

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

Lanyue Lu[†], Peter Varman[†], Kshitij Doshi^{*}

Rice University[†], Intel Corporation^{*}, USA
ll2@rice.edu, pjv@rice.edu, 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 resource, 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 between the service provider and clients stipulate guarantees on throughput [13, 21] or latency [11, 16] for rate-controlled clients. The service provider must provision sufficient resources to meet these performance guarantees based on an estimation of the resource requirements 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 [4]. 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 [9, 15, 19]. These workloads are char-

acterized 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 excess resource commitments with associated monetary 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. In our approach we modify the characteristics of the arriving workload so that its behavior is dominated by the majority well-behaved portion 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. The performance SLA consequently is specified by a distribution of response times rather than a single worst-case measure. By relaxing the performance guarantees for a small fraction, 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 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 for the original workload, since it is closer to the average rather than the worst-case requirements. This translates into reductions in provisioned capacity, and reduced energy consumption as well. Finally, we show how the framework can be used to improve resource estimates 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 benefits of multiplexing is difficult [14]. 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

In this section, we motivate the idea behind workload shaping. The goal is to smoothen the workload to reduce the unpredictability caused by the bursty arrival patterns, that 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 resulting in unacceptable 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 fraction of bursty behavior has a disproportionate effect on overall performance, as well as provisioning and admission control decisions.

By workload shaping we refer to dynamically modifying the characteristics of the arriving workload so that its behavior is dominated by the majority well-behaved portion of the workload; 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. The 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 ap-

plication (or client) is partitioned into two (or more in general) 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 or delay $R1$ and 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 for the tail of the distribution can be used in this phase. These will be discussed and evaluated later in Sections 3 and 4.

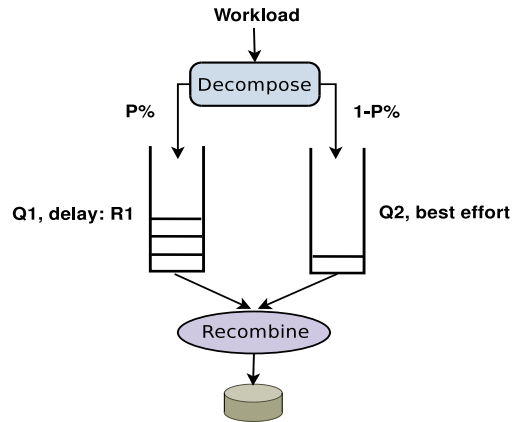


Figure 1. Architecture of workload shaper providing graduated QoS guarantees

Figure 2(a) shows a portion of an OpenMail trace 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. RTT is optimal in the sense that with the same capacity, RTT maximizes the fraction of requests that will meet the response time bound; that is any other decomposition will result in 90% or less of the requests meeting the 10ms deadline. As may be seen Q1 is relatively even at this granularity and this 90% of the original workload can be served to meet the response time bound with a capacity requirement of 1080 IOPS, compared to 9241 IOPS for the original workload. Finally, Figure 2(c) shows the workload following recombination of Q1 and Q2 using the Miser algorithm. 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 Q1 at the earliest such time. Due to the on-

