation if ($t_{expire} = 0$).

- *general function* - The temporal importance could be a complex and monotonically decreasing function.

4 Goals and System Architecture

Next, we outline our target application, experimental goals, simulator and the performance metrics. We present the results of our analysis in Section 5.

4.1 Lecture video capture using Besteffs

The target application video captures all the lectures in a university setting. The interesting aspects of this application scenario are: a) it requires storing large amounts of contents for potentially long durations, b) universities do not have the resources to store the amounts of data generated. The storage requirement for a single semester is far greater than the managed storage available to the entire campus. On the other hand, the campus has more than enough unmanaged storage in desktops and c) the importance of stored objects diminish with time. This observation need not always be true; a new Nobel laureate might see all her past lectures suddenly become important.

Storage for our system will be provided by *Besteffs*. *Besteffs* is an object level, fully distributed storage. Objects are read-only and write once with versioned updates. *Besteffs* uses unused desktop storage as well as in dedicated storage bricks. The system is fully distributed with no centralized components; authentication, authorization and fair resource allocation are implemented in a completely distributed fashion. *Besteffs* is designed to scale to tens of thousands of storage units. Objects are not replicated. *Besteffs* does not provide any more reliability guarantees than the reliability provided for a single copy of an object in the underlying storage.

Rather than solving the problem of designing a general lifetime function, we focus our attention on this target application. Since the lifetime requirements of this system is complex, we first explore temporal abstractions for a simple scenario where all the objects use the same lifetime function. We then explore *Besteffs* for a single instructor using a single storage unit before analyzing the behavior for a fully distributed scenario. Our results show that simple two step lifetime functions can capture the user's intent for these applications. We leave the general problem of choosing a temporal lifetime functions for different applications using the same storage as a future exercise.

4.2 System Objectives

Our experiments are designed to answer questions like:

- *Simple and expressive:* How robust are the representations with respect to the desired application behavior?
- *Predictive:* Given a stream of objects entering the storage system with varying lifetime parameters, can we quantify the kinds of objects that will be reclaimed as well as retained?
- *Scalability:* Can the system behavior scale with the availability of more storage? We prefer object annotations that remain constant while the specific system behavior depended on the available storage.

4.3 Simulation parameters

While we are building our distributed storage for experimental validation, analyzing a system that is expected to operate for a long duration is tedious. We use a simulator to analyze the system behavior over a large time frame (five and ten years for our experiments) on a minute granularity.

4.4 Performance metrics

We use the average storage importance density and the actual lifetimes achieved (measured when objects are evicted) in order to understand the behavior of our storage. Storage importance density can assist the content creators in choosing the right importance function for the current storage.

5 Results

We seek to understand how the storage can be used to accept objects well beyond the point where it appears full. First we describe a simple setup with one storage unit for a single application in order to understand the behavior of a storage system that supports our temporal importance function. Next, we describe a setup to store videos captured for a single lecturer. This scenario requires lifetime functions that depend on the semester and the day within the semester. The parameters for this scenario were derived from an actual video capture by the author in Spring '06 term. Finally,

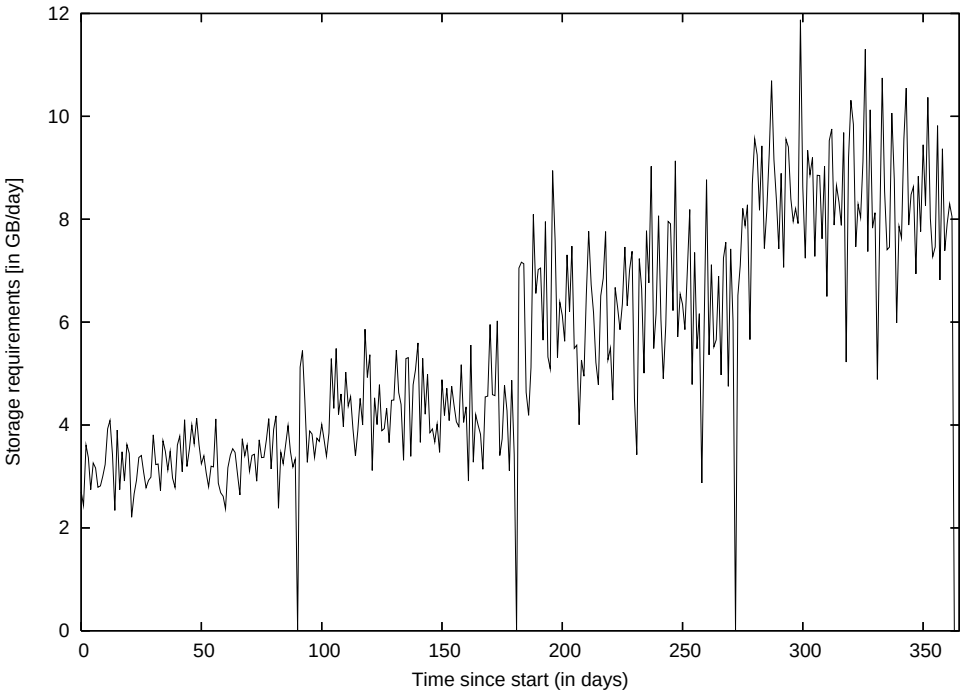
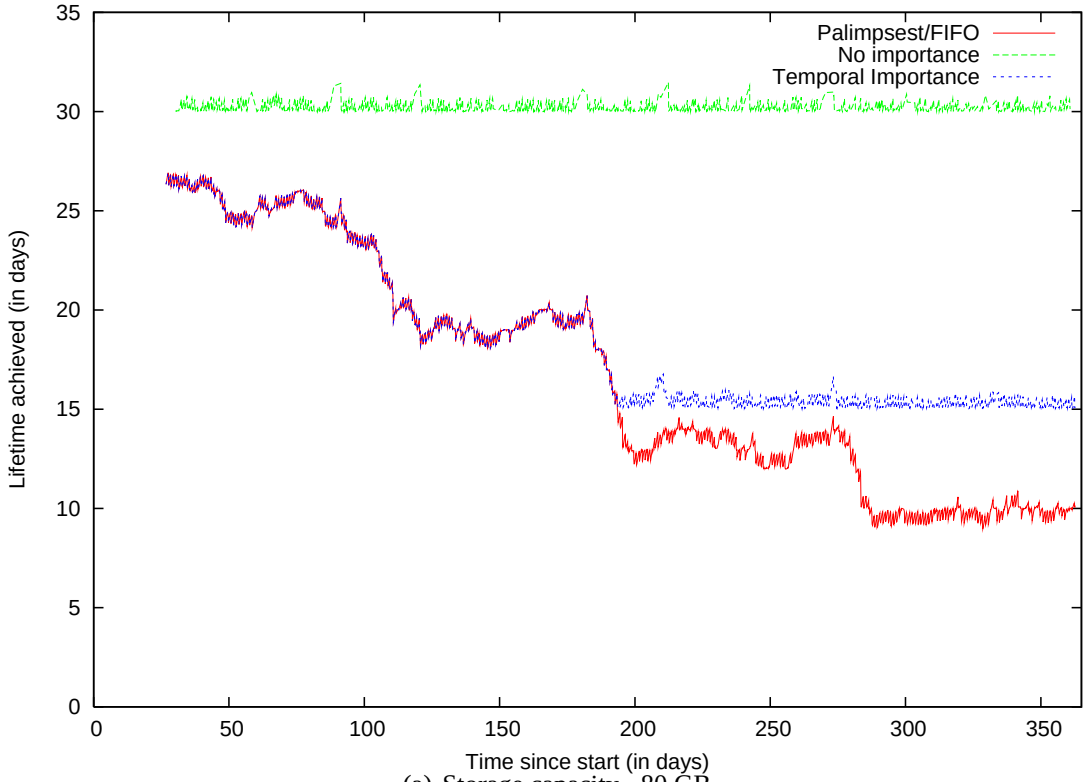


