Piotr Skowron

Uniwersytet Warszawski

Fuzzy Adaptive Control for Heterogenous Tasks in High-Performance Storage Systems

Praca semestralna nr 1

(semestr letni 2010/11)

Opiekun pracy: dr Marek Biskup

Fuzzy Adaptive Control for Heterogeneous Tasks in High-Performance Storage Systems

Piotr Skowron
9LivesData, LLC
p.skowron@mimuw.edu.pl

Marek Tomasz Biskup
9LivesData, LLC
mbiskup@9livesdata.com

Łukasz Heldt
9LivesData, LLC
heldt@google.com

Cezary Dubnicki
9LivesData, LLC
dubnicki@9livesdata.com

ABSTRACT

Beyond handling user reads and writes, storage systems execute multiple background tasks of various types, such as reconstruction of missing parity data and defragmentation. The resources of the system must be divided between user loads and internal tasks using a specific policy.

This work describes Fuzzy Adaptive Control – an innovative mechanism for sharing resources among various types of highly-variable loads. The new approach uses throughput as the task progress indicator avoiding assumptions about task properties such as resource consumption or handling process. It makes this new technique particularly well suited for complex systems where defining an accurate task model is difficult.

The Fuzzy Adaptive Control is evaluated on resource division between user loads and background tasks in HYDRAstor – a commercial high-performance distributed secondary storage system. The presented mechanism is compared with fair queuing variants and is shown to be more stable in the case of irregular workload. The evaluation proves that our approach is responsive to changing load conditions and ensures high resource utilization.

Categories and Subject Descriptors

D.4.2 [**OPERATING SYSTEMS (C)**]: Storage Management(Secondary storage); D.4.8 [**OPERATING SYSTEMS (C)**]: Performance(Queuing theory); D.4.1 [**OPERATING SYSTEMS (C)**]: Process Management(Scheduling); I.2.8 [**ARTIFICIAL INTELLIGENCE**]: Problem Solving, Control Methods, and Search(Scheduling, Control theory, Heuristic methods)

1. INTRODUCTION

Simultaneously to handling user loads (read/write operations), modern storage systems execute multiple types of background tasks, such as replication, parity checks, parity reconstruction, defragmentation, and garbage collection [19, 22, 23]. Even though user loads has the highest priority, in certain system states, for instance after disk replacement, which triggers software RAID rebuilding, or when the system is almost full, which requires fast garbage collection, background tasks become equally important and are allowed to decrease the speed of foreground tasks. Even in a healthy system, maintenance tasks such as garbage collection and defragmentation should not be starved regardless of their lower priority. This problem motivates our work on a controller that divides resources between user loads and background tasks.

We base our research on HYDRAstor [24, 6] – a high performance distributed secondary storage system that uses erasure coding, data compression and duplicates elimination. Even though we faced the problem considered in this paper in the context of HYDRAstor architecture, we believe the problem and the solution is universal and can be applied to other kind of storage systems, including low-latency primary storage with random data access pattern and smaller units of work for background tasks.

To depict the problem, let us consider the characteristic of tasks in HYDRAstor, summarized in Table 1. Tasks in HYDRAstor are not simple IO operations, but require multiple disk access, significant CPU time and network communication. They are of two categories. The first category is latency sensitive user loads coming in chunks of around 50kB, executed with high parallelism reaching 2500 tasks at a time with relatively high latency, up to 1.5s (but only 150ms for duplicate writes[1] or for reads). The second category is large (up to 100MB) unbreakable background tasks taking even 15s. The size of a single background task cannot be reduced easily because a task processes entire data container (e.g. to sort it) and the reduction of its size or of the unit of work for a task would have a negative impact on performance and implementation complexity.

User loads come from an external application (the HYDRAstor filesystem [29], in this case) and can be highly irregular [9]; tasks may be unavailable for a moment and then appear in large num-

task type	user writes	user reads	bg. tasks
characteristic	highly irregular pattern		unbreakable
parallelism[2]	2500	250	10
latency/duration	1.5s	150ms	up to 15s
size	$\approx$ 64KB	$\approx$ 64KB	1MB-100MB

Table 1: The characteristic of the tasks in HYDRAstor.

[1]HYDRAstor uses data duplicate elimination (see [1, 21]); Writing duplicates may be an order of magnitude faster than writing non-duplicates.

[2]The number of tasks that should be executed in parallel to obtain optimal performance.

bers. Because of high latency (150-1500ms), many tasks must be handled concurrently to achieve high bandwidth. The concurrency must be adjusted carefully at run time in response to changes in workloads and in external conditions in order to keep the latency below a predefined level (which is required, for instance, by the filesystem prefetching algorithms [29]). Background tasks are created inside HYDRAstor based on a schedule or on automatic problem detection (e.g. disk or node failure).

The paper addresses a general problem of scheduling heterogeneous (background and user) tasks in storage systems with high concurrency and latency control. The tasks are heterogeneous in the sense described above: (i) they use different and multiple resources, (ii) they have diversified size and (iii) they have different patterns of availability. To the best of our knowledge, this problem has not been addressed before – most literature on proportional resource allocation considers tasks of similar characteristic, or tasks using a single resource [11, 30, 16, 17, 31, 10, 3].

As we will show below, it is the heterogeneity, namely, the long duration of background tasks combined with irregularity of user loads, that makes the most common queuing techniques [2, 8, 14] infeasible for scheduling work in this setting. In the lack of user tasks, work-conserving queuing mechanisms always admit heavy background tasks, burdening the system significantly and leaving no resources for upcoming user tasks. Slow responsiveness of the storage system affects the application, which, in turn, cannot provide the system with sufficient amount of user tasks, amplifying the problem. To keep the system ready for admitting highly irregular user loads it is necessary to delay background tasks even in the presence of unused resources. This may lead to underutilization of resources, but we are willing to pay this price to guarantee quality of service to the user. Our new mechanism, unlike fair queuing, controls the speed of admitting various tasks and smoothly changes these speeds. This way, we avoid burdening the system with heavy background tasks whenever user loads are temporarily unavailable.

There are two additional problems that come along with task scheduling in HYDRAstor and similar systems. Firstly, HYDRAstor is a distributed system, which introduces additional complexity to the scheduler. In order to maintain simplicity, we present a mechanism for controlling local tasks and later we show that with some constraints on the configuration parameters of our control mechanism, the local instances of the Fuzzy Adaptive Control working independently provide that the system works correctly as the whole. Secondly, in a complex distributed storage system it is virtually impossible to measure precisely the resource consumption of various tasks, especially that they are handled by many software components on many servers. We overcome this difficulty by scheduling user tasks and background tasks according to a desired proportional division of observed throughputs (the amount of data processed per second) – *throughput shares*. This solution is motivated by the fact that these proportions reflect the proportions of actual work done by various loads, which is of more interest to users than low level details, such as resource consumption (c.f. [26]). Similar assumption was made in [11, 20, 17].

This paper has the following contribution: (i) We introduce a novel Fuzzy Adaptive Control mechanism enforcing proportional division of load throughputs while maintaining high resource utilization. (ii) We present how it can be used in a distributed environment. (iii) We have implemented the new mechanism in a commercial high-performance distributed secondary storage system – HYDRAstor [24, 6]. The described control mechanism has been used for two years by real-world customers. (iv) We evaluated the new mechanism on a 60-server configuration with 480TB raw capacity (that is 10PB with 95% duplicate elimination ratio) achieving 10GB/s write speed, which makes it one of the biggest and fastest global dedupe appliance in the world up to date.

