OASIS: Self-tuning Storage for Applications

Kostas Magoutis
IBM T. J. Watson
magoutis@us.ibm.com

Prasenjit Sarkar
IBM Almaden
psarkar@almaden.ibm.com

Gauri Shah
IBM Almaden
gauris@us.ibm.com

Abstract

This paper introduces OASIS*, a self-tuning storage management architecture that allows applications and the storage environment to negotiate resource allocations without requiring human intervention. The goal of* OASIS *is to maximize the utilization of all the storage resources in the storage area network subject to fairness in the allocation of resources to applications. To achieve this goal,* OASIS *continually inspects the application's I/O behavior, and determines the application's storage requirements automatically. Subsequently,* OASIS *communicates this information from the applications to a storage manager by means of a communication protocol that isolates the complexities of applications and storage from each other. The* OASIS *framework includes algorithms to dynamically and continuously map application requirements into appropriate low-level resource allocations. A key advantage of* OASIS *over best-effort systems is the ability to fairly share storage resources between workloads of varying characteristics. What distinguishes* OASIS *from other managed systems however, is its self-tuning nature. We implemented a prototype of the* OASIS *architecture and performed experiments on a set of competing synthetic workloads derived from traces. Our results show that* OASIS *is able to detect the bandwidth and latency requirements of the competing workloads and generate a fairer allocation of storage resources than a best-effort approach. More importantly, experience with a real-life database scenario, shows that* OASIS *is able to satisfy the bandwidth requirements of competing multi-threaded workloads without any storage administrator input. In particular,* OASIS *is able to identify an under-performing workload and ensure it receives a fair share of the overall storage system resources, resulting in a performance increase by as much as a factor of five for that workload over best-effort resource allocation.*

1. Introduction

A significant challenge in today's data centers is the growing complexity of the deployed applications as well as that of the overall architecture of the data centers. This challenge is manifested by the high cost of system administration and the cost and complexity of deploying applications that are robust under variations in application workload and infrastructure changes in the data center. Furthermore, information flow in complex applications follows several paths through the various components of the data center, making analysis of the applications challenging [1].

Shared scalable storage systems are an integral part of modern data centers and have been studied extensively in the past [7]. Data centers that employ shared scalable storage systems typically do so for the ability to scale storage capacity independently of server capacity, to increase the utilization and availability of storage resources, and for the ability to share data between collaborating applications. The increasing complexity of applications and of the data center infrastructure complicates the task of allocating storage resources that match the requirements of these complex applications. There is increasing evidence that best-effort resource allocations cannot satisfy the demands of applications [4, 10, 12]. This is fundamentally intuitive: the more complex an environment is, the more challenging it is to find a set of resource allocations that maximize the satisfiability of the applications requesting them.

This paper deals specifically with the problem of generating storage resource allocations that match application requirements. One possible approach is to have system administrators describe in detail the application's storage resource requirements, typically represented as service-level objectives [2, 4, 12], and use planning tools to generate appropriate allocations. This practice however, is fraught with several difficulties. First, system administrators are usually unable to give a detailed description of storage resource requirements of their applications and rely instead on rough estimates based on intuition or on past experience with the application workload. Second, the specification of storage resource requirements of an application can only be statistical in nature and thus cannot describe the complex information flows in an application with sufficient accuracy. Third, the time and cost required to generate the storage resource requirements of an application is an issue for organizations, particularly when the exper-

tise required is hard to come by. A natural consequence of the time and cost issues is that the service-level objectives specified by the largest data center system administrators tend to be punitive rather than descriptive.

In OASIS, we adopt a fundamentally different approach in allocating storage resources to applications. Instead of requiring costly and potentially erroneous system administrator input in trying to determine application requirements on storage, we rely on automatic and continuous measurements of the application's I/O behavior to generate storage requirements. These storage requirements are periodically conveyed to the storage subsystem using a communication protocol. A key characteristic of this protocol is the specification of mutually comprehensible requirements (*e.g.*, bandwidth, latency) without any additional application-specific or storage-specific details. In this way we are able to separate applications and storage in two independent domains, the application domain and the storage domain, hiding the complexity of each domain from the other.

The goal of OASIS is to maximize the utilization of the resources on the storage area network subject to fairness in the allocation of resources to applications. To achieve this goal, OASIS incorporates algorithms to map the application's storage requirements into appropriate low-level resource allocations. These algorithms use storage resource models and fairness policies to ensure that the allocation of resources to applications meet their demands as closely as possible. What this means in practice is that OASIS can use information from applications and storage resources to control aggressive storage flows and shift resources to less aggressive applications that could potentially be resource-starved in best effort systems.