Figure 2: Sizes of object for storage

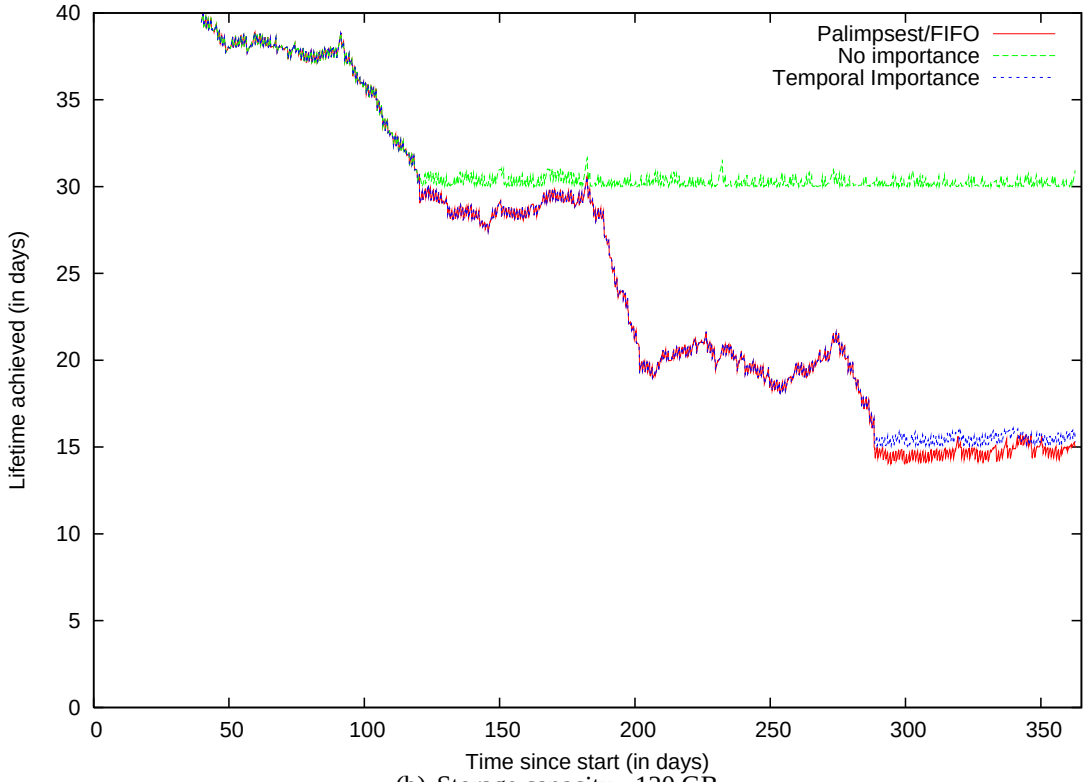
we illustrate the behavior of our system for a university level video lecture capture. For each setup, we describe the specific application's lifetime requirements, lifetimes achieved and our ability to predict the storage condition for assisting the applications in selecting the annotations.

5.1 Single application class

In this scenario, objects constantly arrive into the system at a rate that is randomly distributed up to 0.5 GB an hour for the first three months. Over the following three month intervals, this rate increases to 0.7 GB/hr, 1.0 GB/hr and 1.3 GB/hr, respectively. These values are chosen as a reasonable approximation of constant arrival rates that slowly increase with time. We plot the storage requirements for a whole year in Figure 2. Note that in realistic deployments, these rates may depend on the time of the day and account for holidays and other events. We simulated these access traces on a desktop with varying disk capacities. We only show the results for 80 GB and 120 GB disks in this section. In a traditional storage system, this space will be fully used up in about 40 to 50 days. We experimented with three policies: FCFS policy similar to Palimpsest, a lifetime policy without a temporal importance component ($L(t)$ = 1, t_{expire} = 30 days) and a lifetime function with temporal importance as ($L(t)$=1 ($t < 15$ days) and decreases linearly, t_{expire} = 30 days). This lifetime captures the notion *the object is definitely important for 15 days, might be important for another 15 days and probably not after 30 days.*