2. RELATED WORK

The problem of scheduling tasks in storage systems, distributed or not, is common and frequently described. Most available literature addresses the problem of sharing resources among tasks of the same type [11, 30, 16, 17, 31, 10] or scheduling packets of known size [3]. In contrast, our work addresses the issue of dividing resources among user load and background tasks.

Scheduling methods for heterogeneous tasks are fairly more complicated [27]. Most of them use one of queuing mechanisms: YFQ [2], SFQ and FSFQ [8, 14], or their modifications [30, 31, 3, 35, 10]. In these solutions, tasks are put into queues and then sent to the system in a way that achieves the desired division of throughputs. These basic solutions suffer from a fixed level of concurrency and therefore, in case of server performance drop or changes in workload (e.g. heavier tasks), are unable to respect latency requirements. Some solutions combine the application of the queuing theory with a feedback loop that controls the latency by adjusting the number of concurrently processed tasks [33, 13].

Queuing approach is most often followed in the available literature and in most of regular cases was proven to give fair throughput division with almost zero settling time. We address these solutions (as the most important ones) in context of HYDRAstor, which must accommodate strong irregularity of user tasks. The evaluation presented in Section 3 shows that queuing approach does not give correct resource division in such case; this fact motivated us to introduce our new Fuzzy Adaptive Control.

New trends in the application of the control theory encourage to use model-based solutions [35, 12, 18, 34, 4]. Feedback controllers can be used for ensuring proportional resource division. One controller may regulate the speed of work of the load sources (input signal) in order to obtain the desired proportions of their throughputs (output signal). Another controller may regulate the concurrency level (the number of user tasks handled in parallel). Finding a proper model, however, requires identifying how the presence of each task influences the performance of the others. With three kinds of user tasks (writes, reads, duplicate writes), several maintenance tasks, and a few data deletion phases[4], the model would have many dimensions. Moreover, feedback controllers using linear (or even dynamic) models are less adequate for optimization problems like optimizing resource utilization. Control theory gives us well studied methodologies for ensuring stability and the best settling time of a controller but with the complicated architecture of our system (see [24, 6]) and, in consequence, with the difficulty of modeling it accurately, we decided to consider different possibilities of ensuring proportional resource division.

Resource division between tasks of different nature can also be accomplished through virtualization [32, 15, 25, 20, 26]. Each virtual machine hosts a single application performing an appropriate class of tasks. Although virtual machines allow for accurate division of CPU cycles between applications, such approach results in architectural and implementational limitations (the tasks of a particular category should be executed by a single software component since they need to be run on a separate virtual machine) and introduces performance overhead [28].

Other existing solutions try to execute background tasks in idle periods [22, 7, 23] but such methods are inadequate when servers

[4]Data deletion is a process of marking blocks of data for removal. In our system data deletion can be called on demand and we treat it as a special kind of background task.

constantly handle user tasks and when background tasks can effectively be executed in parallel to user activity.

A problem of running background tasks and avoiding resource contention, but in the context of a standalone process in an operating system, was considered in [5] and in articles cited therein. Interestingly, in [5], the contention is detected without finding the actual bottleneck, as in our solution; also the progress of tasks is measured instead of their resource consumption.

3. QUEUING EVALUATION

Heterogeneous tasks exhibit challenges that make standard work scheduling approaches surprisingly wrong. In this section, we present the nature of these problems in the most common approach (see Sections 1 and 2) – queuing. In queuing, tasks of different categories are put into separate queues. The scheduler selects one queue using some policy, and executes the first task from the selected queue. Specific mechanisms differ in the way how queues are selected; for instance, the goal in our case is to achieve desired throughput proportions.

For our analysis, we use a simplified model that hardly captures the complexity of HYDRAstor, but nevertheless, precisely exposes the root cause of the problems of scheduling irregular tasks using a work-conserving scheduler. Although our model was based on HYDRAstor, we believe that the presented problems are relevant for a wider class of systems (and not only for storage systems).

3.1 System Model, Task Arrival Model and Simulator

We model one category of user loads (write tasks) and two categories of background tasks: type I (small) and type II (big). Irregularity of write arrivals [5] is shaped as batched arrivals with duration of *x* milliseconds, with interarrival times within a batch sampled from Poisson distribution, followed by a period of unavailability with duration of *y* ms. Such pattern is denoted by (*x* ms / *y* ms). We assume that background tasks are always available.

Next, we model the system as a black box which processes many tasks at the same time, where the task processing duration depends on the current concurrency level – the number of requests (tasks) in the system, RIS. Figure 1 presents the (simplified) dependency between system throughput and RIS for user writes in HYDRAstor, which we directly use to model duration of write tasks (to add some stochastics, we sample task duration from a normal distribution centered at the value taken from Figure 1). As we can see, increasing the concurrency level increases system throughput. But beyond some point, submitting more tasks does not result in significant performance improvement and only increases the latency.

Different categories of background tasks affect the overall performance in a very different way and, as indicated in the previous section, modeling the interdependencies of tasks accurately is almost impossible. Nevertheless, we model background tasks in the same way as write tasks, except that they are heavier (one type I task is equivalent to 5 RIS, and type II – 25 RIS), and longer (type I – 3 times the latency from Figure 1, and type II – 10 times). These parameters were also measured in HYDRAstor. The characteristic of the modeled tasks is summarized in Table 2.

The topic of this paper is a work scheduler which lies between the model of tasks and the model of the system, as well as between real tasks and a real system. The scheduler divides work between the three task types according to given throughput shares, and controls RIS in order to maintain a predefined write task latency. We

[5]This scenario may correspond to copying files between two storage systems.

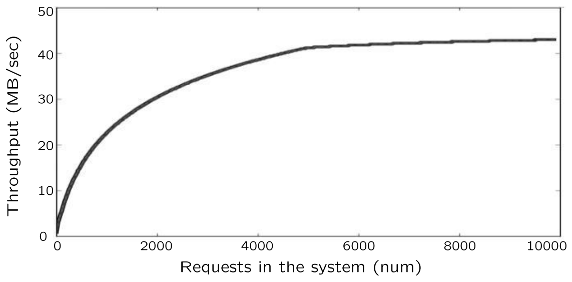

Figure 1: The relation between the number of tasks in the system and the system throughput in the system simulator.

task type	write	bg. I	bg. II
weight	1	5	25
avg. duration	$1d^6$	$3d$	$10d$
pattern	batches	continuous	
batch pattern	Poisson pattern	—	

Table 2: The characteristic of the tasks in simulator.

used the latency of 1500 milliseconds, taken from HYDRAstor, which is reached at 2500 RIS (e.g. 2500 user writes or 1000 user writes and 300 background tasks of type I). The work scheduler is evaluated using a system simulator, that simply calculates (or rather draws) the duration for a given task at the current RIS, without performing any I/O operations.

This simplified model is enough to show that standard queuing techniques suffer for imbalanced work distribution and penalize user loads for its irregular availability pattern (Section 3.2).

3.2 Evaluation results

In this section we present the evaluation of a queuing scheduler enriched with feedback RIS controller, similar to the one presented in [33]. We repeat this exercise in Section 7.1 for our Fuzzy Adaptive Controller to show that it works better.

