Decomposing Workload Bursts for Efficient Storage Resource Management

Lanyue Lu, *Member, IEEE,* Peter J. Varman, *Senior Member, IEEE,* Kshitij Doshi, *Member, IEEE*

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 requirements, management costs, and energy consumption. We present a novel 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 and power consumption significantly, 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 evaluating the performance of several storage workloads using both simulation and Linux implementation.

Index Terms—Workload Decomposition, Graduated QoS, Storage System, Resource Management, Scheduling

1 Introduction

The increasing complexity of storage management and the economic benefits of consolidation are driving storage systems towards a service-oriented paradigm, in which personal and corporate clients lease space and access bandwidth on shared storage servers. In a typical setup, Service Level Agreements (SLAs) 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. The run-time system must isolate the clients to avoid interference, and schedule their requests appropriately [5]. A fundamental challenge in data center operations is the need to deal effectively with *high-variance* bursty workloads arising in the network, storage server, and multi-tiered systems [6; 7; 8; 9]. These workloads are characterized by unpredictable periods of high activity 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 adjacent well-behaved regions of the workload as well. Consequently, 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, leading to 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 improve client performance and reduce resource provisioning at the server. In our approach we modify the characteristics of the arriving workload so that its behavior is dominated by the largely 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. By slightly relaxing the performance requirements for a small fraction of the workload, 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 negative consequences 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, being 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 estimates of the aggregate resource requirements of multiple concurrent clients. Due to statistical variations, the peaks of the workloads are unlikely to line up simultaneously. Simple aggregation of the clients' requirements leads to significant over provisioning of the capacity, but estimating the savings from statistical multiplexing is difficult [10]. We show that aggregation of the capacities needed by the reshaped workloads provides more realistic estimates of system resource requirements. This paper extends our previous work [11] in two ways. First we analyzed the optimality of our workload decomposition technique and applied it to reduce power consumption in storage systems. In addition, we implemented the scheduling framework in the Linux kernel, and evaluated it on an actual system using recent enterprise storage workloads.

The rest of the paper is organized as follows. Section 2 describes the workload shaping framework and its use for better capacity and power provisioning. An optimal decomposition algorithm RTT is described and analyzed 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 and performance degradation caused by bursty arrival patterns. With traditional scheduling methods the effects of these bursts are not confined to the localized regions where they occur, but spill over and affect the following well-behaved regions of the workload as well. Consequently, even a small amount of bursty behavior can have a disproportionate effect on performance and provisioning. 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. Figure 1 shows a two-class scheme with queues Q1 and Q2 respectively. 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 suitably multiplexed to satisfy the individual performance constraints. Scheduling algorithms with different response time distributions can be used in this phase as discussed in Section 3.

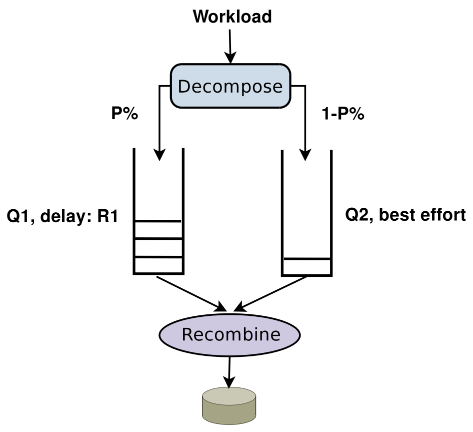


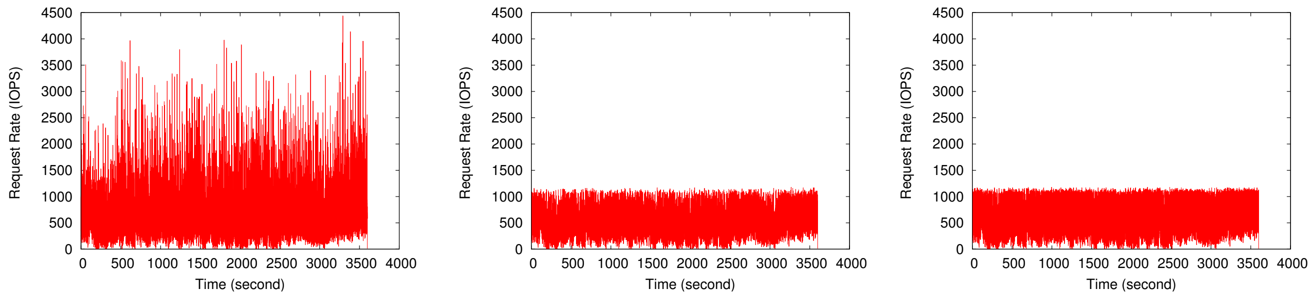
Fig. 1. Architecture of workload shaper providing graduated QoS guarantees

Figure 2(a) shows a portion of an OpenMail trace [12] 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 waiting in 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.* $A(t) = \sum_{j=1}^{K} n_j$, where $a_K \leq t < a_{K+1}$. 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 pending requests. The Service Curve (SC) is shown by a line of slope C during a server busy period. At any time, the vertical distance between SC and AC is the number of pending requests (either queued or in service). 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 number of pending requests 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 overshooting the SCL result in overload, and some requests must be dropped in order for the rest to meet their deadline. When 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. The modified AC now lies below the SCL which means that all these later requests will meet their deadlines. A different choice of dropped requests is shown in Figure 3(c), where requests at times 2 and 3 are removed. For this example at least two requests must necessarily miss their deadlines. On the other hand dropping two requests at time 1 is a poor choice,



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

Fig. 2. Shaping the OpenMail trace by Decomposition and Recombination

since a request arriving at time 3 will still miss its deadline. Note also that the decomposition method needs to be online and make a decision to keep or drop a request without knowing the future request pattern. We show in Section 3 that our online decomposition algorithm RTT 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 [13] 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 called Miser to combine the two portions of the workload. Miser allows better shaping of the tail of the workload than a Fair Queueing scheduler, but in some cases may increase the fraction of requests missing their deadlines.

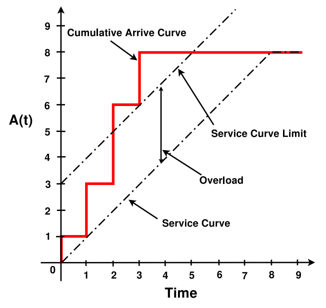
2.2 Capacity Provisioning

We profile the workload of a client to determine the capacity reservation needed to meet its QoS guarantees. For a response time bound δ, we estimate the minimum server capacity C_{min} required for a stipulated fraction f of the requests to meet their deadlines. This is done by an efficient search of the state space as follows. We use the optimal RTT algorithm (described in Section 3.1) to find the maximum number of requests that can meet the response time bound δ, with a fixed capacity C. If this 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 within a logarithmic number of probes to the minimum capacity C_{min}. We provision a capacity of $C_{min} + \Delta C$, where the latter is used to prevent starvation of the best-effort requests in Q2. In our simulation experiments an additional capacity of $\Delta C = 1/\delta$ was found to be sufficient to obtain good performance for the entire workload.

