Graduated QoS by Decomposing Bursts: Don't Let the Tail Wag Your Server

Lanyue Lu and Peter Varman
Rice University, USA
Email: ll2@rice.edu, pjv@rice.edu

Kshitij Doshi
Intel Corporation, USA
Email: kshitij.a.doshi@intel.com

ABSTRACT

The growing popularity of hosted storage services and shared storage infrastructure in data centers is driving the recent interest in resource management and QoS in storage systems. The bursty nature of storage workloads raises significant performance and provisioning challenges, leading to increased infrastructure, management, and energy costs. We present a novel dynamic workload shaping framework to handle bursty workloads, where the arrival stream is dynamically decomposed to isolate its bursts, and then rescheduled to exploit available slack. We show how decomposition reduces the server capacity requirements dramatically while affecting QoS guarantees minimally. We present an optimal decomposition algorithm RTT and a recombination algorithm Miser, and show the benefits of the approach by performance evaluation using several storage traces.

1. Introduction

The increasing complexity of managing stored data and the economic benefits of consolidation are driving storage systems towards a service-oriented paradigm, in which personal and corporate clients purchase storage space and access bandwidth to store and retrieve their data. This paper deals with issues of performance and provisioning of server resources in storage data centers. In a typical setup, Service Level Agreements (SLA) between the service provider and clients stipulate guarantees on throughput [1], [2] or latency [3], [4] for rate-controlled clients. The service provider must provision sufficient resources to meet these performance guarantees based on estimates of the resource demands of the individual clients, and the aggregate capacity requirements of the client mix admitted into the system. The run-time scheduler must isolate the individual clients from each other so that they receive their reservations without interference from misbehaving clients with demand overruns, and schedule their requests on the server appropriately [5]. A fundamental challenge in data center operations is the need to deal effectively with *high-variance* bursty workloads arising in the network and storage server traffic [6], [7], [8]. These workloads are characterized by unpredictable bursty periods during which the instantaneous arrival rates can significantly exceed the average long-term rate. In the absence of explicit mechanisms to deal with it, the effects of these bursts are not confined to the localized regions where they occur, but spill over and affect otherwise well-behaved regions of the workload as well. As a consequence, although the bursty portion may be only a small fraction of the entire workload, it has a disproportionate effect on performance and provisioning decisions. This ”tail wagging the dog” situation results in the server being forced to make unduly conservative estimates of resource requirements, resulting in excessive provisioning and energy consumption costs, and unnecessary throttling of the number of the clients admitted into the system.

In this paper we present a novel approach to improving client performance and slimming resource provisioning at the server. In our approach we modify the characteristics of the arriving workload so that its behavior is dominated by the majority well-behaved portions of the request stream; the portions of the workload comprising the tail are identified and isolated so that their effects are localized. This results in more predictable behavior, and significantly lower resource requirements. Consequently, the performance SLA is specified by a distribution of response times rather than a single worst-case measure. By slightly relaxing the performance requirements, a significant reduction in server capacity can be achieved while maintaining stringent QoS guarantees for most of the workload. The server can pass on these savings by providing a variety of SLAs and pricing options to the client. Storage service subscribers that have highly streamlined request behavior, and who therefore require negligible surplus capacity in order to meet their deadlines, can be offered service on concessional terms as reward for their ”well-behavedness”.

This paper makes the following specific contributions. We present a new framework for run-time scheduling of a client's workload based on decomposition and recombination of the request stream. This reshaped workload helps localize the effects of bursts so that a large percentage of the workload has superior response time guarantees, while keeping the behavior of the tail comparable to that achieved by traditional methods. The resource requirements for the reshaped workloads are shown to be significantly lower than that for the original workload, since it is closer to the average rather than the worst-case requirement. This translates into reductions in provisioned capacity, and reduced energy consumption as well. Finally, we show how the framework can be used to improve estimates of the aggregate resource requirements of multiple concurrent clients. Due to statistical variations, the peak inputs of the workloads are unlikely to line up simultaneously. Estimates based on simple aggregation of the requirements of each client therefore tend to overestimate the requirements significantly, but estimating the savings due to statistical multiplexing is difficult [9]. We show that aggregation based on the capacity of the reshaped workload provides more realistic estimates of resource requirements, compared to dealing with the unshaped workload.

The rest of the paper is organized as follows. Section 2 describes the reshaping framework and provides a high-level description of the decomposition and recombination phases. An optimal decomposition algorithm RTT is described in Section 3 along with several recombination schemes, including a new slack-based algorithm called Miser. Detailed evaluation results are presented in Section 4. Related work is summarized in Section 5, and conclusions are presented in Section 6.

2. Workload Shaping

The goal of workload shaping is to smoothen the workload to reduce the unpredictability caused by bursty arrival patterns, which makes capacity planning difficult and degrades performance. Although the average utilization of the system tends to be low, the unpredictable bursts of high activity overwhelm server resources leading to poor performance. With traditional scheduling the effects of these bursts are not confined to the localized regions where they occur, but spill over and affect otherwise well-behaved regions of the workload as well. Consequently, as noted before, a small amount of bursty behavior has a disproportionate effect on performance, provisioning, and admission control decisions. The workload shaping procedure consists of two complementary operations: *decomposition* and *recombination*, as shown schematically in Figure 1.

In the decomposition phase, the workload of a single application (or client) is partitioned into two or more classes with different performance guarantees. The requests belonging to the different classes are directed to separate queues. In the scheme shown in Figure 1 there are two classes, identified by queues Q1 and Q2 respectively. In this example, requests belonging to Q1 will be guaranteed a response time $R1$ while requests in Q2 are served in a best-effort fashion. In the recombination phase the requests of the two classes are multiplexed in a suitable manner to satisfy the individual performance constraints. Different scheduling algorithms which provide different response time distributions can be used in this phase. These will be discussed and evaluated in Sections 3 and 4.

