Scalable Service Differentiation in a Shared Storage Cache

Bong-Jun Ko
Columbia University
kobj@ee.columbia.edu

Kang-Won Lee Khalil Amiri Seraphin Calo
IBM T.J. Watson Research Center
{kangwon, scalo}@us.ibm.com, kamiri@ca.ibm.com

Abstract

Motivated by the need to enable easier data sharing and curb rising storage management costs, storage systems are becoming increasingly consolidated and thereby shared by a large number of users and applications. In such environments, service differentiation becomes increasingly important. Since caching is a fundamental and pervasive technique employed to improve the performance of storage systems, providing differentiated services from a storage cache is a crucial component of the entire end-to-end QoS solution. In this paper, we discuss a QoS architecture for a shared storage proxy cache which can provide long-term hit rate assurances to competing classes. The proposed architecture consists of three components: (a) per-class feedback controllers that track the performance of each class, (b) a fairness controller that allocates excess resources fairly in the case when all goals are met, and (c) a contention resolver that decides cache allocation in the case when at least one class does not meet its target hit rate. We compare the performance of various feedback per-class controllers, and provide guidelines for designing QoS mechanisms for such a dynamic environment.

1 Introduction

Motivated by the need to enable easier data sharing and curb rising storage management costs, storage systems are becoming increasingly consolidated and thereby shared by a large number of users and applications. Furthermore, current trends suggest a potential market for storage system outsourcing, where the same storage system at a service provider is used to store the data of several classes of remote customers. In such environments, service differentiation among different application and user classes becomes increasingly important.

Caching is a fundamental and pervasive technique employed to improve the performance of storage systems. Consequently, providing differentiated services from a storage cache is a crucial component of the entire end-to-end QoS solution. In this paper, we define the problem of service differentiation in a storage cache as that of achieving specified hit rate goals for a number of competing classes sharing the

cache. More specifically, the problem is that of dynamically allocating cache resources across classes to achieve the specified goals.

Designing a storage cache to support differentiated services invokes interesting questions about the proper choice of policies and mechanisms. For example, when all hit rate goals are achieved and excess resources exist, what policy should govern the allocation of such excess resources? On the other hand, when dynamic changes risk some applications to temporarily miss their hit rate goals, how should the system resolve the contention across the competing classes?

In this paper, we discuss a scalable QoS architecture for a storage proxy cache which can provide long-term hit rate assurances to competing classes. The proposed architecture consists of three components: (a) per-class feedback controllers that track the performance of each class, (b) a fairness controller that allocates excess resources fairly in the case when all goals are met, and (c) a contention resolver that decides cache allocation in the case when at least one class does not meet its target hit rate. We compare the performance of various per-class feedback controllers, and provide guidelines for designing QoS mechanisms for the shared storage environment.

Through a trace-driven simulation study, we arrive at the following conclusions.

- It is possible to design an effective per-class controller which controls space allocation for a given class so as to track a specified target hit rate. In particular, the retrospective controller proposed in this paper provides a practical means to achieve this goal without incurring excessive fluctuation in space allocation.
- Once the minimum target service levels have been achieved, it is desirable to allocate excess cache resources fairly across competing classes. In our architecture, the fairness controller implements this function.
- In addition to achieving the long term service levels, our QoS architecture can handle temporary overloads based on a high level policy and ensure that high priority classes do not violate their performance targets at any time.

The remainder of this paper is organized as follows. Section 2 describes the general framework, and in particular the

target environment, our assumptions and the design challenges. Section 3 presents our QoS architecture and describes its various components. Section 4 evaluates the proposed mechanisms for adjusting cache space allocations. Section 5 discusses related work and Section 6 concludes the paper.

2 Framework

We consider a shared storage system consisting of *storage clients* accessing a remote *storage location* via *storage proxies*. Disk read and write requests from the storage clients are sent to the (possibly remote) storage location through a proxy cache. Storage proxy caches hide disk access latency by caching frequently accessed disk objects.¹ The storage location is a back-end storage system such as a collection of disk or disk arrays.