The queuing scheduler takes tasks (if available) from task sources and puts them into separate queues, depending on task type. It uses a simple feedback controller to compute appropriate RIS level (called *slots*) given the referral latency. The difference between RIS computed by the controller (slots) and actual RIS is called *free slots*. If the number of free slots is greater than zero, a task can be taken from some queue and sent to the system. Write tasks occupy one slot (increase RIS by one), tasks I – 5 slots and tasks II – 25 slots. Proportional throughput division is ensured by choosing tasks from appropriate queues. We use 1:1:1 proportions in the evaluation.

The queues in this scheduler mitigate the effects of irregularities in workload. We tried several small sizes of queues: 50, 100, 300 tasks, because large queues require significant amount of memory and, what is even more important, increase the average task latency beyond the value actually maintained by the controller – a queue of the size 1000 write tasks would increase the write task latency by about 600ms (in addition to 1500ms). Using each of the aforementioned sizes gave similar effect so the results are presented only for the size 100.

If user tasks do not come in batches, but instead are produced

[6]*d* denotes the average duration of a single user write that depends solely on RIS – the total weight of the tasks currently processed by the system. This dependency is presented in Figure 1.

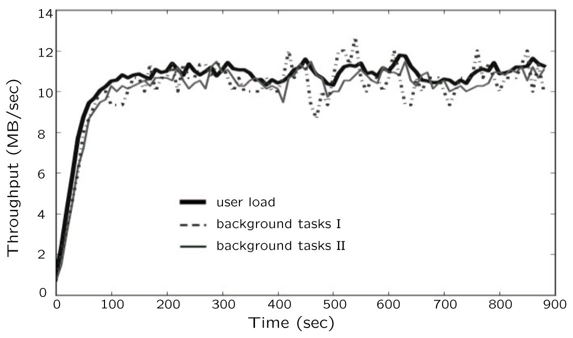

Figure 2: Throughputs of user writes and two categories of background tasks on the system simulator with queuing using a controlled slots mechanism. All three load sources are working full speed. The fluctuations are the result of short period of averaging.

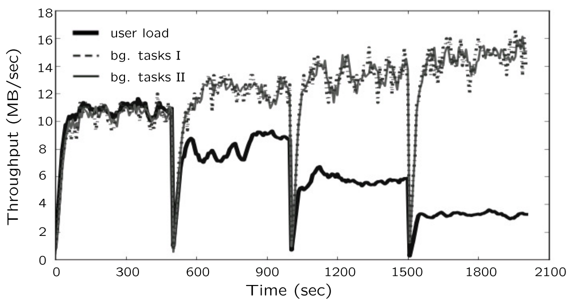

Figure 3: Throughputs of user writes and two categories of background tasks on the system simulator with queuing using a controlled slots mechanism. User writes (thick solid line) are irregular. The expected throughput proportions are: 1:1:1. The four cases (300ms/50ms; 300ms/150ms; 1s/300ms; 500ms/500ms) are presented sequentially on one plot.

continuously, then the queuing mechanism is very accurate and stable, as shown in Figure 2. An irregular write pattern, which is more relevant for real systems (c.f. [9]), is problematic. We analyzed four cases: (i) (300ms/50ms) (ii) (300ms/150ms) (iii) (1s/300ms), (iv) (500ms/500ms); and found that throughput proportions were far from being equal[7]. The reason was that during write task unavailability, the write task queue was emptied and the scheduler started many background tasks which occupied slots and prevented write tasks to start during busy periods (if write latency is 1.5s, tasks of type II take 15s).

It may seem that long-term throughput division in the queuing scheduler can easily be improved. We tried to enrich queuing with memory of past throughput, so that when user writes appear after a quiet period, they are preferred till they are compensated for the previous inactivity period. The results are shown in Figure 3.

Surprisingly, even with such improvement, the throughput of user writes was up to 4 times lower than expected. The only case when the queuing mechanism worked as desired was (i) (300ms/50ms). This is because 50 milliseconds of quiet period is not enough to empty the queue of write tasks and, as system is not short of user writes, the background tasks are under control. In cases (ii), (iii) and (iv), the throughput of writes is lower than of background task. Even though the scheduler gave the priority to write tasks, they were not able to catch up with background tasks. Again, this is caused by slots occupied by background tasks. When write busy period finishes, the write queue is emptied even more eagerly than in case of the memory-less scheduler, and again, the scheduler will start background tasks that will occupy slots during the next busy period.

Our experiments have shown that because of the interaction of several parameters: the duration of busy and quiet periods, queue lengths[8] and the task latency, the throughput of write and background tasks cannot be kept at the given proportions by the queuing scheduler (instead of expected 1:1:1 proportions in some cases we obtained 1:5:5). The fundamental problem of this scheduler is being work-conserving, i.e. tasks are sent to the system immediately if resources (slots) are available. We argue that in case of irregular user workload and large background tasks, no work-conserving mechanism can ensure proper resource division[9].

4. FUZZY ADAPTIVE CONTROL

4.1 Architecture

In the previous section, we analyzed a work-conserving, central scheduler which divides resources between tasks from several queues, and we proved it does not work for irregular user load. As a consequence, we prefer a non-work-conserving scheduler that reserves certain bandwidth for upcoming write tasks, even in the presence of free resources, or more precisely, in the presence of free slots, as we want to keep the RIS-latency controller.

A natural approach to this problem is to split the controller into two layers, as pictured in Figure 4. The upper-layer, hereinafter called *controller*, will divide the total throughput between task categories, taking into account their actual needs and the configured throughput shares. In the cases presented in Figure 3, where the shares are 1:1:1, the controller will allocate the same throughput for each queue. The lower layer will consist of one scheduler, called *throttler*, per queue, which keeps the bandwidth of its queue on the level ordered by the controller.

Considering the aforementioned problems, we want the controller to maintain a stable throughput of each throttler, and to increase or decrease the throughputs gradually, depending on the availability of tasks and on system utilization. In this way, in the timescale of write task fluctuations, the bandwidth of background tasks will remain almost constant. During a temporary unavailability of write tasks, free slots will be preserved for upcoming tasks.

This approach generates two problems: (i) to divide the throughput between throttlers, the controller has to know the total system throughput available, which is nontrivial as it depends on the current conditions and may be highly irregular; (ii) gentle controller reaction may lead to system underutilization if system conditions change rapidly, for instance, if background tasks of type II suddenly run out and free resources appear in the system.

To address these problems we modify slightly the natural approach. Firstly, we move the RIS-latency controller to the lower layer, to the write task throttler. As a result, it controls the user load only, but not the background tasks. It may seem that this change

[7]We do not present the plot with these results because it is very similar to Figure 3, where we presented an improved version of queuing (see next paragraph).

[8]Increasing the capacity of the buffer will improve accuracy of the queuing mechanism but too big buffers are unacceptable, as mentioned before.

[9]We do not give a rigorous proof here. The irregular workloads described in the previous paragraph, together with short queue sizes, can serve as a counterexample to other work conserving schedulers as well.