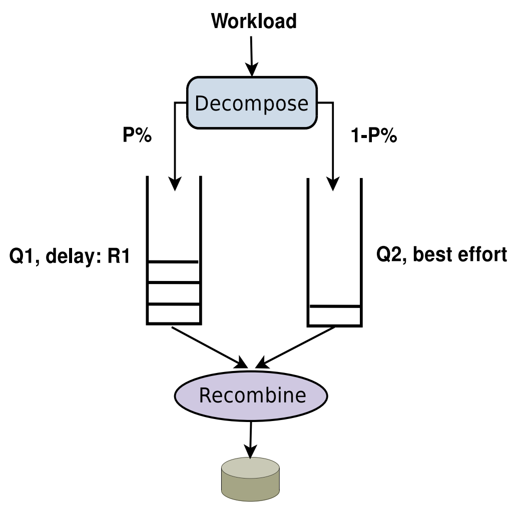
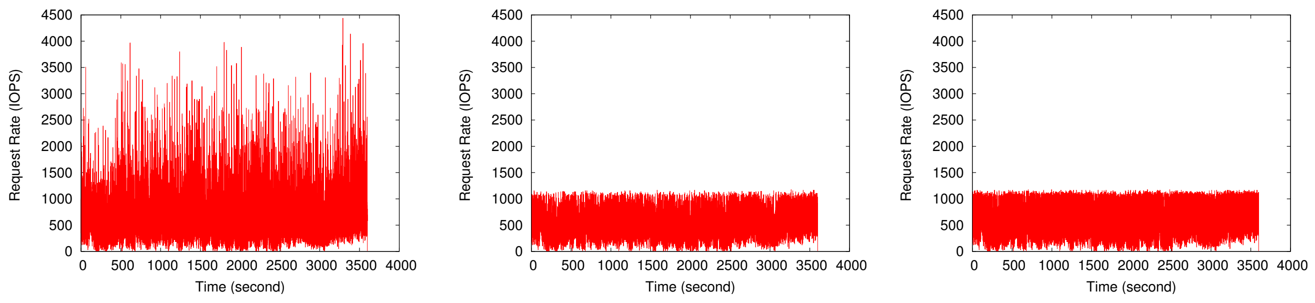


Figure 1. Architecture of workload shaper providing graduated QoS guarantees

Figure 2(a) shows a portion of an OpenMail trace [10] of I/O requests (displayed using aggregated requests in a time window of 100 ms). Note that the peak request rate is about 4440 IOPS while the average request rate is only about 534 IOPS. Figure 2(b) shows the class Q1 containing 90% of the requests after decomposing the workload using our decomposition algorithm RTT (described later). The capacity of the server is chosen so that all requests in Q1 meet a response time of 10 ms. As may be seen Q1 is relatively even at this granularity. All requests in Q1 can meet their response time bounds with a capacity of only 1080 IOPS, compared to 9241 IOPS required for the original workload. Finally, Figure 2(c) shows the workload following recombination of Q1 and Q2 using the Miser algorithm (described later). This algorithm monitors the slack in the arrivals where it can schedule a request of Q2 without causing any of the requests of Q1 to miss their deadline and schedules a request from Q2 at the earliest such time.

2.1. Decomposition and Recombination

The workload is characterized by its arrival sequence that specifies the number of I/O requests n_i arriving at time a_i, $i = 1, \cdots, N$. The Cumulative Arrival Curve (abbreviated AC) $A(t)$ is the the total number of I/O requests that arrive during the interval $[0, t]$; *i.e.* $A(t) = \sum_{j=1}^{i} n_j$, where $a_i \leq t < a_{i+1}$. Figure 3 (a) shows AC as a staircase function with jumps corresponding to the arrival instants. The server provides service at a constant rate of C IOPS as long as there are unfinished requests. The Service Curve (SC) is shown by a line of slope C beginning at the origin during a busy period when the server is continuously busy. At any time, the vertical distance between SC and AC is the number of pending requests (either queued or in service).



(a) Original workload (b) 90% of workload after Decomposition (c) 100% of workload after Recombination

Figure 2. Shaping the OpenMail trace by Decomposition and Recombination

Each request has a response time requirement of δ, so that requests arriving at a_i have a deadline of $d_i = a_i + \delta$. If the number of pending requests exceeds $C \times \delta$ it signals an *overload* condition. Since at most $C \times \delta$ requests can be completed in time δ, some of the requests pending at an overflow instant must necessarily miss their deadlines. In Figure 3 (a) the line above and parallel to the Service Curve is an upper bound on the amount of pending service that can meet their deadlines. We call this the Service Curve Limit (SCL).

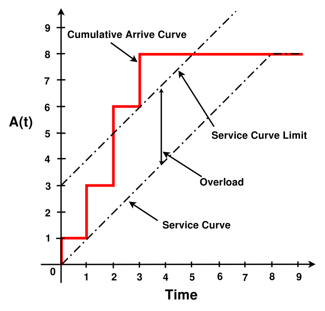
The operation of a decomposition algorithm can be described easily with respect to the Service Curve Limit. The goal is to identify requests to drop from the workload (in actuality dropped requests are merely moved to Q2 and served from there). Consider time instants like 2 and 3 in Figure 3 (a) where the AC exceeds SCL. From the previous discussion, requests exceeding the SCL limits cause an overload condition and some requests must must be dropped in order for the rest of the requests to meet their deadline. If requests are dropped from the workload, the AC shifts down by an amount equal to that removed. This is shown in Figure 3 (b) which shows the situation following the removal of 1 request at time 1 and another at time 2. As can be seen the modified AC lies below the SCL which means that all requests in the new AC will meet their deadlines. A different choice of two requests to remove is shown in Figure 3 (c), where one request each at times 2 and 3 are removed. One can argue that for the given capacity and response time requirements, at least two requests in this workload must necessarily miss their deadlines (as in the two choices above). On the other hand dropping two requests at time 1 is a poor choice, since a request arriving at time 3 will still miss its deadline. Note also that the decomposition method needs to be online in that it needs to make a decision on whether or not to drop a request based on the past inputs only, without knowing the future patterns of requests. We show in Section 3 that our decomposition algorithm RTT satisfies these properties: it is online and minimizes the number of dropped requests for a given capacity and deadline.

