An Integrated Model for Performance Management in a Distributed System

Scott A. Brandt, Carlos Maltzahn, Anna Povzner, Roberto Pineiro, Andrew Shewmaker, Tim Kaldewey
Computer Science Department
University of California, Santa Cruz
{scott,carlosm,apovzner,rpineiro,shewa,kalt}@cs.ucsc.edu

Abstract

Real-time systems are growing in size and complexity and must often manage multiple competing tasks in environments where CPU is not the only limited shared resource. Memory, network, and other devices may also be shared and system-wide performance guarantees may require the allocation and scheduling of many diverse resources. We present our on-going work on performance management in a representative distributed real-time system—a distributed storage system with performance requirements—and discuss our integrated model for managing diverse resources to provide end-to-end performance guarantees.

1 Introduction

Many computer systems ranging from small, embedded computers to large distributed systems have Quality of Service (QoS) requirements. Examples include flight control systems, defense systems, automotive systems, multimedia systems, transaction processing systems, virtual machines on shared hardware, and many others. Even traditional best-effort systems have hidden QoS requirements that are frequently expressed in terms of responsiveness.

Addressing the QoS requirements in all but the most trivial of systems may require the management of many resources: CPU, memory, network, cache, storage, power, and others. While a large amount of research has been conducted on how to provide QoS for individual resources, relatively few approaches—notably those of Lee [6] and Hawkins [3]—address overall system QoS or end-to-end QoS in distributed systems. We focus on end-to-end QoS in a distributed system using commodity hardware.

Interaction and dependencies between resources in complex/distributed systems require integrated solutions to provide overall performance guarantees. For example, compression algorithms may save network bandwidth and/or storage space, but at the cost of higher CPU utilization. Overall, the guarantees provided by a chain of resources can be no stronger than in the weakest link of that chain and

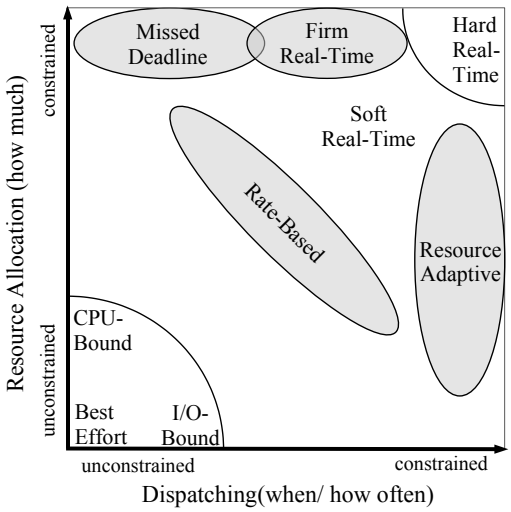


Figure 1: Classification of performance requirements in terms of Resource Allocation and Dispatching (RAD).

incompatible strategies for enforcing guarantees in different components may violate the overall QoS requirements, even if both components meet all of their individual requirements. For example, a network may provide a specified QoS by transferring a desired amount of data to a networked storage device, but in smoothing the network traffic to meet its QoS requirements, it may destroy the burstiness in the original workload that enables sequential accesses required for the disk to meet its I/O performance requirements.

Our goal is to develop a unified model for end-to-end QoS in complex and distributed systems that enables overall performance guarantees via the integrated management of all of the resources in the system. Our solution should support all types of processing guarantees ranging from best-effort to hard real-time. It should also allow the composition of guarantees on the individual resources for system-level performance guarantees independent of workloads, whether known or unknown *a priori*.

Our solution is based on the RAD scheduling model [1], originally developed in the context of CPU scheduling and subsequently extended to include other resources (*e.g.*, disk

I/O [11]). In the RAD model, resources are allocated in terms of *Resource Allocation* and *Dispatching* or, alternatively, *Rate* and *Period*. Resource allocation determines the amount of resources provided to a process over time, *e.g.*, percentage of CPU usage, network utilization, or disk head time. Dispatching determines the times at which the (reserved) resources must be delivered, effectively determining the granularity of the reservation. We have shown these two parameters to be sufficient to describe and support a wide range of scheduling policies ranging from best-effort to hard real-time [7], depicted conceptually in Figure 1.

In the RAD model, rate and period specify the desired performance, which must be enforced by the scheduler for the particular resource. The details of the scheduler depend upon the characteristics of the resource. We have developed schedulers for several resources, including CPU [1] and disk [11]. Our current work extends our disk scheduling research and adapts the RAD model to include network and I/O buffer cache management and begins to examine the interdependencies among those guarantees.

Our current focus is on managing the performance of distributed storage systems. Distributed storage shares many of the important properties of other distributed systems of interest to the embedded real-time community, such as sensor networks. In a distributed storage system, there are many independent I/O initiators operating on results in local memories and transferring data over a shared network to common targets. Where real-time data capture is important, sensor networks must also deal with local and distributed storage performance management (as well as power management).

