RePAIR: Reservation-Based Proportionate Allocation For IO Resources

Ajay Gulati
VMware Inc
agulati@vmware.com

Arif Merchant
HP Labs
arif@hpl.hp.com

Peter Varman
Rice University
pjv@rice.edu

Abstract

Guaranteed IO resource allocation is a challenging problem in storage servers due to the variability and unpredictability in their dynamic capacity. Existing scheduling mechanisms for CPU and memory allocation are insufficient in providing the guarantees that are needed by applications in today's shared system infrastructures.

In this paper, we examine the problem of providing proportional bandwidth allocation, with a minimum reservation per application for shared IO access. We look at two practical use cases: (1) I/O resource management in virtualized servers, and (2) Distributed storage infrastructure. We present a scheduling algorithm, Reservation Based Fair Queuing *RBFQ*,that provides proportionate allocation while meeting per-application reservation constraints. *RBFQ* uses two novel ideas *(i) real-time dual tagging* and *(ii) prioritized scheduling* to accomplish its goals. We also propose *idle credits* as a mechanism to handle bursts in IO workloads and improve IO efficiency. Experiments with *RBFQ*, using both a prototype implementation in VMware's ESX Server and a distributed storage environment, show that it provides good isolation, and meets reservations in the presence of fluctuating capacity, dynamic workloads, and hot-spotting by various clients.

1 Introduction

The importance of resource management strategies continues to grow with the increased deployment of shared infrastructure for computation, storage, and networking. A variety of new system architectures ranging from hosted virtual machines with SAN-attached storage to cluster-based distributed storage systems, are giving rise to new IO-sharing requirements. These systems present a new set of challenges for flexibly allocating resources that go beyond those arising in CPU, memory, and network bandwidth management. A basic set of requirements that a scheduler should provide are: (i) *reserved allocation*: guarantee a client a minimum amount of service, (ii) *proportional allocation*: provide clients service based on their weights reflecting their importance or priority, and (iii) *maximum allocation*: cap the maximum service a client may obtain. The problem is complicated in the context of allocating IO resources since the capacity of the storage server accessed by a host (or a set of workloads) can change significantly and unpredictably over small time intervals. This may occur due to workload characteristics like the degree of sequentiality in the workload, unpredictability in the load on a shared storage array due to bursts by other servers, or access hot-spots in a distributed storage system. We describe two use cases to illustrate these scenarios in Section 2. Most existing resource schedulers are based on the implicit assumption of either a fixed capacity server or predictable capacity variations at relatively large time scales.

Currrent fair-queuing schedulers do a good job of providing clients (workloads) with *proportional shares* of resources such as CPU [29, 26, 10, 8, 14], memory [30, 13], network bandwidth [24, 15, 9, 12] and storage [27, 19, 18, 7, 23], by dividing up the server capacity among the active clients in proportion to their weights. If the available capacity changes, the amount of service received by a client changes in concert, so as to maintain the same ratio in the allocations to the active clients. Unfortunately if the capacity decreases significantly, a client with a small weight can end up receiving an unacceptably small amount of service. A more useful scheduler in this situation will guarantee a minimum floor on the service a client will receive, in preference to insisting on proportionate allocation. When the system capacity increases, the scheduler will again try to reestablish proportionate allocation. For resources like CPU, memory and network bandwidth, the capacity of the resource is known or can be accurately estimated a priori, and this can be effectively exploited to obtain minimum, maximum and weighted allocation [30]. However, those mechanisms are insufficient for storage IO due to its unpredictable capacity. We illustrate the issues in the next section using examples.

In this paper, we examine the problem of providing proportional IO resource allocation *subject to minimum requirements for all applications*. Towards this end we present **RBFQ**, a reservation-based fair queuing algorithm. RBFQ uses two techniques: *real-time dual tagging* and *prioritized scheduling*. RBFQ entails two major points of difference from previous scheduling algorithms [30,26,19,9,12]. First all our tags are based on real time rather than virtual time used in the classic Weighted Fair Queuing algorithm [9] and its successors. This is highly advantageous in this situation since it is much easier to track capacity changes simply by observing the finish times of requests relative to their real-time tags. It is difficult to see how to do this naturally using virtual time because it only tracks relative service allocations. Fair scheduling solutions based on real-time tags have been proposed in [35,22,16], but they do not consider the problem of providing reservations. Second, we use two sets of tags (local tags and global tags) to control the minimum and proportional-share based allocations simultaneously, and the algorithm seamlessly chooses the correct tag to use based on the current system capacity. We also introduce *idle credits*, a mechanism to handle bursts in IO workloads and to improve overall IO efficiency by encouraging batch scheduling. Finally, we present an adaptation of RBFQ to distributed storage architectures where storage services are provided by a cluster of hosts or arrays. Our extensive evaluation both as a prototype implementation in VMware ESX Server 3.5 as well as a on simulated distributed storage systems, shows that RBFQ meets its scheduling goals. We also show that our *idle credit* mechanism can handle bursts to provide smaller latency and improve overall IO throughput by clustering IOs.

In Section 2, two use cases illustrate the problem and how our algorithm allocates the available capacity between clients. Section 3 presents our system model and algorithm *RBFQ* in detail. Performance evaluation and a discussion of relevant issues is presented in Section 4. Related work is presented in Section 5 followed by conclusions in Section 6.

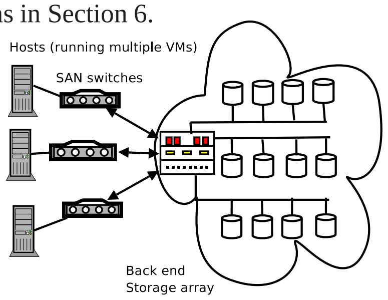


Figure 1: VM System Architecture

2 Use cases

We describe two use cases to motivate further the need for a proportional share scheduler with minimum reservations, and to illustrate how our scheduler would divide the capacity in each case.

Case 1: Virtual Machine Hosting - In most virtualized data centers, virtual machines (VMs) share a host's resources and a common storage system accessed through a SAN (Figure 1). Each VM requires some minimum resource allocation in order to run stably — e.g., 256 MIPS of CPU, 512MB of memory, and 1 MB/s of storage bandwidth. Unlike CPU and memory, storage bandwidth available to a host is unpredictable, because it depends on the access pattern (sequential I/Os can be served more efficiently than random ones) and because the storage array is shared with other hosts. As such, existing proportional share schedulers cannot guarantee a minimum storage bandwidth to the VMs, even if the aggregate bandwidth available is adequate to meet the minimum requirements of all the VMs. For example, consider a case with two VMs, VM1 and VM2, with I/O share allocations in the ratio 1:5 and a minimum I/O requirement of 1MB/s. If the aggregate storage bandwidth available to the host is 3MB/s, VM1 will receive 0.5 MB/s, which is below its required minimum, even though there is sufficient capacity for both VMs to receive 1MB/s.

Aggregate Bandwidth(B)	VM1	VM2
Weight	1	5
Minimum requirement	1 MB/s	1 MB/s
B $\leq$ 2 MB/s	B/2	B/2
2 MB/s $\leq$ B $\leq$ 6 MB/s	1	B-1
B $\geq$ 6 MB/s	B/6	5B/6

Table 1: Allocation based on current throughput

We illustrate how our scheduler will allocate capacity in this scenario. If the bandwidth, $B \geq 6$ MB/s, then the scheduler should allocate $B/6$ and $5B/6$ respectively to the VMs; this guarantees their minimum bandwidth of 1 MB/s and provides the desired $1:5$ ratios as well. If $B < 2$ MB/s, then it is not possible to provide the minimum to both VMs, and the scheduler will divide the available bandwidth in the ratio of their minimum requirements ($1:1$ in this case). For the interesting intermediate case (when $2 \leq B \leq 6$), the bandwidth is allocated as close as possible to the $1:5$ ratio, *subject to the constraint* that each VM gets its minimum requirement; in this example the VMs will receive 1 MB/s and $(B-1)$ respectively. Note that allocating any additional bandwidth to $VM1$ at the expense of $VM2$ will make the ratio of allocation further away from the desired $1:5$ ratio. The desired allocation is summarized in Table 1.

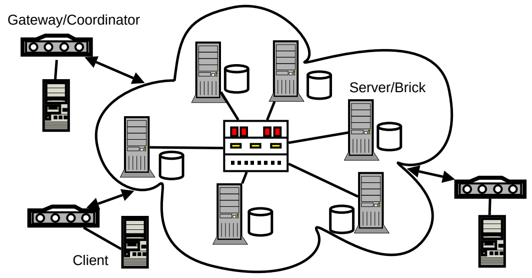


Figure 2: Distributed Storage System

Case 2: Distributed Storage Infrastructure - Distributed storage systems (see Figure 2) built from commodity disks and software-controlled allocation and routing have been proposed as a cost-effective, scalable alternative to expensive centralized storage arrays. Several research prototypes (e.g., CMU's Ursa Minor [11], HP Labs' FAB [21], IBM's Intelligent Bricks [33]) have been built, and nascent commercial offerings are becoming available. Client data reside in logical volumes spread over multiple servers (bricks). Each client accesses its data through a specific gateway (coordinator).

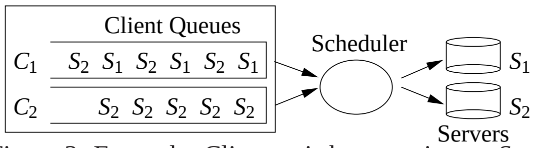

Client Queues
C_1
S_2 S_1 S_2 S_1 S_2 S_1
C_2
S_2 S_2 S_2 S_2 S_2
Scheduler
S_1
S_2
Servers

Figure 3: Example: Client c_2 is hot-spotting on S_2

The aggregate service received by a client is made up of service at individual servers in its volume. A danger of proportionate aggregate service allocation is that excess service received by a client on one server may be compensated by starving the client on another server [32]. We call this *resource-specific starvation*. For example Figure 3 shows two storage bricks S_1 and S_2 of capacity B IOPS each accessed by two clients c_1 and c_2 with equal weights. Suppose that c_1 is uniformly accessing both storage nodes, while c_2 is directing all its requests only to node S_2. A proportional-scheduler (e.g. [17,19,15,9,12]) will provide both clients with B IOPS; however, c_1 will be served entirely from S_1 and will receive no service at S_2.