A key difference of OASIS from other managed storage systems is its self-tuning nature. In other words, the input to OASIS decision-making is not an external, human-specified goal but rather a set of internal, dynamic, and automatically collected measurements of application storage requirements. Finally, another key characteristic of OASIS is that it operates on the generic management interfaces exposed by the applications and storage subsystems.

The rest of this paper is structured as follows: In Section 2 we position OASIS in relation to past research work. In Section 3 we describe the architecture of OASIS and provide a high-level overview of the interactions between applications, the storage manager, and the storage subsystem. Following that, we describe the algorithms used to coordinate resources between these components. In Section 4 we describe our prototype implementation of the OASIS architecture and the experiments we used to evaluate OASIS. In particular, the evaluation focuses on the management of bandwidth and latency requirements of (a) competing synthetic workloads that model application I/O behavior observed in traces; and (b) two real-life application workloads

that concurrently access a database. Our results show that OASIS is able to detect the bandwidth and latency requirements of the competing synthetic workloads and to generate a fairer allocation of storage resources than a best-effort approach. Experience with the real-life database scenario shows that OASIS is able to satisfy the bandwidth requirements of the two competing workloads, a random write-dominated transactional workload and a sequential read-dominated aggregation workload, in a fair manner and without any storage administrator input. In comparison, a best-effort allocation model exhibits resource constraints when applied to the same workloads. Finally, the paper ends with conclusions and suggestions for future work.

2. Related work

Extensive research on self-tuning operating systems and databases in recent years [6, 15, 16, 17] has been motivated by the increasing complexity of these systems and the high level of expertise required to manage them. Much of this prior work has focused on automatically determining optimal settings for appropriate system controls. In OASIS we focus on self-tuning complex storage environments for allocating storage resources to applications.

The OASIS approach is based on the principle that resource allocations must be made with the goal of maximizing the utilization of all storage resources subject to fairness in the allocation to the applications. As such, the OASIS approach is similar in principle to the framework defined by Kelly *et al.* [11], which defines fairness in resource allocation as the optimization of the aggregate utility function of all principals in a system. The underlying mechanisms in such systems typically involve policies that grant resources in an additive manner and revoke them proportionately in a multiplicative manner. An example of such a discipline is the congestion control policies found in TCP. A characteristic of such systems is that they tend to favor short flows (*i.e.*, those using a small number of resources) as opposed to long flows.

Previous work on storage resource management systems focuses primarily on achieving service-level objectives (SLOs) set by a system administrator. Typical SLOs include an upper bound on the I/O latency of a flow assuming that the offered load (*e.g.*, throughput) will not exceed a certain threshold. Systems such as Facade [12], SLEDS [4], and Triage [10] aim to achieve latency SLOs by regulating the rate of application I/O streams entering the storage environment. Other systems, such as YFQ [3] and Cello [14] attempt to balance application requirements and efficiency of resource utilization. YFQ extends the Weighted Fair-Queuing (WFQ) algorithm originally applied to CPU and networks to disks in order to enforce proportional sharing of disk bandwidth. All these systems use a different notion of fairness to that outlined in the previous

paragraph. Their notion of fairness is based on *service differentiation*, *i.e.*, protecting the level of service offered to a workload from surges in the demands of other workloads.

One common aspect of all systems designed to enforce SLOs is that these SLOs must be specified by storage administrators and are potentially erroneous. The alternative of using statistical measurements to generate storage SLOs is hard and not likely to be followed in practice due to the difficulty of collecting detailed measurements of the complex information flows in an application. Even when detailed past measurements are available, it is hard to extrapolate these measurements to different storage environments or changing application workloads. At the same time, statistical specifications of application-storage interaction cannot capture all information about I/O flows. Past studies have shown that even for a simple application, there will be time intervals where the storage resource specifications significantly underestimate the application workload due to the nature of storage accesses [9].

While OASIS does not preclude (and can in fact coexist with) mechanisms designed to enforce SLOs, its focus is fundamentally on automating the interaction between applications and the storage environment while maximizing resource utilization. In OASIS, system objectives are determined automatically and dynamically based on the current demands of applications, instead of relying on human input. OASIS *learns* those requirements by monitoring the I/O demands of the application. Our decision to go with an automated approach rather than a human one is to avoid the degree of error involved in manually estimating application commodity requirements. The learning mechanism may have a time delay before it can approximate the commodity requirement for an application but it will be able to adjust dynamically to new changes much faster than manual input.