Distributed storage performance management is challenging for a variety of reasons:

- End-to-end performance guarantees require the integrated management of at least four resources: the client buffer cache, the network, the storage server buffer cache, and the disk.
- Disk I/O is workload-dependent and individual requests are stateful and non-preemptible with response times that are only partially deterministic, varying by 3–4 orders of magnitude between best and worst-case performance.
- Independently-acting storage clients transfer data via a shared network. Rate enforcement ensures that the overall traffic is feasible, but traffic shaping must be used to avoid network congestion leading to packet loss [5, 10].
- Client and server I/O buffer caches must manage variance in the application I/O patterns and present the requests to each device so as to maximize its predictability and optimize its performance.

We discuss the RAD resource management model and explain its application to each of the system resources, providing results from our proof-of-concept implementations where available.

2 Architecture

Our target system is a distributed storage system consisting of clients accessing common storage devices over a shared network. The system is closed—we control all of the relevant resources in the system, including the clients' CPUs, buffer cache, and network access, and the servers' network access, buffer cache, and storage devices. No non-compliant traffic exists on the network and no non-compliant clients may access the storage. Although we control the resources, we do not control the applications, which may issue requests at any time.

Aside from the scale of our system, which may include up to many thousands of nodes and petabytes of storage, it is also representative of distributed embedded systems such as sensor networks or distributed satellite communications systems[1].

Our goal is to provide I/O performance guarantees to applications running on the client nodes. Application requirements have many forms: guaranteed throughput for a multimedia application; a guaranteed share of the raw disk performance for a virtual machine; and guaranteed latency for a transaction processing system. Regardless of the form of the requirements, our goal is a unified resource management system that ensures the performance of each workload through all of the resources, independent of other workloads.

Making and keeping I/O guarantees in a distributed storage system requires the integrated management of a number of resources, as shown in Figure 2, including the disk, the storage server buffer cache memory, the network, and the client buffer cache memory. The overall guarantees can be no stronger than can be provided in any individual component and the guarantees must be composable in order to provide an end-to-end guarantee.

We base our work on the RAD integrated scheduling model [1]. Originally developed for CPU scheduling, RAD separates scheduling into two distinct questions: *Resource Allocation*, or how much resources to allocate to each task, and *Dispatching*, or when to allocate the resources a task has been allocated. These two questions are independent and separately managing them allows a scheduler to simultaneously handle tasks with diverse real-time processing requirements ranging from best-effort to hard real-time [7].

[1] Although most satellite communication systems are monolithic custom (single) satellites, we are working with researchers at IBM Almaden on a DARPA-funded distributed communication satellite architecture that has many properties in common with our (ground-based) distributed storage system and which will use similar RAD-based resource management.

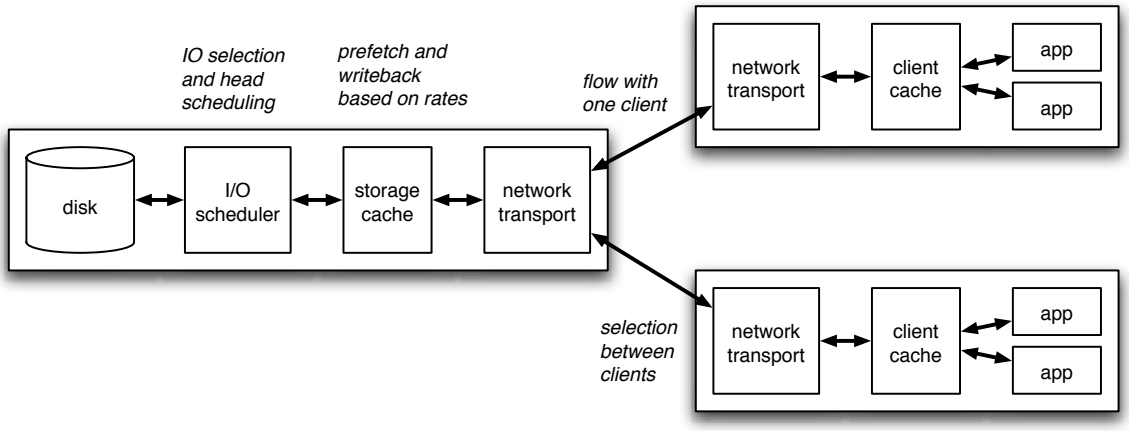


Figure 2: Components in the I/O path.

In the RAD model, feasibility of a task set is trivially verifiable by summing up the resource usage of each of the tasks sharing a resource; if the sum is less than 100% of the available resource(s), the task set is feasible. Scheduling is done at run-time and may be accomplished with any rate-enforcing optimal real-time scheduler[2]. For CPU scheduling we use a version of EDF with timers to interrupt jobs that have used up their allocated budget for the current period.

A *resource broker* is responsible for translating varied application requirements into a uniform representation of application needs and for performing the feasibility verification required for robust admission control. This depends upon the existence of a uniform resource allocation and scheduling model for all managed resources.

In order to manage the diverse resources in our system, we have had to extend the RAD model in a number of different ways. Achieving good disk performance requires both a guaranteeable metric of performance as well as careful management of the workload to ensure and maintain the physical and temporal contiguity of related requests. We manage disk performance in terms of disk head time, which is reservable and guaranteeable up to 100% of the available time [4]. We also add a third layer to the model allowing the reordering of disk requests [11]. Disk requests are dispatched according to both deadline requirements and performance heuristics.