2.1 Service level goals

We assume that requests submitted to the cache are tagged according to the application class they belong to, for example based on application types (e.g., c_1 for the file server workload, c_2 for database accesses) or user ID (e.g., c_1 for privileged users, c_2 for regular users). Denote by $C = \{c_1, c_2, \dots, c_n\}$ the set of application classes. The service level goals of the QoS scheme is to satisfy a given average access latency for disk I/O operations measured over a long-term interval. More specifically, the service level agreement with the clients can be described as follows: “the average access latency of class c_i must be less than or equal to l_i^* (msec) measured over T_m (min).” Here l_i^* represents the target access latency of class c_i , and T_m represents a measurement time window. T_m is typically in the order of a few tens of minutes or a few hours.

2.2 Our approach

In the shared storage model described above, the storage access latency l_i of class c_i is mainly determined by three parameters: average access latency to the local proxy (l_{local}), hit rate in the proxy cache (h_i), and average access latency to the remote storage location (l_{remote}). More precisely,

$$l_i = h_i * l_{local} + (1 - h_i) * l_{remote}. \quad (1)$$

Assuming that l_{local} is a small constant and l_{remote} is the same across different classes, i.e., there is no network level service differentiation, the access latency l_i is effectively determined by the hit ratio h_i .

In other words, it is possible to control the access latency l_i of class c_i by controlling the observed hit rate h_i at the proxy, and therefore, our proposed QoS mechanism tries to

satisfy the access latency requirement of each class by controlling the hit ratio of the class. More specifically, the proposed QoS mechanism controls the cache space allocated to each class to meet its hit rate target, which will result in an overall response time $l_i \leq l_i^*$. We call such hit rate as the *reference hit rate* of class c_i and denote it by t_i . Using this notation, our service goal can be stated as: “for every class c_i , the average hit rate h_i measured over T_m is greater than its target t_i .”

Note that our service goal defines a performance metric that guarantees a minimum level of service to the clients over a prespecified period. This requires the existence of a provisioning module, which ensures that the aggregate client requests can be satisfied by the current cache space and performs admission control based on long term workload analysis. The design of such a provisioning module is outside the scope of this paper.

2.3 Design challenges

There are several challenges in designing an effective QoS solution for a shared storage cache. First of all, the QoS architecture must handle a potentially large number of competing classes. Dynamically partitioning a shared cache among multiple application classes involves responding to complex and dynamic interactions between classes. The problem can be modeled as designing a MIMO (multi-input-multi-output) system (see [3] for example). However, the complexity of a MIMO system increases significantly as the number of classes increases. In this paper, we strive to provide a scalable and efficient solution to handle multiple service classes.

Secondly, designing an effective cache space controller to closely track a given hit rate places another challenge. From the trace-driven study, we find that allocating more cache space does not always translate into increased cache hit rate, in particular, when the working set size increases at a greater rate than the cache space increase. This time-varying property, coupled with workload heterogeneity across different applications, highlight the need for a controller that is robust to workload heterogeneity and changes, and to the choice of controller configuration parameters.

Thirdly, although our storage architecture relies on an external provisioning module to meet the long term service goals, short term contention can occur due to dynamic variations in the workload. Since such variations are prevalent in practice as we will see in the next section, an effective mechanism to handle temporary overloads must be provided. On the other hand, it is often desirable to allocate excess cache space resources, when all target service levels are met, fairly across applications. A mechanism to ensure such fairness is also required.

In the next section, we propose an architecture and a set of mechanisms that ensure the target service level goals and address the design challenges described in this section.

¹We assume both read and write caching. In the case of write caching, we assume a standard techniques for maintaining consistency across distributed caches [1, 2].

3 Proxy QoS architecture

The QoS architecture we propose for the shared cache consists of three modules that interact with each other.

1. *Per-class Controller*: Each application class is associated with a per-class controller. The per-class controller is a feedback controller that determines the cache space allocation for the class based on the current measured hit rate and the current space allocation for that class. We note that each per-class controller operates independently of the others, basing its operation only on the feedback information from its own class. In this way, controller complexity is kept to a minimum.
2. *Contention Resolver*: As previously mentioned, temporary contention for cache resources can occur. The contention resolver is responsible for handling such cases by deciding how cache space is allocated in response to conflicting controller requests. The contention resolver makes its decision based on the level of contention, the requests from the per-class controllers, and according to high level policies.
3. *Fairness Controller*: The fairness controller in our architecture computes the fair share of each class based on the current performance estimate and the reference target hit rate of each class. It then adjusts the target hit rate of each class that the per-class controller must track. In this paper, we consider a simple fairness policy: distribute excess resources in such a way that the resulting hit ratio is proportional to the reference hit rate of the class.