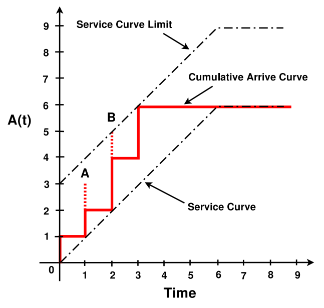
Figures 4(a) and 4(b) show the QoS variation of the Financial Transaction workload from UMass Storage Repository [14] as the capacity is varied. They show the server capacity in IOs/sec (IOPS) needed for a fraction f of the requests in the workload, to meet response time bounds of 50 ms, 20 ms and 10 ms, for f between 90% and 100%, and 99% to 100% respectively. As can be seen, the capacity required falls off significantly by exempting between 1% and 10% of the workload from the response time guarantee. For a 10ms latency, the capacity increases 7.5 times (from 200 IOPS to 1500 IOPS) when f increases from 90% to 100%, and by a factor of 4.2 in going from 99% to 100%. Corresponding capacity increases by factors of 5.0 (10%) and 3.5 (1%) for a response time of 20ms can be observed. In fact, for response times of 10ms, 20ms and 50ms, the capacity increases in going from 99.9% to 100% guarantees are by factors of 3.0, 2.7, and 1.6 respectively. Similar trends for other storage workload are noted in our experiments, and presented in detail in Section 4.

Multiple Concurrent Clients: In a data center environment, the service provider needs to provision sufficient resources for several clients simultaneously sharing the system. A brute-force approach is to simply aggregate the worst-case capacities required for each client. However this results in poor server utilization and unnecessarily conservative admission control. There are two main issues: first, as shown above the worst-case capacity requirements of a client are usually several times the average demand; secondly, adding the individual capacity requirements presumes that the peaks of all the individual workloads occur simultaneously, an extremely unlikely situation in practice.

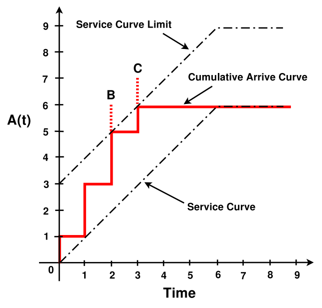
We argue that aggregation of the capacity requirements of individually reshaped workloads provides a good estimate of the server capacity needed for handling multiple concurrent



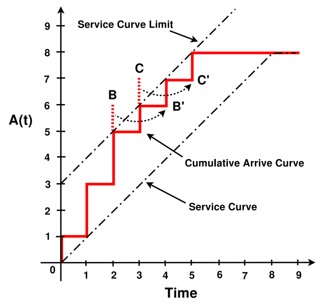
(a) Workload model



(b) One way of drop requests

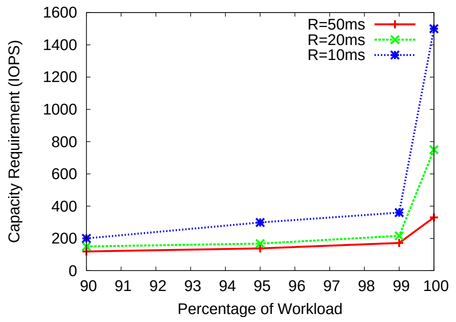


(c) RTT-based request dropping

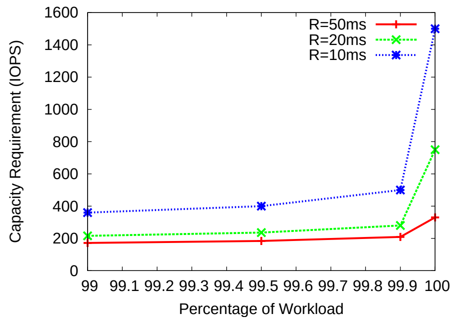


(d) RTT-based Recombining

Fig. 3. Illustrating the Decomposition and Recombination process



(a) f: from 90% to 100%



(b) f: from 99% to 100%

Fig. 4. Capacity required for different percentages of the workload to meet a specified latency bound (Financial Transaction trace)

clients. That is, the capacity required by the reshaped workload serves as a measure of "effective bandwidth" [10] of the client. We evaluate this in Section 4 and show that using the aggregated effective bandwidths provides a much better estimate of the required server capacity.

2.3 Power Efficiency

Many proposals for power management of general servers and storage systems are based on exploiting the temporal periodicity in workloads [15; 16; 17]. The load fluctuates in daily, weekly or even monthly cycles, with periods of high load and periods of low load. The periodic pattern provides opportunities for predicting the future traffic and varying the number of active servers (or storage pools) accordingly, to meet the performance requirements in different time periods. By keeping the inactive servers in the low power (or powered down) state, power consumption is reduced during periods of low utilization.

At the end of a time epoch, the load in the next epoch is predicted, and just enough servers are spun up (or down) to satisfy the performance QoS in that interval. Although the longer term trends of the workload are predictable, the workload is very bursty at a finer granularity, meaning that the instantaneous arrival rates in some time intervals can be higher than the long-term rate by an order or two in magnitude. Thus during the high load period, estimates based on worst-case patterns result in significant over-provisioning of capacity and increased power consumption. On the other hand, spinning up a powered-down commodity disk can take tens of seconds, and starting a server from the sleeping state needs up to several minutes to power on and warm up; hence, changing the number of active servers dynamically at a fine granularity is not a viable option. Consequently, to meet the QoS performance requirements, a large number of servers, (in some cases maybe all the servers), are always kept powered on, although they are critical only for short periods of bursty or worst-case activity. This results in significant power consumption even if most of the time the workload is relatively low.

A recently proposed method PowerNap [18] advocates the elimination of server idle power by quickly transitioning in and out of an ultra-low power state in response to instantaneous load. However, PowerNap assumes the use of SSD storage devices, which are not widely deployed at present. Furthermore, PowerNap focuses on saving idle power consumption, while we target decreasing the peak power by decreasing resource requirements. PowerNap provides a complementary approach orthogonal to our method, and targets idle power rather than peak provisioning.