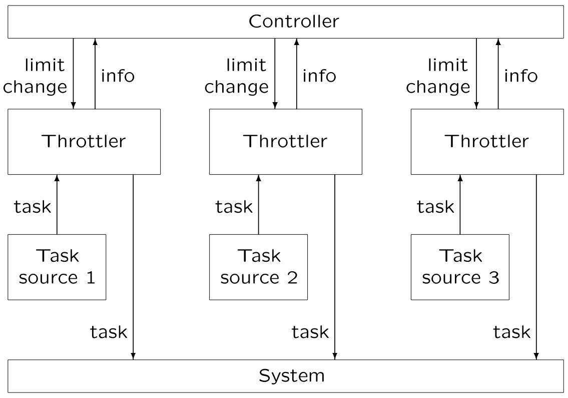

Figure 4: The high-level mechanism of adaptive resource management.

makes the RIS controller unable to respond to starting or finishing background tasks, but it is not the case. Running background tasks increase system utilization, which in turn increases the latency of write tasks and results in the reduction of RIS. When background tasks finish, the latency decreases, and RIS controller increases the number of slots. So even though background tasks do not occupy any slots, they reduce the number of available slots, which gives a similar effect. This change does not bring any major value by itself, but is required by the second change.

Secondly, we change the way the write throttler is controlled by the upper-layer controller. Instead of the throughput, we use an upper bound on RIS, which according to Figure 1, indirectly affects the throughput. In contrast to the natural approach where write throughput was upper-bounded, in this modified approach, write tasks are started whenever there are free slots. If the system becomes underutilized, the latency drops and the write throughput increases even at the same RIS level. Therefore, this choice of the control variable reduces the underutilization of the system, which is particularly important for our solution because the controller must converge slowly in order to maintain background task throughput constant during fluctuation time scale of user load. The property of using spare resources by available write tasks will also help the upper-layer controller to estimate the total available throughput of the system and divide the throughput accordingly.

Finally, we wrap the solution in a general form (Figure 4) that abstracts from the actual way the throughput of each throttler is controlled. Namely, each throttler takes a parameter, called *limit*, such that the throughput of tasks is a non-decreasing function of the limit. In our example, the limit is the upper bound on RIS for write tasks and the throughput, directly, for background tasks. Periodically, the controller orders each throttler to *increase*, *decrease* or *do-not-change* its limit, depending on the configured throughput shares, on the current throughput of each task type and on the current condition of the system, for instance whether the system is over- or under-utilized. The decisions are then translated by each throttler to a particular value of its limit. Even though we presented only two proposals for the limit, RIS and throughput, our algorithm can be applied to other choices.

The conditions of the system are inferred from descriptive information, hereinafter called *info*, provided by each throttler, for instance, whether tasks are available, or whether the *limit* is really slowing down the tasks (the limit on throughput may not slow down background tasks if it is set too high and the actual throughput is capped by system resources). Thanks to this information we are able to find the reason why some type of tasks is slower than expected: (i) because the limit is too low, (ii) because the limit of other tasks is too high and there are no resources in the system, (iii) because there are not enough tasks of this type. The linguistic form of this information and of the decision, translated at some point to numerical values, prompted us to use the word *fuzzy* in the name of our solution.

The high-level view of the architecture is pictured in Figure 4. The controller periodically (the interval is chosen to satisfy the constraints presented in Section 6 – for HYDRAstor it is 500 ms) receives an *info* from each throttler and using this information orders a particular changes of the limit of throttlers. Each throttler translates the limit change into a particular limit value. The throttlers take tasks from their sources and send them to the system, throttling them using the computed limit.

This fuzzy controller seems to be complicated and not well founded mathematically, but it cannot be easily replaced by standard MIMO feedback controllers (see [12]) because linear controllers can be used to maintain the throughput at a predefined level, but not to maximize the total throughput. In particular, they cannot guarantee that the given throughput levels are achievable (if some task type it too slow, it may not be sufficient to increase its limit, but limits of other task types must be decreased). Also standard linear controllers require an accurate model of the system, which is too hard for a complex system like HYDRAstor. Finally, designing a linear feedback controller to work correctly in a distributed environment seems really hard.

4.2 Algorithm overview

The controller has two responsibilities: finding the desired throughput (called *target throughput*) for each throttler, and adjusting limits of throttlers towards their target throughputs. The controller uses information from the throttlers (*info*), which will also be described here.

To find the target throughputs, the controller has to take the maximal system throughput available and divide it between all task types according to the configured shares. The maximal throughput cannot easily be found, so the controller uses the sum of the current throughput of each task type (*info* value: *throughput*). To keep the resources fully utilized, the controller tries to increase the throughput of all task types whenever possible. Even if the current throughput does not utilize all resources, the system will reach maximum utilization after a few steps, and the total throughput will closely approximate the maximum available throughput. Using a RIS bound as the limit for write tasks helps to adjust the total throughput faster (if only user load is available) because write tasks are started whenever slots are available, regardless of the current throughput. The division of the total throughput between throttlers should take into account whether loads indeed have tasks to be processed (*info* value: *needsResources*, a boolean value). For instance, if an external application provides write tasks at speed of 100 MB/s, the controller is not able to increase write throughput above this value. In such case, the controller must divide all remaining throughput, if any, between other task types.

As an example, assume that the throughput is supposed to be divided in 1:1:1 proportions, the throughput is 100 MB/s for writes, 200 MB/s for tasks I, and 300 MB/s for tasks II, and that tasks I and II are always available, but write tasks come at 100 MB/s only. In this case, the total system throughput is estimated to be 600 MB/s. Taking into account 1:1:1 division, the controller should allocate 200MB/s for each task type (this is the *policy throughput*). But as writes are not available above 100 MB/s, the target throughputs will be 100 MB/s for writes and 250 MB/s for tasks I and 250 MB/s

```
Input: policyShare[] – an array of shares per task type (sums up to 1),
       info[] – information collected from throttlers
Output: decision[] – an array of decision per task type
        (INCREASE, DECREASE or DO_NOT_CHANGE)
1  decision[] ← [DO_NOT_CHANGE, DO_NOT_CHANGE, ...];
2  someoneWasIncreased ← false;
3  someoneNeedsResources ← false;
4  totalThroughput ← sum(info[].throughput);
5  policyThroughput[] ← policyShare[] × totalThroughput;
6  someoneNeedsResources ← (true in info[].needsResources);
7  targetThroughput[] ← recountTargetThroughputs(
8      policyThroughput[], info[]);
9  // Try to increase limits
10 foreach tasks in info[] do
11     if (tasks.needsResources = true) and (tasks.limitAchieved
       = true) and (currentThroughput[tasks] ≤
       targetThroughput[tasks]) then
12         decision[tasks] ← INCREASE;
13         someoneWasIncreased ← true
14 // Try to decrease limits - only if no
       limit was increased
15 if (not someoneWasIncreased) and someoneNeedsResources
   then
16     foreach tasks in info[] do
17         if (currentThroughput[tasks] ≥
           targetThroughput[tasks]) then
18             decision[tasks] ← DECREASE;
19 foreach tasks in info[] do
20     throttler[tasks].passDecision();
```

Algorithm 1: The algorithm of the controller.

for tasks II (the unused 100MB/s of write throughput was divided between tasks I and II).

In the second phase, the controller compares the current throughput of each task type to the target throughput, and orders a change depending on the result. It has to take into account that some task types may work slower than requested not because their throttler limits them, but because the whole system has insufficient resources. The information whether a task type has insufficient resources (*info* value: *limitAchieved*, a boolean value), is inferred by each throttler from the limit set by the controller and from the actual value that is controlled by the limit. For instance, if the limit for the throughput of tasks I is 200MB/s but they achieve only 100MB/s because of insufficient resources, their limit is not achieved and increasing it will not help; similarly, if the limit for RIS is 2300, but the RIS-latency controller maintains the level of 1900 RIS because higher RIS would cause the latency exceed the referral one, the limit is not achieved either. Note that the limit is usually not achieved also in case the task type does not need resources.