Figure 1 shows the various components in our QoS architecture. In this discussion, we describe the operation of the proposed QoS mechanisms assuming that time is divided into discrete units, called *rounds*. At the beginning of each round, the hit rate of each application class during the previous round is recorded. Based on the hit rate measurement h_i of class c_i , the fairness controller computes a new target hit rate t_i^* ($> t_i$).

The new target hit rate t_i^* is communicated to the per-class controller of the class c_i . The per-class controller then computes the space allocation s_i required for the class to achieve t_i^* , and makes a request to the contention resolver. Upon receiving the space requests from all the per-class controllers for the new round, the contention resolver determines the actual space allocation s_i^* to each class. Space allocations for the new round are thus decided. The hit ratios for each class are recorded at the end of the round, and the above procedure repeated.

We note that the length of the round involves making a tradeoff between the stability of the system and its adaptability. If we stretch the duration of the round, we can better account for the delay in cache control. However, this will slow down the speed at which the system can adapt to changes. In this paper, we set the round to be long enough such that the number of accesses occurring during the round represents a

Figure 1. The storage proxy QoS framework

small multiple of the number of blocks allocated to the class. From our extensive measurements, we have determined that this duration is the smallest time interval required to ensure that the measured hit rate has reasonable accuracy.

Our modular architecture has several advantages compared to a monolithic controller alternative. First, the complexity of controller design is reduced since each component performs fairly simple and well-understood operations. On the other hand, designing a monolithic controller that is functionally equivalent to our system results in substantial controller complexity.

Second, the modular design allows us to easily “plug in” new modules as they become available. For example, we can simply upgrade the per-class controller without having to change the fairness controller and the contention resolver. Similarly, we can implement different fairness or contention resolution policies, according to high-level administrative goals.

In what follows, we discuss the detailed design of each component in our architecture.

3.1 Per-class controller

In essence, the per-class controller is a feedback controller that takes the current cache space allocation s_i and the measured hit rate h_i as input parameters, and produces the new cache space allocation s_i^* for the class to meet t_i^* as an output. We require the per-class controller to effectively track the target hit ratio even when the user workload changes dynamically. In addition, we want the hit rate variation and the changes in the space allocated to the class to be small. In this paper, we consider three classes of control algorithms (linear control, gradient-based control, PID control) as well as our scheme, which we refer to retrospective control.

3.1.1 Linear controller

The linear controller is the simplest among the four controllers. It adjusts the cache space allocation according to

the following rule:

$$s_i(n+1) = s_i(n) + \alpha(t_i - h_i(n)). \quad (2)$$

Recall that t_i denotes the target reference hit ratio and $h_i(n)$ denotes the measured hit ratio in the n^{th} round. In short, the linear controller simply adjusts cache space according to the difference in the target and the measured value. Thus, the performance of the controller is highly sensitive to the constant weight α .

3.1.2 Gradient-based controller

This controller improves on the linear controller by adapting the constant weight, α , according to its estimate of the gradient of the space-hit rate curve. By estimating the slope, we expect the controller to adapt more effectively to the dynamics of the workload. To estimate the gradient of the curve, we take the ratio of the measured change in hit rate to the corresponding change in space allocation in the previous interval.

$$s_i(n+1) = s_i(n) + \frac{\epsilon}{\frac{\Delta h_i}{\Delta s_i}} \times (t_i - h_i(n)), \quad (3)$$

where $\Delta h_i = h_i(n) - h_i(n-1)$ and $\Delta s_i = s_i(n) - s_i(n-1)$. In effect, the controller estimates the gradient of the space-hit rate curve by keeping track of the history of the changes in space allocation and the corresponding changes in hit rate.

3.1.3 PID controller

The PID controller is one of the most widely used controllers in industrial feedback control systems. It consists of three feedback terms: proportional, integral, and derivative terms. In our case, the operation of the PID controller can be described as follows.

$$s_i(n+1) = s_i(0) + K_P e_i(n) + K_I \sum_j^{n-1} e_i(j) + K_D \Delta e_i(n), \quad (4)$$

where $e_i = t_i - h_i(n)$, the difference between the reference and the measured value, and $\Delta e_i(n) = e_i(n) - e_i(n-1)$. The three terms added to $s_i(0)$ in the above equation denote proportional, integral, and derivative components, respectively. By controlling the gain of each term, we can change the characteristics of the controller. For example, setting a large proportional feedback gain (K_P) typically leads to faster response at the cost of increased instability. On the other hand, increasing the derivative gain (K_D) has a dampening effect and tends to improve stability.