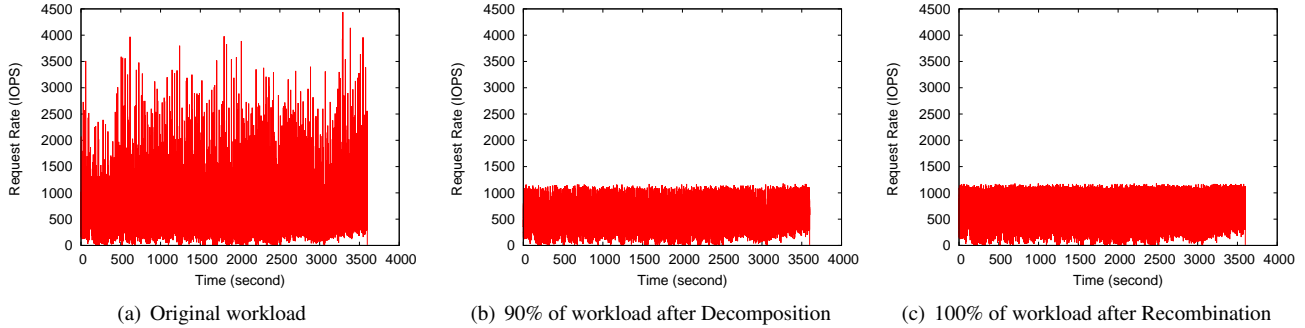


Figure 2. Shaping the OpenMail trace by Decomposition and Recombination

line nature of the recombination process, one can argue (see Section 3.2) that guaranteeing all requests of Q1 when interleaving requests of Q2 is not possible in the worst case by any on-line method, without either placing restrictions on the arrival pattern of Q1 or by increasing the server capacity a small amount. We choose the latter strategy since it is under the control of the resource allocator; in Section 3 we quantify the amount of excess capacity required to guarantee all requests of Q1 when serving both Q1 and Q2 together.

2.1 Decomposition and Recombining Methods

The workload is characterized by its arrival sequence that specifies the number of I/O requests n_i arriving at time a_i , $i = 1, \dots, N$. The Cumulative Arrival Curve (abbreviated AC) $A(t)$ is 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 the 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 at the server). 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 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 will miss their deadlines (as in the two choices mentioned 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 shown 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 3 and 4 when there is slack in the server. Several alternative 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 [18] where bursts of write requests

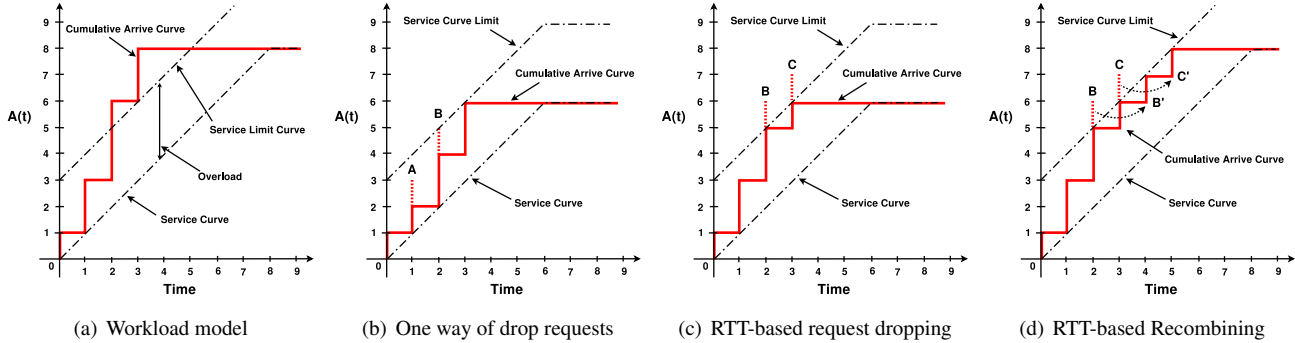


Figure 3. Illustrating the Decomposition and Recombination process

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 server or the need for dedicated resources on the main server, a good strategy is to treat the two parts of the workload as independent and multiplex them on the same 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. In particular the overflow workload is active only during bursts and the capacity during 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 Scheduler, but may in the worst-case slightly increase the fraction of requests missing their deadlines. We provide a theoretical upper bound on the amount of additional capacity required by Miser to prevent this from ever occurring.

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; *i.e.* **Given a response time bound δ , find the minimum server capacity C 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. By performing a binary-search we converge rapidly (within $O(\log C)$ iterations) to the desired minimum capacity C_{min} required to guarantee response time for a fraction f of the workload.

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 have been proposed [14]. 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, as we noted earlier, the worst-case capacity requirements of a client are usually several times that required for the average workload; 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. Different statistical QoS approaches have been proposed to address this usually based on statistical assumptions of the arrival process, to evaluate the chances of overload.

We argue that using the capacity estimate of the reshaped client workload not only reduces the capacity provisioning for a single client, but can provide a good estimate of the capacity required for multiple clients as the sum of these individual capacities. Intuitively this is because the variance in the individual workloads have been reduced by reshaping, and worst and average cases have become closer to

each other. We evaluate this in Section 4 and show that using the aggregated requirements of the reshaped workloads provides a very good estimate of the capacity needed for multiplexing multiple concurrent clients.