2.3.1 Power Model

The experiments in Figure 4 show that by using a graduated QoS policy to shape the workload, one can reduce the capacity requirements significantly. We exploit this observation to reduce power consumption in typical data center environments. Our power savings due to decomposition come from powering up fewer disks than the baseline case. In contrast to the approaches described above, we focus on decreasing the peak power consumption instead of keeping disks in lower power states as long as possible.

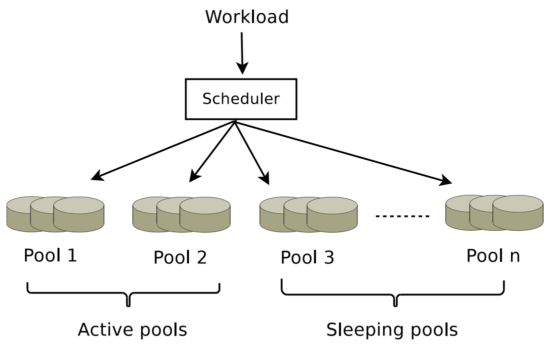


Fig. 5. Runtime scheduler for storage pools

Figure 5 shows the basic architecture of the target storage system logically organized as multiple *storage pools*. A pool may be considered as a logical volume that stores the entire data set. For reliability and performance, data is replicated and stored redundantly in several pools. Data replication is very common in current large-scale data centers and enterprise storage systems. For example, Hadoop DFS [19] and Google File System [20] both maintain at least three copies of each data block. With the increasing popularity of cloud storage service like Amazon S3, redundant data storage both within and across data centers is being increasingly employed to meet reliability, availability, and performance requirements.

At the start of an epoch, a subset of the storage pools are placed in the powered on state (ON pools) and the rest are powered down in the sleep mode (OFF pools). The number of ON pools is estimated by analyzing the workload using the workload decomposition algorithm RTT (explained in section 3). We first use RTT to statically profile the workload to get the capacity requirement C_{total} for providing the QoS guarantees for the workload during this epoch. If the capacity of each pool is C_{pool}, then a conservative estimate of the number of pools that must be powered ON during this epoch is $\lceil C_{total}/C_{pool} \rceil$. Using the example in Figure 4, it requires a server capacity of $C_{total} = 1500$ IOPS to guarantee a 10ms response time for 100% of the workload, while satisfying 99% of workload with a 10ms deadline only requires a capacity of around 400 IOPS. The remaining 1% has a much larger deadline and requires only a small additional capacity. Assuming, for instance, that each pool has capacity C_{pool} of around 500 IOPS, then provisioning 100% of workload needs 3 pools while provisioning 99% of the workload only needs 1 active pool while keeping the 2 pools in the OFF state for power conservation.

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 latency 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 bound for the requests in the primary queue is δ.

3.1 RTT Decomposition

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*:

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.