Here is a summary of the contents of an *info*:

- *needsResources* — indicates if the tasks can work faster given more resources,
- *throughput* — the average throughput generated since the previous *info* was collected,
- *limitAchieved* — indicates whether the task type has achieved its *limit* since last measurement.

4.3 The algorithm

The upper-layer controller is presented in Algorithm 1. As input, it takes the information collected from the throttlers (*info*) and the policy that specifies the share for each task type. Each array in the pseudo-code is indexed with task types (in our case, there are three of them: writes, tasks I and tasks II, but the algorithm is general).

In Steps 1-6, the algorithm initializes auxiliary variables. In Step 7, the algorithm computes the target throughputs. Task types that do not need resources are left only with the throughput they achieve, and the remaining part of their policy throughputs is distributed among task types that need resources. To be more precise, if a task type T does not need resources and its current throughput is lower than the policy throughput ($needsResources = false$ and $currentThroughput < policyThroughput$), then the task is not able to use its entire policy throughput, so:

$$targetThroughput[T] \leftarrow currentThroughput[T]$$

Otherwise ($currentThroughput \geq policyThroughput$ or $needsResources = true$) the target throughput is set to:

$$targetThroughput[T] \leftarrow policyThroughput[T]$$

In the first case, the spare throughput:

$$policyThroughput[T] - currentThroughput[T]$$

is divided between other task types according to their policy shares.

We say that a task type T is eligible to get more resources if it needs resources and $currentThroughput[T] \leq targetThroughput[T]$. The algorithm tries to increase the speed of such a task type, either by increasing the limit for this type of tasks (if the task source achieves the given limit) or by decreasing the limit for other types (if it does not). At least one of the task types that need resources is eligible to get more resources, which is formally expressed by the following invariant:

LEMMA 1. *If there are task types for which $needsResources = true$, then at least one of them has $currentThroughput \leq targetThroughput$.*

PROOF. All types of tasks which do not need resources ($needsResources = false$) have $targetThroughput \leq currentThroughput$ (consider two cases separately: $currentThroughput < policyThroughput$ and $currentThroughput \geq policyThroughput$ in the rule of computing *targetThroughput*). This implies:

$$\sum_{\substack{taskType\ \boldsymbol{T}:\\ needsResorces=false}} targetThroughput[T] \leq \sum_{\substack{taskType\ \boldsymbol{T}:\\ needsResources=false}} currentThroughput[T]$$

Additionally, the sum of all target throughputs is equal to the sum of all current throughputs (both are equal the total throughput), so:

$$\sum_{\substack{taskType\ \boldsymbol{T}:\\ needsResources=true}} targetThroughput[T] \geq \sum_{\substack{taskType\ \boldsymbol{T}:\\ needsResources=true}} currentThroughput[T]$$

Now it is clear that $currentThroughput \leq targetThroughput$ for at least one task type with $needsResources = true$. □

System saturation is a state when a task type T has *needsResources* = ‘true’ and *limitAchieved* = ‘false’: T has work to do, its *limit* allows for faster speed, but it cannot proceed faster because of insufficient resources in the system. In order to increase its speed of work, the *limit* for other task types should be decreased. On the other hand, if the system is not saturated (each task type either does not need resources or achieves the given *limit*) the algorithm will increase the *limit* for a task type, which is specified by the following lemma:

LEMMA 2. *If there is work in the system (at least one task type needs resources), then either the system is already saturated or the limit for one of task types will be increased.*

PROOF. Let us assume there is some work in the system and consider task types which need resources. Either one of them has *limitAchieved* $= false$, which means the system is already saturated, or every task has *limitAchieved* $= true$. In the second case, we can use Lemma 1 and infer that there is at least one task type T which satisfies all of the following:

$$needsResources[T] = true$$

$$limitAchieved[T] = true$$

$$currentThroughput[T] \leq targetThroughput[T],$$

so the limit for T will be increased by the algorithm. □

5. IMPLEMENTATION ISSUES

The structure of the Fuzzy Adaptive Controller is fairly simple, but there are a few details which need to be mentioned. First of all, throughput information from the throttlers may fluctuate highly, especially if background tasks are large. For the mechanism to be stable, the algorithm requires relatively stable reports. Therefore some averaging is necessary. In HYDRAstor we used moving averages. Secondly, we did not describe defuzzification of the limits. The implementation in HYDRAstor is sketched in Section 5.1, but other implementations are possible. Note that this is an important part of the whole solution as the convergence speed and the stability of the controller depends on it. We did not prove the stability of the controller, but an inherent property of our solution is that the changes of the limits are gradual, therefore fluctuations should not be large.

5.1 Defuzzification of the limit

The decision made by the controller, as described in Section 4.3, needs to be converted from the linguistic form: *increase/decrease/do-not-change* to a particular change of the *limit* (defuzzification). In HYDRAstor we used a simple implementation where the current throughput level is multiplied by a constant greater, smaller or equal one, respectively. We added one optimization – when a throttler receives the second or the third (or further) decision in the same direction, the constant is larger/smaller, so that the changes are faster.

5.2 Defining throughput shares

The topic of defining throughput shares are out of scope of this paper, but we will give an overview of what is done in HYDRAstor. Firstly, the shares depend on the current state of the system – for instance, the share for background tasks is higher if resiliency of data is lower (more parity data is missing) or if free space is running out (garbage collection should work faster). Secondly, defining throughput shares is easy for engineers but not for users, so our system provides the user with a set of profiles, each of them defining the shares for each state of the system. The profiles have intuitive names (such as "resiliency" for reconstruction background tasks to work faster, or "performance" to give the priority to user load) and can be easily chosen by the user.

6. DISTRIBUTED ENVIRONMENT

In this section we describe how our Fuzzy Adaptive Control behaves in a distributed environment, where tasks are handled by many servers. Again, we base our system model on HYDRAstor, which is a content addressable storage. In HYDRAstor, each write task, which corresponds to a block being written, is delegated to a particular server, depending on the hash value of the block. As hashes can be considered random, all servers are equally burdened with user write tasks. A server which is slower than others (for instance, because it devoted some amount of its resources to background tasks), will affect other servers because they will receive write tasks with the same rate. On the other hand, background tasks in HYDRAstor may be local, executed on a particular server, or distributed, occurring on several servers, not necessarily all of them.

Scheduling tasks in such environment is hard because even a local background task may reduce the speed of write tasks on the server where it is executed, which, in turn, reduces the overall write speed and leaves unused resources on other servers. If these servers start too many background tasks, they may further reduce the overall write speed and the scenario is repeated.

We model a distributed system with tasks delegated to multiple servers [30, 6] in a similar way. Such remote tasks are organized in a sequence where the i-th task should be executed at a specific server $s(i)$. The tasks are sent to the servers in the ascending order of their numbers. In this model, as in HYDRAstor, slow execution of task on one node may influence their speed of execution on other servers. Although the server $s(i+1)$ may have free resources, it will not be given the task $i+1$ for execution until $s(i)$ admits the previous task i. As a result, the speed of execution of the tasks is bounded by the speed of their execution at the slowest server. For simplicity, we assume that background tasks are all local (which is not the case in HYDRAstor).