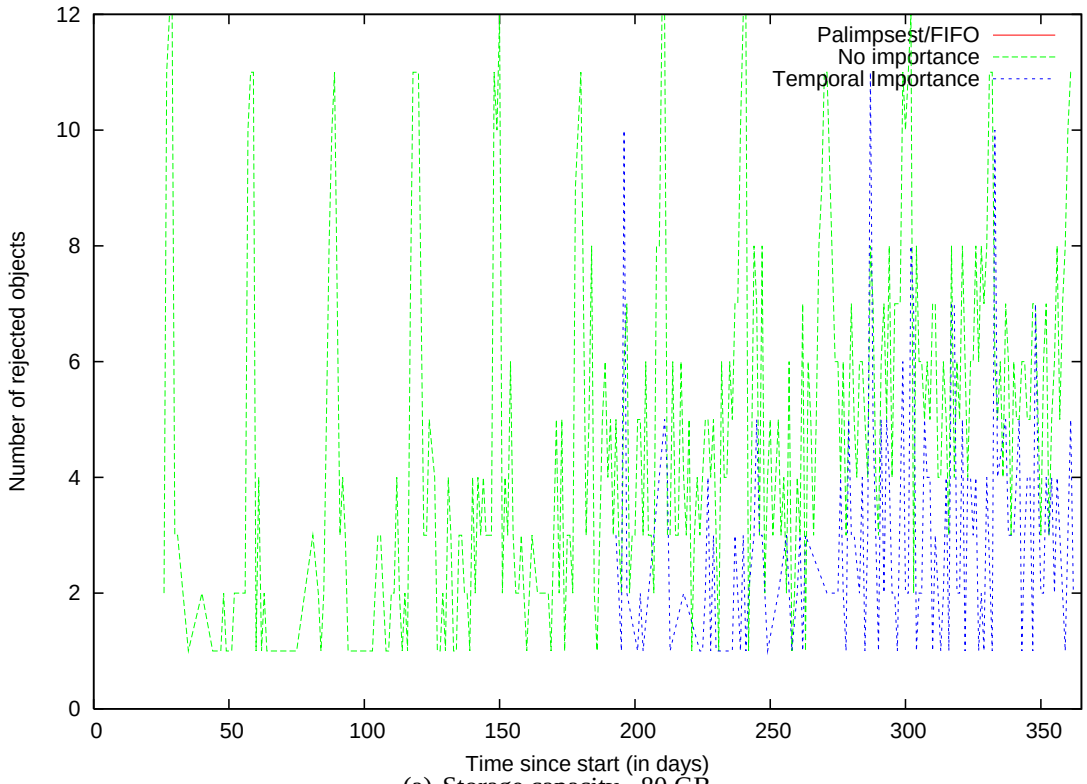


(a) Storage capacity - 80 GB

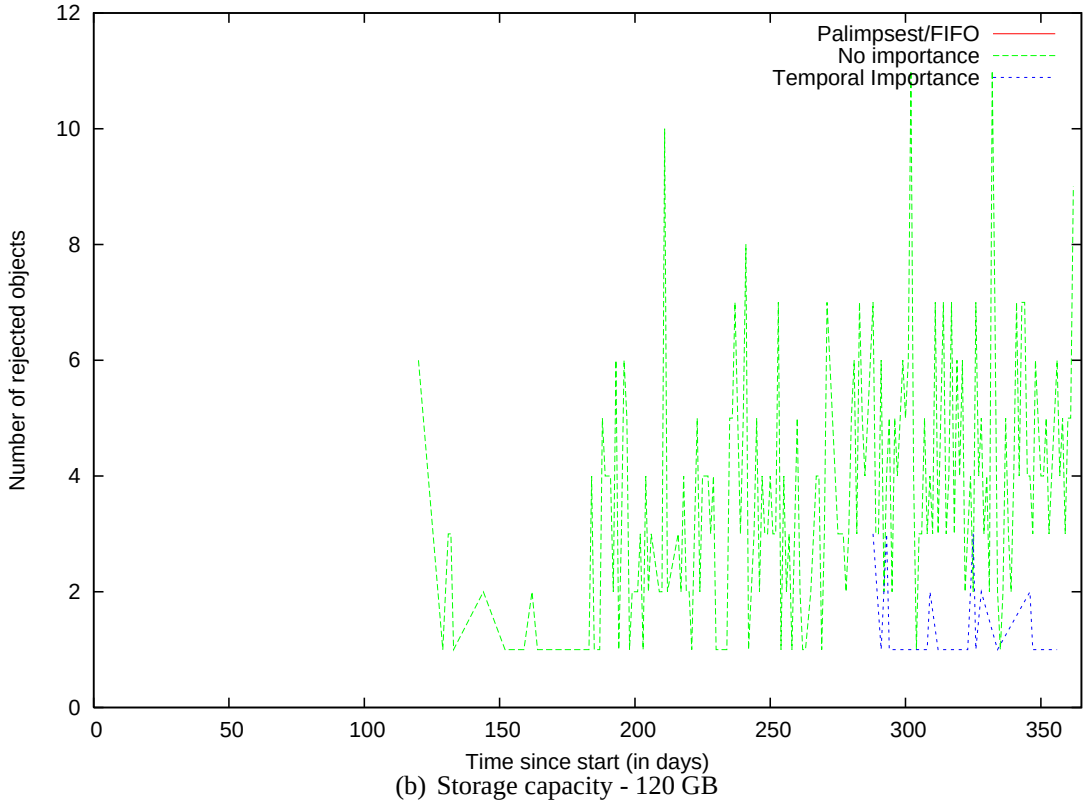


(b) Storage capacity - 120 GB

Figure 3: Lifetime achieved. (*No importance* is at the top, followed by *Temporal importance* and *Palimpsest* (Sec. 5.1.1))



(a) Storage capacity - 80 GB



(b) Storage capacity - 120 GB

Figure 4: Requests turned down because of full storage (Sec. 5.1.1) (storage is never full for Palimpsest)

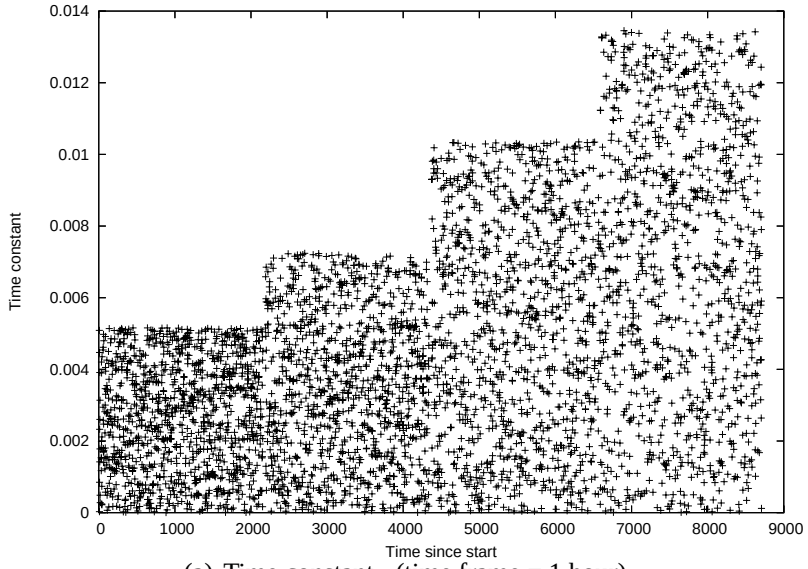
5.1.1 Observed Behavior: Lifetimes achieved

First, we plot the lifetimes achieved for the different policies for desktop storage of size 80 GB and 120 GB in Figure 3. Note that the lifetimes are measured when the objects are evicted. Hence, the graphs only start from 40 days or so even though the system was functional on day one. We also plot the number of requests for which the storage was full in Figure 4. From Figures 3(a) and 3(b), we note that when there is plenty of storage, all these policies perform in a similar fashion. However, as the storage pressure increases, their behavior diverges. The policy without temporal importance gives all stored objects their requested lifetime of 30 days. On the other hand, this policy rejects many more objects than a policy that implements the temporal importance function. Under severe storage pressure, the temporal importance also begins to reject newer objects. The key difference from Palimpsest is the control available to the content creator (who can trade lifetimes for the ability to provide storage to newer contents).

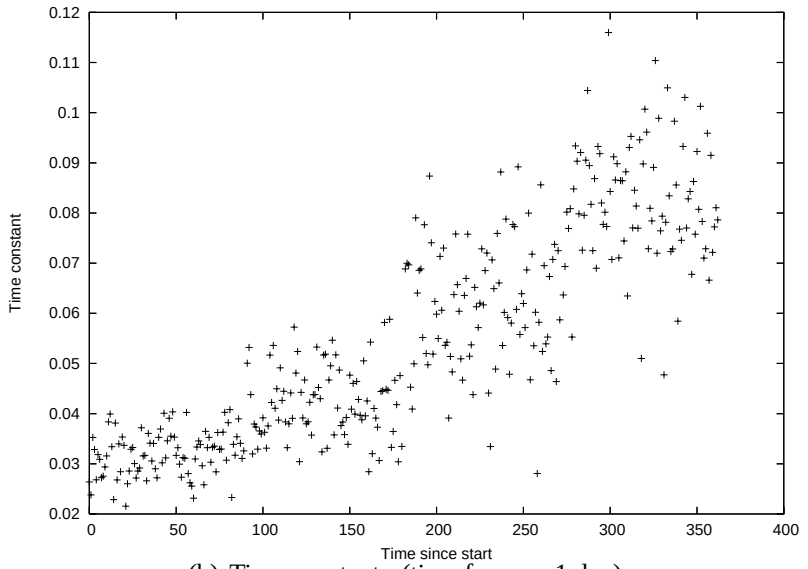
5.1.2 Observed Behavior: Predicting the actual object lifetimes achieved while associating lifetimes

Next, we analyze the effectiveness in predicting the object lifetimes achievable at the time that the object enters the system. Naturally, a system without temporal importance will guarantee the full 30 day availability for any object that was accepted for storage. The application developer need not be involved after the initial storage for such a system. Palimpsest describes a time constant which describes the rate of arrival with respect to the total storage. Time constants require the time frame for analyzing the rate of arrival and so we plot the results by analyzing every hour, day and month in Figure 5. We note that the measured time constant varied considerably, especially for analyzing every hour. The results for analyzing every day also exhibit heteroscedasticity [6] of the variance, wherein the variance of the time constant is not the same for all time intervals and depends on the arrival rate. Hence, an application that stores a new object is bound to misinterpret the storage pressure. Since the application is fully responsible for rejuvenating important objects, this poses a problem. Unless the arrival rates are predictable, the data needs to be analyzed over a long duration in order to identify the trends. In our setup, these required intervals can be as high as a month. Otherwise, an application might misinterpret the arrival rates and wake up later than necessary, potentially losing the object to reclamation.