Our simple storage network behaves somewhat like a single CPU in that each transmit port may only serve one client's data at a time. Unlike a CPU, the control of the network is decentralized; each client must independently decide when it will start and stop transmitting data. The RAD model remains relatively intact for network scheduling, but the scheduler is quite different. We introduce a novel network scheduler called Less Laxity More that is intended to approximate the behavior of Least Laxity First without centralized control.

Our work on buffer cache management currently focuses primarily on the storage server. Although cache memory can be relatively trivially partitioned according to the memory needs of each process, the RAD model determines the partition by indicating exactly how much cache is needed for each process. Each task must be able to store a multiple of the amount of data that may be transferred per period. Interestingly, this means that the best case for the disk is also the worst case for the cache, as described in Section 5. The cache may also be used for rate and period transformation between the client and the disk, allowing the client to temporarily transfer data at a higher rate than the disk allows, and to transfer data with a smaller period than is feasible for our disk scheduler.

Because each of the resources is managed via the RAD model, the guarantees are easily composable. Although the utilization of different resources vary for a given task, the deadlines will be the same, allowing for simple synchronization of the use of the different resources. Overall, if the reservation for a given I/O stream is satisfiable on each of the resources, the stream can be admitted and its I/O performance can be guaranteed.

The following sections discuss our management of each of the resources in more detail.

3 Guaranteed disk request scheduling

Our real-time disk scheduler is designed to meet three goals. First, the scheduler must provide guaranteed, integrated real-time scheduling of application request streams with a wide range of different timeliness requirements. The mechanical nature of disks adds an additional set of require-

[2] A sub-optimal scheduler may also be used, with a suitably modified feasibility test

ments. Sequential accesses experience orders of magnitude lower latencies than random accesses, and good disk scheduler can significantly improve performance by reordering requests to increase sequentiality. Thus, as a second goal, our disk scheduler must provide not just guaranteed performance but good performance. Finally, in a shared storage system, performance of an I/O stream may be affected by seeks introduced by competing I/O streams. Therefore, the scheduler must also isolate I/O streams from the behavior of others so that none of the streams cause another to violate its requirements.

Traditional real-time disk schedulers guarantee reservations on throughput [2, 13, 12]. However, due to the orders of magnitude difference between best-, average-, and worst-case response times, hard throughput guarantees on general workloads require worst-case assumptions about request times allowing reservations of less than 0.01% of the achievable bandwidth. Our Fahrrad real-time disk I/O scheduler [11] uses a different approach based on *disk time utilization* reservations [4]. A reservation consists of the *disk time utilization u* and the *period p*. Disk time utilization specifies an amount of time a disk will make available for a given request stream to service its I/O requests. The period specifies the granularity with which the request stream must receive its reserved utilization. Reservations are associated with I/O request streams, which represent related sets of requests that may come from a single user, process, application, or a set of these.

Fahrrad implements the RAD model and adapts it to disk scheduling. Since the basic goal of our scheduler is to provide a full range of timeliness guarantees, Fahrrad implements the two layers of the RAD model: resource allocation and dispatching. Resource allocation is done via the broker, which ensures feasible resource allocation and maps application requirements into disk time utilization and period. I/O request dispatching, which chooses which I/O stream requests to process, is based loosely on EDF. Because disk I/O is stateful, adapting the RAD model to disk scheduling requires the addition of a third layer concerned with I/O request *ordering*. Fahrrad allows request ordering by logically gathering as many requests as possible into a set with a property that the requests in the set can be executed in any order without violating any guarantees. We now describe each layer in greater detail.

Resource allocation is made via the broker and consists of two parts: translation of application requirements into a common representation—disk time utilization and period—and admission control on the basis of this representation. Most applications express their I/O performance requirements in terms of throughput and latency[3]. In order to make utilization reservations, applications specify their desired throughput and/or latency and their expected I/O behavior to the broker. Given knowledge about disk performance characteristics, the broker translates throughput and I/O behavior into utilization. When nothing is known about I/O behavior, the broker assumes worst-case request response time, resulting in no worse performance than with throughput-based schedulers.

Applications with no real-time requirements are associated with a best-effort I/O request stream that receives a minimum or remaining unreserved utilization. Latency requirements translate directly to the period reservation. If an application sends I/O requests according to its reservation, its requests will be queued no longer than one period. Since the reservation is guaranteed by the end of each period, the latency is bounded by that period.