Fuzzy Adaptive Control can be used in such distributed environment with a separate instance (i.e. with an independent controller and throttlers) working on each server independently. As in the local case, and in fact it is a local case from the point of view of the controller, when the user load at a particular server does not achieve its target throughput (even if the speed of the user load on this server adjusted to the speed at the bottleneck server), the control mechanism will allocate the unused resources to local background tasks, and take advantage of the unused resources. (Recall that *infos* describe only the local state of the tasks and the user load which is slowed down by a remote bottleneck server, reports *needsResources* false, indicating there is space for different loads at the server.) However, the controller must guarantee that background tasks will not influence the speed of user loads on the server. Ensuring such guarantees is not straightforward as overshoots and fluctuations in load characteristic are common. In the subsequent subsections, we present an analysis of consequences of careless recapturing unused resources. Then, we introduce constraints on the controller wake-up interval that guarantee the performance of the user load is not affected by the local load let into the system to recapture unused resources.

6.1 Recapturing unused resources

In a distributed environment, the decrease of the limit for a user load at any server, s, makes the user load slow down at each server. After such decrease the unused resources may appear on another server s'. Such server is not a bottleneck for a user load so it infers that it may safely increase the limit for local background tasks; because overshoots and fluctuations are possible, the increase of the limit for the background tasks may make this server become a new bottleneck for a user load, further decreasing its global speed. It may happen that in consecutive steps the servers will alternately increase the throughput of the local tasks, eventually causing the starvation of the user load. The local tasks are not throttled because in each time step one of the servers sees that the user load does not need local resources (because it is not a bottleneck for a user load at the moment), causing another decrease of its throughput in the next step.

To prevent this problem we introduce an additional constraint on the controller wake up interval. With the given constraints, after increasing a limit for background tasks, the server s' from the example above will not become a bottleneck for any other load (here, for user load) at least till the next time the controller wakes up. In the next step after the increase of background tasks, even if the user load on s' indicates that it needs local resources, the server will still have abilities to accommodate some additional incoming load. E.g., after increase of the background tasks the user load latency may become close to the neutral one, so *needsResources* would be true, but the latency would not yet exceeded the neutral one, so the tasks would not be rejected nor the number of slots would be throttled down. Because the user load indicates it needs resources, the limit for background tasks which was previously increased will be throttled down in the next step back to its previous value, causing no degradation of the user load speed.

6.2 Constraints on wake up interval

Consider the implementation of *needsResources* based on whether there is a task waiting in a buffer. The server is able to admit tasks unless the buffer is full. Let w_a be the controller wake up interval, $size_b$ is the size of the buffer and T_{max} is the maximum throughput of the source; and assume $w_a < \frac{size_b}{T_{max}}$. With such assumptions if *needsResources* was *false* then, after w_a, the buffer will not be full, which results in no slow down in tasks admission.

The constraints on w_a force the desired property: if the tasks of a distributed load (here, user load) do not need resources, and so the local tasks are let in, the speed of the admission of the distributed tasks will not be affected till the next wake-up of the controller.

With such constraints, even with no coordination between the servers, the control mechanism will enforce correct proportions of the throughputs on at least one loaded server. If the servers are homogeneous (like in case of HYDRAstor) each server will hold the correct proportions.

7. EXPERIMENTAL EVALUATION

In the first part of this section we present the results of experiments on the simulator described in Section 3.2. The second part describes experiments conducted on HYDRAstor.

7.1 Artificial workload

Figure 5 presents results of the (500ms/500ms) experiment conducted on the simulator with the Adaptive Fuzzy Controller (other three scenarios, (300ms/50ms, 300ms/150ms, and 1s/300ms, gave similar results, so we present only the worst case). The throughput values, averaged over 30s intervals, show significant fluctuations, which is an inherent property of the system model. If we used 300s averaging period, we would see that the proportions of throughput differ from the expected 1:1:1 by no more than 3% after the initial phase. The long 170s stabilization phase is a price we pay for the stability – the throughput of background tasks (for each type of tasks independently) is being slowly and steadily increased until it evens with the throughput of user load.

7.2 HYDRAstor

Further experiments were conducted on HYDRAstor. Apart from user requests (writes, duplicate writes and reads) it has two main classes of background tasks: data deletion (marking blocks for removal), and maintenance tasks (reconstructions, defragmentation, garbage collection, etc.) – corresponding to background tasks of type I and II analyzed throughout the paper. A single server can execute concurrently all three classes of tasks. The tasks, even though handled by different software components, use the same resources, such as disks, CPU, memory or network.

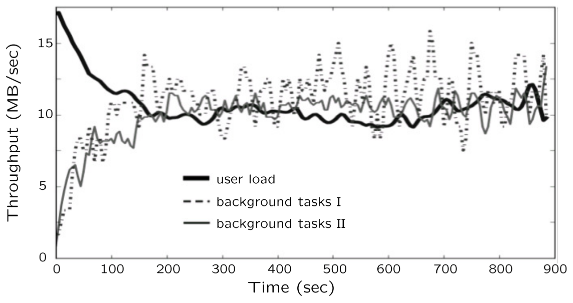

Figure 5: The throughputs of user writes and two categories of background tasks on system simulator using fuzzy adaptive control mechanism. User writes (thick solid line) are irregular (500ms/500ms).

	user load	deletion	maint.
Critical reconstruction	20%	10%	70%
Critical garbage col.	20%	35%	35%
Normal reconstruction	50%	30%	20%
Garbage collection	50%	30%	20%
Regular tasks	70%	30%	0%

Table 3: Examples of policies. The throughput shares depend on the state. For example, critical garbage collection takes effect when free space in the system is low.

The throughput shares for the task categories in the experiments were changing dynamically dependently on the state of the system. For example, after failures that significantly reduce the resiliency level of some data, there was the critical reconstruction policy in effect, which gave significant share to background tasks. Example policies correspond to particular states of the system. The Fuzzy Adaptive Control always uses the policy for the current state.

7.2.1 Experimental setup

For the experiments, we have used HYDRAstor in two configurations: (1) a 60-server configuration and (2) a 6-server configuration. The latter was used because our access to the 60-server configuration was limited. Also, by using two different sizes, we show that our solution works in different scales, in particular in a large system where tasks, especially data deletion, are more irregular (compare Figures 8 and 9).

Each storage node had two quad-core 64-bit 3.0 GHz Intel Xeon processors, twelve 7200 RPM Hitachi HUA72101AC3A SATA disks and 24GB of memory. Each server held two logical storage nodes, with 6 disks each (for details about HYDRAstor architecture see [24, 6]). The servers were connected by the 4x1Gb network interfaces.

7.2.2 High resource utilization

The aim of the first experiment on HYDRAstor, performed on the smaller system, was to show that the Fuzzy Adaptive Control keeps high resource utilization, in particular, if user tasks do not consume all assigned resources, maintenance tasks are started. It also showed that the system utilization is high regardless of which resource is a bottleneck.

The test uses the “Regular tasks” policy from Table 3. Data deletion did not run, so the whole 100% of the throughput was assigned to user load – maintenance tasks were supposed to be started only if there are unused resources in the system and if running tasks do not affect the throughput of user load.