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 acts as the overflow area for 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 . Section 3.1 presents the details and theoretical properties of the decomposition algorithm RTT. Methods for recombining the split stream are described in Section 3.2.

3.1 RTT Decomposition

Algorithm 1: RTT Decomposition

```

RTT.Decompose()
begin
  maxQ1 = C × δ
  if lenQ1 ≤ maxQ1 - 1 then
    begin
      Add request to Q1
      Increment lenQ1
    end
  else
    Add request to Q2;
  end
end

```

The *primary queue* Q_1 has bounded length $C \times \delta$, to control the latencies of requests accepted into it. 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. Despite its simplicity, we will prove below that RTT satisfies the following optimality property.

RTT Optimality Property: For a given workload, capacity and response time bound, RTT correctly identifies a maximal-sized set of requests that can meet the deadline, among all online or offline partitioning algorithms.

To show the 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 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 $x < 0$.

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

Proof: By definition, the number of requests with deadline less than or equal to $a_k + \delta$ equals the number of requests arriving at or before time a_k , which equals $A(a_k)$. Similarly the maximum amount of service that can be completed by time $a_k + \delta$ is $S(a_k + \delta)$. Hence, if $A(a_k) > S(a_k + \delta)$ then $\lceil A(a_k) - S(a_k + \delta) \rceil$ of the $A(a_k)$ requests that arrive in the interval $[0, a_k]$ will miss their deadlines. Hence at least $sgn(A(a_k) - S(a_k + \delta))$ requests will need to be dropped in the interval $[0, a_k]$. The largest of these values over all times $a_k, k = 1, \dots, N$ is a lower bound on the number of requests that need to be dropped. \square

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

Proof: Let a_k be the last arrival instant in the busy period at which RTT drops a request. The total service done by RTT in the interval $[0, a_k]$ is $C \times a_k$. Let the total number of requests dropped by RTT prior to a_k be Δ . Now n_k requests arrive at a_k , and any requests which result in a queue length over $maxQ_1$ must be dropped at a_k . That is service to be dropped at a_k is given by $A(a_k) - \Delta - C \times a_k - maxQ_1$. Hence the total service that cannot be completed in $[0, a_k]$ is the sum of the requests dropped at a_k plus the number dropped before a_k (i.e. Δ), and equals $A(a_k) - C \times a_k - maxQ_1 = A(a_k) - C \times (a_k + \delta) = A(a_k) - S(a_k + \delta)$, since RTT is continuously busy in this period. The number of dropped is therefore at most $sgn(A(a_k) - S(a_k + \delta))$. \square

Let intervals I_1, I_2, \dots, I_m be successive *busy periods* of RTT during the time $[0, T]$. In particular $I_1 = [a_{j_1}, b_1]$, $I_2 = [a_{j_2}, b_2] \dots I_k = [a_{j_k}, b_k]$, $I_m = [a_{j_m}, b_m]$; RTT is continuously busy from time a_{j_k} (the start of an interval I_k) till some time b_k , $b_k < a_{j_{k+1}}$, when it becomes idle; it remains idle till the start of the next interval equal to the arrival time $a_{j_{k+1}}$. The following Lemma will be proved by Induction.

Lemma 3: Let OPT be an optimal algorithm that drops the minimal number of requests in $[0, T]$. Then $\forall k, 1 \leq k \leq m$, 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 in I_k .

Proof: We prove the Lemma by induction on the interval number k .

Base Case: For the base case consider the interval I_1

corresponding to $k = 1$. Now RTT server is continuously busy in the interval I_1 and the initial amount of service done by RTT at the start of the interval is zero. Now by Lemma 2 the number of requests dropped by RTT in I_1 equals the lower bound of the number of requests that must miss their deadline in that interval, and hence both OPT and RTT will drop Δ_1 requests. Now RTT is continuously busy throughout I_1 and no further work arrives till the start of interval I_2 ; the idle time cannot be reduced further.