3.1.4 Retrospective controller

The control approaches mentioned so far make limited use of the history which can be accumulated on-line in a shared storage cache. In particular, the system can explicitly maintain histories of past application request streams and derive

relatively accurate predictions about what the hit rate would be under various cache space allocations. This idea has motivated the design of a new controller that we propose in this paper. We call it the *retrospective controller* since it refers to the history of past accesses. In order to make accurate predictions, the controller maintains the *summary MRA (most recently accessed) block list* for the disk blocks which have been accessed in the recent past. This includes blocks that do not exist in the cache, for example blocks which have been evicted and replaced by other blocks. Each entry in the summary list maintains the disk block id, and the access count within the last measurement interval associated with that disk block. When the measured hit rate of the class falls short of the reference hit rate, the controller computes the number of blocks which should be added to the class' space allocation, so that the target hit rate is achieved. This is deduced by consulting the summary MRA list. On the other hand, if the measured hit rate is higher than the reference hit rate, the controller examines the cache entry and determines the number of cache blocks which can be safely removed.

$$s_i(n+1) = s_i(n) + F_i(t_i), \quad (5)$$

where the function $F_i(t_i)$ returns the number of disk blocks to add or subtract (when F_i is negative) from the current space allocation by looking at the summary MRA list.

To calculate $F_i(t_i)$, the controller must traverse the list adding up the number of accesses to each block to determine how much hit rate *could have been* achieved by storing a certain number of blocks in the cache. Note that the summary list is maintained in LRU order to simulate the LRU cache management algorithm.

We can view the retrospective controller as having a more global view of the space-hit rate curve whereas the linear or gradient controller captures the slope of that curve only at the neighborhood of the current space allocation point. In general, it can simulate any cache replacement algorithm that the cache may implement.

The access count values in the summary list entry must decay with time since they should be eventually forgotten in favor of more recent histories. This is done by maintaining an exponentially decaying average of the history using a decay parameter $0.5 < \beta < 1$. We examine the sensitivity (or lack thereof) of the controller performance to β in the next section.

3.2 The fairness controller

Before discussing the operation of the fairness controller, we must first define the notion of fairness that we are striving for. In this paper, we consider an intuitive definition of fairness which dictates that excess resources are distributed such that the effective hit rate $h_i(> t_i)$ is proportional to the reference hit rate t_i . To achieve this goal, the fairness controller performs a simple calculation to modify the target hit rate of each class.

$$t_i^*(n+1) = t_i \frac{S}{\sum_j \left(\frac{s_j^*(n)}{h_j(n)} t_j \right)} \quad (6)$$

where S is the total cache space and $s_i^*(n)$ is the cache space allocated to class c_i (i.e., $\sum s_i^*(n) = S$). In essence, the fairness controller tries to estimate the fair hit rate targets (higher than their reference hit rates) that will consume the entire cache space. Note however that the fairness controller must compute the distribution of the excess resources in the hit rate domain while the actual distribution must be done in the space domain. We can show that the fairness targets computed by Eq. 6 minimizes the deviation of the total space demand of the per-class controllers from the actual cache space, assuming that the space-hit rate curve can be approximated by a time-varying linear function. A simple proof is presented in the appendix.

3.3 The contention resolver

In the absence of contention (cache space $>$ total demand), the contention resolver only needs to make minor adjustments to allocation requests from the per-class controllers. This step is necessary because the target hit rates specified by the fairness controller are not perfect and the per-class controllers independently compute their cache space allocation requests without any coordination between them. This is done by a simple scaling operation.

On the other hand, when contention occurs (cache space $<$ total demand), the contention resolver must handle this temporary overload. In general there are two policies. The first policy is to treat all classes equally and allocates $s_i * \frac{S}{\sum_i s_i}$ to every class. With this “proportional allocation” all classes observe temporary service violation although the long term service goals are still ensured.