We now describe the operation of a recombination algorithm. The goal is to service the overflowing requests that have been placed in Q2 concurrently with the guaranteed requests in Q1. For instance, in Figure 3 (d) the two requests that were dropped at times 2 and 3 are scheduled from Q_2 at times 4 and 5 when there is slack in the server. Several strategies with different tradeoffs can be employed for the recombination. One simple approach is to offload the overflowing requests to a separate physical server where they can be serviced without interfering with the guaranteed traffic (this is similar in principle to the write offloading strategy in [11] where bursts of write requests are distributed to a number of low-utilization disks for service). In cases where this offloading is not feasible, perhaps due to lack of a suitable off-load sever or the need for dedicated resources available only on the main server, a good strategy is to treat the two parts of the workload independently and multiplex them on the server using a Fair Queuing scheduler to keep them isolated. This approach actually has significant capacity benefits over the dedicated offload server approach (as we show in Section 4), due to the benefits of statistical multiplexing. Since the overflow workload is active only during bursts, the capacity during its idle periods can be profitably used by the guaranteed portion of the workload to improve its response time profile. We also propose a new slack-based scheduling algorithm to combine the two portions of the workload. This method called Miser, allows better shaping of the tail of the workload than a Fair Queueing Scheduler, but may in the worst-case slightly increase the fraction of requests missing their deadlines, unless a small amount of additional capacity is provided.

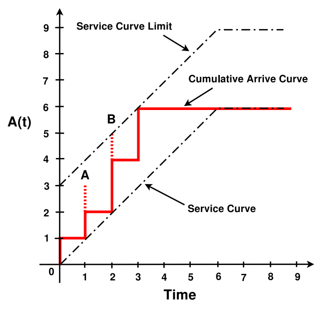
2.2. Capacity Provisioning

In this section we address the issue of how much server capacity needs to be reserved in order to meet a client's requirements. We consider both the cases of provisioning for a single client and for multiple, concurrent clients.

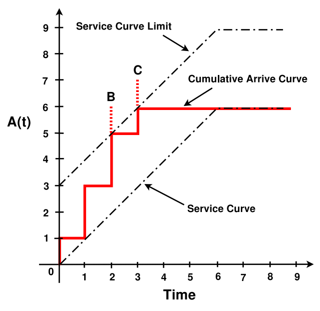
Single Client: We profile the workload to determine the capacity reservation needed to meet a stipulated QoS requirement. That is, given a response time bound δ, find



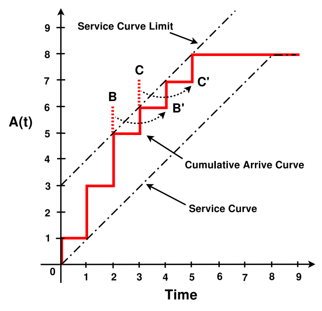
(a) Workload model



(b) One way of drop requests



(c) RTT-based request dropping



(d) RTT-based Recombining

Figure 3. Illustrating the Decomposition and Recombination process

the minimum server capacity C_{min} required to guarantee that a given fraction f of the requests of the given workload meets their deadlines. Decomposing the workload in this way results in much smaller server capacity requirement, while still maintaining a high QoS, albeit less than 100%.

Although it is possible to find direct optimization methods for the problem stated, we found that a deterministic search of the solution space provided the answers with low computational overhead for even very large traces. We search the space as follows. For a given C and δ we use the RTT algorithm (detailed in Section 3.1) to find a decomposition that maximizes the number of requests meeting their deadlines. If the fraction meeting the deadline is higher than the required fraction f we reduce the capacity and try again; else we increase the capacity and retry. Since f is a monotonically non-decreasing function of C, a binary-search converges rapidly (within $O(logC)$ iterations) to the desired minimum capacity C_{min}.

We provision a capacity of $C_{min} + \Delta C$, where the latter is used to prevent starvation of the secondary class in Q2. In our experiments an additional capacity of $\Delta C = 1/\delta$ was found to be sufficient to obtain good performance of the entire workload.

Multiple Concurrent Clients: In a data center environment, the service provider needs to provision sufficient resources for several clients simultaneously sharing the system. Accurate provisioning is an extremely difficult problem and several approaches based on statistical models of the arrival process have been proposed [9]. A brute-force approach is to estimate the worst-case capacity required for each client and then provision at least that much for each of the clients. This approach results in poor server utilization and overly cautious admission control policies. There are two main problems: first, the worst-case capacity requirements of a client are usually several times the average demand; secondly, adding the individual capacity requirements presumes that the worst-cases of all the individual workloads line up simultaneously, an extremely unlikely situation in practice.

We argue that aggregation of the capacity requirements of reshaped clients provides a good estimate of the server capacity needed for handling multiple concurrent clients. That is, the capacity required by the reshaped workload serves as a measure of "effective bandwidth" [9] of the client. We evaluate this in Section 4 and show that using the aggregated effective bandwidths provides a very good estimate of the required server capacity.

3. Workload Shaping Algorithms

The system model is shown in Figure 1. The scheduler maintains two queues Q_1 and Q_2. The *primary queue* Q_1 has bounded length to control the latencies of requests accepted into it. The *overflow queue* Q_2 holds requests that are not accepted into Q_1 because their latency cannot be guaranteed. The server has a capacity C and the response time bounds for the requests in the primary queue is δ.