Resource-specific starvation can be avoided by mandating a minimum amount of service for a client at each server, say 20 IOPS in the above example. If the server capacities are both 100 IOPS, the scheduler would provide 120 IOPS to c_1 (100 IOPS at S_1 and 20 IOPS at S_2) and 80 IOPS to c_2 (all at S_2). If S_2's capacity is increased to 140 IOPS, each client would receive 120 IOPS, thereby meeting minimum requirements as well as aggregate (global) allocation in the ratio $1:1$.

3 RBFQ Scheduling

In this section we describe the model of the system and precisely specify the objectives of the scheduling policy. In Section 3.1 we describe the scheduling algorithm for the case of a single centralized resource scheduler. In Section 3.2 we describe a distributed implementation of the algorithm for the case of multiple servers servicing IOs in a distributed manner.

The system consists of clients $c_i, \; i = 1, 2, \cdots n$, that request service from a storage array. Each client c_i has two parameters that determine its service allocation: a global *weight* w_i and a *minimum service demand* M_i. The former represents the proportionate share of the capacity that the client should ideally receive. The service received by c_i is denoted by S_i. The capacity of the server at any time is denoted by C. If the capacity is insufficient to provide all clients with their minimum service demands (the rare case), then the available capacity is allocated to the clients in proportion to their minimum demands. Otherwise, the scheduler will try to allocate service in proportion to the global weights of clients, while ensuring that each client receives its minimum demand.

Let M_i/w_i and S_i/w_i denote the *normalized minimum service* and *normalized received service* for client c_i. Note that providing service to a set of clients in proportion to the global weights is equivalent to providing them with an equal amount of normalized received service; that is, $M_i/M_j = w_i/w_j$ exactly when $M_i/w_i = M_j/w_j$.

Without loss of generality let us renumber the clients to be in non-decreasing order of normalized minimum service: that is, $M_1/w_1 \leq M_2/w_2 \leq M_3/w_3 \cdots \leq M_n/w_n$. Depending on the capacity, some set of the clients will receive exactly their minimum service demand M_i, while the remaining will receive service in proportion to their weights, an amount greater than their minimum demands. We define below a series of *critical capacities* $C_k, k = 1, \cdots n$ at which there is a change in the set of clients receiving their minimum service versus those receiving additional service. Define $C_k = (M_k/w_k)\sum_{i=1}^{k} w_i + \sum_{i=k+1}^{n} M_i$. For notational convenience define quantities $C_0 = 0$ and $C_{n+1} = \infty$. As may be observed (see Appendix B for formal proofs), the C_k form a nondecreasing sequence. Hence for a given system capacity C, there exists a unique index k, $0 \leq k \leq n$, such that $C_k \leq C < C_{k+1}$. When the capacity $C \geq C_n = (M_n/w_n)\sum_{i=1}^{n} w_i$, then all clients receive service in proportion to their w_i while also satisfying their minimum requirement M_i. When the capacity C lies between C_k and C_{k+1}, then clients c_1 through c_k will get service in proportion to w_i while meeting their minimum requirement, while clients c_{k+1} through c_n will receive exactly their minimum requirement.

Example: Consider four clients with minimum service requirements $120, 75, 50, 25$ and corresponding weights $20, 5, 10, 1$ respectively. The normalized minimum service numbered in increasing order are $M_1/w_1 = 50/10 = 5, M_2/w_2 = 120/20 = 6, M_3/w_3 = 75/5 = 15, M_4/w_4 = 25/1 = 25$. The critical capacities are: $C_1 = 5 \times 10 + (120 + 75 + 25) = 270$, $C_2 = 6 \times (10 + 20) + (75 + 25) = 280$, $C_3 = 15 \times (10 + 20 + 5) + (25) = 550$ and $C_4 = 25 \times (10 + 20 + 5 + 1) = 900$. Figure 4 shows the service received by the flows for different amounts of system capacity.

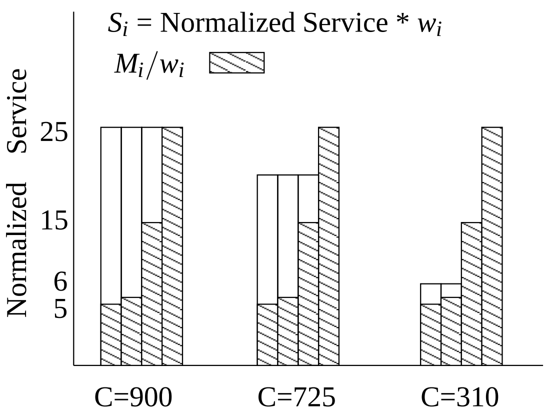


Figure 4: Allocation for different capacity values

When the capacity $C \geq 900$ then c_1, c_2, c_3 and c_4 receive service in the ratios of w_i, *i.e.* $10 : 20 : 5 : 1$. As shown by leftmost set of bars in Figure 4 the normalized received service for each client in this case is equal. For example, if $C = 900$ then the normalized received service for each c_i is $M_4/w_4 = 25$. Hence, the actual amount of service obtained by c_i will be $25 \times w_i$: $S_1 = 250, S_2 = 500, S_3 = 125$, and $S_4 = 25$. Note that these all meet the minimum requirements $M_1 = 50, M_2 = 120, M_3 = 75$, and $M_4 = 25$.

When the capacity C falls below 900 but is at least 550, then c_4 will receive exactly its minimum amount $M_4 = 25$, while c_1, c_2 and c_3 will divide the remaining capacity in the ratios of w_i. For example, if $C = 725$ then (middle set of bars in Figure 4), the normalized received service for c_1, c_2 and c_3 is $(C - M_4)/(w_1 + w_2 + w_3) = 20$. The actual amount of service obtained by these three clients will be $20 \times w_i$, which is $S_1 = 200$, $S_2 = 400$, and $S_3 = 100$. $S_4 = M_4 = 25$. Note again that these all meet the minimum requirements $M_1 = 50, M_2 = 120, M_3 = 75$, and $M_4 = 25$.

When the capacity C falls below 550 but is at least 280, then c_3 and c_4 will receive only their minimum amounts $M_3 = 75$ and $M_4 = 25$. c_1 and c_2 will divide the remaining capacity in the ratio $10 : 20$. For example, if $C = 310$

Symbol	Meaning
g_i^r	Global tag of request r of client c_i
l_i^r	Local tag of request r of client c_i
M_i	Minimum requirement of client c_i
w_i	Global share (weight) of client c_i

Table 2: Symbols used and their descriptions

(rightmost bars in Figure 4) then the normalized receive service for c_1 and c_2 is $(C - M_3 - M_4)/(w_1 + w_2) = 7$, so $S_1 = 70$ and $S_2 = 140$. $S_3 = M_3 = 75$, and $S_4 = M_4 = 25$. Note as usual that these at least the minimum requirements $M_1 = 50, M_2 = 120, M_3 = 75$, and $M_4 = 25$.

In the next section we provide the algorithm for RBFQ scheduling. In Appendix B we provide proofs of the properties stated below of RBFQ scheduling in the fluid-flow model of service allocation, where service is provided in continuous increments. Simulation results of a study of the micro-level behavior of the algorithm are provided in Appendix A.

RBFQ Scheduling Property: Let $c_i, i = 1, \cdots n$ be a set of continuously backlogged clients in some interval $[0, T]$. Let C be the server capacity in this interval, and suppose that $C_k \leq C < C_{k+1}$. Then the service S_i received by c_i in the interval satisfies: (i) $\forall i = 1, \cdots n$, $S_i \geq M_i$ (ii) $\forall i, k < i \leq n$, $S_i = T \times M_i$ (iii) $\forall i, 1 \leq i \leq k$, $S_i = T \times w_i \times (C - \sum_{j=k+1}^{n} M_j)/\sum_{j=1}^{k} w_j$.

3.1 RBFQ Algorithm

We now present our reservation-based fair queuing scheduling algorithm RBFQ for a centralized single-server system.

The algorithm uses two main ideas: *real-time dual tagging* and *prioritized scheduling* to achieves its goals. Each request arriving at the server is assigned two tags: a *local* and a *global* tag. Local tags are based on the minimum service requirements M_i, while global tags are based on the weights w_i.

The intuitive idea behind the algorithm is to logically interleave a local scheduler and a global scheduler in a fine-grained manner. The scheduler alternates between a local phase (using local tags) and a global phase, (using global tags). In a local phase the scheduler provides each client with its minimum requirement for a small future window of time. At the appropriate time the scheduler switches to the global phase and begins allocating service in the ratio desired to meet the global weights for the rest of the window. Note the algorithm does not choose or calculate the size of the time window. Any such choice of window size or switching threshold requires assumptions about the capacity which is inherently not predictable in our model. Our algorithm switches between the phases automatically based on *real-time tag* values it assigns requests, and the times at which they complete (that depends implicitly on the current capacity). The scheduler is quite simple and has three main components: (1) Tag Assignment (2) Tag Adjustment and (3) Request Scheduling. We will explain each of these routines in more detail below.

```
Request Arrival:
(1) Let $t$ be arrival time, for request $r$ from $c_i$
(2) Tag-Adjustment( )
(3) Tag-Assignment( )

Tag-Assignment( ):
(1) $l_i^r = \max\{l_i^{r-1} + 1/M_i,\ t\}$   /* Local tag */
(2) $g_i^r = \max\{g_i^{r-1} + 1/w_i,\ t\}$   /* Global tag */

Tag-Adjustment():
(1) Adjust Global Tags: Add or subtract a constant amount
    from each global tag so that the minimum global start tag
    equals current time $t$

Request Scheduling (when server is free):
(1) Let $D$ be the set of requests with local tag $\leq$ t
(2) if ($D$ not empty)
(3)     Dispatch the request $r \in D$ with minimum local tag
(4) else
(5)     Dispatch the request $r$ with minimum global tag
(6)     Subtract $1/M_k$ from local tags of selected client $C_k$
```

Figure 5: Components of RBFQ algorithm

Tag Assignment: This routine assigns *local* and *global* tags to a request r from client c_i arriving at time t. The value assigned to the *local tag* of request r of c_i, denoted by l_i^r, is the larger of the current time (t) and $1/M_i + l_i^{r-1}$. When a client c_i becomes active after a period of inactivity, its first request is assigned a local tag equal to the current time; local tags of subsequent requests are spaced $1/M_i$ apart, as long as the client remains backlogged. This ensures that in an interval of T seconds, a client will have roughly TM_i requests with local tags in that interval. Hence, if the scheduler can ensure that a request begins service by the time indicated in its local tag, then the client will be receiving service at a rate of at least M_i as desired.