3. Architecture

In this section we introduce the OASIS architecture starting from a high-level overview of the interactions between its three main components: applications, storage manager, and storage subsystem. Next, we describe the communication protocol by means of which these interactions are realized. Finally, we discuss the algorithms that dictate the allocation of resources to applications.

3.1. Environment

In this study we assume an environment where a set of applications $P_1 \ldots P_n$ are deployed on a set of hosts and are accessing a set of storage subsystems over a storage network. To simplify analysis, we assume that each application corresponds to a single host, although multiple applications may map to the same host as shown in Figure 2.

In the general case, the architecture allows the assignment of multiple applications to multiple hosts. The allocation of storage resources to applications is managed by a storage manager as shown in the diagram of Figure 1. For the purpose of this study we assume a single storage manager, although the architecture does not preclude the presence of multiple storage managers.
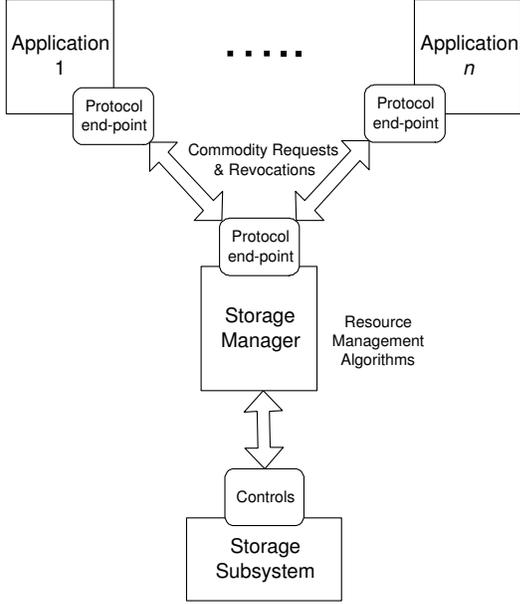
3.2. Interaction

The model of interaction between applications and the storage manager is guided by the principle that these are two complex entities in different domains of expertise, and is thus impractical to embed knowledge about the one domain into the other. We thus chose to keep the application and storage manager domains logically separate and use a communication protocol for resource negotiation between them. This model of interaction has many advantages:

- The application and storage domains can continue to evolve in terms of functionality and complexity, while the mechanics of their interaction via the communication protocol remain unchanged.

- The communication protocol has been designed to operate on the generic management interfaces exposed by applications and storage subsystems.

- The level of isolation between the application and storage domains can make it easy to extend this interaction model to other domains.

The OASIS communication protocol conveys application requirements to the storage manager in terms of *commodities*, which are virtual negotiable entities well-understood by both domains. Examples of commodities include bandwidth, latency and space. We assume that the application domain knows how to translate its requirements into commodities and that the storage manager domain knows how to translate commodities into low-level resource allocations.

Figure 1 shows the interaction between the OASIS components. Each application, as well as the storage manager, are associated with a *protocol end-point*. The *application protocol end-point* is the part of the communication protocol that is responsible for continuously monitoring the application's current demand for commodities as well as its utilization of the commodities that have been granted to it, to analyze the monitored information, and to periodically communicate requests for additional commodities to the *storage manager protocol end-point*. The design and functionality of the application protocol end-point (described in detail in Section 3.3), which operates without requiring any administrator input, is the key behind the automated operation of OASIS. The application protocol end-point is

Figure 1. High-level architecture. This figure shows the interaction between the applications, the storage manager and the storage subsystem. The applications and the storage manager interact via *commodity* requests, allocations, and revocations. The commodities are mapped into storage subsystem *resources* that are managed by the storage manager.

considered functionally distinct from the application for the remainder of the paper.

The storage manager protocol end-point is the part of the communication protocol that is responsible for receiving requests for commodities and translating them into allocation of resources in the storage environment. In contrast to a commodity, a *resource* is a manageable entity in the storage subsystem that can be controlled by the storage manager. The storage manager protocol end-point is described in detail in Section 3.4.