3.1. RTT Decomposition

Algorithm 1: RTT Decomposition

```
RTT_Decompose( )
begin
    $maxQ_1 = C \times \delta$
    if $lenQ_1 \leq maxQ_1 - 1$ then
        begin
            Add request to $Q_1$
            Increment $lenQ_1$
        end
    else
        Add request to $Q_2$;
end
```

The decomposition algorithm *RTT*, shown in Algorithm 1, is used to partition the requests dynamically into the two queues. The algorithm is extremely simple. If the arriving request will cause the length of the primary queue Q_1 ($lenQ_1$) to exceed its maximum length ($maxQ_1$), the request is diverted to the overflow queue; else it joins the end of the primary queue. The maximum length of Q_1, $maxQ_1 = C \times \delta$. Despite its simplicity, RTT satisfies the following *optimality property*:

For a given workload, capacity, and response time bound, RTT identifies a maximal-sized set of requests that can meet the deadline.

To show RTT optimality, we first show that in any period that RTT is continuously busy, the number of requests it drops is the minimum possible. Lemma 1 shows a lower bound on the number of dropped requests in any interval, and Lemma 2 shows that RTT matches that bound in a busy period. Following this, we consider an arbitrary period of operation in which RTT may alternate between idle and busy periods. We show inductively in Lemma 3, that RTT cumulatively drops no more than a hypothetical (possibly offline) optimal algorithm OPT at the end of any busy period.

Recall from Section 2 that a_i represents a request arrival instant, and $A(t)$ and $S(t)$ represent the cumulative arrivals and service up to some time t. Also, define the function $sgn(x) = \lceil x \rceil$ for $x \geq 0$, and $sgn(x) = 0$ for $x < 0$. The proofs of the Lemmas have been omitted due to space but can be found in the technical report available in [12].

Lemma 1: Given server capacity C, a lower bound on the number of requests in the workload that cannot meet their deadlines is given by $max_{1 \leq k \leq N}\{sgn(A(a_k) - S(a_k + \delta))\}$.

Lemma 2 In any busy period $[0, a_N]$, RTT will drop no more than $max_{1 \leq i \leq N}\{sgn(A(a_i) - S(a_i + \delta))\}$ requests.

Let intervals I_1, $I_2, \cdots I_m$ be successive *busy periods* of RTT during the time $[0, T]$. The folowing Lemma can be proved by Induction.

Lemma 3: Let OPT be an optimal algorithm that drops the minimal number of requests in $[0, T]$. $\forall k, 1 \leq k \leq m$, OPT drops at least Δ_k requests in I_k and incurs an idle period of at least η_k, where Δ_k is the number of requests dropped by RTT in I_k and η_k is the amount of idle time of RTT till the end of I_k.

3.2. Recombining Algorithms

We now describe four methods for combining the workload spilt by RTT and scheduling them at the server. Their performance evaluation is described in Section 4. **FCFS**: The requests are not partitioned and serviced in a FCFS manner. This serves as a base case for the evaluation. **Split**: The requests are partitioned by RTT and the overflow requests in Q_2 are served by a separate physical server. The primary server's capacity C_{min} is based on profiling the workload, and a small additional amount ΔC is provided to the secondary server. **Fair Queueing**: The requests are partitioned by RTT and the two queues Q_1 and Q_2 are served using a proportional share bandwidth allocator (like WF2Q [13], SFQ [14], pClock [3]) that divides the server capacity in the specified ratio. The total capacity of the server is $C_{min} + \Delta C$, but by sharing a single physical server we hope to harness the benefits of statistical multiplexing. **Miser**: The scheduler uses slack in the scheduling of the primary queue to schedule requests in Q_2 as early as possible. Unlike the previous two methods, where the additional capacity ΔC only affected the performance of the requests in Q_2, here the two queues are more closely coupled. Being online the composite algorithm (RTT + Miser) could sometimes drop more than the theoretical minimum number of requests. One can show that if $\Delta C = C_{min}$, then this can never occur. Our simulations show that even with a small amount of additional service $\Delta C = 1/\delta$, very few requests in Q_1 are delayed beyond the deadline in practice, and the tail distribution of Q_2 is much nicer.

Algorithm 2: Miser Scheduling

```
On a request arrival:
begin
    RTT_Decompose( );
    /* Compute Slack*/
    if request $r_i$ in $Q_1$ then
        $r_i\_slack = \lfloor maxQ_1 - lenQ_1 \rfloor$
        $minSlack = min\{minSlack, r_i\_slack\}$
end

On a request departure:
begin
    /*Dispatch a request*/
    if ( $minSlack \geq 1$ ) & ( $Q_2$ is not empty ) then
        Issue request from $Q_2$ in FIFO order
    else
        Issue request from $Q_1$ in FIFO order

    /*Update Slack*/
    if scheduled request $r_i$ is from $Q_1$ then
        if $r_i\_slack = minSlack$ then
            $minSlack = min_{i \in Q_1}\{r_i\_slack\}$
    else
        for $\forall i \in Q_1$ do
            $r_i\_slack = r_i\_slack - 1$
        $minSlack = minSlack - 1$
end
```

Algorithm 2 shows the actions taken by Miser on request arrivals and completions. On a request arrival the routine *RTT_Decompose* is first invoked to classify the request. If placed in the primary queue it is assigned a slack value equal to the number of places still available in Q_1. A request in the overflow queue Q_2 is scheduled when the smallest slack value of the requests in Q_1 is at least 1.

4. Experimental Evaluation

In this section, we evaluate the workload shaping based scheduling framework using the storage system simulation tool DiskSim [15]. We implemented the RTT decomposition algorithm at the device driver level which catches all the incoming requests before they reach the underlying disks. The workload is decomposed by RTT and put into separate queues. When the disk driver needs to dispatch a new request to the disk, our recombining scheduler is called to choose the next request for service.