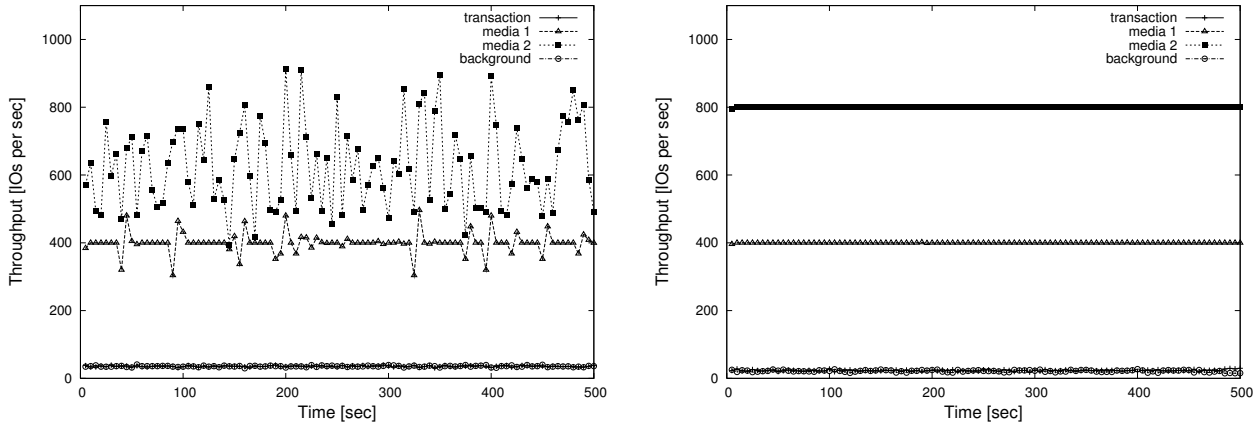
Once translated into the utilization and period, the broker decides that the reservation is feasible as long as the total sum of the utilizations on a given disk (plus a little extra) are less than or equal to 100%. The extra reservation is needed to account for blocking due to the non-preemptibility of I/O requests. In our task model, preemptible jobs are divided into non-preemptible I/O requests analogous to non-preemptible portions of CPU jobs. We have shown previously that a task set is feasible as long as we reserve enough extra time for one worst-case request in the task with the shortest period [11].

In order to guarantee the reserved budget $b = u * p$ for a given stream, the broker has to make an additional reservation. Since service times of I/O requests are not known *a priori* and I/O requests are non-preemptible with a large potential worst-case request time (WCRT), the scheduler cannot issue a request unless there is a worst-case request time left in the current period. Thus, in order to guarantee the desired budget b, the broker must actually budget $b + WCRT$ [11].

Fahrrad guarantees the reserved utilization for each request stream by correctly measuring and accounting for all I/O requests issued and seeks occurred. Fahrrad temporarily assumes that each request takes worst-case time, and allows $\lfloor b_i/WCRT \rfloor$ requests from stream i in the current period into the reordering set. Each time a request completes, the scheduler measures its execution time and updates the budget based on actual execution times. If there is enough budget left to issue one or more worst-case requests, the scheduler continues to dispatch additional requests until the reservation is met. I/O streams whose reservation has been met must wait until their next period to receive more service.

The architecture of Fahrrad is shown in Figure 3, which implements the dispatching and ordering layers of the RAD model. The architecture consists of *request stream queues*, the *Disk Scheduling Set* (DSS), the request *dispatching policy*, and the request *ordering policy*. Each request queue contains the requests from a single I/O stream and requests are ordered by their arrival times. The request dispatching

[3] An exception to this is virtual machines, which want a share of the disk performance with latency bounds.



(a) Linux (b) Fahrrad

Figure 4: Behavior of mixed workload during 500 seconds, with and without Fahrrad. Points are the average for 5-second intervals.

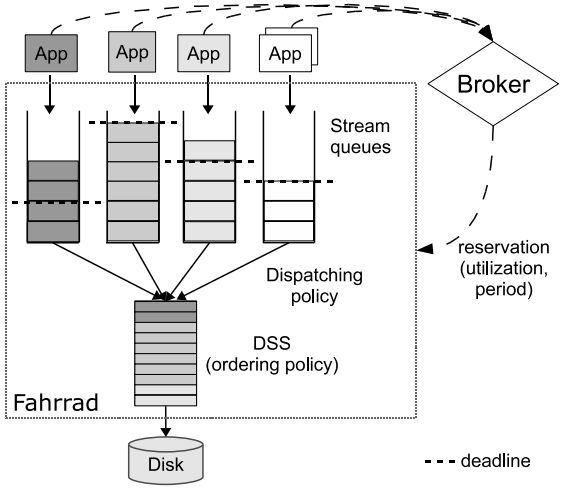


Figure 3: Fahrrad architecture.

policy takes requests from request queues and sends them to the DSS such that DSS always contains the largest set of requests that can be executed in any order without violating any utilization reservations. All requests in the DSS are assumed to take the worst-case time and the number of requests in the DSS is dictated by the earliest deadline in the system. The request dispatching policy moves all requests that have to be executed in the current period from the stream with the earliest deadline. Any remaining time is filled with requests from other stream queues. In order to minimize inter-stream seeking, the dispatching policy tries to maximize the number of requests from the same stream in the DSS (from streams with later deadlines and thus looser scheduling constraints). Since requests are assumed to take worst-case time, the scheduler always guarantees that the stream with the earliest deadline will meet its reservation regardless of the order in which the requests are sent from the DSS to the disk. If requests take less than worst-case time, the dispatcher allows more worst-case requests to the DSS if there is enough space left. The ordering policy takes requests from the DSS and sends them to the disk in an order that optimizes head movement.

While Fahrrad tries to minimize the interference between I/O streams by minimizing inter-stream seeking, some seeks between streams are unavoidable. In order to guarantee isolation between streams, we account for extra seeks caused by inter-stream seeking by reserving "overhead" utilization. We account for these seeks in the reservations of streams responsible for inter-stream seeking and bill these streams for the additional seeking. In this way, the I/O performance achieved from the reserved utilization depends only upon the workload behavior.