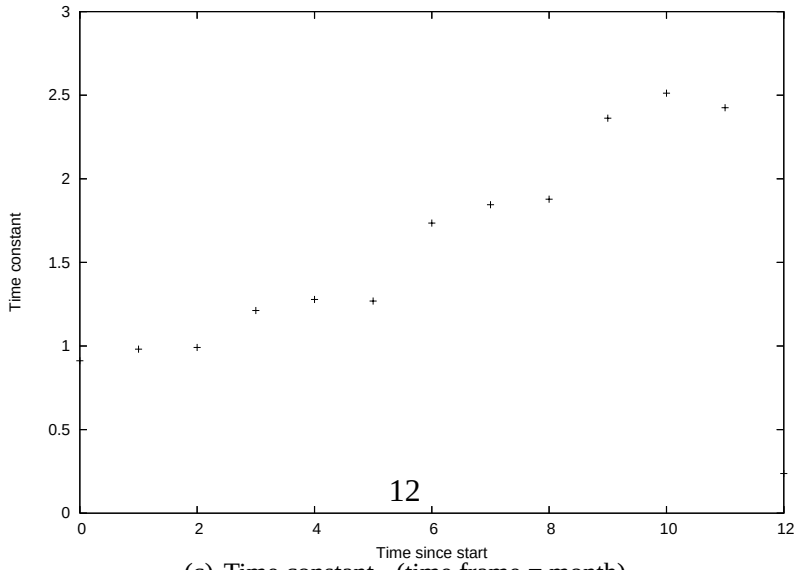
Next, we plot the instantaneous storage importance density by scaling each byte by its current importance. The storage importance density is a number in the range [0..1]. Expired objects and unallocated storage contribute an importance of zero. We plot the results in Figure 6. In order to fully appreciate this graph, we randomly took a snapshot of the storage when the instantaneous storage density was 0.8369 (random) and plotted the cumulative distribution of the importance values of the stored bytes in the storage in Figure 7. At this instant, 57% of the bytes have storage importance one and are non-preemptible. Objects with importance less than 0.25 cannot be stored. The average storage importance density of 0.8369 is a reasonable predictor of this state of the storage. The content creator is forced to make a decision up front. On the other hand, if the storage density is 1.0, the disk is full for all incoming objects. The difference between the storage density and the object importance gives some indication of the object longevity.



(a) Time constant - (time frame = 1 hour)

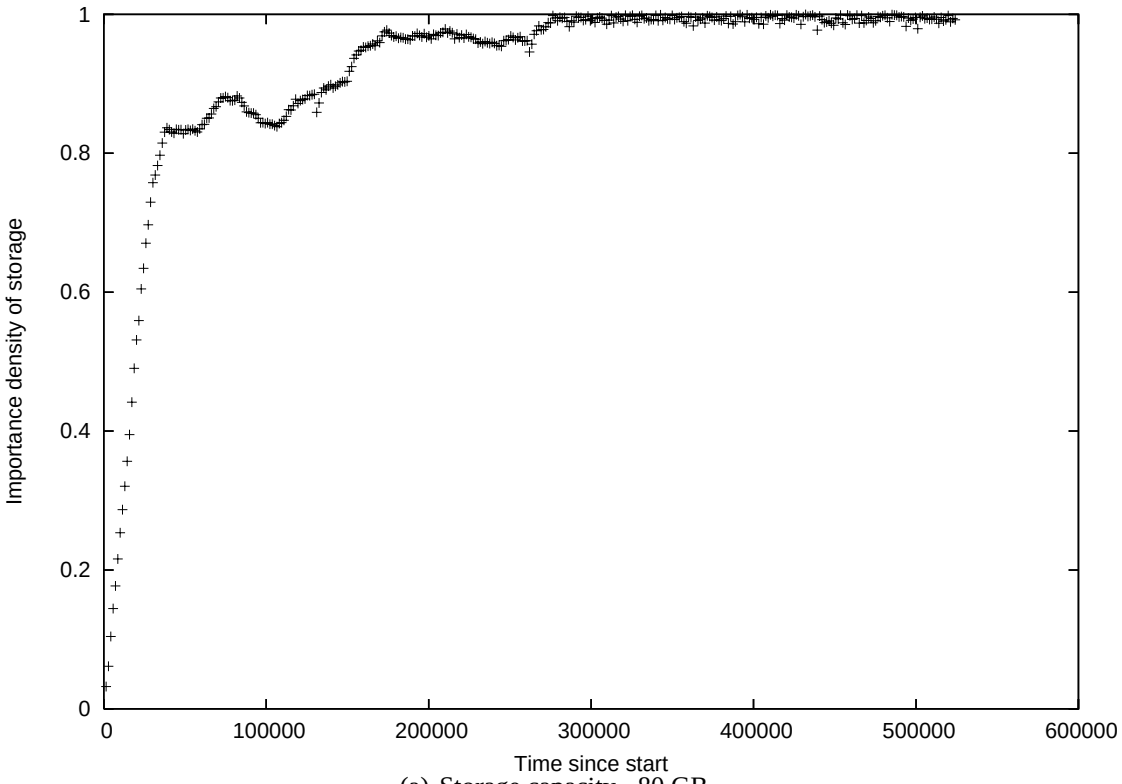


(b) Time constant - (time frame = 1 day)