Inductive Step: For the Induction Hypothesis we assume the Lemma is true for all intervals up to I_k and show it holds in the interval I_k . The proof is similar to the base case, additionally noting that by the Induction Hypothesis, OPT has incurred no less idle time than RTT till the start of I_k , and hence cannot have done more service till this time. Then by Lemmas 1 and 2, OPT will need to drop at least Δ_k requests in I_k as well. \square

3.2 Recombining Algorithms

Algorithm 2: Miser Scheduling

```

On a request arrival:
begin
  RTT.Decompose();
  /* Compute Slack*/
  if request  $r_i$  in  $Q_1$  then
     $r_i.slack = \lfloor \max Q_1 - len Q_1 \rfloor$ 
     $minSlack = \min\{minSlack, r_i.slack\}$ 
  end
end

On a request departure:
begin
  /*Dispatch a request*/
  if  $minSlack \geq 1$  then
    Issue request from  $Q_2$ 
  else
    Issue request from  $Q_1$ 
  end

  /*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
    end
  end
end

```

We now describe several strategies for combining the workload spilt by RTT and scheduling them at the server. We describe four scheduling methods to combine the two parts of the workload. Their performance evaluation is described in Section 4. **FCFS:** The requests are not partitioned and serviced in 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 Queuing:** 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 [5], SFQ [10], pClock [11]) 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. Due to its online nature the composite algorithm (RTT + Miser), could sometimes drop more than the theoretical minimum number of requests. We can show theoretically that if $\Delta C = C_{min}$, then this can never occur. Our simulations show that even with a small amount of additional service ΔC , very few (if any) requests in Q_1 are delayed beyond the deadline in practice, and the tail distribution of Q_2 is much nicer.

Algorithm 2 shows the actions taken on request arrival and request completion at the server for the scheduler Miser. 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 is at least 1.

4 Experimental Evaluation

In this section, we evaluate the workload decomposition based scheduling framework using the storage system simulation tool DiskSim [1]. 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 [3] and HP Research Labs [2]. All of these are low-level block storage I/O traces. The WebSearch traces are from a popular search engine and consists 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) com-

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	410	473	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	299	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

parison 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.

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 capacity required for different workload fractions to meet a specified response time target for the three different workloads. Response time bounds of 5, 10, 20, 50 ms and f between 90% to 100% of the workload are considered. As can be seen in Table 1, the capacity required falls off significantly by exempting between 1% and 10% of the workload from the response time guarantee. For instance, with a 5 ms response time, extending the response time guarantee from 90% to 100% of the workload requires large capacity increases: almost 4 times (from 590 to 2325 IOPS) for the WebSearch workload, 7.5 times (from 400 to 3000 IOPS) for FinTrans workload, and more than 10 times (from 1350 to 13990 IOPS) for OpenMail workload. Even going from 99% to 100% the capacity required increases by a factor of 2.4 (from 960 to 2325 IOPS) for WebSearch, a factor of 5 (from 600 to 3000 IOPS) for

FinTrans and a factor of 3.5 (from 3950 to 13990 IOPS) for OpenMail. For higher response times, the capacity required also increases by significant, though smaller factors, as can be seen in the Table. For instance, for OpenMail workload, the required capacity for 100% guarantees is still several times that required to guarantee a reduced fraction: specifically, for response time bounds of 10 ms, 20 ms and 50 ms respectively, the capacity required 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) that can be present in the workload can be gauged by looking at the range from 99% and 100% of FinTrans workload, 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

The results of Section 4.1 show that meeting the guarantees of a relatively small fraction of the workload accounts for a large share of the server capacity requirement. We now investigate the effects of the bursts on the response time of the workload. In a shared data center, scheduling across clients may be done using a fair scheduler or other isolating mechanism, and scheduling at the low level of storage array