Figure 4 shows the performance obtained with Fahrrad. It compares a mixed-application workload running on a standard Linux system (a) and one with Fahrrad (b). The workload combines two “media” streams, a transaction processing workload with highly bursty request arrivals, and a random background stream simulating backup or rebuild. Fahrrad meets both the utilization guarantees and throughput requirements of the I/O streams and its throughput exceeds that of Linux by about 200 I/Os per second.

4 Guaranteeing storage network performance

Most general-purpose networks provide a best-effort service, striving for good overall performance while offering no guarantees. Network hardware with built-in QoS features exists, but is relatively expensive and is usually limited to static configurations that distinguish between classes of

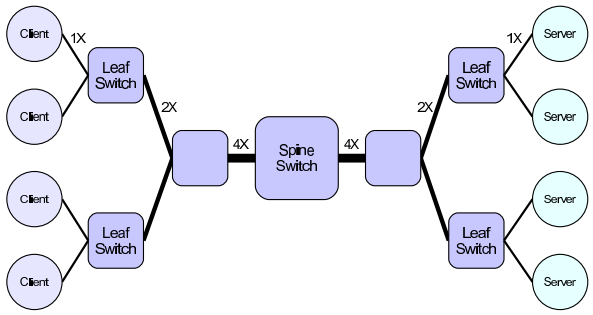


Figure 5: Fat tree network

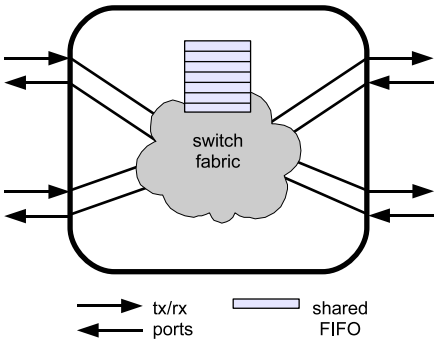


Figure 6: Simple model of a standard Ethernet switch

traffic rather than individual streams. We are interested in a more general cooperative end-to-end protocol that does not rely on specialized network hardware. Adapting the RAD model to the Network (RAD on the Network, or Radon) allows for flexible, general, and fine-grained performance guarantees using commodity network hardware.

We distinguish three major classes of networked storage: Network Attached Storage (NAS), Storage Area Networks (SAN), and distributed file systems. NAS is the most common and least expensive storage network, where one or more servers individually provide a file system interface over a commodity networks. More expensive SANs are composed of storage arrays connected with a high performance network, e.g. Fibre Channel, addressed as a local device. Distributed file systems come in two flavors, for Wide Area and Local Area Networks. Wide area systems generally serve large numbers of users, operate over a large variety of technologies, and are generally grown rather than designed. On the other hand, local area systems are usually designed to provide a high performance parallel file system for a defined clientele. We focus on local area distributed file system.

Figure 5 is our canonical storage network–a closed, full bisection bandwidth, fat tree network of standard Gigabit Ethernet switches. Each of the switches have a set of ports connected via a switch fabric and shared memory for queuing requests, as shown in Figure 6. Packets contending for the same destination port are queued. Continuous contention (congestion) may cause once isolated streams to interfere with each other. In the worst case, the queue will overflow and packets will be lost. Distributed file systems experience a particular case of congestion called incast [5, 10] where a file spread among many servers is sent in simultaneous bursts to a client, which can overflow a switch buffer with little or no warning signs.

Given the theoretical capabilities and limitations of commodity storage networks, the question, “How much of the resource is actually reservable?" has to be answered. We performed a simple characterization with a commodity Gigabit Ethernet switch supporting jumbo frames. Figure 7 shows that one to seven nuttcp UDP clients communicating with the same host achieve linear scaling for aggregate load up to 900 Mbps while experiencing packet loss of under 3% averaged over 10 seconds. Achieved load leveled off with an offered load greater than 900 Mbps while packet loss increased dramatically. The performance of a single connection appears to be limited by a host’s NIC, as a single client reaches a maximum throughput of approximately 600 Mbps over the network and above 1000 Mbps using the host’s loopback device.

These results show that well paced, short burst, fixed rate streams are able to achieve good individual performance with low packet loss while achieving 80% utilization of the network resource. With accurate congestion detection and bounded responses, we expect to be able to further increase overall utilization and decrease packet loss.

Before introducing our model for network resource management, we will briefly describe the most widely deployed end-to-end network protocol, TCP/IP. Network performance is determined by the flow control mechanism, which manages the rate at which data is injected into the network in the absence of congestion. Congestion control mechanisms, adapt the rate and timing of transmissions when congestion is detected. TCP/IP is one of most successful protocols ever developed, but its congestion control algorithms do not allow for any performance guarantees. It continuously tries to increase throughput at the sender by increasing the window (burst) size and uses packet loss as a congestion signal to throttle the sender drastically. Even for a single connection, this results in a sawtooth pattern for throughput over time and a large variance in packet delays, as a the queue continually overflows and drains.