The experiment consists of four phases, each one with different characteristic of write tasks. In the first 30 minutes, external backup application worked full speed generating 60 MB/s per logical node (this is throughput after compression). After 30th minute, the backup application generated constant throughput of 40 MB/s, which is not enough to saturate the system. In the 60th minute, the write speed was changed to 10 MB/s. Finally, in the 90th minute, the write speed was changed to 45 MB/s but only duplicated data was written. The maintenance tasks that ran in background were doing defragmentation (sorting data containers).

The results of this experiment are presented in Figure 6. The plot shows the throughput of write tasks (in the last period, divided into total- and non-duplicate-write throughput) and maintenance tasks on one storage node. Figure 7 shows the utilization of the processor and hard disks, allowing to identify the bottleneck resource in each phase of writing.

In the first period, when the application worked at full speed the processor was a bottleneck[10]. Here the average latency was equal L_{ref} so the system was kept loaded; any further increase of the speed of any task type would result in violating the latency contract.

In the second period, where write tasks did not saturate the system unused resources were allocated to maintenance tasks. These tasks, according to our expectations, did not affect the throughput of the writes. In this phase, hard disks were a bottleneck. One could notice that the total throughput of tasks in the second phase is significantly higher than the total throughput in the first phase, even though write tasks are only a bit slower. Also CPU and disk utilization, together, is higher, which means that the resources are used more efficiently. Our controller, however, is not supposed to maximize resource utilization, but to maintain predefined policy of resource division. As in the first phase the policy assigns 100% resources to write tasks, the controller is not allowed to start background tasks that would affect the speed of writes. The controller works as expected.

In the third period, when the backup application worked even slower, the maintenance tasks were allowed to achieve higher throughput. In this phase, there were fewer write tasks executed concurrently so it was easier to maintain their latency bounded. As a result, we managed to obtain almost 100% utilization of hard disks.

In the fourth period, we had duplicate writes, which utilize a bit of CPU but hardly disks, and background tasks, which use disks. Therefore, there was little interaction between the two, and both workloads achieved high throughput.

7.2.3 Policy changes

As we mentioned previously, the policy of resource division is likely to change during system operation because it should depend on the system state. In the next two experiments, run on 6- and 60-server configuration, respectively, we analyze changes of the policy. In both experiments, the backup application writing to the system was configured to work at full speed for the whole duration of the experiment. In the first experiment, for the first 30 minutes, the system executed defragmentation tasks so it used the "Regular tasks" policy from Table 3. Next, in the 30th minute, we switched off one storage node to simulate a failure and trigger reconstruction tasks, so the system changed the policy to "Normal reconstruction" that gives more resources to maintenance tasks. After another

[10]Note that user writes are not simple IO operations because HYDRAstor uses data compression, erasure coding, and data deduplication, see Section 1.

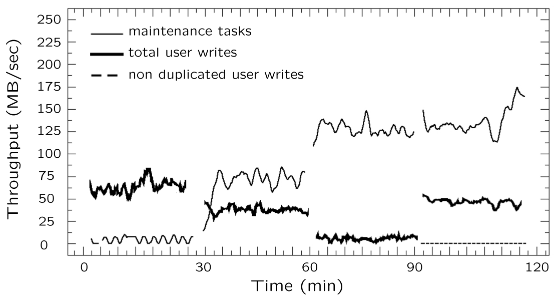

Figure 6: The throughputs of maintenance tasks and user writes. Duplicates were present only in the last phase.

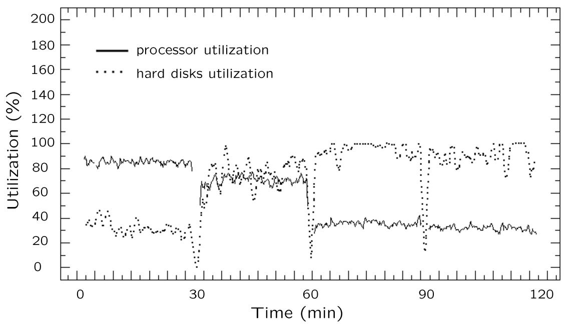

Figure 7: Utilization of the processor and hard disks.

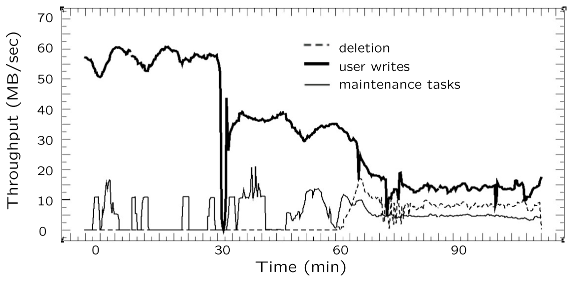

Figure 8: The throughputs of maintenance tasks, user writes and data deletion – 6-server configuration

30 minutes (60th minute), we started data deletion, which did not change the policy, but created tasks of a new type.

Figure 8 presents the throughput of user writes, maintenance tasks and data deletion on one of the storage nodes. The expected throughput proportions (writes:maintenance tasks:deletion) should be 100%:0%:0% in minutes 0-30, 71.5%:28.5%:0% in minutes 30-60, and 50%:20%:30% after 60th minute. The obtained proportions differ from the expected by less than 2% (after averaging) – this error comes from the load fluctuations.

Figure 9 presents the throughput achieved by user writes, maintenance tasks and data deletion for the experiment on the 60-server configuration. In this experiment we powered off 4 servers to trigger reconstruction, started data deletion (to have three kinds of tasks) and changed throughput shares to 30%:10%:60% (writes:maintenance tasks:deletion). Such a large system generates more irregular data deletion load in comparison with the 6-node configuration and the high throughput allocation for deletion is meant to verify

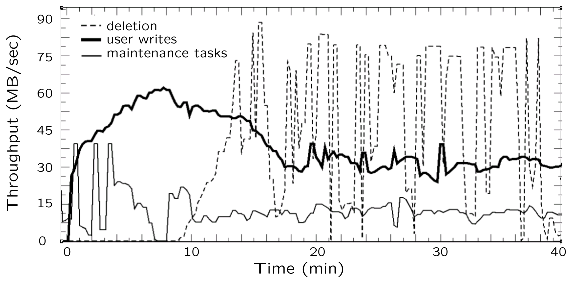

Figure 9: The throughputs of maintenance tasks, user writes and data deletion – 60-server configuration

the system behavior in case of very variable conditions (the controller will have to reallocate the throughput very often because deletion tasks will often be missing). Also the user load is highly irregular (it is averaged on the plot). The results prove that even in the presence of the irregularity the control mechanism keeps stable throughput proportions.

8. CONCLUSIONS AND FUTURE WORK

We presented a novel mechanism for dividing resources among tasks of different load types. The new approach is based on abstraction of the tasks and avoids assumptions about their characteristics. Therefore, it is suitable for distributed systems, where standard methods of defining a model fail due to complex system architecture. The mechanism was implemented in the commercial system HYDRAstor, with the focus on achieving high performance of the controlled system. We theoretically and experimentally proved that the throughput proportions in the controlled system converge to the desired ones and confirmed that in a steady state the algorithm keeps the system saturated (the resources are properly utilized). According to the experiments, the controlled system is stable (there are no serious fluctuations of throughputs nor overshoots). Also, the reaction time is suitable for the workloads of a secondary storage system (for other types of systems, the time scale of all operations may be reduced and the whole solution should work in a similar way). The evaluation was done on a 60-server system with the write performance of 10GB/s.