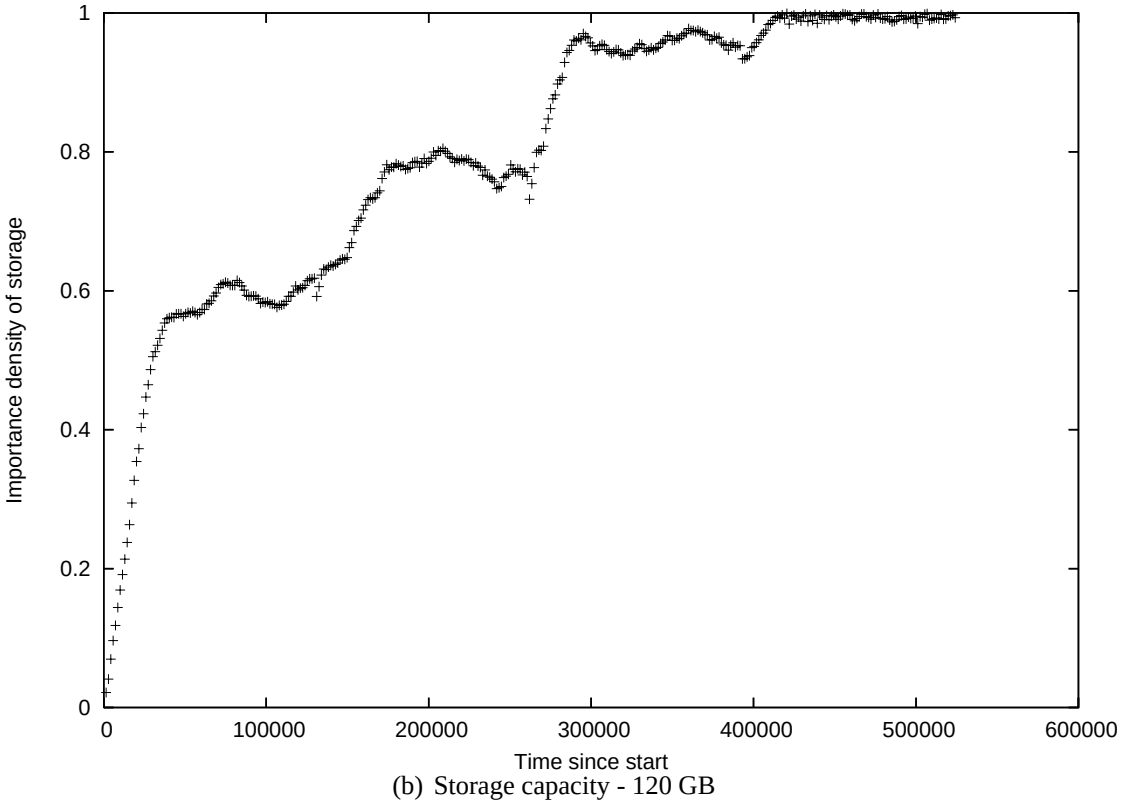


(c) Time constant - (time frame = month)

Figure 5: Time constant (Sec. 5.1.2)



(a) Storage capacity - 80 GB



(b) Storage capacity - 120 GB

Figure 6: Instantenous storage importance density (Sec. 5.1.2)

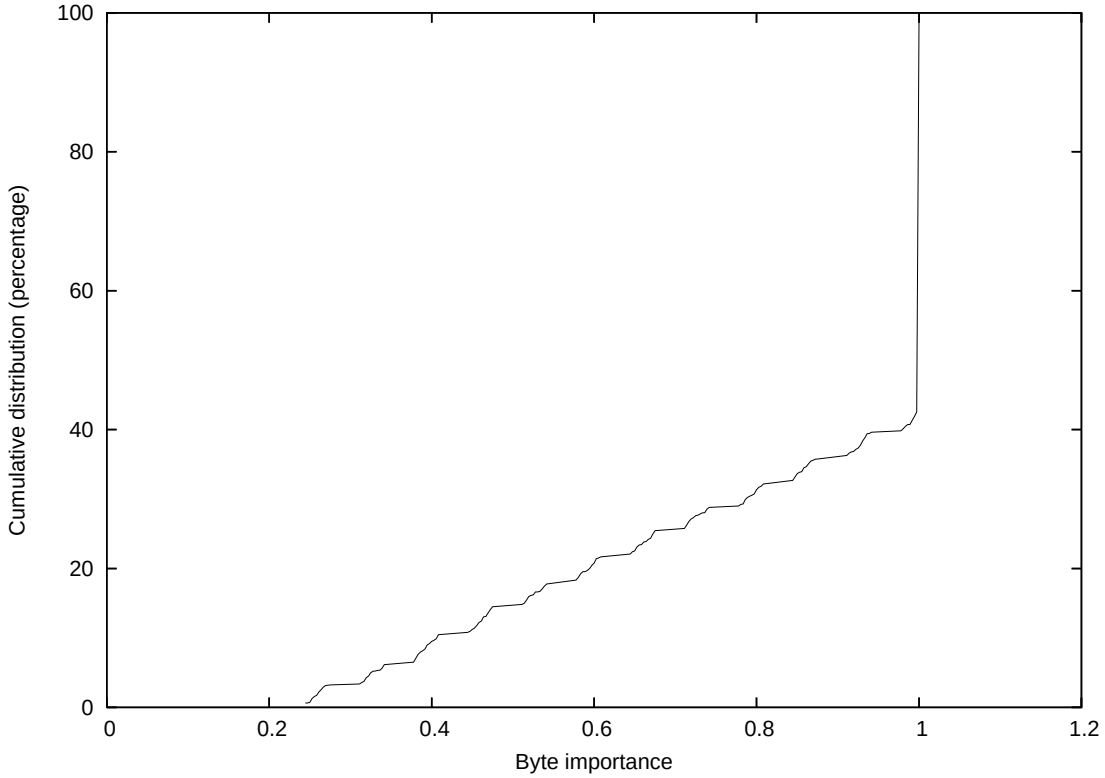


Figure 7: Cumulative distribution of the byte importance at an instant when importance density was 0.8369 (Sec. 5.1)

Term	$TermBegin$ $(dayofyear)$	$t_{persist}$ $(indays)$	t_{wane} $(indays)$
Spring	8	120 - today	730
Summer	150	210 - today	365
Fall	248	360 - today	850

Table 1: Lifetimes for lecture capture system

5.1.3 Summary of observation

The above set of experiments showed that the temporal lifetime function can allow the users some control over the lifetime and premature reclamation. The policies are scalable to the addition of more storage. Systems such as Palimpsest do not provide any guarantees from the system and leave the management entirely to the user. Given the nature of object arrivals, the user might not get good feedback on how to administer the objects. The storage importance density encapsulates the state of the storage and provides better feedback to the user. The application can decide at the outset the kinds of behavior it requires and whether the storage can provide such behavior. For many scenarios, the application need not continue to manage an object that was accepted for storage. The two step importance function appears to be appropriate for this scenario.

5.2 Lecture video capture for a single instructor using a single storage unit

Next we focus on a scenario which involves objects with varying lifetime requirements. We analyze the interplay of these importance functions. The lifetimes depend on the semester, day in the semester, as well as the specific user. We expand the system to encompass the whole university in Section 5.3.

5.2.1 Application lifetime requirements

This application treats recent lectures as more important than older lectures. Lectures should be fully available while the class is in session for the semester. In order to get some sense on how users might actually use a captured lecture, we captured our lectures for a 38 student, undergraduate Operating Systems course and analyzed the popularity of our lecture videos. Note that the lectures were captured in Spring 2006 semester; we still do not have the longer term trends. This particular core course is offered every spring. The graduate OS course is offered in the fall. Though not necessary, the graduate students might use these videos for reference. Our traces do not show this effect yet. We plot the number of times that the lectures were accessed in Figure 8. The plot also shows the days that the lectures were made available and significant events such as exams that might affect the access frequency. We continue to monitor the longer term trends.