The *global tag* assigned to a request depends on the total number of requests of that client that have completed service by the time it is dispatched. The global tag g_i^r is the larger of the arrival time of the request or $1/w_i + g_i^{r-1}$. Like the local tags, the first request from an activated client is tagged with the current time; subsequent backlogged requests are spaced by $1/w_i$.

Tag Adjustment: Tag adjustment is used to calibrate the global tags dynamically against real time. This is similar to the idea of synchronizing clients with current virtual time, in virtual time based techniques for fairness [6, 9]. Since the spacing of global tags is based on relative weights while the initial tag value is determined by the actual arrival time, there needs to be some mechanism to synchronize the tag values with real time. In the absence of such a mechanism starvation may occur. Two possible situations can arise depending on the weight of the client relative to the capacity available after providing the reserved service: global tags may get ahead of the real time or may lag real time. In the first case, a client that begins sending requests after a temporary idle period, will assign tags to its requests that will be less than the tags of the backlogged clients whose tags have gone ahead of real time. Those clients will be starved of service until the tags of the new client catch up. In the other case, the newly active client will have its tags greater than the tags of the backlogged clients that are lagging real time. Hence this client will be starved of service until the tags of the other clients catch up to the present time.

The routine adjusts all the global tags by adding (or subtracting) a fixed amount from all of them. so that the smallest global tag equals the current time. The relative ordering of existing tags is not altered by this transformation; however, newly activated clients start with a tag value equal to the currently smallest tag, and consequently compete fairly with existing clients.

Request Scheduling: The scheduler alternates between local and global phases. First, the scheduler checks if there is any request with a local tag less than or equal to the current time. If such requests exist, then the request with smallest local tag is dispatched for service. This is defined as a local phase. The local phase ends (and global phase starts) at a scheduling instant when all the local tags exceed the current time. The interval up to the next smallest local start tag constitutes a global phase. At any time within a global phase, all clients have received their minimum requirements up to this time. During this interval the scheduler therefore allocates service to achieve global proportionality. Also whenever a request from client c_i is scheduled in a global phase, the local tags of the outstanding requests of c_i are decreased by $1/M_i$. This maintains the condition that local tags (i.e. reserved service) are not affected by the scheduling in a global phase, and are always spaced apart by $1/M_i$.

Figure 5 provides a high-level description of the algorithm. The straightforward implementation has a complexity of $O(n)$, for n active clients (VMs in our case); this can be improved to $O(\log n)$ using linked data structures. We discuss below an example to show the working of the algorithm.

Example: There are two clients c_1 and c_2 with $M_1 = 20, M_2 = 40$ IOPS, and $w_1 = 3, w_2 = 1$ that are continuously backlogged starting from time $t = 0$. Assume that $C = 100$ IOPS, though the algorithm does not need to know this value, and it may vary with time. The schedule should allocate 40 IOPS (its minimum requirement) to c_2 and the remainder 60 IOPS to c_1,

The local tags of c_1's requests will be spaced $1/M_1 = 50$ ms apart, with values, 0, 50, 100, 150, etc. as shown in Table 3. Similarly, the local tags of c_2 are spaced apart by 25 ms, with values 0, 25, 50, 75, etc. The spacing of the global tags should be in the inverse ratio of the w_i: hence the global tags of c_1 are $1/w_1$ apart, with values $0, 1/3, 2/3, 1, 4/3, 5/3, 2$, etc., while those for c_2 will be $0, 1, 2, 3, 4, 5$, etc.

At $t = 0$ a local phase begins. The first requests of both clients are eligible for scheduling (since their local tags are not greater than the current time). Since both tags have the same value, one of them (say c_1), is chosen to be served. This request will complete at $t = 10$ ms with the assumed server capacity. At that time, the first request of c_2 is still eligible, and will be scheduled completing at $t = 20$ ms. At this time, all local tags are later than the current time (the smallest local tag is 25 ms). Hence the scheduler switches to the global phase, and looks at the global tags of the first unscheduled request of each client (the second request of both). The global tag (1/3) of c_1 is less than

Request Number	1	2	3	4	5
Local tags c_1 (ms)	0	50	100	150	200
Local tags c_2 (ms)	0	25	50	75	100
Global Tags c_1	0	1/3	2/3	1	4/3
Global Tags c_2	0	1	2	3	4

Table 3: Tag Example: $M_1 = 20$, $M_2 = 40$, $w_1 : w_2 = 3 : 1$

the global tag (1) of c_2; hence request 2 of c_1 is serviced, finishing at $t = 30$ ms. *Also the local tags of requests from c_1 are all reduced by* 50 *ms, so that requests* 3, 4, 5, *etc have tags* 50, 100, 150 *etc.* This mechanism keeps the local allocation unaffected by the global scheduler. At this time, the scheduler switches to the local phase, since c_2 has a local tag with value 25 ms. This request completes at time $t = 40$ ms, and the scheduler switches once again to the global phase. The third request of c_1 has a smaller tag $(2/3)$ than the next request of C_2 (tag value 2), and so will be scheduled, completing service a time 50. The local tags of c_1 are again adjusted to be 50, 100 etc. At this time, 3 requests of c_1 and 2 requests of c_2 have been served. The conditions are now exactly as at $t = 0$, and in the next 50ms, once again 3 requests from c_1 and 2 from c_2 will be serviced, and so on. As can be seen, c_1 is getting 3 IOs every 50 ms or 60 IOPS and c_2 is getting 2 requests every 50 ms or 40 IOPS.

3.2 Distributed Implementation

In the distributed model there are several independent servers, and the requests from a client c_i are routed to the appropriate server from a gateway (or coordinator) node. All requests from a particular client are routed through its gateway, but different clients may have different gateways. There is no communication between the servers or between the gateway nodes. A gateway forwards a client's request to a server along with some auxiliary information, and receives completion acknowledgments from the server when the request completes at that server.

The scheduling algorithm RBFQ is implemented at each server. There are two modifications to the algorithm to account for the distributed model, both in the Tag-Assignment component. First, the minimum service requirement for a client c_i is now different for for each server; hence at server s_j, the minimum requirement used in computing the local tag should be $M_{i,j}$ (instead of M_i). Secondly, recall that the global tag is used to provide proportionate aggregate service to each client. In this case the aggregate service received by a client is the total of the service it has received from all the servers. This information will be provided implicitly by the coordinator by piggybacking an integer δ_i to each request that it forwards to a server. The coordinator for c_i keeps a running count (δ_i) of the number of requests of c_i that have completed service. The coordinator forwards the current value of δ_i along with c_i's request to a server. The difference between two successive δ_is received by a server (denoted by Δ) indicates the number of requests that c_i has completed on all servers in that time interval. (Note that in the single server case, Δ will always be 1). In routine Tag-Assignment the value of Δ is used to compute the global tag by modifying step 2 as follows: $g_i^r = \max\{g_i^{r-1} + \Delta/w_i,\ t\}$. That is the new request may receive a tag further into the future, to reflect the fact that c_i has received additional service at other servers. The greater the value of Δ, the less priority the request has for service. The information used by a server (Δ) may, in the worst case, be inaccurate by up to 1 request at each of the other servers. The scheme however avoids complex synchronization between servers or between gateways.

3.3 Idle Credits

Storage workloads are known to be bursty in nature and requests from the same application often have high spatial locality. This is especially true for requests originating from a VM since its requests go to a specific virtual disk, that is often laid out in a contiguous manner on the storage volume (unless the disks are thin provisioned). This is done by keeping a bounded lag between the global tag of the workload and current time. In particular, when a idle workload gets active, we compare the previous global tag with current time t. If the difference is higher than a burst parameter $\sigma_i \times 1/w_i$, we make the global tag equal to $t - \sigma_i * (1/w_i)$. This parameter σ_i can be defined per VM and it determines the maximum amount of credits that can be gained by becoming idle. Note that adjusting only the global tag has the nice property that σ_i cannot affect the minimum reservations, but if there is spare capacity in the system, it will be preferentially given to the workload that was idle. Also if the idle workload sends a burst of IO requests, they will all get scheduled with higher priority in a batch, providing low latency to the workload. This is quite desirable for certain workloads such as those arising from database applications and decision support systems. This will also increase the overall IO efficiency as requests from same application tend to have high locality.

3.4 Upper Bound on Allocation

In many cases, storage administrators want to limit the amount of bandwidth a workload can obtain. This can help throttle some operations such as backups, virus scanners etc. from adversely affecting others and increasing the overall latency seen by all workloads. In our framework, this can be easily implemented by keeping a separate tag u_i to keep track of overall allocation. For a given upper bound B_i MB/s, the tag u_i will get incremented by $iosize/(B_i * 1024)$ for each IO with size *iosize* KB. During request scheduling, we can stop issuing IOs from a VM if $(u_i > current\ time)$. This would give us a simple fine grained control for putting an upper bound on the bandwidth allocated to VM (workload).

4 Performance Evaluation

In this section, we present extensive experimental evaluation of *RBFQ* using both a prototype implementation in VMware ESX Server 3.5 hypervisor [4] and a simulated storage environment using DiskSim [1]. Our goal is to show the following properties of *RBFQ* in this evaluation:
(1) Show that *RBFQ* provides allocation in proportion to global weights, while meeting reservation constraints, and thus it can provide any behavior in between locally and globally fair scheduling.
(2) Show that *RBFQ* can handle bursts effectively and reduce latency by giving idle credit.
(3) Evaluate the performance of *RBFQ* in a distributed storage environment using real traces and show that it can handle dynamic workloads.

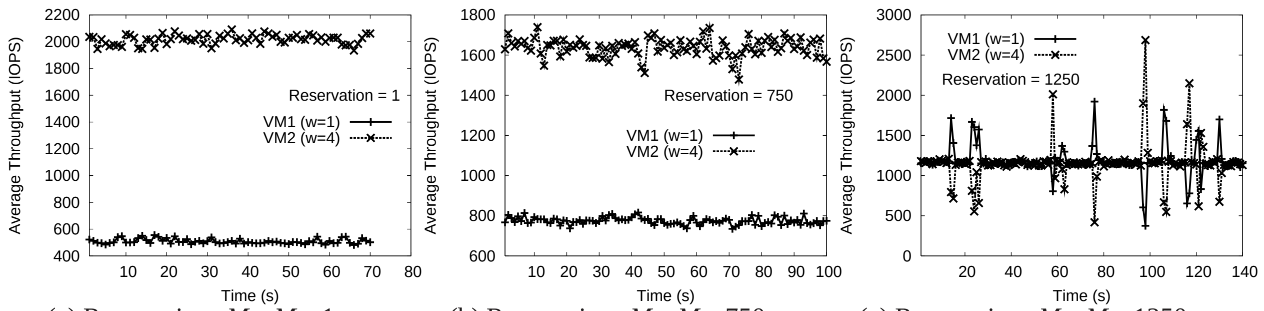
For each of the use cases, we first present the details of the experimental setup, followed by the results highlighting each of the properties.