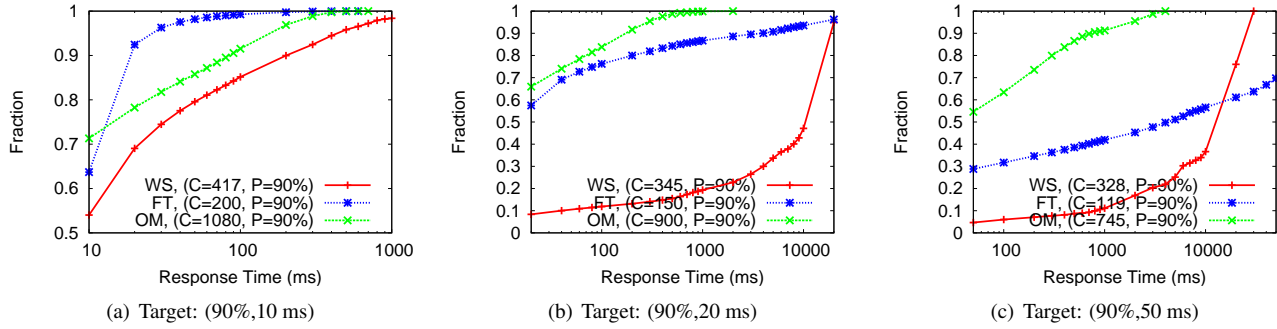


Figure 4. Response time CDF of FCFS scheduling: Different response time targets

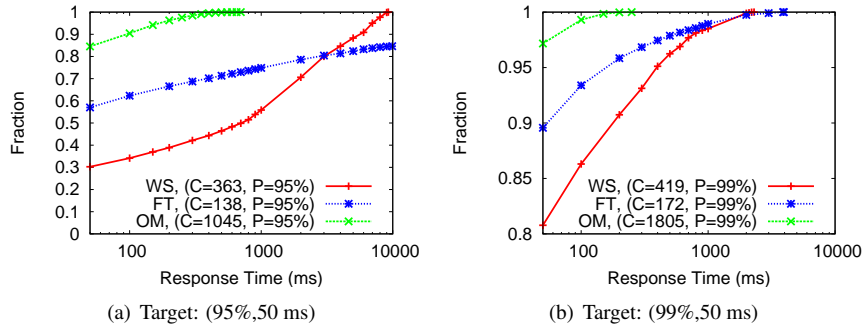


Figure 5. Response time CDF of FCFS scheduling: Different percentage meeting targets

uses some throughput maximizing ordering from among the requests in the low-level queue. However, requests of a single client are usually handled in a simple FCFS manner. The following experiments show that in the presence of bursty traffic within a single client workload, this can result in poor response time profiles. That is, the bursts in the workloads are not sufficiently isolated to prevent them from affecting the behavior of the non-bursty part of the workload, and isolation needs to be enforced specifically by a scheduler.

The cumulative response time distribution obtained for the unpartitioned workloads using FCFS scheduling is shown in Figure 4. Figures 4(a), 4(b) and 4(c) show the response time distribution for the three workloads assuming target response times of 10 ms, 20 ms and 50 ms respectively. 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), at a capacity of 417 IOPS, only 54% of the unpartitioned WebSearch workload meet a 10 ms latency bound. In contrast, in the partitioned workload 90% of the workload would meet the response time bound (see Table 1). The unpartitioned workload reaches 90% compliance only for a response time around 200ms. A similar behavior is shown by the OpenMail workload for a 10 ms response time bound and a capacity of 1080 IOPS. In

the unpartitioned workload, only 71% of the requests meet the response time bound, and the system reaches a 90% compliance at around 90 ms. In contrast, the decomposed workload achieves 90% compliance with the 10ms latency (see Table 1). 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 20 ms. At a capacity of 345 IOPS, only 8% of the unpartitioned WebSearch workload meets the 20 ms deadline, compared to 90% of the partitioned workload. For FinTrans and OpenMail workloads, the corresponding percentages of guarantees are 57% and 66% respectively. In Figure 4(c), 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 numbers are still a low 29% and 55% respectively. The reason for this drop in FCFS performance is in stark contrast to the improvement in performance of the decomposed workload. With a more relaxed response time (50ms instead of 10ms), the partitioned workload can meet the same 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