3.3. Application Protocol End-Point

The functionality of the application protocol end-point can be better understood by first considering the system view of Figure 2, which shows two applications P_1 and P_2 submitting I/O requests to a set of host I/O queues $Q[P_1]_1$, $Q[P_1]_2$, $Q[P_2]_1$, and $Q[P_2]_2$. In general, $Q[P_i]_j$ is the queue that allows application P_i access to a host-mounted logical disk LUN_j. This logical disk is backed by a set of storage volumes exported by one or more storage subsystems.

I/O requests between application queues and the storage subsystems flow over I/O paths referred to as *data paths*. A data path has several storage elements associated with it: the host I/O queue, the network path(s) from the queue to
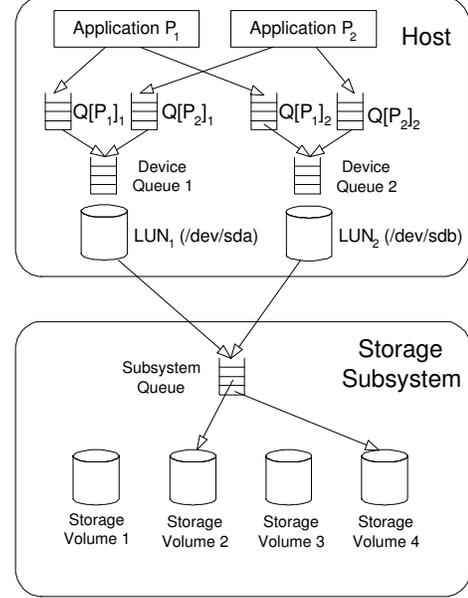
Figure 2. Sample OASIS **Topology.** In this example, two applications P_1 and P_2 access volumes LUN_1 and LUN_2 imported from a storage subsystem via the queues Device Queue 1 and Device Queue 2. The volumes LUN_1 and LUN_2 map to Storage Volume 2 and Storage Volume 4 in the storage subsystem.

the storage subsystems, and the I/O queues at the intermediate switches. We use the data path as the primary point of analysis for an application's resource requirements. Note that a single I/O stream to a virtual storage volume can comprise multiple data paths if the virtual storage volume maps to multiple physical storage volumes.

The application protocol end-point associated with application P_i monitors the queues $Q[P_i]_j$ over time intervals $(t, t + \delta t)$ and observes the following quantities: the average I/O arrival rate $D_{i,j}^t$; the average I/O completion rate $X_{i,j}^t$; the average I/O latency $R_{i,j}^t$; and the average I/O size $S_{i,j}^t$. The interval δt is determined empirically by observing the reaction time of the storage manager protocol end-point; it is also the minimum time interval between successive invocations of the communication protocol by the application end-point.

Next we focus on two commodities: bandwidth and latency.

3.3.1. Bandwidth. The application protocol end-point periodically communicates its bandwidth requirements to the storage manager protocol end-point to ensure that the latter has up-to-date information about the bandwidth needs of each queue $Q[P_i]_j$ and can manage the resources appropriately. In the initial communication, the application protocol end-point communicates the total bandwidth requirement at that time for the queue in question. If band-

width is represented as a commodity C, then the number of units $c_{i,j}^t(\delta t)$ requested from the storage manager protocol end-point at time t over time period δt can be represented as the product

$$c_{i,j}^t(\delta t) = D_{i,j}^t(\delta t) \cdot S_{i,j}^t(\delta t)$$

For each request for bandwidth, the storage manager protocol end-point performs a *bandwidth allocation* (expressed, for example, in sectors/second). The exact amount of the allocation is specified in the response message from the storage manager protocol end-point. The application protocol end-point keeps track of the total bandwidth allocation $A_{i,j}^t$ granted to it. The storage manager can *revoke* bandwidth from one or more applications when the system becomes resource-constrained, as explained in Section 3.4. In such cases, the storage manager explicitly notifies the corresponding application protocol end-points, which adjust their record of allocation to account for the decrease.

The application protocol end-point communicates with the storage manager protocol end- point whenever any of the following two conditions are true: *(i)* the average arrival rate exceeds the bandwidth allocation; or *(ii)* the observed average completion rate is lower than the bandwidth allocation even though the demand is equal to or higher than the allocation. The amount of bandwidth requested in case *(i)* is equal to the arrival rate. In case *(ii)*, the completion rate is less than both the arrival rate and allocation. This indicates a resource shortage for the commodity in question. However the application protocol end-point still communicates with the storage manager protocol end-point to indicate that the application's demands have not decreased, and the current level of allocation is still accurate. The amount of communication traffic between the application and storage manager protocol end-points can vary depending on the workload characteristics and the responsiveness of the system.