4.1 VM Hosting Evaluation

We implemented RBFQ by modifying the SCSI scheduling layer in the I/O stack of VMware ESX Server 3.5 to construct our prototype. A single hypervisor-based server manages multiple virtual machines(VMs), where each VM is associated with resource settings for CPU, memory, I/O etc. In this case we mainly focus on storage I/O resource and focus of I/O intensive workloads. We used two kinds of VMs: (i) Linux operating system (RHEL3) with 10GB virtual disk, 3GHz CPU and 512 MB memory, (ii) Windows Server 2003 operating system with 16GB virtual disk, 3GHz CPU and 1 GB of memory. All VMs share a 10 disk, RAID 5 LUN of size 600GB hosting a clustered file system *vmfs*. The VMs are hosted by a Dell Poweredge 2950 server with 2 Intel Xeon 3.0 GHz dual core CPUs, 8GB of RAM and Qlogic HBAs connected to an EMC CLARiiON CX3-40 storage array over a Fiber Channel network. The server is configured to keep 32 IOs pending per LUN at the array, in order to have some concurrency at the array and also to avoid overflow of adapter or port queues when multiple hosts try to simultaneously send a large number of IO requests. Each VM acts as a client and has associated reservation and weight parameters. In practice these will be determined by the workload on the VM. For the experiments we specify each parameter in terms of IOPS, although they can be easily converted to bytes by using a simple mapping from IOs to IO sizes.

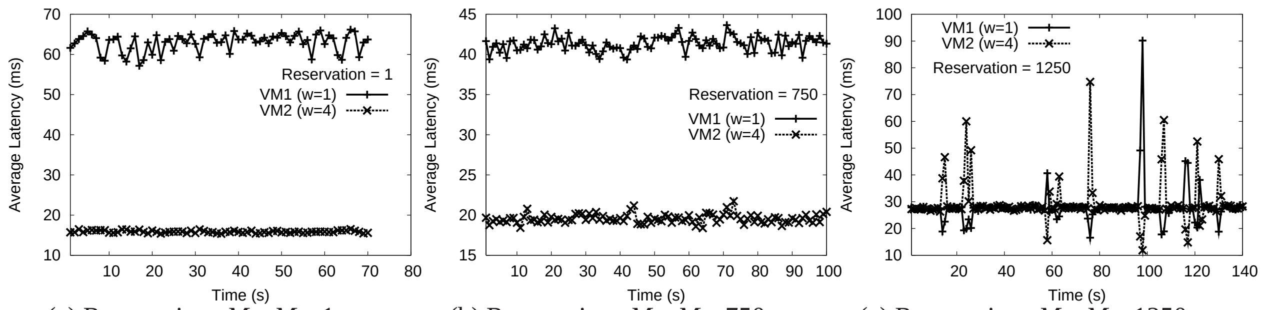
4.1.1 Global Fairness with Reservation

We experimented with 2 VMs running RHEL3, with weights in ratio 1:4. Each VM runs a workload which is configured with 4 parameters: IO size, number of IOs, concurrency and run length. Here concurrency denotes the number of pending IOs at any instant, and run length denotes the length of sequential runs. Thus a run length of 4 means four sequential IOs followed by one random IO. We experimented with concurrency = 32, run length = 1 and IO size = 16K. We measured the IOPS seen by the workload; other VM traffic is almost non-existent compared to that generated by the workloads. Figure 6(a) shows the number of IOPS obtained by two VMs, when both reservations M_i are set to 1 IOPS. We observe that VM1 and VM2 get approximately 500 and 1950 IOPS re-



(a) Reservations M_1=M_2=1 (b) Reservations M_1=M_2=750 (c) Reservations M_1=M_2=1250

Figure 6: These figures show the IOs/sec observed by VM1 and VM2 when their weights are in ratio 1:4 and reservations are set to 1,750 and 1250 IOPS



(a) Reservations M_1=M_2=1 (b) Reservations M_1=M_2=750 (c) Reservations M_1=M_2=1250

Figure 7: These figures show the latency observed by VM1 and VM2 when their weights are in ratio 1:4 and reservations are set to 1,750 and 1250 IOPS

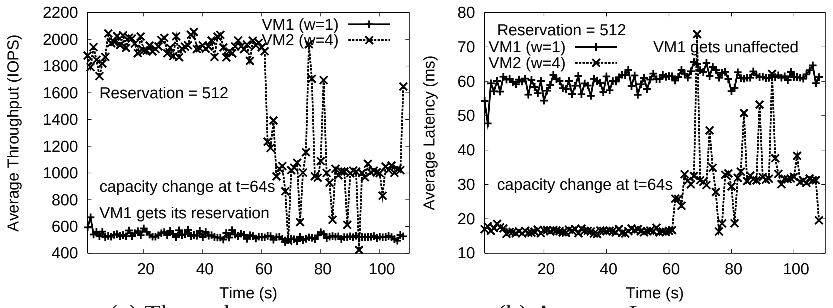
spectively, which is in proportion to their weights. Next we experimented with two more values of reservation: 750 and 1250 IOPS. The results are shown in Figures 6(b) and (c). Note that as we increase the reservation value, $VM1$ always gets its reservation even though bandwidth is no longer in the ratio of weights.

These results show that when reservation is very low, the bandwidth is allocated in proportion to the global weights (Figure 6(a)). When reservation is 750, the VM with lower weight gets it reservation and the remaining is allocated to the other VM as shown in Figure 6(b). Finally, when capacity is smaller than the total reservations, the allocation is in proportion of the M_i (1:1) as shown in Figure 6(c).

Figure 7 shows the latency observed for the reservation values of 1, 750 and 1250 respectively. The latencies are inversely proportional to throughput obtained, since the workloads are always backlogged.

4.1.2 Variable Capacity

Storage capacity can fluctuate widely due to changes in the workload characteristics or an increase in IO activity at the array from other VMs. We look at a scenario where the workload on the array changes due to IO activity by the Windows Server 2003 VM. We again experimented with 2 Linux VMs with weights in the ratio 1:4 and reservations set to 512 IOPS. At t=64 sec, we started an IO intensive workload with 32 IOs pending at all times from the Windows server VM. Figure 8(a) and (b) show the average throughput and latency obtained by the two VMs. Before $t = 64$, both VMs were getting throughput in proportion to their weights (approx 510 and 2000 IOPS respectively). As the capacity changes, VM1 still keeps on getting similar throughput and latency, whereas VM2



(a) Throughput (b) Average Latency

Figure 8: Average throughput and latency for VM1 and VM2, when capacity suddenly gets reduced at t=64 due to starting of another VMs workload. Note that VM1 always meets its reservation of 512 IOPS and its latency remains unaffected. VM2 on the other hand gets smaller IOPS and sees higher latency.

finds it IOPS reduced and faces higher latencies. This shows that reservations can provide robust isolation and smaller latency in the presence of capacity fluctuations.

4.1.3 Variable VM Workloads

We also experimented with diverse VM workloads, where each VM is sending workload with different characteristics. We used four Windows Server 2003 VMs, each running IOmeter [2] to generate the workload. The workloads are described in Table 4. Each VM keeps 32 IOs pending at all times. We also measured the base IOPS, MB/s and latency observed by each VM when run in isolation on the host. The reservation is set to 1 and weights are set in ratio 1:1:2:2 for the four VMs. We ran all four VMs simultaneously and the results for IOPS and average latency are shown in Table 5. Note that VMs get the IOPS in proportion to their weights, and latencies are again inversely proportional to the IOPS. Then we set the reservation for each VM to be 512 IOPS and the results are presented in last column of Table 5. Note that first two VMs get high IOPS of around 500 instead of 330 in order to meet their reservations. Other VMs see a corresponding decline in their throughput.

VM	Workload (size, read%, random%)	Throughput (IOPS)	Avg Latency
VM1	4K, 70%, 80%	3200	10 ms
VM2	8K, 90%, 100%	2850	11.8 ms
VM3	16K, 75%, 20%	1950	17.7 ms
VM4	8K, 100%,10%	4100	7.9 ms

Table 4: Workloads issued by different VMs

4.1.4 Bursty VM Workloads

Next we experimented with the idle credits given to a VM, which is bursty in nature. Recall that idle credit allow a workload to get service in a burst only if the workload has been idle in past and also the reservations for everyone is met. This ensures that if someone remains idle for a while, they get preference, next time when there is spare capacity in the system. We used two Windows VMs with workloads 4K,100% reads, 80% random and 16K, 100% read, 20% random.First VM sends a bursty workload that issues 128 IOs every 400ms. We use the value of idle credits of 1 and 64 for our experiment. Table 6 show the IOPS and average latency obtained the bursty VM for different values of idle credit. The number of IOPS remains almost same because idle credits don't impact the overall bandwidth allocation over a period of time and VM1 has a bounded arrival rate. VM2 also see almost the similar IOPS for different values of idle credits. However, we notice that latency observed by the bursty VM gets lower as we increase the idle credits. VM2 also see similar or slightly smaller latency, perhaps due to the increase in ef-

VM	Res=1, [IOPS, ms]	Res=512, [IOPS,ms]
VM1	330, 96ms	490, 68ms
VM2	390, 82ms	496, 64ms
VM3	660, 48ms	514, 64ms
VM4	665, 48ms	530,65ms

Table 5: Throughput and Latency observed by VMs running different workloads for Reservation = 1 and 512 IOPS

VM	σ=1, [IOPS, ms]	σ=64, [IOPS,ms]
VM1	312, 49ms	316, 30.8ms
VM2	2420, 13.2ms	2460, 12.9ms

Table 6: Throughput and Latency observed by VMs running different workloads for idle credit = 1 and 64 IOPS

ficiency of doing a bunch of IOs from a single VM which are likely to be closer to each other in logical block number space on the LUN. This shows that idle credits gives us a useful mechanism to to help the bursty workloads with their latency.