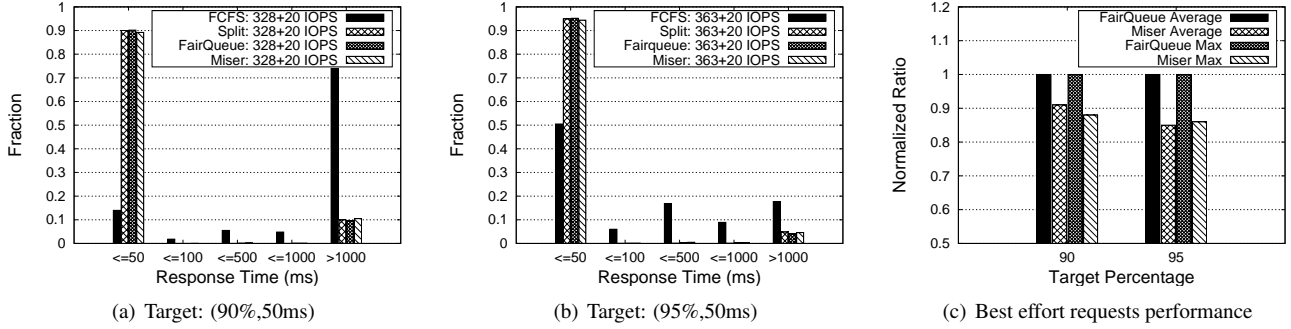


Figure 6. Performance comparison of FCFS, Split, Fair Queuing and Miser: WebSearch workload

part of the workload as well.

When the percentage of guarantees increases to 95% or 99%, the corresponding capacity needed also increases, which will improve FCFS’s performance. In Figure 5, the performance of FCFS at a capacity for which RTT can guarantee 95% and 99% of a workload with 50 ms deadline is shown. In Figure 5(a), the corresponding percentages of guarantees of FCFS for WebSearch, FinTrans and OpenMail are still low: 30%, 57% and 85% respectively. In Figure 5(b), when the target increases to 99%, the corresponding percentages of guarantees of FCFS for WebSearch, FinTrans and OpenMail are 81%, 90% and 97% 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 amount of capacity provided for the workload is held constant, equal to $C_{min} + \Delta C$; C_{min} is the capacity required to guarantee the chosen fraction f of the workload (as obtained from Table 1, and ΔC was chosen to be a small amount $1/\delta$. FCFS uses the total capacity for the unpartitioned workload. For Split and FairQueueing the capacity is divided in the ratio C_{min} to ΔC for the primary and overflow portions of the workload respectively. In Split, the servers cannot be shared and consequently if either the main or overflow server becomes idle, the capacity is wasted even if the other part of the workload has pending requests. On the other hand, FairQueueing 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 6, 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 decomposi-

tion 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 6(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 6(a) and 6(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 the burst in the workload to give good performance to other well behaved requests, leading to larger delays of the overflowing requests. But as the above figures show, the total number of long delayed requests (greater than 1s in the Figures) is less than in FCFS, even though the largest value may be higher.

Finally we compare the performance of Split, FairQueueing and Miser. For Split, the capacity is partitioned without any sharing between the two classes, leading to very bad performance of the secondary class. In this experiment, both the average and maximal response time of secondary class in Split is an order of magnitude bigger than that of FairQueue and Miser. FairQueueing assigns the weighted capacity to the two classes without any preference. The overflow class can only use the spare capacity of the primary when the latter has no requests. However, for Miser, it dynamically monitors the slack of the primary class, and uses it to improve the performance of the secondary class requests. Figure 6(c) shows the average and maximal response time of the secondary class of Miser normalized to that of FairQueueing in the above