The second policy considers a scenario when some classes are more important than the others. Under this policy, which we refer to as “prioritized allocation,” the contention resolver tries to ensure that high priority classes do not experience short term service violations. A naive approach to implement the prioritized policy is to allocate the cache space to the highest priority class first, then to the next highest priority class and so on, until the all cache blocks are fully allocated. We find that this *reactive* approach does not work well because of the inherent delay in caching: allocating more space does not immediately translate into an improvement in hit rate.

Therefore, we consider a more *proactive* scheme, which provisions more resources to higher priority classes even when there is no contention. This goal is achieved by specifying differentiated adaptation rates to different priority classes when reducing cache space allocations. In particular, we specify a slower reduction rate to the higher priority classes than those of the lower classes. In this way, the high priority classes release their allocated cache space more slowly than the lower priority classes do and therefore, they are less likely to suffer from sudden space constraints due to workload changes. We examine the impact of these policies on the temporary service violation rate in the next section.

Figure 2. Dynamic changes in hit rate under a fixed size cache

4 Evaluation

We evaluate our storage QoS scheme using a trace-driven simulation. For this evaluation, we implemented a cache simulator that implements a simple LRU (least-recently used) algorithm per class based on the disk space allocated to the class. We use two disk traces with different characteristics. The first one is a set of traces collected from the Sprite distributed file system [4]. The second set of traces were collected from the 1998 Worldcup Soccer Web server [5]. Figure 2 shows the changes in hit rate for both traces for a fixed cache size. From the figure, we observe that the Sprite trace is much more dynamic (hit rate ranging from 10% to 80%) than the Web server trace, in terms of hit rate variation. The figure reports the cumulative hit rate under a fixed cache size.

4.1 Evaluation of the per-class controllers

We rate the per-class controllers based on the following metrics: (a) how closely the actual hit rate tracks the target hit rate, (b) how sensitive the performance is to the choice of controller parameters, and (c) how gracefully the cache space allocation changes. We first consider the first two metrics by examining the hit rate adaptation of the controllers.

4.1.1 Hit rate adaptation

Figure 3 represents the measured hit rate under the various per-class controllers for the Sprite file system trace. We set the target hit rate to 0.4. We try two different parameter settings for each controller to highlight the sensitivity of each approach to the particular choice of controller parameters.

Figure 3(a) summarizes the results for the linear controller. In general, the linear controller becomes more adaptive as the linear weight, α , increases. However, this comes at the cost of highly fluctuating cache space allocation as we shall see in the next section. Therefore, the linear weight parameter must be carefully selected to balance hit rate performance and space efficiency. In general, however, this is a difficult task because a single α will not work well for different workloads, and the optimal value of α is not known a priori.

Figure 3. Performance of the per-class controllers ($t_i = 0.4$) and their sensitivity to the parameters: Sprite trace

Figure 3(b) graphs the hit ratio under the gradient controller. Overall, we observe that the gradient controller adapts poorly to workload dynamics. This is because accurately estimating the slope of the space-hit rate curve is quite difficult for dynamic workloads. In this case, the measured hit rate changes quickly with small or no changes in space allocation, and therefore the gradient controller has to make a conservative estimate to avoid instability. As a result, its effectiveness in this case is quite limited.

Figure 3(c) shows the performance of the PID controller. As we observe from the figure, the performance of the PID controller is quite sensitive to parameter settings. In fact, tuning the three parameters to get good performance is a non-trivial task. Note that a set of good parameters for one class may not be effective for the other, and vice versa. In addition, the close tracking of the target hit rate at larger values of PID parameters ($K_P = 500,000, K_I = 500,000, K_D = 500,000$) comes at the cost of highly fluctuating space allocation to the class. We discuss this point further in the next section.

Figure 3(d) reports on the results for the retrospective controller. Overall, the hit rate achieved by the retrospective controller does not track the target rate as tightly as the best settings for the linear and PID controllers. However, it is comparable to the other cases of the linear and PID controllers and is much better than the gradient controller. We also notice that the retrospective controller is least sensitive to parameter settings among the four controllers. In the retrospective controller, there is a single parameter β to tune. We find that within a wide range of β , i.e., $0.5 < \beta < 1.0$, the retrospective controller performs reasonably well.