Loosely based on the observations, the desired retention policy is as follows: spring semester starts after the first week in January and proceeds till May. After a month break, the summer term runs for two months. After another break, the fall semester starts in the second week of September and runs till the end of the year. All objects captured in spring are considered to be important

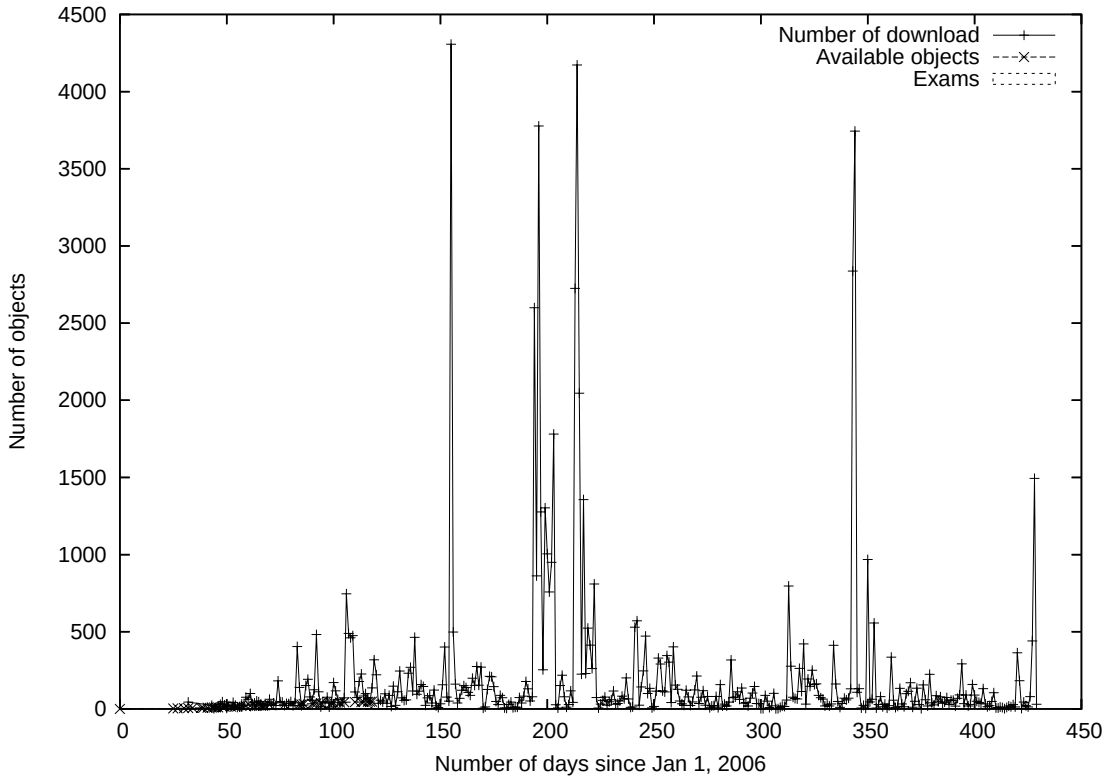
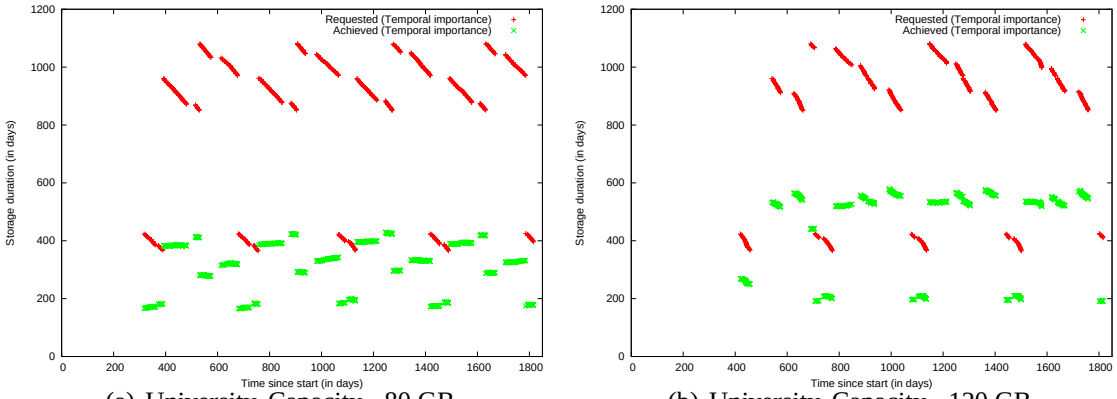
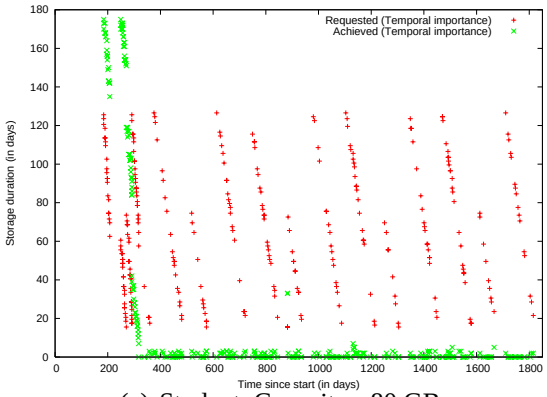


Figure 8: Number of lecture downloads per day (we were briefly slash-dotted during the spikes)

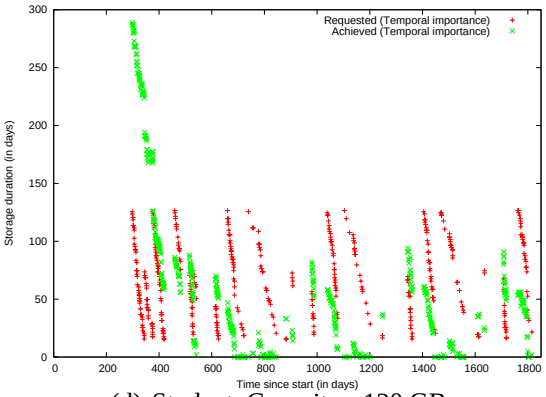


(a) University, Capacity - 80 GB

(b) University, Capacity - 120 GB



(c) Student, Capacity - 80 GB



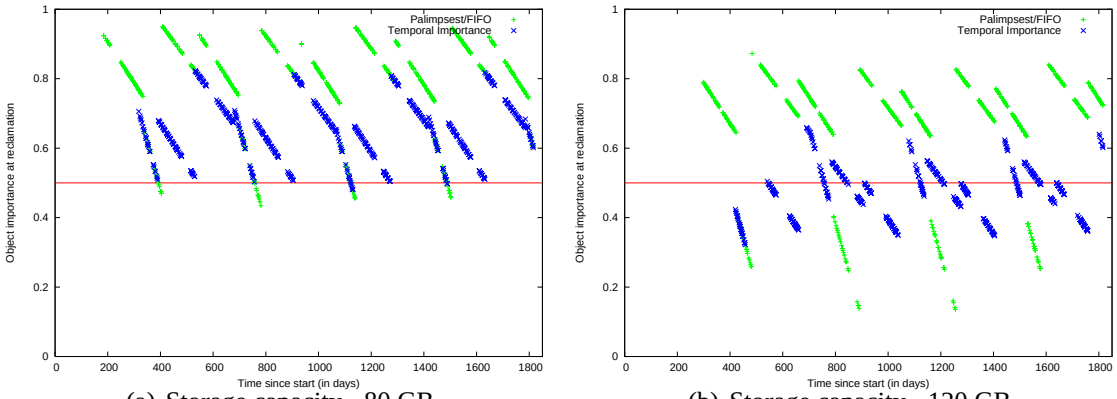
(d) Student, Capacity - 120 GB

Figure 9: Lifetime achieved (our two step importance function)(Sec. 5.2.2)

till the end of the semester. Their importance gradually wanes over the next two years till the end of spring two years later. Classes in summer proceed similarly during the semester with the importance waning in a year after the end of the semester. Classes in fall prolong their historical data up to the end of the spring semester in two years. The system allows up to three students to randomly add their own video interpretation of the lecture. However, student created MPEG4 streams are forced to 320x240 resolution (native resolution of the Apple video iPod and Sony PSP) and have a 50% importance (as compared to the university maintained cameras which start with 100% importance) till the end of the semester, with values gradually dropping in importance two weeks after the end of the term. For our simulations, these lifetime requirements are expressed as a two step function, as illustrated in Table 1. Though these parameters are reasonable, they are not definitive for all lecture capture systems. We continue to monitor the trends in our system.