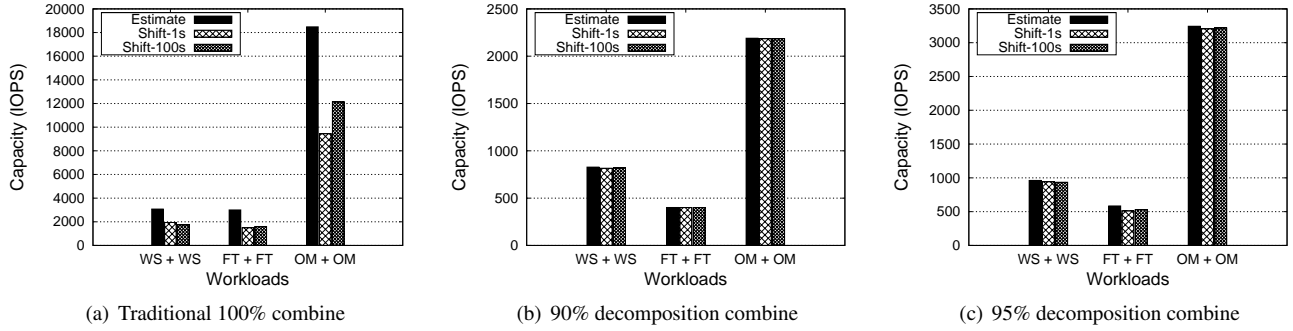


Figure 7. Capacity required for the same workloads multiplexing

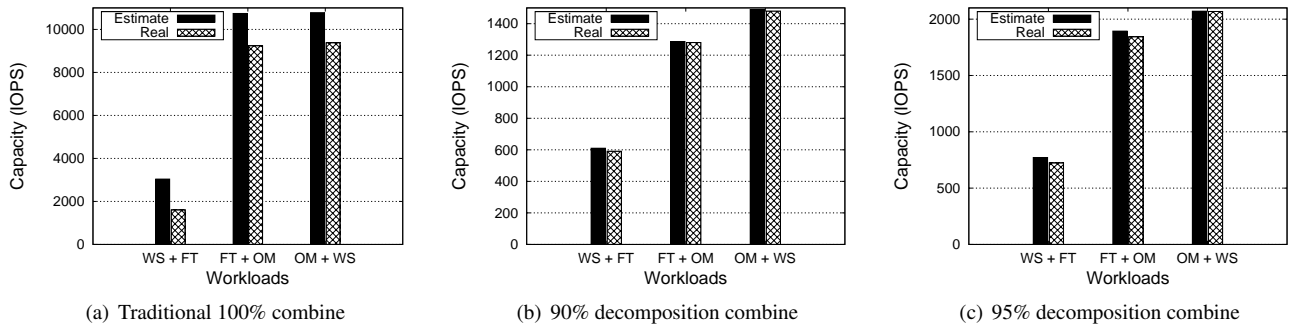


Figure 8. Capacity required for the different workloads multiplexing

experiments. We can see that for WebSearch workload, 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 overestimate the capacity, since it assumes strong correlation between the bursts of different clients. We evaluate the resource requirements for combinations of the same (Figure 7) and different (Figure 8) workloads based on a maximum response time of 10 ms, and compare it with the estimated value which is the sum of the individual capacities of the workloads.

Figure 7(a) shows the capacity needed for combining two identical workloads. The estimated capacity for the pair of workloads is twice the capacity needed by each individual workload, because in the worst case their bursts or peaks overlap exactly. Shift-1s and shift-100s means that one workload is shifted in time by 1 or 100 seconds, then merged with the other workload, to reflect a real multiplexing of the workloads. In Figure 7(a), we can see that for

WebSearch, FinTrans and OpenMail, the capacity needed respectively for Shift-1s is 63%, 50% and 51% of the estimate. For Shift-100s, the capacity needed is 56%, 53% and 66% of the estimate. So, if the bursts or peaks of the two workloads are not overlapped exactly as would be, the worst case provisioning is much more than actually needed.

To avoid overprovisioning and provide a good estimate for the required capacity, we argue that capacity provisioning based on workload decomposition works well in real cases. In Figure 7(b) and 7(c), we show the capacity requirements based on decompositions of 90% and 95%, with the response time guarantee 10 ms, for combining the same workloads of Figure 7(a). After decomposition, the actual capacity needed by shift-1s and shift-100s is very near the estimated capacity, with error of 1% for WebSearch, error of 0.1% for FinTrans and error of 0.2% for OpenMail. Similar results can be found for 95%, with relative errors of 3%, 12.5% and 1% for WebSearch, FinTrans and OpenMail respectively.