4.2 Distributed Storage Evaluation

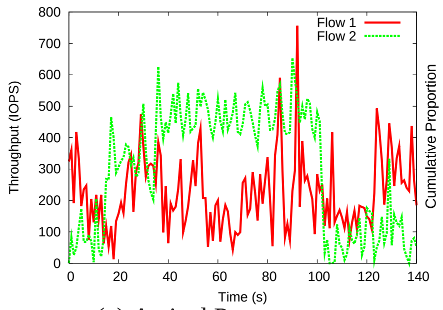
In this section, we present the results of *RBFQ* implementation for a distributed storage system. For distributed storage simulation, we implemented *RBFQ* in Disksim [1], which is well known disk and storage system simulation tool. In this case, we chose a configuration with multiple controllers, each controlling one or more disks. We represent each storage node by a single controller (running *RBFQ*). Clients access the controllers (servers) through driver code and the driver per client acts as a gateway, which piggy-backs the δ value with each request. Disksim simulates the complex behavior of disks in terms of actual seek and rotational delays, caching at controller and bus conflicts.

We ran *RBFQ* at each of the controllers controlling a single Seagate Cheetah disk with 18GB capacity. First, we verified the simpler cases where a storage server is always able to meet the reservation set by the client, irrespective of the global share of the client based on capacity. Some of the storage simulation results are also presented in Appendix A. We experimented with two real traces: 1 websearch and 1 oltp workload. The traces are taken from a trace repository [3] available online. The web search traces are collected at a popular search engine and the oltp trace is collected at a financial data processing engine with many clients. Since the traces are big, we present results for a section (200 seconds) of the trace for better clarity. For both workloads, we mapped some parts of the trace to flows and storage servers. We are presenting only latency results because these are open workloads and throughput is based on arrivals instead of weights.

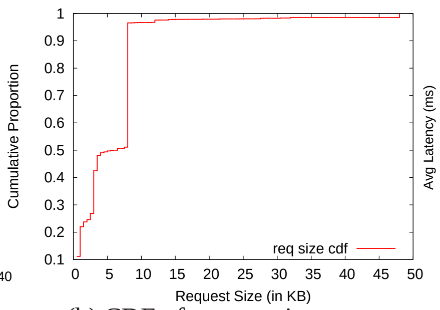
oltp1: This trace consisted of 19 devices. We used 2 flows (f_1, f_2) and 3 servers (s_1, s_2, s_3) for this experiment. We mapped the first ten devices to f_1 and remaining to f_2. Also the devices are mapped to two servers for flow 1 (s_1, s_2) and three servers for flow 2 (s_1, s_2, s_3). Flow 2 has reservation on s_1 , $M_{21} = M_{22} = M_{23} = 10$ IOPS and a weight of $w_2 = 1$. Two sets of parameters are used for f_1. For the first experiment, the weight $w_1 = 1$ and the reservation is set to 10 on each server. For the second experiment, $w_1 = 2$ and the reservation is set to 200. This essentially simulates a situation when there is no differentiation and compares that with the reservation and fairness provided by RBFQ. Figure 9(a) shows the arrival pattern for the requests for each disk. Average request size is only 5.8KB, which is expected from an oltp trace and 98% requests are $\leq 16KB$. Figure 9(c) shows the latency observed by the flow f_1 for the two sets of parameters. Note that the latency is reduced when we provide better isolation by doubling the weight and adding a reservation. Also RBFQ is very effective in absorbing bursts compared to the case with no differentiation.

websearch1: This trace consisted of 3 devices. We used 2 flows (f_1, f_2) and 2 servers (s_1, s_2) for this experiment. Flow 1 consists of IOs on first two devices mapped to (s_1, s_2) respectively. Flow 2 consists of the third device mapped only to s_1. The actual request rate for both flows is shown in Figure 10(a). Figure 10(b) shows the cdf of requests sizes. Average request size for the trace is 15 KB with 99.9% requests of size $\leq$32KB. Flow 2 has reservation reservation on each server equal to 20 IOPS and a weight of $w_2 = 1$. Two sets of parameters are used for f_1. The minimum reservation on each server in the two cases are 20 and 100 respectively, and the weight is $w_1 = 1$ in both cases.

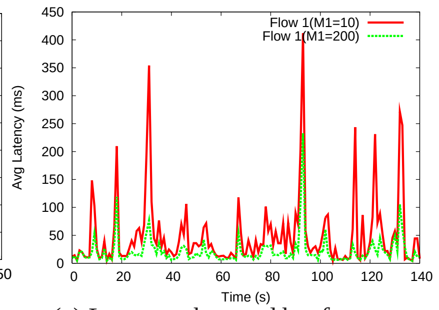
Figure 10(c) show the average response time observed



(a) Arrival Pattern

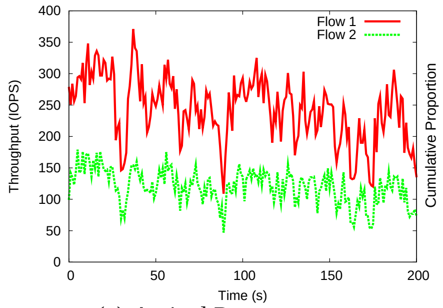


(b) CDF of request size

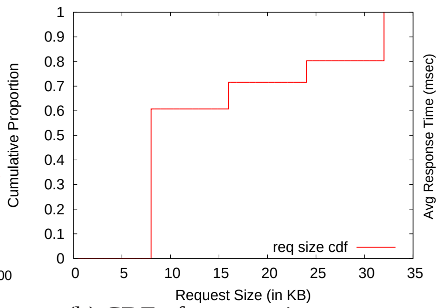


(c) Latency observed by f_1

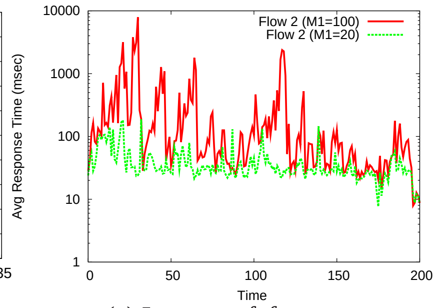
Figure 9: Oltp1: (a) and (b) show the arrival pattern for two flows and the cdf of request sizes in the trace respectively. (c) shows the average latency(ms) observed by f_1 when the local requirement for f_1 is set to 10 and 200 and its weight is doubled.



(a) Arrival Pattern



(b) CDF of request size



(c) Latency of f_2

Figure 10: Websearch1: (a) and (b) show the arrival pattern for two flows and the cdf of request sizes in the trace respectively. (c) shows the average latency(ms) observed by f_2 when the local requirement for f_1 is set to 20 and 100

by f_2, when local requirement of f_1 is 20 and 100 IOPS. Note that f_2 sees much higher latencies when reservation for f_1 is increased to 100 IOPS. This is because most of the server capacity is dedicated to f_1 and RBFQ prefers meeting local guarantees over global fairness. In practice one should be careful not to set very high local requirements for flows.

4.3 IO size and Efficiency

In our experiments, we have allocated service in terms of IOPS, but it can be easily implemented in terms of KB/s, simply by scaling each I/O with the size of the request. Based on our experience with the cost of IO requests, we think that using a certain sized IO for each unit of scheduling may be better than simply using IO size. This is because the cost of 4KB IO isn't half of the cost of 8KB IO at the array. We plan to extend the scheme to consider a fixed sized IO (i.e. 16KB) as the unit of allocation. Thus the cost of the IO will get incremented on the basis of the allocation unit instead of linearly.

Secondly, there is an efficiency concern with running a proportionate allocation mechanism for I/O scheduling. Most storage arrays have a large number of disks and they schedule multiple IOs to disks permitting aggressive seek optimization and concurrency. Our server also keeps 32 IOs active per LUN. In most cases, we noticed that keeping a 32 IOs pending at the array itself ensures high concurrency. We used a batch size of 8 IOs per VM before switching to another VM in most cases. Changing this value did not lead to much improvement in most cases. This is again a configurable parameter, that one can tune to trade-off fairness and efficiency based on the workload characteristics.

5 Related Work

There are several existing scheduling algorithms, but they are either centralized, providing proportional service without any guarantee of minimum service levels, or distributed algorithms that cannot provide globally proportional service to applications. We summarize the existing approaches below; *RBFQ* can emulate any of them and provides additional benefits as well.

A simple way to provide proportional service is to run a *fair queuing algorithm* such as WFQ [9], SFQ [15], DRR [24], self-clocked [12] WF^2Q [6] for scheduling. All these fair schedulers assign tags to requests based on their arrival times and the weight of the client. In a distributed environment, these algorithms can be run either as a centralized scheduler, providing globally fair service without any local service guarantees, or at each of the servers locally, providing locally proportional service without global fairness guarantees. Other storage-specific algorithms [19, 27, 18] that are variants of fair queuing methods have similar limitations. Argon [27] and Aqua [34] propose service time based disk allocation to provide fairness as well as high efficiency. However, measuring service times per request in our use cases is quite difficult because multiple requests can be pending at an array.

For CPU scheduling and memory management, several approaches have been proposed for integrating reservations with proportional-share allocations [31, 25]. Guaranteeing minimum resource allocations for CPU is relatively straightforward, since CPU capacity is typically fixed. For a given configuration with fixed proportional-share weights, each weight also implies a minimum worst-case reservation. Since 2003, VMware ESX Server has provided both reservation-based and proportional-share controls for both CPU and memory resources in a commercial product [4, 5, 30]. The ESX Server CPU scheduler implementation also bears many similarities to the RBFQ techniques of dual tagging (multiple virtual times) and prioritized scheduling (two-level scheduling), which are used to efficiently support reservations, proportional sharing, and time-averaged fairness for bursty workloads [28].

For the case of distributed storage, Wang and Merchant [32] proposed a scheme in which they run SFQ(D) [19] at each of the servers, adjusting the tags by an extra delay to account for service received by the client at other servers. This ensures global fairness, but they have shown cases in which a request can have unbounded latency. They also propose a hybrid approach, where the extra delay is statically capped to guarantee the required minimum service at a server. However, there are natural cases where this static upper bound may allocate more service than the minimum required that lead to global unfairness. *RBFQ*, on the other hand, always enforces global fairness whenever the minimum requirements permit it. LexAS [17] is also proposed as a *centralized, batch-oriented scheduler* that assigns the requests in each batch to the servers to reach as close as possible to the desired global allocation based on the stipulated weights. LexAS can provide globally fair allocation in the presence of resource conflicts, but it can also lead to large (potentially unbounded) latency for some applications on some servers.