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
```

3.2 Proof of RTT Optimality

To show the optimality of RTT, we first show that in any period that RTT is continuously busy, the number of requests it drops is the minimum possible. Lemma 3.1 shows a lower bound on the number of dropped requests in any interval, and Lemma 3.2 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 Theorem 3.3, that RTT cumulatively drops no more than a hypothetical optimal algorithm OPT at the end of any busy period.

Recall from Section 2.1 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 3.1. *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 $a_k, k = 1, \cdots N$ is a lower bound on the number of requests that need to be dropped. □

Lemma 3.2. *In any busy period* $[a_1, a_N]$*, the number of requests that RTT will drop is no more than* $max_{1 \le i \le N}\{sgn(A(a_i) - S(a_i + \delta))\}$.

Proof: Without loss of generality we can assume that $a_1 = 0$. 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 requests is therefore at most $sgn(A(a_k) - S(a_k + \delta))$. □

Let intervals I_1, $I_2, \cdots I_m$ be successive *busy periods* of RTT during the time $[0, T]$. In particular $I_1 = [a_{j_1}, b_1]$, $I_2 = [a_{j_2}, b_2] \cdots I_k = [a_{j_k}, b_k]$, $I_m = [a_{j_m}, b_m]$; 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.

Theorem 3.3. *Let OPT be an optimal algorithm that drops the smallest number of requests in* $[0, T]$*. Then* $\forall k, 1 \le k \le m$*, 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. □

3.3 Recombining Algorithms

We now describe four methods for combining the workload split 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 [21], SFQ [22], RFQ [23]) that divides the server capacity in the specified ratio of $C_{min} : \Delta C$. The total capacity of the server is $C_{min} + \Delta C$, but by sharing a single physical server we hope to leverage the benefits of statistical multiplexing.
- **Miser**: The scheduler uses the free slack in the scheduling of the primary queue to schedule requests in Q_2 as early as possible. Unlike the previous two methods the two queues are more closely coupled. Due to its online nature, servicing requests of Q_2 could trigger overflow of future requests in some cases. That is, Miser could sometimes drop more than the theoretically minimum number of requests. Our simulations show that even with a small amount of additional capacity ΔC, few additional requests are delayed beyond the deadline in practice, and the tail distribution of Q_2 is much nicer.

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.

If there are several different workloads with different response time requirements, the underlying server capacity is multiplexed among the workloads using a fair sharing scheduler like [3]. Decomposition and recombination are performed independently for each workload, which has its own Q1 and Q2.

4 Experimental Evaluation

In this section, we evaluate the workload shaping based scheduling framework using the storage system simulation tool DiskSim [24] and actual Linux implementation. For the simulation we assumed a simple constant-rate server model in order to concentrate on the workload characteristics and properties of the scheduling algorithms. For the actual implementation we implemented the framework as a Linux device driver module and compared it with FCFS by replaying several real storage traces, with a mix of different request sizes, types, and access patterns (random and sequential). The Linux implementation was tested on a server equipped with a hard drive as the storage device.

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
        Remove a request from $Q_2$ in FIFO order,
        and dispatch it
    else
        Remove a request from $Q_1$ in FIFO order,
        and dispatch it

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

For the DiskSim experiments, we used traces of three different storage applications: Web Search Engine (WebSearch), OLTP application (FinTrans) and Email service (OpenMail). The traces are obtained from the UMass Storage Repository [14] and HP Research Labs [12]. All of these are low-level block storage I/O traces, which have not been served by the 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. For the Linux experiments, we used a more recent Microsoft Research Cambridge trace (MSRC) [25]. This is also a block-level trace below the file system cache. It captures activity on 13 different type of servers with 36 storage volumes during a one week period. It contains user home directories servers, source control servers, web servers, print servers, research projects servers, etc.

We conducted six types of experiments: (i) measuring server capacity requirements as a function of the fraction f of requests meeting the latency bound δ; (ii) the tradeoff between the power consumption and the performance guarantees in the graduated QoS model; (iii) response time distribution of a traditional FCFS scheduler; (iv) comparison of the response time distribution of recombination algorithms Split, Fair Queueing and Miser with FCFS and relative to each other; (v) capacity estimation for multiple concurrent clients using decomposition and (vi) response time distribution of Miser using the Linux prototype implementation in the actual test machine.

4.1 Capacity-QoS Tradeoffs

Resource 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 meets a 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 will suffice. 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 between 3.0 and 1.6 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 required capacity. The more aggressive the QoS specifications (lower response time) the greater the potential for capacity savings.

4.2 Power-QoS Tradeoffs

We evaluate the power efficiency of the workload shaping framework with OpenMail, TPC-D and WebSearch traces in this section. The test system consists of multiple storage pools containing several disks as shown in Figure 5. The baseline system provisions enough capacity to serve the entire workload (100%) with a 20ms response time guarantee. By decomposing the workload using RTT, we filter out the burstiest 1% of the workload and serve the rest of the requests with the same 20ms response time guarantee.