Figure 8(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, OpenMail and WebSearch, the actual capacity needed is 86% and 87% of the estimate. The

reason of this high real value is that the capacity needed individually by OpenMail (9241 IOPS) is much higher than WebSearch (1538 IOPS) and FinTrans (1500 IOPS), thus the resulting combined workload at least needs the amount of 9241 IOPS. The capacity provisioning based on workload decomposition also works well for combining different workloads in real cases. In Figure 8(b) and 8(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 8(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 [20, 16, 13, 21, 11] are generally based on Fair Queuing [6, 5, 10] 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 the designing optimal size-aware schedulers to improve performance [12, 22, 17] in Web servers. The basic idea is to separate jobs in terms of their size to avoid having short jobs getting stuck behind long ones. The SRPT scheduler [12] 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 [22] 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 [17] also leverages the size-autocorrelation property of the jobs to do an online

simulate the Short Job First scheduler and delay the long jobs in preference to short ones. Our scheduling framework is designed for storage system, 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 [7] 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 [8]. 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 [14] to reshape inbound traffic and to allocate resources in network systems in order to achieve probabilistically bounded service delays, while simultaneously multiplexing system resources among the requesters to achieve higher utilizations.

6 Conclusion

In this paper 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 steady state 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.

References

- [1] <http://www.pdl.cmu.edu/DiskSim/>.
- [2] Public software (storage systems department at hp labs), 2007. <http://tesla.hpl.hp.com/publicsoftware/>.
- [3] Storage performance council (umass trace repository), 2007. <http://traces.cs.umass.edu/index.php/Storage>.
- [4] M. Aron, P. Druschel, and W. Zwaenepoel. Cluster reserves: a mechanism for resource management in cluster-based network servers. In *Proceedings of ACM SIGMETRICS*, 2000.
- [5] J. C. R. Bennett and H. Zhang. WF^2Q : Worst-case fair weighted fair queueing. In *INFOCOM*, 1996.
- [6] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queuing algorithm. *Journal of Internet Research and Experience*, 1990.
- [7] J. W. Evans and C. Filsfil. Deploying ip and mpls qos for multiservice networks. In *Morgan Kauffman*, 2007.
- [8] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. In *IEEE/ACM Transactions on Networking*, 1993.
- [9] M. E. Gómez and V. Santonja. On the impact of workload burstiness on disk performance. In *Workload characterization of emerging computer applications*, 2001.
- [10] 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.*, 5(5):690–704, 1997.
- [11] A. Gulati, A. Merchant, and P. Varman. *pClock*: An arrival curve based approach for QoS in shared storage systems. In *Proc. ACM Intl Conf on Measurement and Modeling of Comp. Sys. (SIGMETRICS)*, 2007.
- [12] M. Harchol-Balter, B. Schroeder, N. Bansal, and M. Agrawal. Size-based scheduling to improve web performance. In *ACM Trans. Comput. Syst.*, 2003.
- [13] W. Jin, J. S. Chase, and J. Kaur. Interposed proportional sharing for a storage service utility. In *SIGMETRICS*, 2004.
- [14] E. W. Knightly and N. B. Shroff. Admission control for statistical qos: theory and practice. In *IEEE Network*, 1999.
- [15] 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.
- [16] C. Lumb, A. Merchant, and G. Alvarez. *Facade*: Virtual storage devices with performance guarantees. *FAST*, 2003.
- [17] N. Mi, G. Casale, and E. Smirni. Scheduling for performance and availability in systems with temporal dependent workloads. In *Proceedings of IEEE DSN*, 2008.
- [18] 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.
- [19] A. Riska and E. Riedel. Long-range dependence at the disk drive level. In *Proceedings of QEST*, 2006.
- [20] P. J. Shenoy and H. M. Vin. Cello: a disk scheduling framework for next generation operating systems. In *ACM SIGMETRICS*, 1998.
- [21] J. Zhang, A. Sivasubramaniam, Q. Wang, A. Riska, and E. Riedel. Storage performance virtualization via throughput and latency control. In *MASCOTS*, 2005.
- [22] Q. Zhang, N. Mi, A. Riska, and E. Smirni. Load unbalancing to improve performance under autocorrelated traffic. In *Proceedings of ICDCS*, 2006.