4.1.2 Space allocation

Figure 4 plots the dynamic changes in the space allocation recommended by the various per-class controllers to meet the target hit rate of 40% for the Sprite file trace (i.e., space allocation required to achieve hit rates shown in Figure 3). We observe that the linear and PID controllers exhibit extreme fluctuations in space allocation. More precisely, their cache space allocation oscillates between no allocation to 20,000 blocks while the average cache space required is less than 5,000 blocks. On the other hand, the gradient controller and the retrospective controllers adapt much more smoothly as shown in the figures.

We note that it is desirable to have smooth adaptation in cache space allocations for a few reasons. First, when there are multiple classes sharing the storage cache, extreme variations in cache space allocation are likely to generate more contention, and hence result in poor resource utilization. Second, as previously pointed out, increased cache space does not immediately translate into increased hit rate. Thus, not all spikes in cache space allocation will effectively yield increased cache hits. Furthermore, fluctuations in allocations result in low cache utilization as blocks are transferred around between applications before they are effectively used to answer new requests by any application.

Similar trends are observed for the Worldcup Soccer Web server traces, although the plots are not presented here due to space constraints. From these results, we observe that the linear and PID controllers can track the target hit ratios closely, albeit at the expense of large fluctuations in cache space allocation. Furthermore, their performance is highly

Figure 4. Changes in cache space allocation under the various controllers ($t_i = 0.4$): Sprite trace

sensitive to the choice of controller parameters, confirming our conclusion that they require careful tuning. On the contrary, the retrospective controller is less sensitive to parameter settings, and can track the target hit ratio reasonably well, while experiencing limited fluctuation in cache space allocation. Consequently, when the workload characteristics are well known and rarely change, the linear or PID controllers seem to be the best options. However, when the workload is not known a priori or changes in workload are expected, the retrospective controller offers a more robust approach.

In what follows, we further study the effectiveness of the fairness controller and the contention resolver. For these experiments, each class is associated with a retrospective controller.

4.2 Multiple class scenario

In this section, we present the case of three classes sharing a single storage cache. We assign the following target hit rates to the classes: 50% for class 1, 40% for class 2, and 30% for class 3. Other simulation parameters are summarized in Table 1.

Table 1. Simulation parameters for the multiple class case

parameters	values
target hit rates	50% (class 1), 40% (class 2), 30% (class 3)
cache size	30K blocks (Sprite), 4K blocks (Web)
decay parameter (β)	0.75 (for both)

4.2.1 The fairness controller

We first compare the case when the fairness controller explicitly provides a fairness target for each class versus the case where there is no fairness control. When there is no fairness control, any extra cache space is distributed by the contention resolver, which allocates it in proportion to the cache space requests demanded by the per-class controllers.

Figure 5 shows the results with the Sprite trace. It graphs the normalized hit rate, i.e., the measured hit rates normalized to their target hit rates, and a simple fairness index proposed in [6]:

$$f(h'_i) = \frac{(\sum_i h'_i)^2}{\sum_i (h'_i)^2}$$

where h'_i denotes the normalized hit rate for class c_i . When the allocation is fair, this index is 1, and it decreases as it becomes unfair.

Figures 5(a) and 5(c) show the normalized hit rates of the three classes with and without the fairness controller. We find that the normalized hit rates are closer to each other and thus more fair when fairness control is employed (Figure 5(c)). The difference is clearer in Figures 5(b) and 5(d), which plot the fairness index. Note that the fairness index is closer to 1 when the fairness controller is used.

Figure 6 plots the results for the Web server trace. In this case, the variation in normalized hit rates is smoother. In terms of fairness, the fairness controller significantly improves the fairness index as shown in Figure 6(d), compared to 6(b). In addition, we also find that the fluctuation in hit rates decreases in the presence of the fairness controller. Although not shown here, similar results are observed for the other types of per-class controllers.

(a) Normalized hit ratio (without fairness control)

(b) Fairness index (without fairness control)

(c) Normalized hit ratio (with fairness control)

(d) Fairness index (with fairness control)

Figure 5. The performance of the retrospective controller with and without fairness control: Sprite trace

(a) Normalized hit ratio (without fairness control)

(b) Fairness index (without fairness control)

(c) Normalized hit ratio (with fairness control)

(d) Fairness index (with fairness control)

Figure 6. The performance of the retrospective controller with and without fairness control: World-cup Web server trace

Figure 7. Short term service violation vs. priority schemes

4.2.2 Contention resolver

In this section, we briefly compare the performance of different policies implemented by the contention resolver. To compare the impact of each policy on small time scale performance, we define an index that quantifies the short term service degradation. We call it the *service violation index* or SVI.

$$SVI = \sum_n (t_i - h_i(n))^+$$

In effect, SVI denotes the area below the reference target hit rate curve and above the achieved hit rate curve. The larger the SVI value, the more frequent and significant the short term service violations are for the class.