4.1 RAD on Networks (Radon)

The RAD model was originally developed to manage a single resource with a centralized dispatcher. In the case of networks, the RAD model has to accommodate multiple dispatchers for a single resource, where the resource is a transmit port on a switch. The admission process ensures that the aggregate utilization of each switch port is not greater than one. Ideally, dispatchers should be able to cooperatively manage flow control and congestion con-

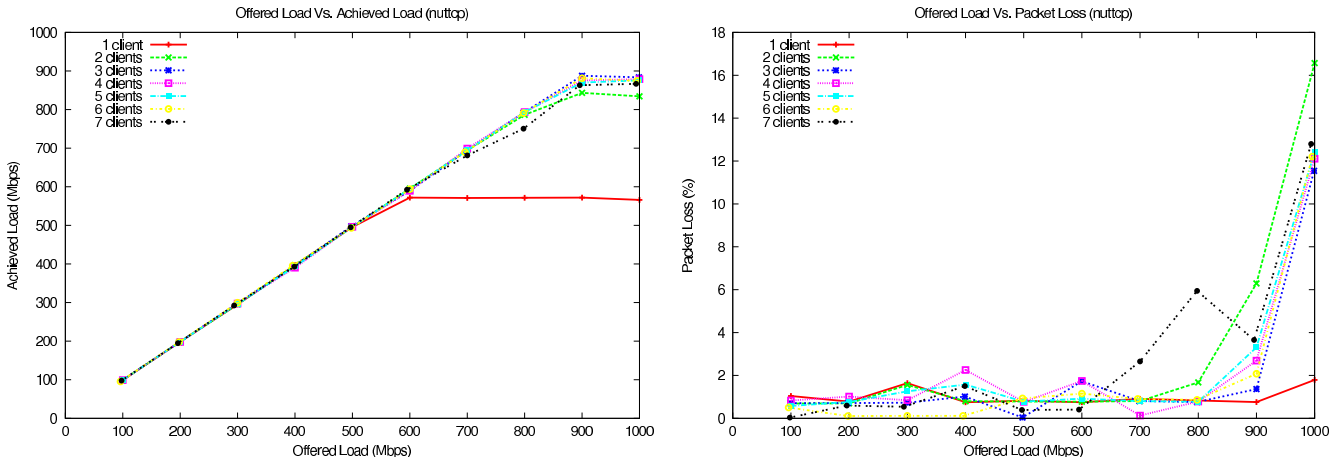


Figure 7: One to seven nuttcp UDP clients offering aggregate loads ranging from 100 to 1000 Mbps

trol based on individual resource allocations, minimizing the use of the queue on a switch. The definitions for the RAD model on networks are as follows:

Resource Allocation A task T_i's reservation (u_i, p_i), where u_i is network time utilization and p_i is the length of the period for which u_i is guaranteed.

Dispatching A task T_i has a budget $e_i = u_i \cdot p_i$, and consists of a sequence of jobs $J_{i,j}$, each having a release time $r_{i,j}$ and a deadline $d_{i,j} = r_{i,j} + p_i$.

The major challenge in guaranteeing network resources is to avoid dispatching synchronized bursts of packets while minimizing communication and synchronization overhead. Ideally this means that a host does not require external information to determine when to dispatch its requests. Scheduling algorithms like Earliest Deadline First (EDF) require all dispatchers contending for the same resource to know the release times of all jobs so that they can agree on the earliest deadline. Furthermore, the clocks of the dispatchers must be synchronized at a granularity corresponding to the smallest possible difference between deadlines. Thus, when a resource is scheduled by multiple dispatchers, a different algorithm is required.

The Least Laxity First (LLF) [8] scheduling algorithm defines the laxity of a job $l_{i,j}$ as the time remaining before the job must be scheduled in order to meet its deadline, $l_{i,j} = d_{i,j} - t - e'_i$, where t is the current time and e'_i is the budget remaining in the period. EDF schedules based on the deadline by which a job must be finished, while LLF schedules based on the deadline by which a job must be started. LLF is optimal for scheduling a single resource in the same sense that EDF is, if a feasible schedule exists, then both will find one [8]. Implementing LLF across multiple dispatchers would require just as much communication and synchronization as EDF, but it lends itself to an approximation suitable for distributed dispatchers because the measure of laxity is relative while deadlines are absolute.

Thus, we propose an approximation to LLF is called Less Laxity More (LLM). As long as no congestion is detected, streams of packets are transmitted as fast as possible up to the allocated budget. When congestion is detected, each sender will use a normalized notion of a job's laxity–percent laxity–the ratio of laxity to the total remaining time until the deadline. More formally:

$$\%laxity = \frac{l_{i,j}}{d_{i,j} - t}$$

This definition of urgency can equivalently be expressed in terms of budget since

$$\%budget = (1 - \%laxity)$$

4.2 Flow Control and Congestion Control

Before developing LLM, we simulated different flow control mechanisms based on the queuing model shown in Figure 8. One mechanism was to implicitly drive flow control by the disk performance reservation. This simulation uses a token bucket model where clients are allowed to submit a storage request to the system when tokens are available. Tokens are managed by the server, which is constantly monitoring the current performance of each client. However, in the most promising simulation, clients replenish tokens required to achieve the reserved performance themselves based on server-assigned rates and periods, while the server directly manages tokens for unused resources. This shows that flow control fits well with the RAD model.