A disk may be in any of three states: *sleep*, *idle* or *active*. In the *idle* state the disk is powered on but is not actively seeking, while in the *active* state it is performing a read or write operation. When in *sleep* mode, the disk is assumed to

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

consume negligible power. The energy consumption of a single disk E_{disk} is calculated by weighting its power consumption in a particular mode by the time spent in that mode. The total energy is the sum of that consumed by each the disk, $\sum_i E_{disk}(i)$.

$$E_{disk} = t_{active} \times P_{active} + t_{idle} \times P_{idle} + t_{sleep} \times P_{sleep} \quad (1)$$

We use the data from an IBM Ultrastar 36Z15 disk model for which the active power is 13.5 W and idle power is 10.2 W.

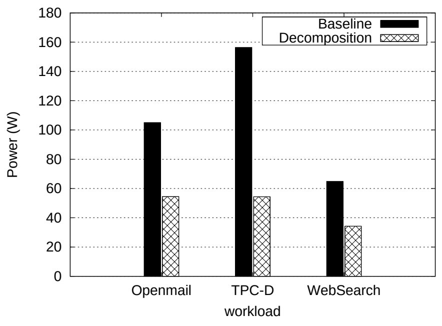


Fig. 6. Power consumption for OpenMail, TPC-D and WebSearch workload

The RTT decomposition algorithm partitions the workload to different classes at runtime to obtain the required 99%-1% split, and issues them to the underlying storage pools. Within the collection of ON pools, requests are sent to the disks in one pool as long as it can handle the workload. This allows the disks in the remaining pools to stay in the lower power idle state, until forced to become active to serve a request from an overloaded disk.

In Figure 6, we compare the power consumption of the baseline system with that obtained by the decomposition. For OpenMail, TPC-D and WebSearch workloads respectively, the baseline system consumes 1.93, 2.88 and 1.90 times the power of the decomposition-based scheme, while only serving 1% additional requests within 20ms. The measured response times using the baseline system and the decomposed workload are shown in Table 2. We note that both finish their specified percentages (100% and 99%) within the guaranteed 20ms bound. From the CDF of the response time, we can see that for response time categories of 10ms,15ms and 20ms, the results of decomposition method are very close to that of the baseline.

4.3 Response Time Distribution of FCFS

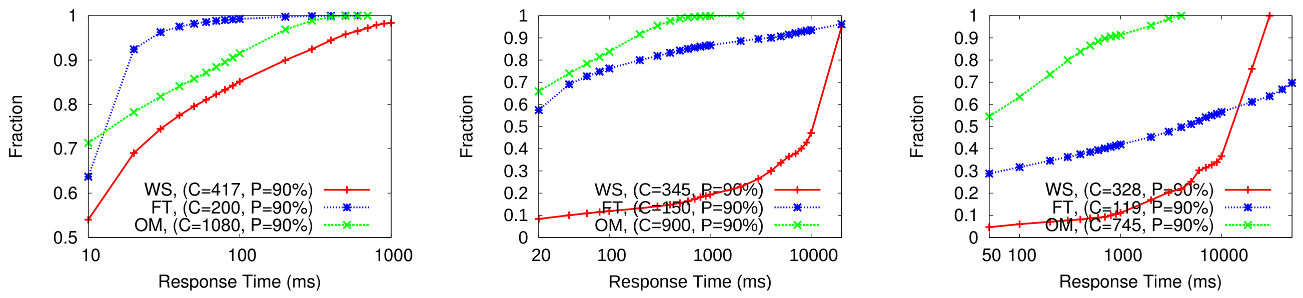
We next investigate the response time distribution when the workload is not decomposed, and simply served using FCFS scheduling. The results show that the bursts in the workload significantly degrade its response time profile. This is because the effects of bursts are not isolated but affect the behavior of the non-bursty part of the workload as well.

The cumulative response time distribution obtained for the unpartitioned workloads using FCFS scheduling is shown in Figure 7. Figures 7(a), 7(b) and 7(c) show the response time distribution for the three workloads assuming target response times of 10ms, 20ms and 50ms respectively. In each case the capacity (C in the figure) is chosen so that 90% of the workload can meet the response time target (P in the figure) if it were optimally decomposed using RTT.

In Figure 7(a), at a capacity of 417 IOPS, only 54% of the WebSearch workload meets 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 10ms 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 90ms, 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 7(b), the response time target is 20ms. 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 are 57% and 66% respectively. In Figure 7(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% for 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

Trace	< 10 ms	< 15 ms	< 20 ms
OM(base)	99.87%	99.98%	100.0%
OM(decom)	94.53%	98.38%	99.00%
TPC-D(base)	99.51%	99.94%	100.0%
TPC-D(decom)	97.78%	98.85%	99.0%
WS(base)	99.92%	99.99%	100.0%
WS(decom)	92.24%	98.58%	99.0%

TABLE 2
Response time performance comparison for OpenMail, TPC-D and WebSearch



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

(b) Target: (90%,20 ms)

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

Fig. 7. Response time CDF of FCFS scheduling for different response time targets

same 90% compliance with a smaller capacity; however, for FCFS the smaller capacity results in the queues built up during the burst to drain even more slowly, increasing the response time for the well behaved part of the workload as well. Thus, as the capacity becomes smaller (in going from Figure 7(a) to Figure 7(c)), the performance of FCFS becomes worse.

4.4 Response Time of Shaped Workload

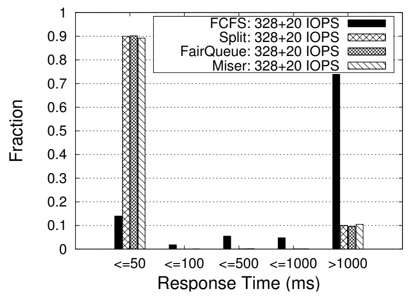
In this section, we evaluate the recombination methods discussed in Section 3.3, Split, FairQueueing and Miser, and compare them with the performance of FCFS. In each case the total capacity provided for the workload is held fixed at $C_{min} + \Delta C$; ΔC was chosen to be a small amount $1/\delta$. FCFS uses the total capacity for the unpartitioned workload. For Split and FairQueueing 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. 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 8, we evaluate the scheduling performance for WebSearch workload with the response time target of 50ms. We can see that Split and FairQueueing achieve the 90% target of 50ms response time following decomposition of the workload. Miser, as noted previously, may incur some additional misses, but is still close to the 90% target. However, FCFS can only finish 14% of the requests within 50 ms. Furthermore, FCFS has 74% of requests with response time bigger than 1000ms, while Split, FairQueueing and Miser have about 10%. Figure 8(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 1000ms, Split has 4.9%, FairQueueing has 4.1% and Miser has 4.6% of the requests respectively, while FCFS has 17.7%. Figures 8(a) and 8(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 long-delay requests (greater than 1s) is much less than for FCFS, even though the largest value may be higher.

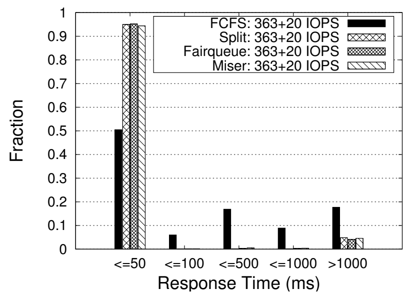