5.2.2 Observed Behavior: Lifetimes achieved

First we plot the lifetimes requested and achieved by the various objects created by the university and students using the two step importance function in Figures 9 and 9, respectively. Similar to Sec. 5.1, we only show the results for using 80 GB and 120 GB local storage. While using an 80



(a) Storage capacity - 80 GB (b) Storage capacity - 120 GB

Figure 10: Importance at reclamation for university created objects(Sec. 5.2.2)

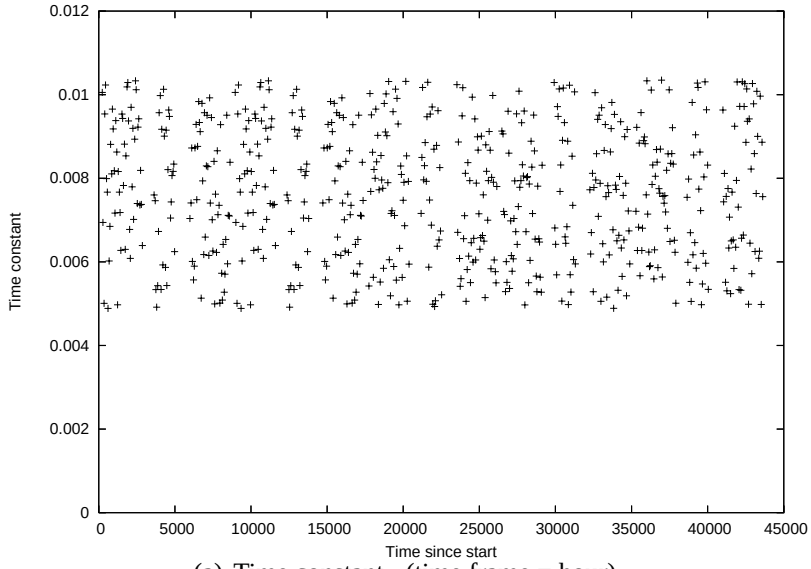
GB local storage, the university generated objects achieve lifetimes of 200 to 400 days depending on the semester in which the objects were created. However, this level of storage is insufficient for the student objects which are mostly rejected after 200 days (lifetimes close to zero after 200 days). As the available storage is increased, the students data are able to achieve some persistence, up to about 70 days.

On the other hand, reclamation policies such as Palimpsest (not illustrated for lack of space) did not offer any differentiation for the different users; student and university created objects achieved lifetimes of around 200 days, even while the system was operating under tremendous storage pressure (80 GB).

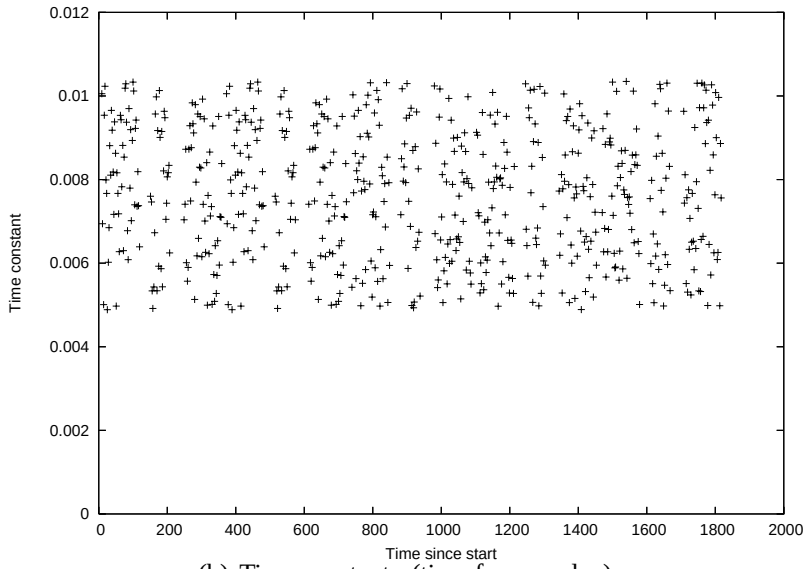
Next, we plot the importance value at reclamation for university created objects for the two storage levels of 80 GB and 120 GB in Figure 10. 50% was the initial importance level of student objects. Note that Palimpsest does not have a notion of object importance, we project the importance from our two step function to show the system behavior. While there is tremendous storage pressure, university objects that fall below 50% importance are evicted. As the pressure eases in the 120 GB storage, objects remain in the storage for importance values as low as 20% (created over the summer). Palimpsest reclaims objects which have higher importance values and leaves objects of lower than 0.5 importance. Such behavior is not preferable.

5.2.3 Observed Behavior: Predicting the actual object lifetimes achieved while associating lifetimes

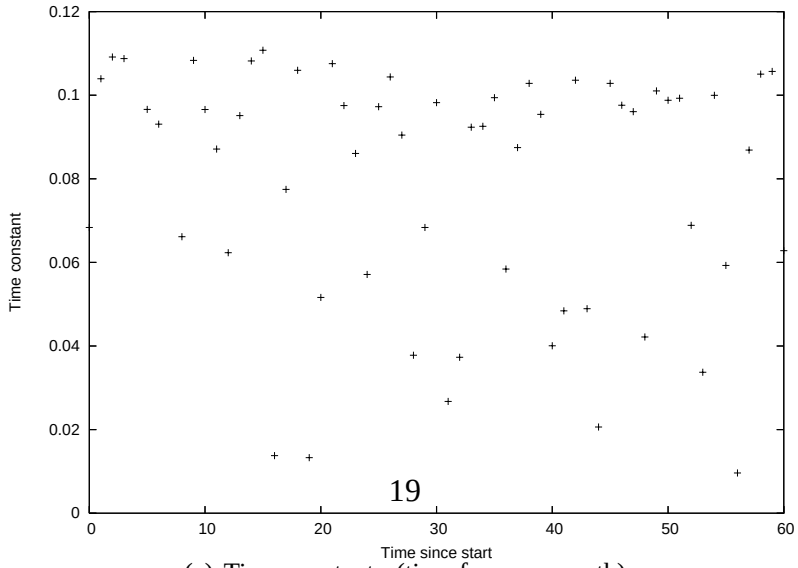
We analyze the Palimpsest's time constant as well as our storage importance density and plot the results in Figures 11 and 12 respectively. From Figure 11, we notice the time constant is not a good predictor even using a time range of a month. On the other hand, Figure 12 shows that the instantaneous average storage density is a good predictor. Applications can use the measured average storage density to understand the expected system behavior. As the storage pressure eases, more objects are retained and the average importance density becomes lower.