Congestion can be detected by observing packet loss or by measuring changes in transmission delay. The response to congestion is traditionally a multiplicative decrease in window size. We suggest bounding the change in window

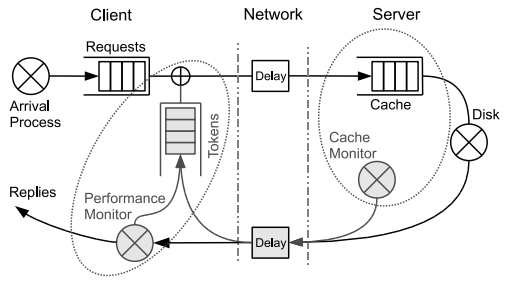


Figure 8: Queuing-theoretic model of Radon

size, making it proportional to percent laxity. Furthermore, we propose explicitly dealing with incast by postponing the dispatch time of the next window based on a model of the current queue depth of the bottleneck switch. Congestion can be detected in its early stages using the measure of relative forward delay as proposed in TCP Santa Cruz [9]. Relative forward delay allows hosts to model the queue depth of a bottleneck switch and allows congestion on the forward and reverse paths to be differentiated. This capability can become crucial on storage networks, where read and write patterns create asymmetric flows.

Flow Control Budget (in packets) $m_i = e_i/pktS$, where $pktS$ is the worst case packet service time

Congestion Control Windows adjusted in size and dispatch time

Window Target $w_{op} = (1 - \%laxity) \cdot w_{max}$

Size Change $w_{change} = \dfrac{-|w_i - w_{op}|}{2}$

Dispatch Offset $w_{offset} = \dfrac{N_{obs}}{pktS} \cdot \text{rand}$

Where w_i is the current window size and N_{obs} is the observed depth of the bottleneck switch's queue. The resulting window size is also obviously bound by the minimum window size and the remaining budget.

Even if individual hosts do not know who among them has the least laxity, they can cooperatively control congestion using the relative measure of their own laxity.

5 Buffer management for I/O guarantees

The goals for our buffer-cache in the context of performance management are two-fold. First, the buffer-cache must provide a single solution that addresses a continuous spectrum of performance guarantees, ranging from best-effort to hard real-time. The buffer-cache must guarantee capture and retention of data as long as needed, but not any longer, before forwarding it to a device. The second goal of the buffer-cache is to enhance the performance of devices, allowing performance reservations for rates and periods that the devices may not be able to provide by themselves.

Buffering serves three main functions. First, buffers are used to stage and de-stage data, decoupling components and introducing asynchronous access to the devices. The second function of the buffers are speed matching between different devices allowing fast transfers to/from slow devices. Finally, they are used to shape traffic between devices, increasing or decreasing burstiness in the workload. The ability to decouple components is driven by the amount of buffers available to the system. Speed matching is dependent upon the transformation of one component's rate to another and vice versa. Finally, the shape of the workload is influenced by the length of the period, among other factors.

Buffering can also be used for caching by placing a small, fast storage device in front of larger, slower device. Distributed storage systems use buffering on a number of system components such as storage clients, storage servers, network switches, and disks. Storage clients, for example, use caching to capture working sets in order to consolidate reads and writes. Storage servers employ caches to stage and de-stage data, capture request bursts, and prefetch data based on sequential access patterns.

In this section we will focus on buffering in storage servers but we believe that many of the principles apply to buffering in general. For the rest of the section we refer to buffering applied to storage servers as buffer caching. For now we also assume that the buffer cache is also non-volatile, as is standard in storage servers.

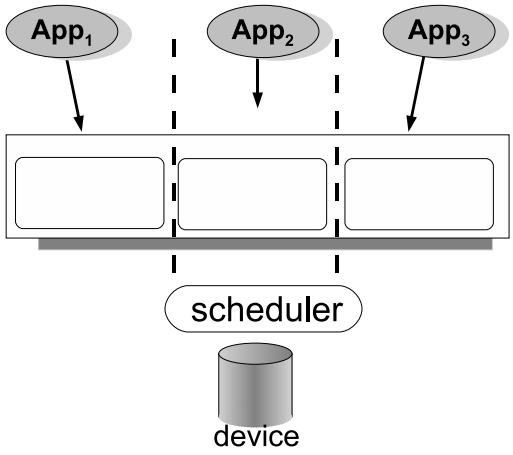


Figure 9: Cache Architecture.

Decoupling components, such as the disk and the network, requires enough buffer space to be allocated. Each application receives a dedicated partition as shown in Figure 9. The amount of buffer space assigned to each stream is a product of the guaranteed rate and period, and also influenced by workload characteristics and performance goals. Streams with performance requirements receive a minimum amount of dedicated buffer space based on worst case

request times on the device, whereas streams with soft-performance requirements might receive buffer space according to average case request times. Streams with no performance requirements are aggregated into a single reservation and may receive any uncommitted resources from streams with performance needs.

Maximizing the utilization of a resource requires worst-case buffer space allocation. We allocate space for as many requests as can be served by a device during a period based on the best case request time for the device. Thus the best-case on the device represents the worst case buffer allocation. This amount represents an upper bound on the buffer space needed within a period for read-only streams. In the case of streams involving writes, allocating extra buffers enables delayed write-back to the disk.

Embedding an application's behavior into the performance reservation (*e.g.*, sequential/random ratio, read/write ratio) allows efficient allocation of resources, for example by allocating less buffers for random streams. Efficient resource allocation can be achieved to the extent an application's workload can be characterized. If such characterization is missing, default worst-case assumptions are made.