For evaluation, we set different priority levels, where class 1 is tagged as high priority, class 2 as medium, and class 3 as low priority. For all three classes, their long term service goals are ensured by the provisioning module. However, temporary contention can cause classes to miss their short term target hit rates. Figure 7 presents the case when the contention resolver implements no priority, reactive priority, and proactive priority policies. Recall that the reactive scheme first allocates space to class 1 (the highest priority) and then tries to allocate to class 2 and then to class 3. Contrary to our expectations, however, the reduction in SVI for class 1 is not noticeable. And the improved performance for class 2 comes at the expense of degraded performance for class 3.

On the other hand, the effectiveness of the proactive priority scheme is more visible. We present two proactive cases with different reduction rates. When the reduction rate is selected appropriately, the SVI values for all three classes are smaller than the other cases with the SVI of class 1 is 0. Even when the reduction rate is set sub-optimally, the impact of prioritization is clear and contention mostly affects the lower priority class.

In summary, we find that the fairness controller makes a significant improvement in achieving fairness across classes. In addition, it can also improve overall resource utilization thanks to smoother variations in cache space allocations.

Also we notice that a priority-based contention resolver must be implemented using a proactive mechanism. Due to the delayed impact of cache space allocation on hit rate improvement, a reactive mechanism cannot prevent short term service level violations for high priority classes.

5 Related work

Control theoretic approaches have been applied in the area of network caching to provide differentiated services to clients [7, 8, 9, 3]. Kelly et al. have proposed a weighted cache replacement policy where users with higher weights receive better service than those with lower weights [7]. While their approach provides some level of service differentiation, it does not guarantee any specific service level.

The closest efforts to ours are [8], [9], [3], which apply a formal feedback control approach to provide proportionally differentiated cache hit rates to multiple classes. In [3], the authors model the cache as a single multi-input-multi-output system, and use an adaptive control scheme which adjusts the control parameter according to system dynamics. This solution has been shown to work well with a small number of classes. However, extending this approach to a large number of classes makes it computationally expensive.

The main idea behind our summary MRA list used in the retrospective controller is similar to those of [10] and [11], which estimate the value of caching a particular document by maintaining a long history of how recently and how frequently it has been referenced in the past. The main difference is that our summary list is mainly used to calculate the required cache size in the future rather than making cache replacement decisions.

6 Conclusions

Driven by the economics of cost of ownership, storage systems are increasingly consolidated and shared by a variety of users and applications. As a result, service differentiation in storage systems is becoming increasingly crucial. As storage caches are pervasive throughout the storage system, providing quality of service within a storage cache is a problem of highly practical value. In this paper, we present a modular QoS architecture for achieving specified performance goals for competing application classes accessing a shared storage proxy cache. This architecture consists of three components: (a) per-class feedback controllers that track the performance of each class, (b) a fairness controller that allocates excess resources fairly in the case when all goals are met, and (c) a contention resolver that decides cache allocation in the case when at least one class does not meet its target hit rate. Through trace-driven simulations, we compare the performance of several per-class controllers. In particular, we propose a retrospective controller design that tracks the performance goal without incurring excessive fluctuations in space allocation. Furthermore, we suggest and evaluate effective mechanisms to allocate excess resources fairly among classes and judiciously

distribute resources to ensure that high priority classes do not violate their performance targets at any time.