We use traces of three different storage applications for our evaluation: Web Search Engine (WebSearch), OLTP application (FinTrans) and Email service (OpenMail). The traces are obtained from UMass Storage Repository [16] and HP Research Labs [10]. All of these are low-level block storage I/O traces, which have not been filtered out by the

Workloads	Response Time Target	Percentage of Workload Meeting Response Time					
		90.0%	95.0%	99.0%	99.5%	99.9%	100%
WebSearch (WS)	5 ms	590	711	960	1055	1310	2325
	10 ms	417	474	603	658	786	1538
	20 ms	345	388	462	487	540	900
	50 ms	328	363	419	437	467	533
FinTrans (FT)	5 ms	400	550	600	800	1000	3000
	10 ms	200	300	360	400	500	1500
	20 ms	150	168	216	236	280	750
	50 ms	119	138	172	184	209	330
OpenMail (OM)	5 ms	1350	2000	3950	4800	6600	13990
	10 ms	1080	1595	2965	3550	4860	9241
	20 ms	900	1326	2361	2740	3480	5766
	50 ms	745	1045	1805	2050	2495	3656

Table 1. Capacity (IOPS) required for specified Workload Fraction to meet the Response Time target

file system cache. The WebSearch traces are from a popular search engine and consist of user web search requests. The FinTrans traces are generated by financial transactions in an OLTP application running at two large financial institutions. OpenMail traces are collected from HP email servers during the servers' busy periods.

We conducted four types of experiments: (i) measuring server capacity requirements as a function of the fraction f of requests that are guaranteed a response time δ (ii) response time distribution obtained by a traditional FCFS scheduler that does not decompose the workload (iii) comparison of the response time distribution of recombination algorithms Split, Fair Schedule and Miser with FCFS and relative to each other (iv) capacity estimation for multiple concurrent clients using the decomposition framework. For reasons of space only a representative sample of the results are presented here; some more results are in our technical report [12].

4.1. Capacity-QoS Tradeoffs

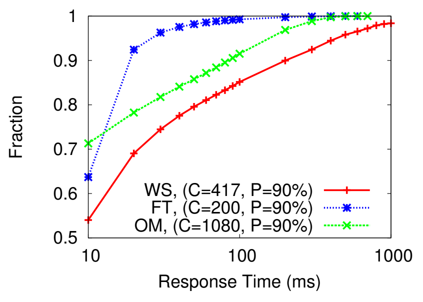
Avoiding resource over-provisioning is a difficult problem due to the unpredictable bursty behavior of real workloads. This set of experiments explores the tradeoffs between the fraction f of the workload that is guaranteed to meet a specified response time bound δ, and the minimum server capacity C_{min} required. The case $f = 100\%$, gives the minimum capacity required for *all* the requests to meet the latency bound. As f is relaxed, a smaller capacity should be sufficient. Our results confirm the existence of a sharp knee in the C_{min} versus f relation, that shows that a very small percentage of the workload necessitates an overwhelming capacity to meet its guarantees.

Table 1 shows the capacity required to meet response time bounds of $5, 10, 20, 50$ ms for f between 90% to 100% using three different workloads. The capacity required falls off significantly by relaxing the response time guarantee for 1% to 10% of the workload. For instance, with $\delta = 5$ ms, increasing f from 90% to 100% requires large capacity increases: almost 4 times (590 to 2325 IOPS) for Websearch, 7.5 times (400 to 3000 IOPS) for FinTrans, and more than 10 times (1350 to 13990 IOPS) for OpenMail. Even going from 99% to 100% the capacity required increases by factors of 2.4 (960 to 2325 IOPS) for Websearch, 5 (600 to 3000 IOPS) for FinTrans and 3.5 (3950 to 13990 IOPS) for OpenMail. For higher response times, the capacity required also increases by significant, though smaller factors. For instance, in OpenMail for δ = 10, 20, and 50ms respectively, the required capacity increases 8.6, 6.4 and 4.9 times in going from 90% to 100% and 3.1, 2.4 and 2 times in going from from 99% to 100%. The extent of burstiness (and potential for capacity savings) can be gauged by looking at the range from 99% and 100% of FinTrans, where increasing f from 99.9% to 100% required capacity increases by factors of 3.0, 3.0, 2.7 and 1.6 respectively for different response times.

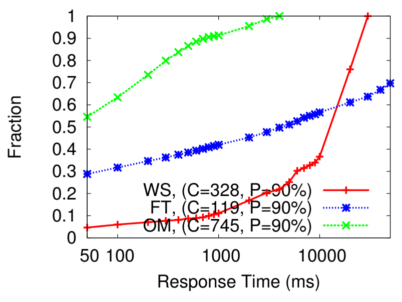
Summarizing, the experiments clearly indicate that exempting even a small fraction of the workload from the response time guarantees can substantially reduce the capacity that needs to be provisioned. The more aggressive the QoS specifications (lower response time requirements), the greater the savings in relaxing the fraction meeting the guarantee. Even a small percentage of burst in the workload (such as 0.1%) can require a large amount of resources to guarantee the response time.

4.2. Response Time Distribution of FCFS

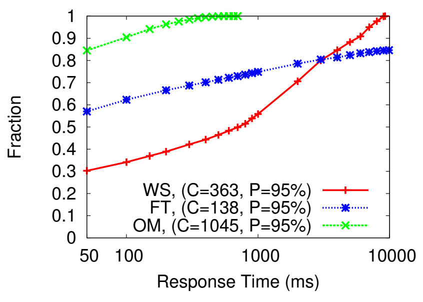
We next investigate the effects of the bursts on the response time distribution of the workload. In a shared data center, scheduling across clients may be done using a fair scheduler, and scheduling at the storage array uses some throughput maximizing ordering of the requests in the low-level queue. However, in the absence of application-specific ordering, the requests of a single client are usually handled in a simple FCFS manner. The following experiments show that the bursts in the workload significantly degrade its response time profile. That is, the effects of bursts are not isolated but



(a) Target: (90%,10 ms)



(b) Target: (90%,50 ms)



(c) Target: (95%,50 ms)

Figure 4. Response time CDF of FCFS scheduling

affect the behavior of the non-bursty part of the workload; isolation needs to be specifically enforced by a scheduler.