In addition to the request for the commodity, the application protocol end-point also passes along the identity of the storage volumes (physical or virtual) that the I/O requests in the input queue are addressed to. This information is needed by the storage manager protocol end-point to identify the data paths for the input queue.

3.3.2. Latency. The latency analysis at the application protocol end-point is also performed on a per input queue basis. The application protocol end-point measures the latency of every I/O request but initially categorizes the latencies per input queue $Q[P_i]_j$.

Let us assume a set of latency data points uniformly sampled over some base time period. The core of the analysis is to compare the sets of latency data points in the base time period and the current time period, and detect whether there is a latency shift. If the latency shift is considered high, the application protocol end-point communicates a latency request to the storage manager. The latency shift detection is calculated by computing the maximal cross-correlation [5] between the two sets of latency data points for a variety of latency shifts.

If the latency shift is negative, then it means that the current time period has lower latency than the base time period. Consequently, the base time period is now set to the current time period. All future comparisons are made to this new base time period.

3.4. Storage Manager Protocol End-point

The goal of the storage manager is to ensure maximum utilization of all the available resources in the system subject to fairness in the resource allocation to the applications. The criteria that we have chosen for our algorithms and evaluation is *proportional fairness* as explained below. However, the storage manager protocol end-point is not limited by this definition of fairness; it can be modified to implement any other definition of fairness too.

Each application computes its demands for certain commodities and conveys them to the storage manager. The storage manager internally converts the commodity demands to resource requirements, and tries to allocate and revoke resources as required. When resources are available in the system, the storage manager will allocate these resources as the requests arrive from different applications. However, when the system is *resource constrained*, the storage manager will revoke resources *proportionally* from other applications in order to allocate them to the application making the new demand.

Previous work in the networking domain [11] defined proportional fairness in a system, as the property that the aggregate of proportional changes in resource allocation is non-positive. For example, if the original resource allocation to application P_i is x_i and the new allocation is x_i^*, then the system is said to be proportionally fair iff:

$$\sum_{i=1}^{n} \frac{x_i^* - x_i}{x_i} \le 0$$

Our algorithms for allocation and revocation of storage resources presented in Sections 3.4.1 and 3.4.2, exhibit this property.

In order to achieve proportional fairness, we would ideally want to base resource revocation on both the current allocation and usage of a particular resource by an application. However, in the case of storage resources such as disk bandwidth, allocation does not necessarily imply a guarantee (*i.e.*, it is often an upper bound). It is thus possible that an application appears to use less than its allocated amount only because the system is resource-constrained overall. For example, consider an application P_i which is allocated A_i units of bandwidth but is using only $U_i < A_i$. In such a

case, it is difficult to know whether the need of P_i has really gone down or whether the system is resource-constrained and P_i cannot receive its allocated bandwidth. Since revoking resources from applications based on their usage may lead to starvation of certain applications, we base our revocation policy solely on the allocation of the resources.

This notion of fairness propagates up to the applications as follows: The application protocol end-point communicates[1] the commodity requirements of applications to the storage manager protocol end-point using the OASIS communication protocol as depicted in Figure 1. The storage manager maps these commodity requests to resource allocations. As long as this mapping is reasonably accurate, the fairness of resource allocations will be reflected in the fairness of commodity allocations. If the application protocol end-points continuously communicate their requirements to the storage manager, the storage manager algorithm will inherently lead to fairness in meeting the demands of the applications for resources.

Next, we describe the steps in processing a commodity request at the storage manager protocol end-point.

Identifying data paths. A request to the storage manager end-point for $c_{i,j}$ units of commodity C along the I/O queue $Q[P_i]_j$ may be relevant to multiple data paths that correspond to that input queue. The storage manager protocol end-point maps the I/O queue into its corresponding data paths by examining the list of volumes in the request for the commodity.

Resource mapping. The storage manager protocol end-point then maps the commodity request to a vector of resource requests for each identified data path. The number of resources depends on the nature of the data path. For example, a host-attached IDE drive has considerably less resources associated with it than a networked storage volume. The mapping is specific to the nature of the data path and cannot be easily generalized; examples of such mapping functions are given in our experimental setup. We define one mapping function per commodity, and in certain cases, the storage manager protocol end-point may *learn* this function over a period of time.