References

- [1] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West, "Scale and Performance in a Distributed File System," *ACM Transactions on Computer Systems*, vol. 6, no. 1, 1988.
- [2] M. N. Nelson, B. B. Welch, and J. K. Ousterhout, "Caching in the Sprite Network File System," *ACM Transactions on Computer Systems*, vol. 6, no. 1, 1988.
- [3] Y. Lu, G. Tao, and T. Abdelzaher, "LDU Parameterized Discrete-Time Multivariable MRAC and Application to A Web Cache System," in *IEEE 2002 Conference on Decision and Control*, December 2001.
- [4] "The Sprite Operating System." On-line document, <http://www.cs.berkeley.edu/projects/sprite/sprite.html>, March 1996.
- [5] "The Internet Traffic Archive: 1998 WorldCup Web Server Trace." On-line document, <http://ita.ee.lbl.gov/html/contrib/WorldCup.html>, April 2000.
- [6] R. Jain, D.-M. Chiu, and W. Hawe, "A Quantitative Measure of Fairness and Discrimination for Resource Allocation in Shared Systems," Tech. Rep. TR-301, Digital Equipment Corporation, September 1984.
- [7] T. P. Kelly, Y. M. Chan, S. Jamin, and J. K. MacKie-Mason, "Biased Replacement Policies for Web Caches: Differential Quality-of-Service and Aggregate User Value," in *4th International Web Caching Workshop*, March 1999.
- [8] Y. Lu, A. Saxena, and T. F. Abdelzaher, "Differentiated Caching Services; A Control-Theoretical Approach," in *21st International Conference on Distributed Computing Systems*, April 2001.
- [9] Y. Lu, T. Abdelzaher, C. Lu, and G. Tao, "An Adaptive Control Framework for QoS Guarantees and its Application to Differentiated Caching Services," in *10th International Workshop on Quality of Service*, May 2002.
- [10] Y. Smaragdakis, S. Kaplan, and P. Wilson, "EELRU: Simple and Effective Adaptive Page Replacement," in *ACM SIGMETRICS Conference*, May 1999.
- [11] H. Bahn, S. H. Noh, S. L. Min, and K. Koh, "Using Full Reference History for Efficient Document Replacement in Web Caches," in *2nd USENIX Symposium on Internet Technologies and Systems*, October 1999.

Appendix Determining the fairness target

Here we show that the fairness targets computed by Eq. 6 minimizes the deviation of the total space demand of the per-class controllers from the actual cache space.

Suppose we model the hit rate vs cache size relationship as a time-varying linear function for each class, i.e., for class c_i in the n -th round, $s_i(n) = g_i(n) \times h_i(n)$, where $h_i(n)$ is the hit rate, $s_i(n)$ the cache space allocated, and $g_i(n)$ a time-varying coefficient.

In order for the new target hit to be proportional to reference target hit, i.e., at any instance n , $t_i^*(n) = K(n)t_i$ for all i , where $K(n)$ is constant over i but may be time varying. We call $K(n)$ as "fairness margin."

Our optimization goal is, at a given round n , to determine the fairness margin $K(n+1)$ (and thus determine $t_i^*(n+1)$) so as to minimize the difference between the available cache size and the sum of requested cache sizes to meet the fairness target. In other words, we want to minimize

$$\left| \sum_i s_i(n) - S \right|, \quad (7)$$

where S is the total available cache space.

Let $s_i^*(n)$ be the cache space actually allocated to class c_i ($\sum_i s_i^*(n) = S$) by the contention resolver. Assuming that we have a 'perfect' per-class controller which can estimate exactly the right amount of cache space needed to meet a given target, and that the above time-varying linearization model holds, it follows that

$$s_i(n) \simeq g_i(n)t_i^*(n), \quad \text{and} \quad s_i^*(n) = g_i(n)h_i(n). \quad (8)$$

Obviously, the optimization goal Eq. 7 is minimized if $\sum_i s_i(n) = S$, and thus,

$$\sum_i s_i(n) = \sum_i g_i(n)t_i^*(n) = S$$

Since $t_i^*(n) = K(n)t_i$,

$$\sum_i g_i(n)K(n)t_i = K(n) \sum_i g_i(n)t_i = S.$$

Therefore, to minimize Eq. 7, at each round n , a new fairness target hit would be set as

$$t_i^*(n+1) = K(n)t_i = t_i \frac{S}{\sum_j g_j(n)t_j}.$$

However, since $g_i(n) = s_i^*(n)/h_i(n)$ which can be measured from the effective hit rate and allocated cache size, we conclude that

$$t_i^*(n+1) = t_i \frac{S}{\sum_j \left(\frac{s_j^*(n)}{h_j(n)} t_j \right)}. \quad (9)$$

□

Note that the success of this fairness policy depends on the correctness of the time-varying linearization model and the ability to estimate the required allocation to achieve the target performance level. In the QoS architecture proposed in this paper, the latter factor will be decided by the proper design of the per-class controller. On the other hand, the former depends on a good estimation of the hit rate vs cache size function.