The cumulative response time distribution obtained for the unpartitioned workloads using FCFS scheduling is shown in Figures 4(a) and 4(b) for $\delta = 10$ms and $\delta = 50$ms respectively, for three workloads. In each case the capacity is chosen so that 90% of the workload can meet the response time target if it were optimally decomposed using RTT.

In Figure 4(a), with a capacity of 417 IOPS only 54% of the unpartitioned WebSearch workload meets a $\delta = 10$ ms latency bound compared to 90% compliance for the workload decomposed by RTT. The unpartitioned workload reaches 90% compliance only for δ around 200ms. Similarly, unpartitioned OpenMail workload with $\delta = 10$ms and $C = 1080$ IOPS achieves only 71% compliance compared to 90% for the partitioned workload, and reaches 90% compliance only at around δ of 90ms. For the FinTrans workload, a capacity of 200 IOPS resulted in 64% of the unpartitioned workload, and 90% of the partitioned workload meeting the 10ms response time bound.

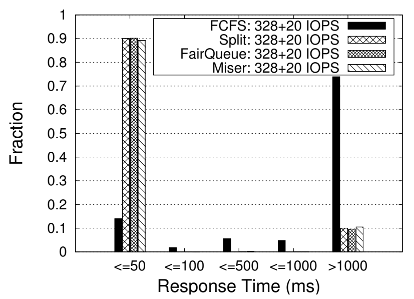
In Figure 4(b), the response time target is relaxed to 50 ms. In this case, for WebSearch only a tiny 5% of the requests meet the 50 ms deadline, compared to 90% of the partitioned workload. For FinTrans and OpenMail the corresponding figures are still a low 29% and 55% of the workload respectively, With a more relaxed response time (50ms instead of 10ms), the partitioned workload can meet the 90% compliance with a smaller capacity; however, for FCFS the smaller capacity results in the queues built up during the burst to drain slower, increasing the response time for the well-behaved part of the workload as well. Figure 4(c) shows the performance of FCFS using the same capacity at which RTT guarantees a 50 ms response time for 95% of workload. The corresponding percentages of requests meeting the 50ms bound for WebSearch, FinTrans and OpenMail using FCFS are a low 30%, 57% and 85% respectively.

4.3. Response Time of Shaped Workload

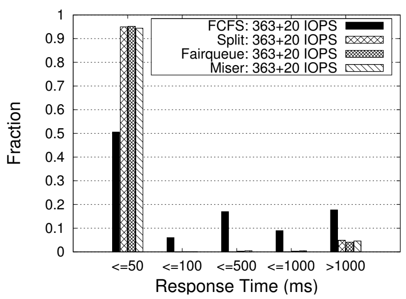
In this section, we evaluate the recombination methods discussed in Section 3.2, Split, FairQueueing and Miser, and compare them with the performance of FCFS. In each case the total capacity provided for the workload is held fixed at $C_{min} + \Delta C$; ΔC was chosen to be a small amount $1/\delta$. FCFS uses the total capacity for the unpartitioned workload. For Split and FairQueuing the capacity is divided in the ratio C_{min} to ΔC for Q_1 and Q_2 respectively. In Split, the servers are not shared. FairQueuing multiplexes the capacity of a single server so that excess capacity can be flexibly moved from one part to the other, while guaranteeing a minimum reservation to each. Miser opportunistically uses the capacity to schedule the overflow requests depending on the amount of available slack.

In Figure 5, we evaluate the scheduling performance for Websearch workload with the response time target of 50 ms. We can see that Split and FairQueueing achieves the 90% target of 50 ms response time following decomposition of the workload. Miser, as noted previously, may incur some additional misses, but is still very close to the 90% target, even with just $\Delta C = 20$ IOPS additional capacity. However, FCFS can only finish 14% of the requests within 50 ms. Furthermore, FCFS has 74% of requests with response time bigger than 1000 ms, while Split, FairQueue and Miser have about 10%. Figure 5(b) shows the performance of these schedulers with percentage target 95% and $\delta = 50$ ms. Split, FairQueueing and Miser still outperform FCFS with 95% guarantees of 50 ms response time, while FCFS finishes only 51% within 50 ms. For the response time larger than 1000 ms, Split has 4.9%, FairQueueing has 4.1% and Miser has 4.6% of the requests respectively, while FCFS has 17.7%.

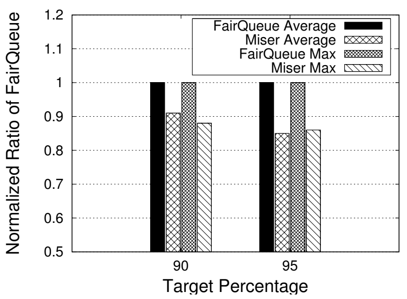
Figures 5(a) and 5(b) show that Split, FairQueueing and Miser are better able to guarantee a higher percentage of requests with small deadlines. But Split, FairQueueing and Miser have larger maximum response time than FCFS, because a decomposition-based scheduler will delay requests in Q_2 over those in Q_1 leading to larger delays for the overflowing requests. But as shown, the total number of



(a) Target: (90%,50ms)



(b) Target: (95%,50ms)



(c) Best effort requests performance

Figure 5. Performance comparison of FCFS, Split, Fair Queuing and Miser: Websearch workload

long-delay requests (greater than 1s) is much less than for FCFS, even though the largest value may be higher.