6 Conclusions

In this paper, we presented a novel scheduling algorithm, *RBFQ*, that provides proportionate service allocation while meeting the reservations for clients accessing a shared storage system. Our algorithm efficiently handles variable storage bandwidth and is sensitive to the bursty nature of workloads. We demonstrate our approach for two use cases: IO scheduling in virtualized servers and distributed storage systems such as FAB and IceCube. Our evaluation shows that *RBFQ* meets the desired goal of providing bandwidth to clients in proportion to their weights while meeting minimum reservations. *RBFQ* is efficient to implement in centralized virtual machine IO scheduling and doesn't require communication among the storage nodes in the distributed system.

7 Acknowledgement

Acknowledgement: Part of the work was done while the first author was a PhD student at Rice University. An earlier version of the algorithm appeared as a brief announcement in PODC 2007. The support of the National

Science Foundation under Grant CCR-0541369 is gratefully acknowledged. We would also like to thank Carl Waldspurger for valuable discussions and feedback.

References

[1] The disksim simulation environment (version 3.0). http://www.pdl.cmu.edu/DiskSim/.

[2] Iometer. http://www.iometer.org.

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

[4] VMware ESX Server User Manual, December 2007. VMware Inc.

[5] VMware resource management guide, 2007. http://www.vmware.com/support/pubs/resource-management/vi-pubs-res-mgmt.html.

[6] J. C. R. Bennett and H. Zhang. WF^2Q: Worst-case fair weighted fair queueing. In *INFOCOM (1)*, pages 120–128, 1996.

[7] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz. Disk scheduling with quality of service guarantees. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems, Volume 2*. IEEE Computer Society, 1999.

[8] A. Chandra, M. Adler, P. Goyal, and P. Shenoy. Surplus fair scheduling: a proportional-share cpu scheduling algorithm for symmetric multiprocessors. In *OSDI*, pages 4–4, Berkeley, CA, USA, 2000. USENIX Association.

[9] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queuing algorithm. *Journal of Internetworking Research and Experience*, 1(1):3–26, September 1990.

[10] K. J. Duda and D. R. Cheriton. Borrowed-virtual-time (bvt) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. *SIGOPS Oper. Syst. Rev.*, 33(5):261–276, 1999.

[11] M. A.-E.-M. et. al. Ursa Minor: Versatile cluster-based storage. In *USENIX FAST*, 2005.

[12] S. Golestani. A self-clocked fair queueing scheme for broadband applications. In *INFOCOMM'94*, pages 636–646, April 1994.

[13] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum. Cellular disco: resource management using virtual clusters on shared-memory multiprocessors. In *SOSP*, pages 154–169, New York, NY, USA, 1999. ACM.

[14] P. Goyal, X. Guo, and H. M. Vin. A hierarchial cpu scheduler for multimedia operating systems. *SIGOPS Oper. Syst. Rev.*, 30(SI):107–121, 1996.

[15] P. Goyal, H. M. Vin, and H. Cheng. Start-time fair queuing: A scheduling algorithm for integrated services packet switching networks. Technical Report CS-TR-96-02, UT Austin, January 1996.

[16] A. Gulati, A. Merchant, and P. Varman. *p*Clock: An arrival curve based approach for QoS in shared storage systems. In *ACM Sigmetrics*, 2007.

[17] A. Gulati and P. Varman. Lexicographic QoS scheduling for parallel I/O. In *Proceedings of the ACM SPAA*, pages 29–38. ACM Press, 2005.

[18] L. Huang, G. Peng, and T. cker Chiueh. Multi-dimensional storage virtualization. In *SIGMETRICS '04*, pages 14–24, New York, NY, USA, 2004. ACM Press.

[19] W. Jin, J. S. Chase, and J. Kaur. Interposed proportional sharing for a storage service utility. In *ACM SIGMETRICS '04*, pages 37–48, 2004.

[20] J. R. Jump. Yacsim reference manual. http://www.owlnet.rice.edu/ elec428/yacsim/yacsim.man.ps.

[21] Y. Saito et al. FAB: building distributed enterprise disk arrays from commodity components. *SIGPLAN Not.*, 39(11):48–58, 2004.

[22] H. Sariowan, R. L. Cruz, and G. C. Polyzos. Scheduling for quality of service guarantees via service curves. In *Proceedings of the International Conference on Computer Communications and Networks*, pages 512–520, 1995.

[23] P. J. Shenoy and H. M. Vin. Cello: a disk scheduling framework for next generation operating systems. In *ACM SIGMETRICS*, pages 44–55. ACM Press, 1998.

[24] M. Shreedhar and G. Varghese. Efficient fair queueing using deficit round robin. In *Proc. of SIGCOMM '95*, pages 231–242, August 1995.

[25] I. Stoica, H. Abdel-Wahab, and K. Jeffay. On the duality between resource reservation and proportional-share resource allocation. *SPIE*, February 1997.

[26] D. G. Sullivan and M. I. Seltzer. Isolation with flexibility: a resource management framework for central servers. In *USENIX*, 2000.

[27] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger. Argon: performance insulation for shared storage servers. In *USENIX FAST'07*, pages 5–5, Berkeley, CA, USA, 2007. USENIX Association.

[28] C. Waldspurger. Personal Communications.

[29] C. A. Waldspurger. *Lottery and stride scheduling: flexibile proportional-share resource management*. PhD thesis, Cambridge, MA, USA, 1996.

[30] C. A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, 2002.

[31] C. A. Waldspurger and W. E. Weihl. An object-oriented framework for modular resource management. In *IWOOOS*. IEEE Computer Society, 1996.

[32] Y. Wang and A. Merchant. Proportional-share scheduling for distributed storage systems. In *Usenix FAST*, feb 2007.

[33] W. Wilcke et al. IBM intelligent bricks project- petabytes and beyond. *IBM Journal of Research and Development*, 50, 2006.

[34] J. C. Wu and S. A. Brandt. The design and implementation of Aqua: an adaptive quality of service aware object-based storage device. In *Proc. of IEEE/NASA MSST*, pages 209–18, May 2006.

[35] L. Zhang. VirtualClock: A new traffic control algorithm for packet-switched networks. *ACM Trans. Comput. Syst.*, 9(2):101–124.

8 Appendix A

8.1 RBFQ: Distributed Storage Simulation

We have implemented *RBFQ* and other algorithms using a simulation environment called Yacsim [20]. We implemented a process oriented simulation of a distributed storage system using Yacsim [20], which is a discrete event simulator developed at Rice. Each flow, gateway and storage server is represented by a process. The flow process generates requests at a certain rate and with a certain distribution among servers (uniformly accessing all of them or hot spotting on few). A gateway process receives requests from a flow and sends them to server along with the appropriate δ parameter. The gateways keep a bounded number of requests pending at each server, thereby being work conserving. A server process service requests using *RBFQ* algorithm, while assuming an exponential distribution for service times. This setup although simplistic, provides a *platform to compare various approaches and observe their behavior in a controlled environment*. We are presenting these results to show the fine grained comparison of *RBFQ* with others and the subtle effects of various components in *RBFQ* itself.

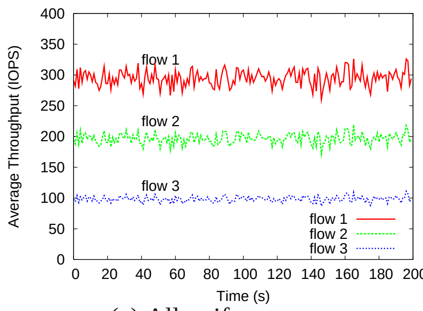
8.2 Yacsim Based Evaluation

8.2.1 Global Bandwidth Allocation

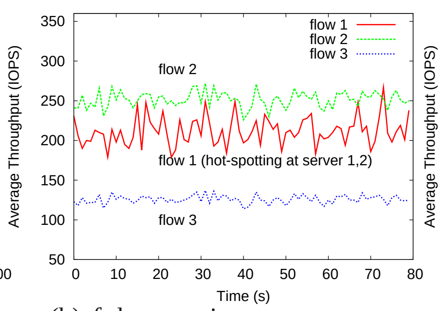
To test the ability of *RBFQ* in providing bandwidth allocation in proportion to global weights, we experimented with three flows/clients (denoted as f_1, f_2, f_3) and three servers (denoted as s_1, s_2, s_3). Each server has an average capacity of 200 IOPS, i.e. service times of requests follow exponential distribution with mean $1/200$ seconds. The global and local needs of flows are $\{300, 80\}, \{200, 60\} and \{100, 30\}$ respectively. *Throughout this section global and local needs represent global shares and reservation per client per server.* Figure 11(a) shows the overall throughput obtained by the flows, when the flows are accessing all the servers. In this case flows get overall bandwidth in ratio of their global weights and local requirements are also met. As mentioned earlier, *RBFQ* does scheduling in two phases : local and global. We looked at the distribution of requests done in each of these phases to get further insight into the behavior. Figure 12 shows the distribution of requests done at each server. Figure 12(a) shows the distribution when all servers are accessed uniformly. Here we notice that the local phase meets the minimum requirement, and the rest of the requests are done during a global phase and overall requirements of every flow are met.

We ran the experiment with similar capacity and requirements, with flow f_1 hot-spotting only on servers s_1 and s_2. Figure 11(b) shows the overall throughput achieved by the flows. Note that share of f_2 and f_3 is increased in the proportionate manner on s_3 but their overall ratio is maintained as 1:2. Also note that both f_2 and f_3 get their local requirements at s_1, s_2 and remaining capacity is allocated to f_1 to achieve global fairness. Figure 12(b) shows the distribution at servers for this case, when f_1 is hot spotting on s_1 and s_2. Note that f_2 and f_3 still meet their local requirements at s_1,s_2, however they don't get any request in global phase because all the remaining capacity is allocated to f_1 in order to meet its global requirement.