An application's reservation is transformed into a resource reservation by means of rate and period transformations. *Rate transformation* and *period transformation* are mechanisms which allow the buffer-cache to decouple an application's reservation from the underlying device's capabilities while maintaining performance guarantees. It is possible to shape bursty workloads, using period transformations, into uniform accesses over long periods of time when that results in making better use of the device (*e.g.*, network). Similarly, by transforming short periods into long periods it is possible to introduce burstiness into the workload, reducing device utilization and overhead (for example, extra seeks on disks).

The period length of write-only streams can be elongated by means of *period transformation*, provided enough buffer space is available to hold the additional updates. It is possible to remove extra seeks in a predictable manner by transforming shorter periods into longer periods. The overhead previously imposed by short periods is then transformed into reservable utilization that could be used for admitting more request streams. Finally, since buffers have no context switch cost between stream requests, it enables reservations with very short periods. This turns some unmanageable scenarios into feasible situations that can be supported by the whole system.

Rate transformation is achieved by means of speculative reads and delayed writes. Rate transformation decouples an application's rate from a device's rate, allowing faster rates than the device can actually support. Finally, exposing faster rates to the application results in faster access times per request, allowing applications to release requests closer to the end of the period without missing deadlines.

6 Conclusion

End-to-end performance management in a complex, distributed system requires the integrated management of many different resources. The RAD integrated scheduling model provides a basis for that management and is adaptable to a variety of different resources. Based on a separation of the two basic resource management questions—how much resources to allocate to each process and when to provide them—RAD supports a wide variety of different types of processes. Our ongoing work demonstrates the applicability of the model to CPU scheduling, disk scheduling, network scheduling, and buffer cache management.

Our future work focuses on fully generalizing the RAD model. The addition of constraints—required processing characteristics such as deadlines—and heuristics—desired processing characteristics such as minimizing jitter or task migrations—give the model sufficient flexibility to describe a wide variety of existing and hypothetical schedulers with different properties. We are also extending the model to apply to additional resources and dimensions, including multi-processor and multi-disk scheduling.

Acknowledgements

This work was supported in part by National Science Foundation Award No. CCS-0621534. Additional support was provided by Los Alamos National Laboratory. The other members of our RADIO research team–Richard Golding and Theodore Wong—contributed significantly to the development of the ideas in this paper.

References

[1] S. A. Brandt, S. Banachowski, C. Lin, and T. Bisson. Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes. In *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS 2003)*, pages 396–407, Dec. 2003.

[2] J. L. Bruno, J. C. Brustoloni, E. Gabber, B. Özden, and A. Silberschatz. Disk scheduling with quality of service guarantees. In *Proceedings of the 1999 IEEE International Conference on Multimedia Computing and Systems (ICMCS '99)*, pages 400–405, 1999.

[3] W. Hawkins and T. Abdelzaher. Towards feasible region calculus: An end-to-end schedulability analysis of real-time multistage execution,.

[4] T. Kaldewey, T. Wong, R. Golding, A. Povzner, C. Maltzahn, and S. Brandt. Virtualizing disk performance. In *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2008)*, Apr. 2008.

[5] E. Krevat, V. Vasudevan, A. Phanishayee, D. G. Andersen, G. R. Ganger, G. A. Gibson, and S. Seshan. On application-level approaches to avoiding TCP throughput collapse in cluster-based storage systems. In *Proc. Petascale Data Storage Workshop at Supercomputing'07*, Nov. 2007.

[6] C. Lee, J. Lehoczky, D. Siewiorek, R. Rajkumar, and J. Hanson. A scalable solution to the multi-resource QoS problem. In *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS 1999)*, Dec. 1999.

[7] C. Lin, T. Kaldewey, A. Povzner, and S. A. Brandt. Diverse soft real-time processing in an integrated system. In *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS 2006)*, pages 369–378, Rio de Janeiro, Brazil, Dec. 2006.

[8] A. K. Mok. *Fundamental Design Problems of Distributed Systems for Hard Real-time Environment*. PhD thesis, Massachusetts Institute of Technology, May 1986.

[9] C. Parsa and J. J. Garcia-Luna-Aceves. Improving TCP congestion control over internets with heterogeneous transmission media. In *Proceedings of the 7th IEEE International Conference on Network Protocols (ICNP)*. IEEE, 1999.

[10] A. Phanishayee, E. Krevat, V. Vasudevan, D. G. Andersen, G. R. Ganger, G. A. Gibson, and S. Seshan. Measurement and analysis of tcp throughput collapse in cluster-based storage systems. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

[11] A. Povzner, T. Kaldewey, S. Brandt, R. Golding, T. M. Wong, and C. Maltzahn. Efficient guaranteed disk request scheduling with fahrrad. In *Eurosys 2008*, April 2008.

[12] L. Reuther and M. Pohlack. Rotational-position-aware real-time disk scheduling using a dynamic active subset (DAS). In *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS 2003)*, Dec. 2003.

[13] T. M. Wong, R. Golding, C. Lin, and R. Becker-Szendy. Zygaria: Storage performance as a managed resource. In *12th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS06)*, Apr. 2006.