Finally we compare the performance of Split, FairQueueing and Miser. For Split there is no capacity sharing between the two classes; this results in an order of magnitude higher latency (both average and maximum) for Q_2's requests. FairQueueing assigns the weighted capacity to the two classes; Q_2 can only use the capacity of Q_1 if the latter has no requests. However, Miser dynamically monitors the slack in Q_1, and uses it to improve Q_2's performance. Figure 8(c) shows the average and maximal response time of Q_2's requests attained by Miser normalized to those of FairQueueing in the above experiments. We can see that for WebSearch the average response time of Q_2's requests obtained by Miser is about 85% - 90% of FairQueueing, while maximal response time is roughly 85%.

4.5 Multi-flow Consolidation

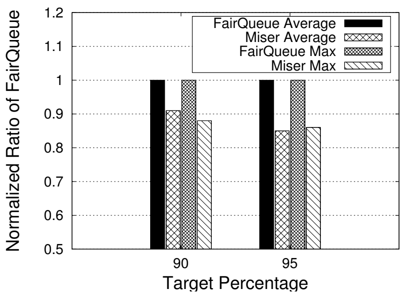
In a shared server environment, accurate prediction of aggregate resource requirements is complicated by workload multiplexing. Simple addition of the clients' requirements overestimate the capacity, since it pessimistically assumes strong temporal correlation between the bursts of different clients. We evaluate the actual capacity needed for different combinations of workloads with a response time of 10ms, and



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

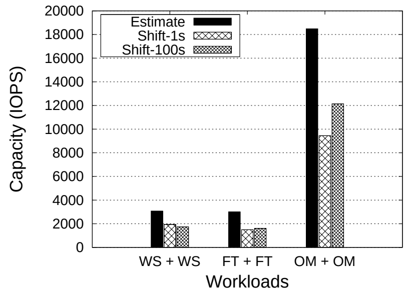


(b) Target: (95%,50ms)

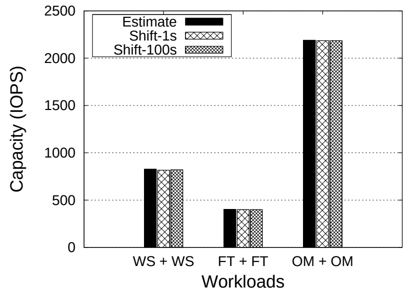


(c) Best effort requests performance

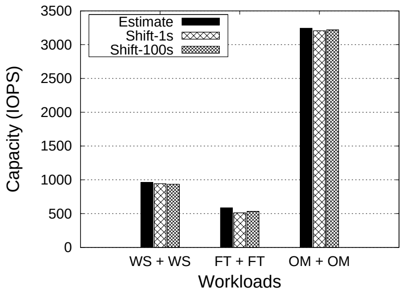
Fig. 8. Performance comparison of FCFS, Split, Fair Queuing and Miser: WebSearch workload



(a) Traditional 100% combine

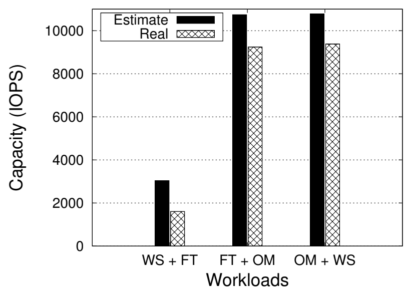


(b) 90% decomposition combine

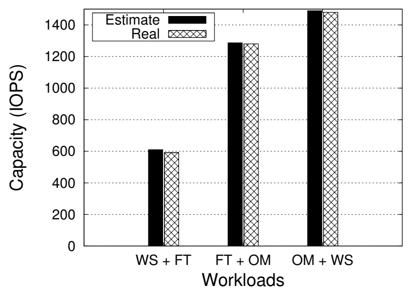


(c) 95% decomposition combine

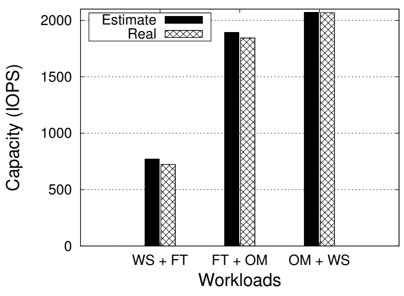
Fig. 9. Capacity required for the same workloads multiplexing



(a) Traditional 100% combine



(b) 90% decomposition combine



(c) 95% decomposition combine

Fig. 10. Capacity required for the different workloads multiplexing

compare it with the simple estimate equal to the sum of the individual client capacities.

We profile the workload trace to determine the minimum capacity needed for the stipulated fraction to meet the response time bound. We then filter the workload using this capacity by running it through the RTT decomposition algorithm; the requests that are not dropped by RTT form the decomposed workload. We aggregate two workloads by simply merging their requests together based on their arrival time, and use the same profiling method to find the capacity required to serve 100% of the combined workload.

Figure 9(a) shows the capacity needed when two identical workloads are combined. The estimated capacity for the pair is twice the individual requirement. Shift-1s and shift-100s means that one workload is shifted in time by 1 second or 100 seconds, then merged with the other workload. Figure 9(a), shows that for WebSearch, FinTrans and OpenMail respectively, the capacities needed for Shift-1s are 63%, 50% and 51% of the estimate. For Shift-100s, the capacities needed are 56%, 53% and 66% of the estimate.

We argue that aggregation based on the decomposed workload provides a better estimate of capacity requirements. In Figure 9(b) and 9(c), we show the capacity requirements based on decompositions of 90% and 95% and a 10ms response time, for the workloads of Figure 9(a). After decomposition, the actual capacities needed by shift-1s and shift-100s are very near their estimates, with errors of 1% for WebSearch, 0.1% for FinTrans and 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. The decomposition process removes the most bursty part from the workload, leaving a smoother, more predictable workload. Thus the estimates based on aggregation after decomposition are much closer to the actual values.

Figure 10(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+OpenMail, OpenMail+WebSearch, the actual capacities needed are 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 for WebSearch (1538 IOPS) and FinTrans (1500 IOPS); thus the resulting combined workload needs at least 9241 IOPS. The capacity provisioning based on workload decomposition also works well for combining different workloads. In Figure 10(b) and 10(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 10(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 of 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 good estimate for the combined workload.

4.6 Linux Prototype Evaluation

We implemented our workload shaping framework Miser as a loadable block device driver module for the Linux 2.6.28 kernel. The device driver module creates a virtual device backed up by a physical hard drive. The module provides system call ioctl for user-level programs to assign scheduling parameters: capacity (IOPS) and response time limit for each virtual device. For Miser, it uses these two input parameters to calculate the queue length threshold. After this initialization, the virtual device can receive IO requests, schedule them using its own algorithm and send them to the backing physical device. We implemented two scheduling algorithms: Miser and FCFS for performance comparison. In order to replay the traces as mentioned above, we implemented a trace processing module in FIO tool [26] for replaying the ASCII format traces. This enhanced tool can honor the request arrival time when sending requests to the hard drive, which maintains the burstiness of the original workload. We evaluate our prototype on a Dell desktop machine running Linux 2.6.28 kernel, with 2.80GHz Intel Pentium(R) D CPU, 3 GB memory and a 1TB Seagate SATA test hard drive. The test drive has about 130 IOPS for 4K random requests and 100 MB/s of bandwidth for 256K sequential requests.

We tested a WebSearch workload from the original set of traces used in the simulation experiments, as well as two new MSRC storage workload traces, and compared the response time distributions of Miser and FCFS.