Resource management. The storage manager first determines the quantity of resources required. If a particular resource is available in the system, the storage manager will allocate it to meet the demands of the applications. The interesting case is when the system is resource constrained and no spare resources are available. The determination of whether there is a resource constraint depends on the nature of the resource. One type of resource is measurable and has

---

[1]Remember that the application protocol end- point is functionally distinct from the application itself and does not require any application modification or human involvement.

a fixed quantity at any given point in time; *e.g.* memory in an I/O buffer cache. The resource constraint condition is easily verifiable by examining the available quantity of that resource. Another type of resource is characterized by a total quantity that is only estimable and varies over time; an example of such a resource is disk bandwidth. The total disk bandwidth at time t depends on the nature of access to the disk at that time and is higher for sequential accesses compared to random. We explain how the storage manager manages resources by focusing on two commodities: bandwidth and latency.

3.4.1. Bandwidth. We assume that a commodity request for bandwidth maps to the disk bandwidth for a storage subsystem. The mapping between a commodity request for bandwidth and the equivalent resource request is the identity function; a request for x units for the bandwidth commodity translates to a request for x units of the disk bandwidth resource.

The total capacity of the storage subsystem with respect to disk bandwidth is not fixed. This capacity cannot be easily calculated, as the capacity varies with time due to many factors including the workload access pattern. So, in the bandwidth resource management algorithm, the storage manager does not calculate the total disk bandwidth capacity. Instead it uses revocations to adjust the total allocations to the estimated capacity of the system whenever the storage subsystem hits a resource constraint. The bandwidth management algorithm is given in Algorithm 1.

Algorithm 1: Bandwidth Management Algorithm

   Data: Bandwidth allocated to appn. i is A_i

   Data: Bandwidth usage of appn. i is U_i

   upon receiving bandwidth request B from application P

1 **if** $A_P = U_P$ **then**

      /*System not resource-constrained */

2   allocate B units of bandwidth to P

3 **else**

      /*System resource-constrained */

4     $A \leftarrow \sum_{i=1}^{n} A_i$

5     $U \leftarrow \sum_{i=1}^{n} U_i$

6   revoke $A_i - \frac{A_i}{A} \cdot U$ from all appns. except P

Upon receiving a request for bandwidth, the storage manager monitors the bandwidth usage U and the bandwidth allocation A of the requesting application. Suppose it observes that the bandwidth usage is less than the allocation. A request for more resources (mapped from the request for more commodities) even when the usage is less than the allocation indicates that the application is not re-

ceiving all the allocated resources. Thus, the storage manager concludes that the system is resource constrained and revokes resources from all the applications. The storage manager calculates a new total allocation of the bandwidth by computing the total amount of bandwidth used by all applications. The rationale behind this is that there is no spare capacity in a resource-constrained environment so the total allocation equals the total amount of bandwidth used by all applications at that given instant. In order to meet this new allocation, the storage manager now revokes resources proportionally from each application (Alg. 1, Step 6). Resources are not revoked from the requesting application because it has already made a request for additional resources and the storage manager avoids penalizing it further. On the other hand, if the usage is equal to the allocation, the storage manager concludes that the system is not resource constrained, and simply allocates the resources to the requesting application (Alg. 1, Step 2).

This algorithm is trivially proportionally fair because in a resource-constrained system, it revokes bandwidth from each application but does not allocate additional bandwidth to any application. The aggregate change is thus always non-positive.

Note that the storage manager algorithm does not distinguish between applications based on the *efficiency* of the I/O streams. This choice recognizes the fact that I/O inefficiency may be due to factors beyond the application's control (*e.g.*, data layout in the storage subsystem). Since we currently do not provide any mechanism to correct I/O inefficiency (*e.g.*, modify data layout for sequential access), we do not penalize applications with inefficient I/O streams. However, we can easily modify Algorithm 1 (Step 6) to revoke fewer resources from efficient applications thereby encouraging efficient I/O streams. On the other hand, the storage manager tends to favor applications which issue multiple I/O requests at the same time. Applications which wait for an I/O request to be completed before issuing the next I/O request are at a disadvantage because bandwidth is only allocated upon request.

3.4.2. Latency. Latency depends on several factors including the storage hierarchy (i.e. the hierarchy of caches and queues starting from the application protocol end-point to the physical disks where the data resides) as well as the hit ratios and queuing bottlenecks at each stage of the storage hierarchy. A comprehensive analysis of latency with respect to all the resources in the storage environment is very complex. However, as the cache is a primary factor that affects latency, we examine how the primary storage subsystem cache determines the latency profile of an application workload.