We showed how existing modifications of fair queuing can be adapted for resource division between user load and background tasks. We compared our Fuzzy Adaptive Control with fair queuing algorithms. The evaluation was done on an artificial model which was a simplification of the HYDRAstor system. We concluded that although fair queuing mechanisms have better reaction time to changing target shares, they may behave unstably in case of irregular user load. Fuzzy adaptive control has, on the other hand, longer settling time, but it is more robust in presence of irregularities of the write load pattern. Policies are changing quite rarely in HYDRAstor so settling time of fuzzy adaptive mechanism is acceptable. We concluded that fuzzy adaptive mechanism is more suitable for our system.

Future work will concern the defuzzification controllers. We consider applying some heuristic algorithm for faster convergence in case of shorter background tasks. Our research will also focus on using automatic mechanism for choosing throughput shares in run-time to maximize the total system throughput.

9. ACKNOWLEDGMENTS

We thank the reviewers for invaluable comments that helped to improve the presentation and put our research in the context of other related work. We thank all our colleagues from 9LivesData, LLC, for all brainstorms, reviews and comments.

10. REFERENCES

[1] D. R. Bobbarjung, S. Jagannathan, and C. Dubnicki. Improving duplicate elimination in storage systems. *Trans. Storage*, 2(4):424–448, 2006.

[2] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz. Disk scheduling with quality of service guarantees. In *ICMCS '99*, page 400, 1999.

[3] H. M. Chaskar and U. Madhow. Fair scheduling with tunable latency: a round-robin approach. *IEEE/ACM Trans. Netw.*, 11(4):592–601, 2003.

[4] Y. Diao, C. M. Garcia-arellano, J. L. Hellerstein, S. S. Lightstone, S. S. Parekh, A. J. Storm, and M. Surendra. Systems and methods for providing constrained optimization using adaptive regulatory control, December 2005. US patent application no 20050268063.

[5] J. R. Douceur and W. J. Bolosky. Progress-based regulation of low-importance processes. *SIGOPS Oper. Syst. Rev.*, 33(5):247–260, Dec. 1999.

[6] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki. HYDRAstor: a Scalable Secondary Storage. In *FAST '09*, 2009.

[7] L. Eggert and J. D. Touch. Idletime scheduling with preemption intervals, proceedings of the twentieth acm symposium on operating systems principles, brighton,. In *Twentieth ACM symposium on Operating systems principles*, pages 249–262, 2005.

[8] P. Goyal, H. M. Vin, and H. Chen. Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks. In *SIGCOMM '96*, pages 157–168, 1996.

[9] S. D. Gribble, G. S. Manku, D. Roselli, E. A. Brewer, T. J. Gibson, and E. L. Miller. Self-similarity in file systems. *SIGMETRICS Perform. Eval. Rev.*, 26(1):141–150, 1998.

[10] A. Gulati and I. Ahmad. Towards distributed storage resource management using flow control. *SIGOPS Oper. Syst. Rev.*, 42(6):10–16, 2008.

[11] A. Gulati, I. Ahmad, and C. A. Waldspurger. PARDA: Proportional Allocation of Resources for Distributed Storage Access. In *FAST '09*, San Francisco, California, USA, February 2009.

[12] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. 2004.

[13] L. Huang, G. Peng, and T.-c. Chiueh. Multi-dimensional storage virtualization. *SIGMETRICS Perform. Eval. Rev.*, 32(1):14–24, 2004.

[14] W. Jin, J. S. Chase, and J. Kaur. Interposed proportional sharing for a storage service utility. *SIGMETRICS Perform. Eval. Rev.*, 32(1):37–48, 2004.

[15] E. Kalyvianaki, T. Charalambous, and S. Hand. Self-adaptive and self-configured cpu resource provisioning for virtualized servers using kalman filters. In *ICAC '09*, pages 117–126, 2009.

[16] M. Karlsson, C. Karamanolis, and J. Chase. Controllable fair queuing for meeting performance goals. *Perform. Eval.*, 62(1-4):278–294, 2005.

[17] M. Karlsson, C. Karamanolis, and X. Zhu. Triage: Performance differentiation for storage systems using adaptive control. *Trans. Storage*, 1(4):457–480, 2005.

[18] S. Keshav. A control-theoretic approach to flow control. *SIGCOMM Comput. Commun. Rev.*, 21(4):3–15, 1991.

[19] C. Lu, G. A. Alvarez, and J. Wilkes. Aqueduct: Online data migration with performance guarantees. In *FAST '02*, page 21, 2002.

[20] C. R. Lumb, A. Merchant, and G. A. Alvarez. Facade: Virtual storage devices with performance guarantees. In *FAST '03*, pages 131–144, 2003.

[21] D. Meister and A. Brinkmann. Multi-level comparison of data deduplication in a backup scenario. In *SYSTOR '09*, pages 1–12, 2009.

[22] N. Mi, A. Riska, X. Li, E. Smirni, and E. Riedel. Restrained utilization of idleness for transparent scheduling of background tasks. In *SIGMETRICS '09*, pages 205–216, 2009.

[23] N. Mi, A. Riska, Q. Zhang, E. Smirni, and E. Riedel. Efficient management of idleness in storage systems. In *ACM Transactions on Storage (TOS)*, volume 5, pages 1–25, June 2009.

[24] NEC Corporation. HYDRAstor Grid Storage System, 2008. http://www.hydrastor.com.

[25] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant. Automated control of multiple virtualized resources. In *EuroSys '09*, pages 13–26, 2009.

[26] S.-M. Park and M. Humphrey. Feedback-controlled resource sharing for predictable escience. In *SC '08*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[27] A. Popescu and S. Ghanbari. A study on performance isolation approaches for consolidated storage. Technical report, May 2008.

[28] B. Quetier, V. Neri, and F. Cappello. Selecting a virtualization system for grid/p2p large scale emulation. In *EXPGRID '06*, June 2006.

[29] C. Ungureanu, A. Aranya, S. Gokhale, S. Rago, B. Atkin, A. Bohra, C. Dubnicki, and G. Calkowski. Hydrafs: A high-throughput file system for the hydrastor content-addressable storage system. In *FAST '10*, pages 225–239, 2010.

[30] Y. Wang and A. Merchant. Proportional-share scheduling for distributed storage systems. In *FAST '07*, pages 4–4, 2007.

[31] M. Welsh and D. Culler. Adaptive overload control for busy internet servers. In *USITS'03*, pages 4–4, 2003.

[32] J. Xu, M. Zhao, J. Fortes, R. Carpenter, and M. Yousif. On the Use of Fuzzy Modeling in Virtualized Data Center Management. In *4th IEEE International Conference on Autonomic Computing*, Washington, DC, USA, June 2007.

[33] J. Zhang, A. Sivasubramaniam, A. Riska, Q. Wang, and E. Riedel. An interposed 2-level i/o scheduling framework for performance virtualization. In *SIGMETRICS '05*, pages 406–407, 2005.

[34] X. Zhu, X. Liu, S. Singhal, and M. Arlitt. Resource entitlement control system, January 2010. US patent application no 7644162.

[35] X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, P. Padala, and K. Shin. What does control theory bring to systems research? volume 43, pages 62–69, 2009.