(a) Time constant - (time frame = hour)

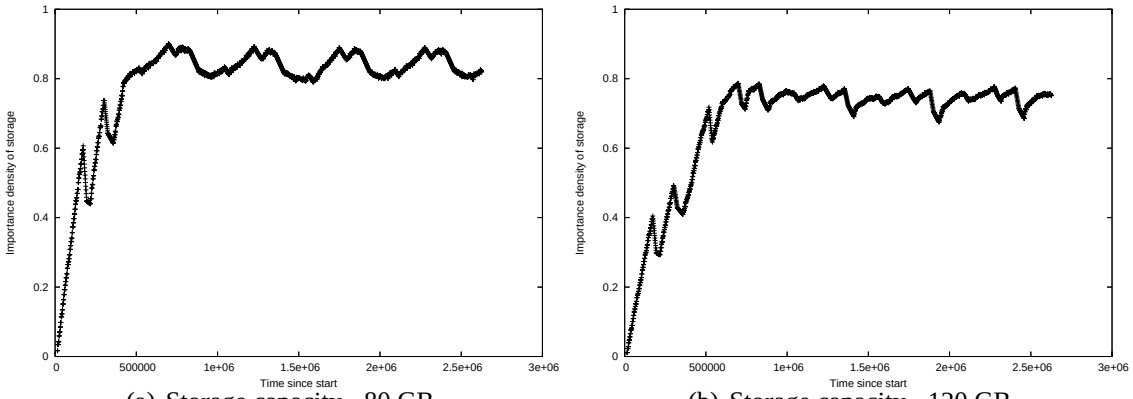


(b) Time constant - (time frame = day)



(c) Time constant - (time frame = month)

Figure 11: Time constant (Sec. 5.2.3)



(a) Storage capacity - 80 GB (b) Storage capacity - 120 GB

Figure 12: Instantenous storage importance density (Sec. 5.2.3)

5.2.4 Summary of observation

We analyzed the efficacy of a two step function to describe a complex lifetime requirement that was appropriate for storing lecture videos. We show that the system behavior matches expectations: the system is scalable with the addition of more storage and that the system behavior is predictable using the average storage importance density.

5.3 University-wide capture

Next, we expand to encompass the entire university. The system stores all the captured video in a university. We target capturing 2,321 courses and storing the videos across about 2,000 university owned desktops. The reclamation algorithm locates a storage unit to store the object and then reclaims storage from a less important object. We choose the storage unit which preempts the least important object using a distributed algorithm as follows: randomly pick x storage units. Random walks on our p2p overlay help us choose a good set of storage units. Each storage unit sorts the objects stored in it in increasing current temporal importance value followed by the amount of the remaining lifetimes. For each of these storage units, the system checks for the *highest importance object that will be preempted* in order to store this particular object. If the highest preempted objects' importance value for a particular storage unit is zero, then the object can be directly stored in this unit. If the highest preempted object from a storage unit has a higher importance value than the current object, the storage unit is *full* for this particular object. Otherwise, we wait for up to m successive tries and choose the storage unit with the lowest value of the highest importance object(s) preempted. Note, the highest importance object preempted is not weighted with the object sizes. For example, we choose a storage unit whose highest importance object preempted is smaller than another unit even though only 1% of the required storage space might come from the highest importance object.

We analyzed the results for a 2,000 node storage network with per node storage sizes of 80 GB, 120 GB (values used throughout this paper). We expect the university to continuously replace older desktops with newer desktops that will likely host larger disks. On the other hand, the lectures will

also likely be encoded in higher bitrate streams. We currently use a 1 Mbps video stream. Our simulator does not implement the interplay of growing storage and increasing space requirements. Our capture system required about 300 TB of storage every year. The storage system that uses 80 GB or 120 GB storage units (for a total of 160 TB or 240 TB of storage) cannot fully store a year's worth of new contents. In the interest of brevity, we only summarize the results.

Average importance density gives a good indication for the capture units to choose the appropriate lifetime parameters. In our target scenario, student videos are pegged to lower importance levels; the available storage to student cameras remains small until more storage is available. Likewise, the availability of more storage allows the system to leverage the extra storage, without explicitly changing the lifetime parameter itself. The cameras continue to use the same lifetime parameters and the actual behavior depends on the available storage.

6 Discussion

Providing traditional persistence guarantees severely restricts our ability to design storage systems that deal with vast amounts of data over long durations. In this paper, we described a temporal lifetime abstraction that can allow the system to elastically accommodate newer contents by automatically reclaiming older and less important objects. We introduced the average *storage importance density* metric to quantify the behavior of the storage. Using a two step temporal importance function, we showed that the system allowed for the flexibility to specify a single temporal importance function that adapted the system behavior to the available storage resources. A user level file system prototype of the system will be available at the author's web page (`http://www.nd.edu/~surendar/`).

Recently, we are investigating temporal importance functions for other scenarios. For example, storage in sensor scenarios might treat unprocessed data as important but retain processed data to accomodate for communications failure in propagating the results. Similarly, peer-to-peer storage systems maintain confidentiality and integrity using encryption and digital signatures. The importance of data corresponds to the guarantees that can be made about its confidentiality and integrity. Under storage pressure, a security-sensitive system could evict the most compromised objects. These scenarios might require the ability to dynamically change the importance values based on triggers such as the receipt of an acknowledgment.

References

[1] Ranjita Bhagwan, Fred Douglis, Kirsten Hildrum, Jeffrey O. Kephart, and William E. Walsh. Time-varying management of data storage. In *IEEE First Workshop on Hot Topics in System Dependability*, Yokohama, Japan, June 2005.

[2] Surendar Chandra. Lecture video capture for the masses. In *Innovation and Technology in Computer Science Education (iTiCSE 2007)*, Dundee, Scotland, June 2007.

[3] Surendar Chandra and Carla Schlatter Ellis. JPEG Compression Metric as a Quality Aware Image Transcoding. In *Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems USITS-99*, pages 81–92, Boulder, CO, October 1999. USENIX Association.

[4] Fred Douglis, John Palmer, Elizabeth S. Richards, David Tao, William H. Tetzlaff, John M. Tracey, and Jian Yin. Position: short object lifetimes require a delete-optimized storage system. In *ACM SIGOPS European workshop: beyond the PC*, page 6, New York, NY, USA, September 2004. ACM Press.

[5] Mor Harchol-Balter and Allen B. Downey. Exploiting process lifetime distributions for dynamic load balancing. *ACM Trans. Comput. Syst.*, 15(3):253–285, 1997.

[6] David G. Kleinbaum, Lawrence L. Kupper, and Keith E. Muller. *Applied Regression Analysis and Other Multivariable Methods*. PWS - Kent Publishing Company, second edition, 1988.

[7] Nielsen/NetRatings Inc. User-generated content drives half of u.s. top 10 fastest growing web brands. `http://www.nielsen-netratings.com/pr/PR_060810.PDF`, August 2006.

[8] Stefan Podlipnig and László Böszörmenyi. A survey of web cache replacement strategies. *ACM Comput. Surv.*, 35(4):374–398, 2003.

[9] Timothy Roscoe and Steven Hand. Palimpsest: Soft-capacity storage for planetary-scale services. In *HotOS*, pages 127–132, Kauai, HI, May 2003.

[10] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, and Alistair C. Veitch. Elephant: The file system that never forgets. In *Seventh Workshop on Hot Topics in Operating Systems*, pages 2–7, Rio Rico, AZ, March 1999.

[11] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir. Deciding when to forget in the elephant file system. In *Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 110–123. ACM Press, 1999.

[12] Darko Stefanović, Kathryn S. McKinley, and J. Eliot B. Moss. On models for object lifetime distributions. In *Proceedings of the second international symposium on Memory management*, pages 137–142. ACM Press, 2000.

[13] John Wilkes and Raymie Stata. Specifying data availability in multi-device file systems. In *ACM SIGOPS European workshop*, pages 1–4. ACM Press, 1990.