A challenge in performing resource management for latency is the estimation of the cache requirements of the various application workloads based on their cache usage

history. The usage of the cache resource by an application is calculated by periodically measuring the hit ratio on the cache allocated to the application[2]. In addition to the hit ratio, the storage manager also keeps track of the cache size allocated to an application. Thus, for every application, the storage manger maintains two arrays: the cache size array, and the hit ratio array for a period where measurements of cache size and hit ratio have been taken. The time period is assumed to be the most recent one as that time period has the maximal workload correlation with the present time [8].

Upon receiving a request for lower latency, the first step is to filter the cache size and hit ratio arrays so as to select the most recent hit ratio for a particular cache size. At the end of the filtering, there will be two distinct arrays: the filtered cache size array, and the filtered history list array. The storage manager uses the two arrays to construct a piecewise linear function F that can be used to calculate the cache size to obtain a particular hit ratio. The slope of F gives an estimate of the expected hit rate per unit cache size. In general, F will change when the behavior of a workload exhibits a shift, such as a change in the working set size or in the sequentiality of its accesses.

When an application protocol end-point detects a latency shift and consequently makes a request to correct the shift, the storage manager protocol end-point employs the cache usage history and the generated piecewise linear function to generate a cache size resource request to the storage subsystem. The cache management algorithm is given in Algorithm 2.

Algorithm 2: Cache Management Algorithm

Data: Cache allocated to appn. i is A_i

upon receiving request for latency from application P

convert request to hit ratio requirement

determine extra cache C required to meet hit ratio

1 **if** *cache not available* **then**

2

   /*Proportional revocation */

3    $A \leftarrow \sum_{i=1}^{n} A_i$

4   revoke $\frac{A_i}{A - A_p} \cdot C$ from all appns. except P

5 allocate cache C to P

Cache allocation can be easily measured so the storage manager can determine if the system is resource constrained or not. If the system is resource constrained, then the storage manager revokes the cache from the applications proportionally (Alg. 2, 4), and allocates it to the requesting application (Alg. 2, 5). If the system is not re-

---

[2]A previous study on the subject [8] demonstrated that periodic measurement is as effective as continuous measurement for the purpose of calculating cache usage.

source constrained, the available cache is allocated as requested (Alg. 2, 5).

Algorithm 2 satisfies proportional fairness as follows: if each application i loses $C \cdot \frac{A_i}{A - A_p}$ of the cache, then the total cache revoked is $C \cdot \sum_{i=1, i \neq p}^{n} \frac{A_i}{A - A_p} = C$. C units of cache are allocated to application P. Thus the aggregate change is $C - C = 0$.

3.4.3. Oscillations. One important aspect of the resource management algorithm is the *dampening of allocation oscillations* in a resource-constrained environment. Consider a resource-constrained system with two competing applications that make alternating demands for resources that have just been allocated to the other application. If the storage manager naïvely responds to the requests, the resource allocation will continually oscillate between the two applications. To avoid these oscillations, the storage manager only allocates a fraction of the amount of resources requested. If the allocation to demand ratio of an application is high, then the application will receive a lot less resources than if the application's allocation to demand ratio is low. This forces an inversely exponential convergence of an application's allocation to its ultimate demand and prevents oscillations.

4. Evaluation

To evaluate OASIS, we implemented a prototype involving a storage manager, a shared storage subsystem, and a number of applications. First, we describe our experimental setup (§4.1), followed by an evaluation of the OASIS resource management architecture with respect to the two commodities of interest, bandwidth (§4.2) and latency (§4.3). We use competing synthetic workloads to compare the fairness of the resource allocation performed by OASIS to that of a best-effort (*i.e.*, uncontrolled) system. The synthetic workloads were generated after evaluating sample traces. We also evaluate the responsiveness of the OASIS resource management system by the speed with which the system responds to application requests for resources. This evaluation also offers insight into the frequency of communication between the protocol end-points.

In Section 4.4 we use a real-life database application scenario to evaluate how OASIS is able to handle the resource requirements of different workloads as compared to a best-effort system. This real-life scenario allows us to examine the behavior of OASIS in situations where:

- there are a varying number of application threads

- there is a mix of sequential and random accesses

- the access patterns of an application thread vary over time

4.1. Experimental Setup

The storage manager, storage subsystem, and applications are implemented on an Intel workstation with four 2.4 GHz CPUs and 4 GB of RAM running Linux 2.6.8. There are no architectural constraints in OASIS that prevent the co-existence of (possibly multiple instances of) each component (applications, storage manager, storage subsystems) in a single system or in a fully distributed environment.