Finally we compare the performance of Split, FairQueueing and Miser (see Figure 5(c)). For Split there is no capacity sharing between the two classes, resulting in an order of magnitude bigger average and maximal response times of Q_2's requests compared to FairQueueing and Miser. FairQueueing assigns the weighted capacity to the two classes; Q_2 can only use the spare capacity of Q_1 when the latter has no requests. However, Miser dynamically monitors the slack in Q_1, and uses it to improve the performance of Q_2's requests. Figure 5(c) shows the average and maximal response time of the requests in Q_2 attained by Miser normalized to that of FairQueueing in the above experiments. We can see that for WebSearch the average response time of secondary class of Miser is about 85% - 90% of FairQueueing, while maximal response time is roughly 85% compared to FairQueueing.

4.4. Multi-flow Consolidation

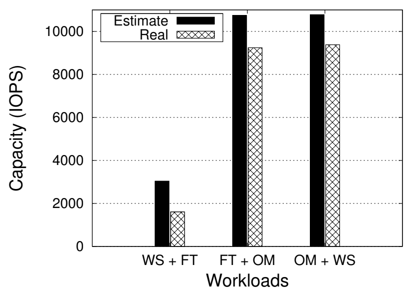
In a shared server environment, resource provisioning is usually hard to predict because of the bursty nature of the workloads. A straightforward aggregation of the reservation requirements of each client provides a simple estimate of the capacity requirements, but tends to severely overestimate the capacity, since it assumes strong correlation between the bursts of different clients. We evaluate the resource requirements for combinations of workloads with $\delta = 10$ ms, and compare it with the estimated value which is the sum of the individual capacities of the workloads. Figure 6(a) shows the results when combining different pairs of the three workloads. For WebSearch and FinTrans, the actual capacity needed is only 53% of the estimate, indicating considerable multiplexing gains in the combination. For FinTrans and OpenMail, and OpenMail and WebSearch, the actual capacity needed is 86% and 87% of the estimate. The reason for this high real value is that the capacity needed individually by OpenMail (9241 IOPS) dominates that required by WebSearch (1538 IOPS) and FinTrans (1500 IOPS).

To avoid overprovisioning and provide a good estimate for the required capacity, we argue that capacity provisioning based on workload decomposition works well in practice. In Figure 6(b) and 6(c), we report the capacity requirements based on decompositions of 90% and 95%, with the response time guarantee 10 ms, for the same workload combinations as in Figure 6(a). We can see that after decomposition, the capacity estimate based on adding the individual capacity requirements is very close to the actual capacity needed, with error of 0.3% for WebSearch + FinTrans, error of 0.05% for FinTrans + OpenMail, and error of 0.7% for OpenMail + WebSearch. Similar results can be found for 95%, with the relative errors 6.2%, 2.6% and 0.1% for WebSearch + FinTrans, FinTrans + OpenMail and OpenMail + WebSearch respectively. By removing the high variance portion of the individual workloads, the simple aggregation of the decomposed workloads provides a very good estimate for the combined workload.

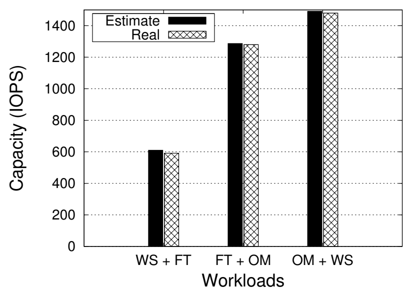
5. Related Work

Recently proposed QoS schedulers for storage servers [1], [2], [3], [4], [17] are generally based on Fair Queuing [13], [14], [18] principles, combined with throughput enhancing mechanisms to exploit locality and concurrency in the storage arrays. These works do not explicitly address the issue of efficiency in resource provisioning, simply requiring the server to be provisioned for the worst-case traffic of each client based on its SLA. As shown in the results this requires significant resource over-provisioning to account for the unpredictable burst-arrival pattern. Our scheduling framework differs the above works by provisioning based on the well-behaved portion that comprises the overwhelming portion of the workload rather than the worst-case bursts, and dynamically decomposing each client workload to conform to the provisioning.

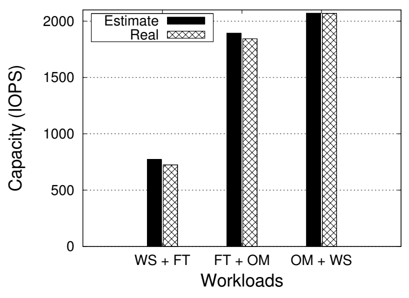
Considerable amount of previous work has been devoted to designing optimal size-aware schedulers to improve performance [19], [20], [21] in Web servers. The basic idea is to



(a) Traditional 100% combine



(b) 90% decomposition combine



(c) 95% decomposition combine

Figure 6. Capacity required for workload multiplexing with response time targets 10 ms

separate jobs in terms of their size to avoid having short jobs getting stuck behind long ones. The SRPT scheduler [19] gives preference to jobs or requests with short remaining processing times to improve mean response time of Web servers. In a clustered server environment, D_EQAL [21] utilizes the size-based policy to assign the jobs to different servers in terms of size distribution, and further enhances this by considering the autocorrelation property of the workload to deliberately unbalance the load to improve the performance. Swap [20] also leverages the size-autocorrelation property of the jobs to simulate the Short Job First scheduler online and delay the long jobs in preference to short ones. Our scheduling framework is designed for storage systems, where the request size is not as diverse as Web applications. The big requests are already partitioned by the OS or storage device driver into smaller-sized block requests, such as up to 32KB. Our work differs from the above works by considering the arrival-autocorrelation (bursty arrival rates) property of the workloads, and then proposes decomposing the workload to different classes dynamically based on their burst characteristics to improve the resource efficiency and performance.