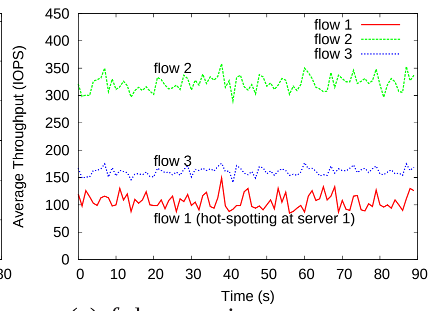
Then we constrain the scenario even further with flow



(a) All uniform

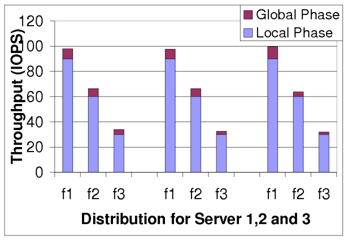


(b) f_1 hot spotting on s_1,s_2

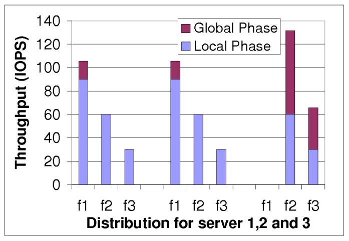


(c) f_1 hot spotting on s_1

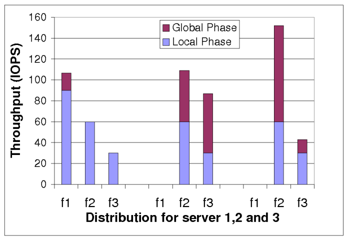
Figure 11: Throughput obtained by different flows under uniform and hot-spotting cases.



(a) All uniform



(b) f_1 hot spotting on s_1,s_2



(c) f_1 hot spotting on s_1

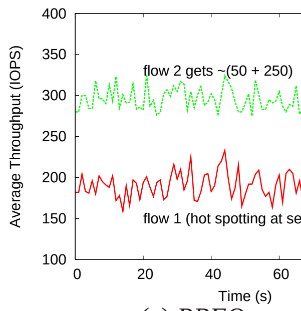
Figure 12: Distribution of requests done in local and global phase by *RBFQ* for various degrees of hot-spotting

f_1 hot-spotting only on server s_1. Figure 11(c) shows the overall throughput achieved by the flows. Note that share of f_2 and f_3 is increased in the proportionate manner on s_2 and s_3. Also note that both f_2 and f_3 get their local requirements at s_1 and remaining capacity is allocated to f_1 to achieve global fairness. This is shown clearly in Figure 12(c) that all the requests done in global phase belong to f_1 and others just get their local requirement. This validates that *RBFQ* always meets local constraints and uses any spare capacity to maintain global fairness.

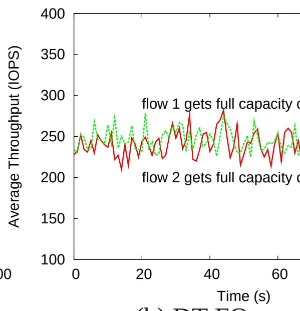
8.2.2 Comparison with other approaches

Now we will compare our results with other existing approaches. We experimented with a simple case consisting of two servers and two flows. The requirements for flows f_1, f_2 are $\{300,150\}$ and $\{100,50\}$ respectively. We assume that global requirement is uniformly distributed over servers. Server capacity is 250 IOPS, simulated using a exponentially distributed request completion times with mean $1/250$. To highlight the differences we made f_2 hot spot only on server s_1. We compare our algorithm with two other approaches: (1) LFQ: Each server is running a local fair scheduler [9] using weights in ratio 3:1 (2) DT-FQ: this approach is based on approach proposed by Yin and Merchant [32], where each server runs SFQ(D) [19] while delaying requests based on δ value sent by gateways. They proposed a hybrid mechanism to avoid resource specific starvation but choosing a parameter to work for all cases is quite difficult. More discussion on this approach is presented in Section 5. Figure 13(a), (b) and (c) show the average throughput obtained using *RBFQ*, DT-FQ and LFQ respectively. The summary of results is as follows:

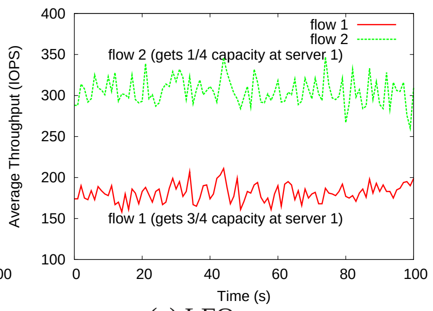
*RBFQ***:** allocates 50 IOPS to f_2 and remaining 200 IOPS to f_1 on s_1. Server s_2 is completely allocated to f_2 because there is no contention there. Thus f_2 *gets its local requirement on* s_1 *and no more*. Thus *RBFQ* tries to be



(a) *RBFQ*

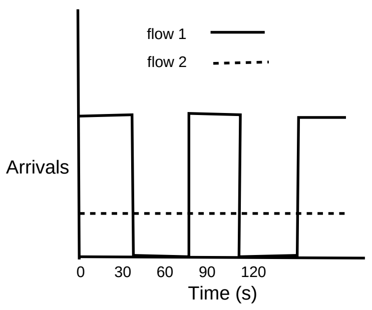


(b) DT-FQ

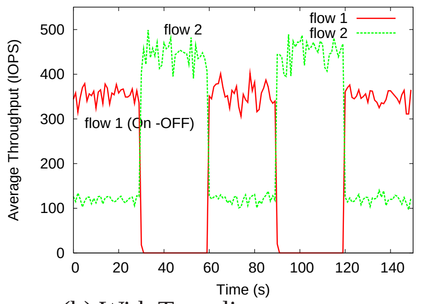


(c) LFQ

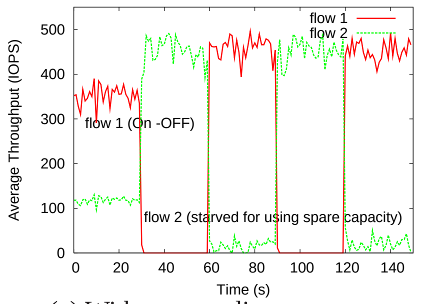
Figure 13: Comparison of *RBFQ* with distributed tagging FQ and Locally fair queuing at each server, when f_2 is hot-spotting at server s_2



(a) Arrival Pattern



(b) With Tag adjustment



(c) Without tag adjustment

Figure 14: Effect on tag adjustment on flows when they become idle and active. Note that f_2 is starved for using spare capacity left idle by f_1, if tags are not adjusted to synchronize them

globally fair while maintaining local constraint.
DT-FQ: allocates server s_1 to f_1 and s_2 to f_2. Distributed tagging approach uses information from the coordinator to space out tags on s_1 for f_2. Thus f_2 is starved on s_1. DT-FQ tries to be globally fair at the expense of missing local requirement. Setting up local requirement to 0 in *RBFQ* has the similar effect.
LFQ: allocates s_1 in a locally fair manner by giving IOPS in ratio 3:1 to f_1 and f_2. Thus f_1 gets 185 IOPS and f_2 gets remaining 65 + 250 IOPS on server s_2. Thus LFQ uses no information about global access pattern of different flows.

8.2.3 ON-OFF Workloads

So far we have looked at cases when the workloads are always backlogged. Now we will look at more dynamic cases with ON-OFF arrivals from workloads, so as to see the effectiveness of our tag adjustment component in keeping the flows synchronized. We experimented with 2 flows, with requirements $\{300,150\}$ and $\{100,50\}$ respectively. However flow f_1 is ON-OFF with a period of 30 seconds. Both flows keep a backlog of 16 requests during ON periods, so that they can take benefit of capacity left idle by anyone. We tested the throughput obtained by two flows, with and without tag adjustment in *RBFQ*. Figure 14(a), (b) and (c) show the arrival pattern, IOPS with tag adjustment and IOPS without tag adjustment respectively. Note that with tag adjustment (Fig 14(b)), as soon as f_1 gets active, both the flows start sharing in the desired manner and f_2 is not penalized for using spare capacity in the system. However, if tag adjustment is not done, flow f_2 is starved when f_1 gets active at 60 and 120 seconds. This is because f_2's tags get higher and are not synchronized with f_1 at those times. This shows that tag adjustment is necessary and it is able to handle dynamic flows in the system. Also during OFF periods for f_1, flow f_2 is able to utilize spare capacity showing work conserving property of our algorithm.

9 Appendix B

9.1 Analysis of Service Allocation

Lemma 1. *The sequence of critical capacities $C_1, C_2, \cdots C_n$, where $C_k = (M_k/w_k)\sum_{j=1}^{k} w_j + \sum_{i=k+1}^{n} M_i$, is a non-decreasing sequence.*

Proof. By definition, $M_k/w_k \leq M_{k+1}/w_{k+1}$, $1 \leq k \leq n-1$. We will show that $C_k \leq C_{k+1}$. Now, $C_k = (M_k/w_k) \sum_{j=1}^{k} w_j + \sum_{i=k+1}^{n} M_i = (M_k/w_k)\sum_{j=1}^{k} w_j + M_{k+1} + \sum_{i=k+2}^{n} M_i \leq (M_{k+1}/w_{k+1}) \sum_{j=1}^{k} w_j + (M_{k+1}/w_{k+1})w_{k+1} + \sum_{i=k+2}^{n} M_i = (M_{k+1}/w_{k+1}) \sum_{j=1}^{k+1} w_j + \sum_{i=k+2}^{n} M_i = C_{k+1}$. □

We consider the interesting case where $C \geq \sum_{i=1}^{n} M_i$, when there is sufficient capacity to give each client its minimum request. Divide the interval $[0,T]$ into subintervals $\Delta T_i = [t_{i-1}, t_i]$, $i = 1, 2, \cdots$, where $t_0 = 0$. In each interval ΔT_i the server provides total service equal to $C \times (t_i - t_{i-1})$, broken into a period of *mandatory service* followed by a period of *discretionary service*. In the former period each client c_j receives mandatory service $M_j \times (t_i - t_{i-1})$. Since $C \geq \sum_{j=1}^{n} M_j$, there is excess capacity of $\hat{C} = C - \sum_{j=1}^{n} M_j$ which is used to provide discretionary service in the rest of the interval.

Definition: The total *actual service* received by c_j up to time t will be denoted by $S_j(t)$. The *normalized service* received by c_j up to time t will be denoted by $\sigma_j(t) = S_j(t)/w_j$.

During the discretionary period service is provided to try and equalize the normalized service received by the clients, by always providing service to the client with the currently smallest amount of normalized service.