WebSearch Workload: The overall request rate of the WebSearch workload is shown in Figure 11(a). We see that this workload is very bursty across the entire time range. We chose the first 500 seconds of WebSearch for our test; its request rate is shown in Figure 11(b). Since our test hard drive only has a capacity of about 130 IOPS for random requests, we slow down the trace by 2x to match the performance of the drive. First, we compare the response time distribution of Miser and FCFS with response time preference of 50ms. As shown in Figure 12(a), about 55% of the requests get response times smaller than 50ms with Miser scheduling, while FCFS only finishes about 0.05% of the requests within the 50ms deadline. About 30% of requests finish between 50ms and 100ms under Miser, while FCFS only gets 12% in this range. Then, we increase the response time target to 100ms for further validation using the same workload; the results are shown in Figure 12(b). We can see that about 70% of requests have response times smaller than 100ms using Miser scheduling, compared with only 12% of requests using FCFS.

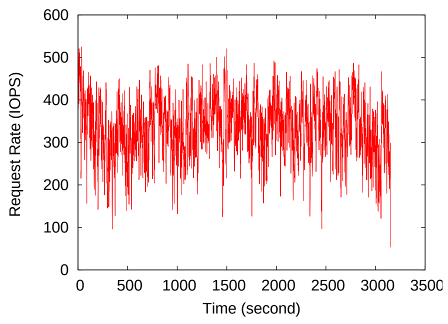
MSRC Workloads: We chose two representative MSRC workloads for performance evaluation. Each workload is a one week long trace. *Src* workload is from a source control server. *Usr* represents user home directories workload.

Src results: The overall request rate of *Src* is shown in Figure 13(a) and the first day request rate is shown in Figure 13(b). We can see that *Src* workload has a periodic pattern over the whole week, and high burstiness in the request rate each day. We slow down the first day's workload by 4x for the test. In Figure 14(a), we set Miser's response time target to 20ms. We find that about 52% of the response times meet the target while only about 1% of requests do so using FCFS. 20% of requests finish in between 20ms and 50ms using Miser while only 7.9% such requests do so with FCFS. In Figure 14(b), we increase the response time target to 50ms. Miser still outperforms FCFS significantly in terms of percentage of requests with small response time. Miser can get about 63% of requests with response time smaller than 50ms compared with 8.8% for FCFS.

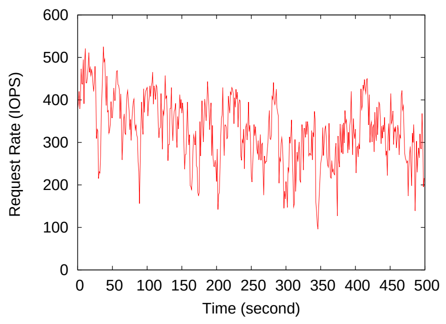
Usr results: As the *Usr* workload also has periodic weekly pattern, we only show the first day's workload in Figure 15(a). There are two busy periods in this workload; we are more interested in the second one shown in Figure 15(b) since it represents a typical ON/OFF bursty pattern. Although the request rate of *Usr* is very high, it is highly sequential in terms of its block address access pattern. Thus we do not need to scale the workload in this test. As Figure 15(c) shows, for this sequential workload, FCFS can do good job of making about 80% of requests' response time smaller than 50ms. However, Miser can still improve the response time distribution over FCFS for smaller response times. As the figure shows, Miser can finish 43% of requests within 10ms while FCFS only completes 3.6% in this range.

5 Related Work

Recently proposed QoS schedulers for storage servers [1; 2; 3; 4; 27] are generally based on Fair Queueing [22; 28; 29] principles, combined with throughput enhancing mechanisms to exploit locality and concurrency in the storage arrays. These works do not explicitly address the issue of reducing resource requirements, simply assuming sufficient capacity or

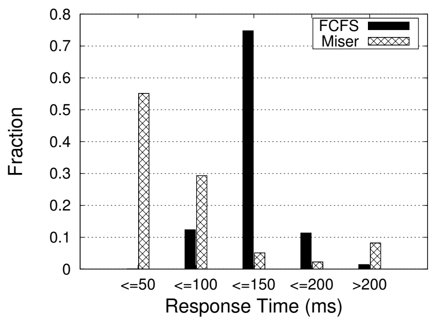


(a) WebSearch request rate

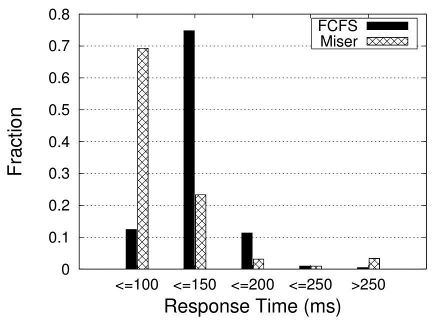


(b) WebSearch request rate: first 500s

Fig. 11. Request rate of WebSearch workload

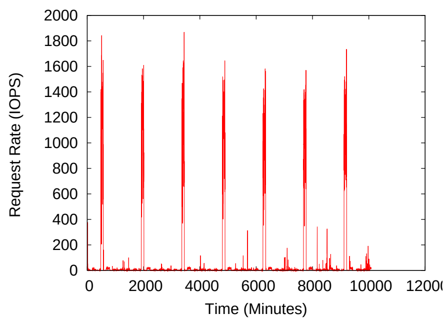


(a) 50ms target

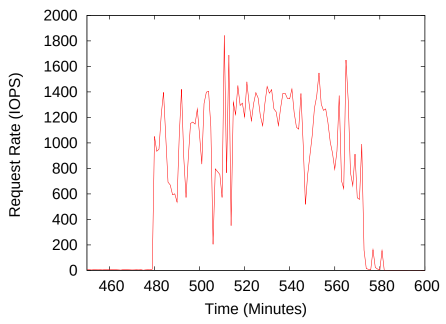


(b) 100ms target

Fig. 12. Response time distribution of FCFS and Miser on WebSearch workload

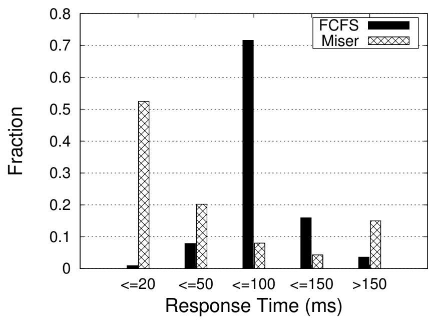


(a) MSRC src request rate

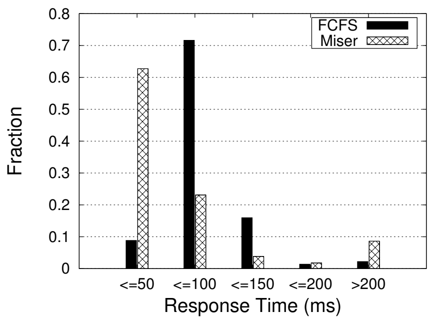


(b) MSRC src request rate: 450M to 600M

Fig. 13. Request rate of MSRC src workload

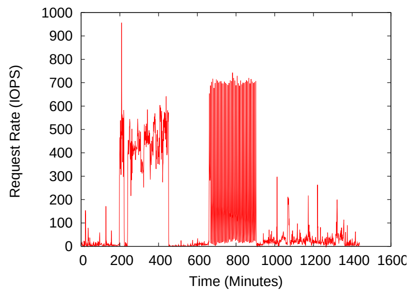


(a) 20ms target

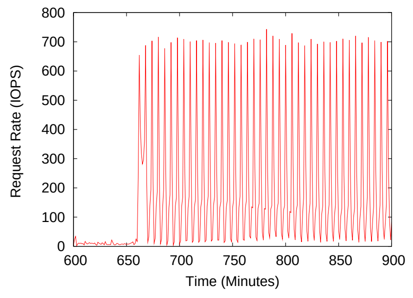


(b) 50ms target

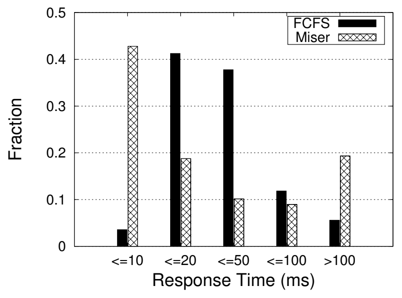
Fig. 14. Response time distribution of FCFS and Miser on MSRC src workload



(a) MSRC usr request rate



(b) MSRC usr request rate: 600M to 900M



(c) 10ms target

Fig. 15. Response time distribution of FCFS and Miser on MSRC usr workload

proportionately allocating the available capacity by weight. As shown in the results this requires significant resource over-provisioning to handle the unpredictable burst patterns and meet response time goals. Our scheduling framework differs from the above works by provisioning based on the overwhelmingly well-behaved portion of the workload rather than the 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 [30; 31; 32] 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 [30] 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 [32] 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 [31] also leverages the size-autocorrelation property of the jobs to simulate the Short Job First scheduler 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 (e.g. 32KB). We differ from the above works by considering the correlation in arrival times (leading to bursts), and proposes decomposing the workload to different classes dynamically based on the overall shape of the traffic rather than on individual request characteristics.

A third body of related work can be found in the considerable literature on network QoS [33] 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 [34]. In storage systems, 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 [10] 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 utilization. In general, the traffic decomposition methods in network literature focus on shaping the aggregated traffic from several clients by throttling or dropping based on their traffic classification, to match transmission bandwidth. We are interested in shaping one specific client traffic to provide good response time distribution without throttling or dropping.

6 Conclusions and Future Work

We addressed the problem of response time degradation in storage servers caused by 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 using both simulation and Linux implementation. 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.

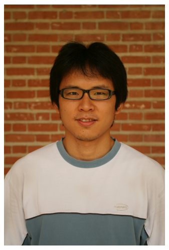