A third body of related work can be found in the considerable literature on network QoS [22] where traffic shaping is used to tailor the workloads to fit QoS-based SLAs. Typically, arriving network traffic is made to conform to a token-bucket model by monitoring the arrivals, and dropping requests that do not conform to the bucket parameters of the SLA. Alternatively, early detection of overload conditions is used to create back pressure to throttle the sources [23]. In storage workload request dropping is not a viable option since the protocols do not support automatic retry mechanisms, and throttling is difficult in an open system and can lead to loss of throughput in disks and storage arrays. Techniques leveraging statistical envelopes have been proposed [9] to reshape inbound traffic and to allocate resources in network systems in order to achieve probablistically bounded service delays, while simultaneously multiplexing system resources among the requesters to achieve higher utilizations.

6. Conclusion and Future Work

We addressed the problem of response time degradation in storage servers caused by the the bursty nature of many storage workloads. Since the arrival rates during a burst can be an order of magnitude or more than the long-term average arrival rate, providing worst-case guarantees requires very significant over provisioning of server resources. Furthermore, even though the bursts make up only a small fraction of the requests, their effects are not isolated but affect even the well-behaved portions of the workload.

We presented a workload shaping framework to address this problem. In our approach, the workload is dynamically decomposed into its bursty and non-bursty portions based on the response time and capacity parameters. By recombining the bursty portions to exploit available slack in the rest of the workload, the entire workload can be scheduled with much smaller capacity and superior response time distribution. We presented an optimal decomposition algorithm RTT and a slack-scheduling recombination method Miser to do the workload shaping, and evaluated it on several storage traces. The results show significant capacity reductions and better response time distributions over non-decomposed traditional scheduling methods. Finally, we showed how the decomposition could be used to provide more accurate capacity estimates for multiplexing several clients on a shared server, thereby improving admission control decisions.

In future work, we are focusing on how the workload shaping affects the disk throughput and how to best apply workload shaping in a multi-client environment. Disk throughput highly depends on the spatial locality in the workload. Thus, during decomposition, we not only need to schedule the requests across different queues to meet response time bounds, but also try to improve the system throughput by considering the data locality in scheduling the queues. In a shared storage environment, each client or application may have different QoS requirements. The workload shaping framework could maintain private primary queues for each client and separate or consolidated overflow queues, while the total bandwidth is shared proportionally among all the clients.

Acknowledgment

Support by NSF grants CNS 0615376 and CNS 0541369 is gratefully acknowledged. We thank the reviewers for their useful comments that helped improve this paper.

References

[1] W. Jin, J. S. Chase, and J. Kaur, "Interposed proportional sharing for a storage service utility," in *Proceedings of SIGMETRICS*, 2004.

[2] J. Zhang, A. Sivasubramaniam, Q. Wang, A. Riska, and E. Riedel, "Storage performance virtualization via throughput and latency control." in *Proceedings of MASCOTS*, 2005.

[3] A. Gulati, A. Merchant, and P. Varman, "*p*Clock: An arrival curve based approach for QoS in shared storage systems," in *Proceedings of SIGMETRICS*, 2007.

[4] C. Lumb, A. Merchant, and G. Alvarez, "Façade: Virtual storage devices with performance guarantees," *Proceedings of FAST*, 2003.

[5] M. Aron, P. Druschel, and W. Zwaenepoel, "Cluster reserves: a mechanism for resource management in cluster-based network servers," in *Proceedings of SIGMETRICS*, 2000.

[6] M. E. Gómez and V. Santonja, "On the impact of workload burstiness on disk performance," in *Workload characterization of emerging computer applications*, 2001.

[7] W. E. Leland, M. S. Taqqu, W. Willinger, and D. V. Wilson, "On the self-similar nature of ethernet traffic," in *IEEE/ACM Trans. Netw*, 1994.

[8] A. Riska and E. Riedel, "Long-range dependence at the disk drive level," in *Proceedings of QEST*, 2006.

[9] E. W. Knightly and N. B. Shroff, "Admission control for statistical qos: theory and practice," in *IEEE Network*, 1999.

[10] "Public software (storage systems department at hp labs)," 2007, http://tesla.hpl.hp.com/publicsoftware/.

[11] D. Narayanan, A. Donnelly, E. Thereska, S. Elnikety, and A. Rowstron, "Everest: Scaling down peak loads through i/o off-loading," in *Proceedings of OSDI*, 2008.

[12] in *http://www.ece.rice.edu/~ll2/distribute/miser.pdf.*

[13] J. C. R. Bennett and H. Zhang, "WF^2Q: Worst-case fair weighted fair queueing," in *Proceedings of INFOCOM*, 1996.

[14] P. Goyal, H. M. Vin, and H. Cheng, "Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks," *IEEE/ACM Trans. Netw.*, vol. 5, no. 5, pp. 690–704, 1997.

[15] Http://www.pdl.cmu.edu/DiskSim/.

[16] "Storage performance council (umass trace repository)," 2007, http://traces.cs.umass.edu/index.php/Storage.

[17] P. J. Shenoy and H. M. Vin, "Cello: a disk scheduling framework for next generation operating systems," in *Proceedings of SIGMETRICS*, 1998.

[18] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queuing algorithm," *Journal of Internetworking Research and Experience*, 1990.

[19] M. Harchol-Balter, B. Schroeder, N. Bansal, and M. Agrawal, "Size-based scheduling to improve web performance," in *ACM Trans. Comput. Syst.*, 2003.

[20] N. Mi, G. Casale, and E. Smirni, "Scheduling for performance and availability in systems with temporal dependent workloads," in *Proceedings of DSN*, 2008.

[21] Q. Zhang, N. Mi, A. Riska, and E. Smirni, "Load unbalancing to improve performance under autocorrelated traffic," in *Proceedings of ICDCS*, 2006.

[22] J. W. Evans and C. Filsfils, "Deploying ip and mpls qos for multiservice networks," in *Morgan Kauffman*, 2007.

[23] S. Floyd and V. Jacobson, "Random early detection gateways for congestion avoidance," in *IEEE/ACM Transactions on Networking*, 1993.