Lemma 2. *Let $c_j, j = 1, \cdots n$ be a set of continuously backlogged clients in some interval $[0,T]$. Then: $\sigma_1(T) \leq \sigma_2(T) \leq \cdots \leq \sigma_n(T)$.*

Proof. We show this by induction over time steps t_i. The Lemma holds at $t_0 = 0$ since $\sigma_j(0) = 0$, for all $1 \leq j \leq n$. Suppose the lemma holds at time t_{i-1}, so $\sigma_j(t_{i-1}) \leq \sigma_{j+1}(t_{i-1})$, $0 \leq j < n$. We will show that the inequalities also hold at t_i.

Let σ'_j denote the total normalized service received by c_j immediately after the mandatory period in the interval ΔT_i. During this period c_j received $M_j \times (t_i - t_{i-1})$ amount of mandatory service. Hence: $\sigma'_j = \sigma_j(t_{i-1}) + (t_i - t_{i-1}) \times M_j/w_j$. Note that $\sigma'_j \leq \sigma'_{j+1}$, for all $1 \leq j < n$, since by the Induction Hypothesis, $\sigma_j(t_{i-1}) \leq \sigma_{j+1}(t_{i-1})$ and by definition $M_j/w_j \leq M_{j+1}/w_{j+1}$. Discretionary service $\hat{C}$ will then be provided to the clients to try and equalize the normalized service of as many clients as possible. Therefore there is some index k (that depends on the value of $\hat{C}$), such that $c_1, \cdots c_k$ receive service during the discretionary period while $c_{k+1}, c_{k+2}, \cdots c_n$ do not receive any discretionary service. The clients $c_j, 1 \leq j \leq k$, that receive discretionary service will equalize their normalized service $\sigma_j(t_i)$, while for all $j > k$, $\sigma_j(t_i) = \sigma'_j$. Also by the definition of k, $\sigma'_{k+1} \geq \sigma_k(t_i)$, else c_{k+1} would have received discretionary service. Hence, we have $\sigma_1(t_i) = \sigma_2(t_i) = \cdots = \sigma_k(t_i) \leq \sigma'_{k+1} \leq \sigma'_{k+2} \cdots \leq \sigma'_n$. Since $\sigma'_j = \sigma_j(t_i)$, for all $j > k$, the Lemma follows. □

Defintion: Let C denote the system capacity and $C_j, j = 1, \cdots n$ denote the critical capacities. Define k^* to be the unique integer $1 \leq k^* \leq n$, such that $C_{k^*} \leq C < C_{k^*+1}$.

We note that a unique k^* exists because of the monotonicity of the critical capacities established in Lemma 1.

Lemma 3. *Let $c_j, j = 1, \cdots n$ be a set of continuously backlogged clients in some interval $[0,T]$. Exactly the clients c_j, $1 \leq j \leq k^*$, receive discretionary service in the interval ΔT_i.*

Proof. First we show that discretionary service will be provided in increasing order of c_i until the excess capacity $\hat{C}$ is exhausted. The normalized service received by c_j up to the start of the discretionary phase of ΔT_i is given by $\sigma'_j = \sigma_j(t_{i-1}) + M_j/w_j$. By Lemma 2, $\sigma_1(t_{i-1}) \leq \sigma_2(t_{i-1}) \leq \cdots \leq \sigma_n(t_{i-1})$, and by definition $M_1/w_1 \leq M_2/w_2 \leq \cdots \leq M_n/w_n$. Hence, $\sigma'_j \leq \sigma'_{j+1}, 1 \leq j < n$, and discretionary service will be added in the order of increasing c_i.

Let α be the largest index for which c_1, $c_2, \cdots c_\alpha$ receives discretionary service in the interval ΔT_i. The total discretionary service provided to these c_i in the interval equals $\hat{C} \times (t_i - t_{i-1})$ and the total mandatory service provided to them in the interval equals $(t_i - t_{i-1}) \times \sum_{j=1}^{\alpha} M_j$. Hence, the total service provided to $c_1, c_2, \cdots c_\alpha$ during ΔT_i is $(t_i - t_{i-1}) \times (\hat{C} + \sum_{j=1}^{\alpha} M_j) = (t_i - t_{i-1}) \times ((C - \sum_{j=1}^{n} M_j) + \sum_{j=1}^{\alpha} M_j) = (t_i - t_{i-1}) \times (C - \sum_{j=\alpha+1}^{n} M_j)$.

Now for all $1 \leq j \leq \alpha$, the normalized service received by c_j up to t_i must at least be equal to the normalized service received by c_α at the start of the discretionary phase of ΔT_i (*i.e.* $\sigma_\alpha(t_{i-1}) + (t_i - t_{i-1}) \times M_\alpha / w_\alpha$). This is because c_α will start receiving discretionary service only after all the $c_j, j < \alpha$ have caught up to its normalized service. Hence in the interval ΔT_i, all $c_j, 1 \leq j \leq \alpha$, must receive at least $\sigma_\alpha(t_{i-1}) + (t_i - t_{i-1}) \times M_\alpha / w_\alpha - \sigma_j(t_{i-1})$ amount of normalized service, and since by lemma 2, $\sigma_\alpha(t_{i-1}) \geq \sigma_j(t_{i-1}), 1 \leq j \leq \alpha$, this quantity is at least $(t_i - t_{i-1}) \times M_\alpha / w_\alpha$. Therefore, the actual service received by $c_j, 1 \leq j \leq \alpha$ in the interval ΔT_i is at least $(t_i - t_{i-1}) \times M_\alpha / w_\alpha \times w_j$.

Consequently, $C - \sum_{j=\alpha+1}^{n} M_j \geq \sum_{j=1}^{\alpha} (M_\alpha / w_\alpha \times w_j)$, which implies $C \geq M_\alpha / w_\alpha \times \sum_{j=1}^{\alpha} w_j + \sum_{j=\alpha+1}^{n} M_j$. By definition, since α is the largest such index for which the inequality holds, we have $\alpha = k^*$. $\square$

Lemma 4. *Let $c_i, i = 1, \cdots n$ be a set of continuously backlogged clients in some interval $[0, T]$. The service $S_j(T)$ received by c_j, $k^* < j \leq n$ in the interval $[0, T]$ equals $M_j \times T$.*

Proof. By Lemma 3, client c_j, $k^* < j \leq n$ does not receive any discretionary service in any interval ΔT_i. The mandatory service received by c_j in the interval ΔT_i is given by $M_j \times \Delta T_i$. Hence, the total service received in $[0, T]$ is $M_j \times \sum_i \Delta T_i = M_j \times T$. $\square$

Lemma 5. *Let $c_i, i = 1, \cdots n$ be a set of continuously backlogged clients in some interval $[0, T]$. The service $S_j(T)$ received by c_j, $1 \leq j \leq k^*$, in the interval $[0, T]$ equals $T \times \eta \times w_j / \Sigma_{r=1}^{k^*} w_r$, where $\eta = C - \sum_{r=k^*+1}^{n} M_r$.*

Proof. We show this by induction over time steps t_i. The lemma trivially holds at $t_0 = 0$. Suppose the lemma holds at time t_{i-1}, so $S_j(t_{i-1}) = t_{i-1} \times \eta \times w_j / \Sigma_{r=1}^{k^*} w_r$, $1 \leq j \leq k^*$. We will show the lemma holds for $S_j(t_i)$.

By Lemma 3 exactly the clients $c_j, 1 \leq j \leq k^*$ will receive discretionary service during the interval ΔT_i. Hence the total normalized service received by each of these clients at t_i will be equal: *i.e.* $\sigma_1(t_i) = \sigma_2(t_i) = \cdots = \sigma_{k^*}(t_i) = \mathcal{N}(t_i)$. Hence for all $1 \leq j \leq k^*$, $S_j(t_i) = \mathcal{N}(t_i) \times w_j$, and by the induction hypothesis $S_j(t_{i-1}) = t_{i-1} \times \eta \times w_j / \Sigma_{r=1}^{k^*} w_r$, $1 \leq j \leq k^*$. Therefore, the total service received by $c_j, 1 \leq j \leq k^*$ during ΔT_i is given by: $\sum_{j=1}^{k^*} S_j(t_i) - \sum_{j=1}^{k^*} S_j(t_{i-1}) = \mathcal{N}(t_i) \times \sum_{j=1}^{k^*} w_j - (t_{i-1} \times \eta / \Sigma_{r=1}^{k^*} w_r) \times \sum_{j=1}^{k^*} w_j = \mathcal{N}(t_i) \times \sum_{j=1}^{k^*} w_j - (t_{i-1} \times \eta)$.

Now, of the total service $C \times (t_i - t_{i-1})$ provided during ΔT_i, all but $(t_i - t_{i-1}) \times \sum_{r=k^*+1}^{n} M_r$ is given to the clients $c_j, 1 \leq j \leq k^*$. Hence, these clients receive a total of $(t_i - t_{i-1}) \times \eta$ service during ΔT_i.

Equating the two expressions: $(t_i - t_{i-1})\eta = (\mathcal{N}(t_i) \times \sum_{r=1}^{k^*} w_r) - (t_{i-1} \times \eta)$. Hence, $\mathcal{N}(t_i) = t_i \times \eta / \sum_{r=1}^{k^*} w_r$. Since, $S_j(t_i) = \mathcal{N}(t_i) \times w_j$, the lemma is established for $t = t_i$. $\square$

Theorem:
Let $c_i, i = 1, \cdots n$ be a set of continuously backlogged clients in some interval $[0, T]$. The service $S_j(T)$ received by c_j in the interval $[0, T]$ satisfies:
(i) $S_j(T) \geq T \times M_j$, $j = 1, \cdots n$,
(ii) $S_j(T) = T \times M_j$, $k^* < j \leq n$,
(iii) $S_j(T) = T \times w_j \times (C - \sum_{r=k^*+1}^{n} M_r) / \sum_{r=1}^{k} w_r)$, $1 \leq j \leq k^*$.

Proof. The results (ii) and (iii) follow directly from Lemmas 4 and Lemma 5 respectively. From (ii) it is clear that (i) holds for c_j, $k^* < j \leq n$. Now since $C \geq C_{k^*}$, we have $C \geq (M_{k^*} / w_{k^*}) \sum_{j=1}^{k^*} w_j + \sum_{i=k^*+1}^{n} M_i$. Hence, for $k^* < j \leq n$, $S_j(T) \geq T \times w_j \times M_{k^*} / w_{k^*} \geq T \times M_j$. $\square$