In future work, we are focusing on how the workload shaping affects the disk throughput and how to best apply workload shaping in a multi-client environment. Disk throughput highly depends on the spatial locality in the workload. Thus, during decomposition, we not only need to schedule the requests across 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

The research was partially supported by NSF grants CNS0615376, CNS054136, CNS0917157. We thank the reviewers for their constructive suggestions and Tate Hornbeck for help in the implementation.

References

[1] W. Jin, J. S. Chase, and J. Kaur, “Interposed proportional sharing for a storage service utility,” in *Proc. of SIGMETRICS*, 2004.
[2] J. Zhang, A. Sivasubramaniam, Q. Wang, A. Riska, and E. Riedel, “Storage performance virtualization via throughput and latency control.” in *Proc. 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 *Proc. of SIGMETRICS*, 2007.
[4] C. Lumb, A. Merchant, and G. Alvarez, “Façade: Virtual storage devices with performance guarantees,” *Proc. of FAST*, 2003.
[5] M. Aron, P. Druschel, and W. Zwaenepoel, “Cluster reserves: a mechanism for resource management in cluster-based network servers,” in *Proc. 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, S. T. M, W. Willinger, and D. V. Wilson, “On the self-similar nature of ethernet traffic,” *IEEE/ACM Trans. Networking*, vol. 2, no. 1, 1994.
[8] A. Riska and E. Riedel, “Long-range dependence at the disk drive level,” in *Proc. of QEST*, 2006.
[9] N. Mi, Q. Zhang, A. Riska, E. Smirni, and E. Riedel, “Performance impacts of autocorrelated flows in multi-tiered systems,” in *Performance Evaluation*, vol. 64, no. 9-12, 2007.
[10] E. W. Knightly and N. B. Shroff, “Admission control for statistical qos: theory and practice,” in *IEEE Network*, vol. 13, Mar. 1999.
[11] L. Lu, P. J. Varman, and K. Doshi, “Graduated qos by decomposing bursts: Don’t let the tail wag your server,” in *Proc. of ICDCS*, 2009.
[12] “Public software (storage systems department at hp labs),” June 2007, http://tesla.hpl.hp.com/publicsoftware/.
[13] D. Narayanan, A. Donnelly, E. Thereska, S. Elnikety, and A. Rowstron, “Everest: Scaling down peak loads through i/o off-loading,” in *Proc. of OSDI*, 2008.
[14] “Storage performance council (umass trace repository),” June 2007, http://traces.cs.umass.edu/index.php/Storage.
[15] J. S. Chase, D. C. Anderson, P. N. Thakar, A. M. Vahdat, and R. P. Doyle, “Managing Energy and Server Resources in Hosting Centers,” in *Proc. of SOSP*, 2001.
[16] G. Chen, W. He, J. Liu, S. Nath, L. Rigas, L. Xiao, and F. Zhao, “Energy-Aware Server Provisioning and Load Dispatching for Connection-Intensive Internet Services,” in *Proc. of NSDI*, 2008.
[17] Q. Zhu, Z. Chen, L. Tan, Y. Zhou, K. Keeton, and J. Wikes, “Hibernator: Helping Disk Arrays Sleep Through The Winter,” in *Proc. of SOSP*, 2005.
[18] D. Meisner, B. T. Gold, and T. F. Wenisch, “PowerNap: Eliminating Server Idle Power,” in *Proc. of ASPLOS*, 2009.
[19] “HDFS: Architecture and Design,” http://hadoop.apache.org/, Aug. 2008.
[20] S. Ghemawat, H. Gobioff, and S.-T. Leung, “The Google File System,” in *Proc. of SOSP*, 2003.
[21] J. C. R. Bennett and H. Zhang, “WF^2Q: Worst-Case Fair Weighted Fair Queueing,” in *Proc. of INFOCOM*, 1996.
[22] P. Goyal, H. M. Vin, and H. Cheng, “Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks,” in *IEEE/ACM Trans. Networking*, vol. 5, no. 5, 1997.
[23] A. Gulati and P. Varman, “RFQ: Redemptive Fair Queuing,” in *Proc. of ESA*, 2008.
[24] Http://www.pdl.cmu.edu/DiskSim/.
[25] D. Narayanan, A. Donnelly, and A. Rowstron, “Write off-loading: Practical power management for enterprise storage,” in *Proc. of FAST*, 2008.
[26] “Flexible IO Tester,” http://freshmeat.net/projects/fio/.
[27] P. J. Shenoy and H. M. Vin, “Cello: a disk scheduling framework for next generation operating systems,” in *Proc. of SIGMETRICS*, 1998.
[28] J. C. R. Bennett and H. Zhang, “WF^2Q: Worst-case fair weighted fair queueing,” in *Proc. of INFOCOM*, 1996.
[29] A. Demers, S. Keshav, and S. Shenker, “Analysis and simulation of a fair queuing algorithm,” *Journal of Internetworking Research and Experience*, pp. 3 – 26, Sep. 1990.
[30] M. Harchol-Balter, B. Schroeder, N. Bansal, and M. Agrawal, “Size-based scheduling to improve web performance,” in *ACM Trans. Comput. Syst.*, vol. 21, no. 2, 2003.
[31] N. Mi, G. Casale, and E. Smirni, “Scheduling for performance and availability in systems with temporal dependent workloads,” in *Proc. of DSN*, 2008.
[32] Q. Zhang, N. Mi, A. Riska, and E. Smirni, “Load unbalancing to improve performance under autocorrelated traffic,” in *Proc. of ICDCS*, 2006.
[33] J. W. Evans and C. Filsfils, “Deploying ip and mpls qos for multiservice networks,” in *Morgan Kauffman*, 2007.
[34] S. Floyd and V. Jacobson, “Random early detection gateways for congestion avoidance,” in *IEEE/ACM Transactions on Networking*, vol. 1, no. 4, 1993.



Lanyue Lu received his B.E. from University of Science and Technology of China in 2006, and M.S. from Rice University in 2009. His research interests are in storage and file systems.

Peter J. Varman received his B. Tech from the Indian Institute of Technology, Kanpur and M.S. and Ph. D. degrees from the University of Texas at Austin. Since then he has been on the faculty of Rice University where he is currently a Full Professor. He had held visiting positions at IBM T. J. Watson and Almaden research centers, NTU Singapore, and Duke University. From 2002 2005 he served as Program Director for Computer Systems Architecture at the National Science Foundation. His research interests are in computer architecture, storage systems, resource scheduling and power management. He has served on several international science and engineering review and advisory panels. He is a past editor of the IEEE Transactions on Computers and serves on the editorial board of the Journal of Combinatorial Optimization.

Kshitij Doshi received his B. Tech from the Indian Institute of Technology, Bombay and M.S. and Ph. D. degrees from Rice University. He has since then worked with AT&T, Novell, and Intel corporations in areas of hardware and software architecture and performance analysis, system virtualization, and storage hierarchies. His current focus also includes database systems architecture and performance and computational energy optimization approaches.