Applications. The applications used for experimentation are (a) a configurable process, hereafter referred to as a *worker*, which issues I/O requests to the storage subsystem according to a specified pattern, and (b) a database workload representative of a real-life scenario. The database workload is described in more detail in Section 4.4.

The worker process produces an I/O workload with the desired characteristics and interfaces with the application protocol end-point described in Section 3. The application protocol end-point in turn communicates with the storage manager protocol end-point to request resource allocation. The worker process takes an input several parameters that guide its behavior: *(i) Number of stages*: The specification can be broken down into an arbitrary number of stages, where each stage determines a different I/O access pattern for the worker. A number of parameters determine the pattern for each stage. *(ii) Sequentiality*: This determines the degree of sequentiality in a stage in the specification. *(iii) Block Size*: This determines the block size used in a stage in the specification. *(iv) Duration*: This determines the duration of the stage in seconds. *(v) Rate*: This determines the rate of I/Os accessing the storage subsystem in a stage of the specification.

Storage Manager. The storage manager protocol end-point and resource allocation algorithms are implemented as described in Section 3 in the context of a Linux application server.

Storage Subsystem. The Intel workstation has five 36GB 7200 RPM SCSI disks which are used as a part of the storage subsystem in this experimental setup. In addition to the base physical infrastructure, the storage subsystem implementation provides disk bandwidth allocation services to the storage manager. The allocation management code resides inside the Linux kernel and acts as an I/O scheduler for the disks in the storage subsystem. The I/O scheduler is implemented using two-level disk scheduling with multiple queues, which is similar to the Cello [14] disk scheduling framework that allocates disk bandwidth to different application classes. The upper level scheduler performs the bandwidth allocation, while the default Linux block scheduler is used as the lower level scheduler.

We also implemented a storage cache in the Intel workstation to manage latency as a commodity between the application and storage manager protocol end-points. In this setup, the latency commodity is mapped to the cache space resource as described in Section 3.4.2. The storage cache is implemented as a user-level cache [13] of disk blocks in the application's address space.

4.2. Bandwidth

In the first set of experiments, we consider the case of workloads with competing demands and focus on managing the disk bandwidth resource.

Competing Workloads. In this experiment we evaluate the OASIS resource management algorithm and compare it to a best-effort system by considering competing workloads that place high demands on the storage system, exceeding the maximum capacity.

To evaluate the competing workload scenario, we consider two identically configured workers that perform raw (unbuffered) 4KB read accesses to a logical disk device. The only difference between the workers is that one performs sequential accesses whereas the other performs random accesses. Each worker produces I/O requests at a constant rate of 60 IOPS for a four-minute interval. The total rated capacity of the logical disk device is about 100 random 4KB IOPS.

The results of the experiment are shown in Figure 3. The first thing to note is that the best-effort scenario is unable to provide an equitable allocation of resources to the two applications. The worker with the sequential workload is able to obtain significantly better bandwidth usage than the one with the random workload. In fact an analysis of the queue backlog of the two workers shows that the average I/O queue length in the worker with the random workload is 13.4 compared to a corresponding value of 2.1 for the worker with the sequential workload. By contrast, the OASIS resource management algorithm manages more equitable bandwidth usage between the two workers, and the difference in the average I/O queue lengths of the two workers is negligible ($< 0.5\%$). One thing to note is that the OASIS system requires some lead time in fully satisfying the demand from the workers, something that is explored next in the responsiveness experiments.

Examining the behavior of the OASIS resource management algorithm further in Figure 3, bandwidth usage converges to the same level for both workers with little variation[3] In fact, the allocations tend to oscillate in the

---

[3]The current version of the OASIS bandwidth management algorithm does not take the efficiency of I/O streams into account when re-distributing disk bandwidth between processes, as explained in Section 3.4.1.

initial stages. However, the storage manager protocol end-point detects that the usage of both workers is less than the allocations and infers a resource constraint. The storage manager protocol end-point consequently dampens the oscillations by revoking progressively smaller amounts of bandwidth allocations as is seen clearly in Figure 3. The dampening is triggered for as long as the utilization of the workers cannot match the corresponding demand.

Responsiveness. In this experiment we evaluate the responsiveness of our system to growing application demand for resources. First, we examine responsiveness in the case of growing demand for bandwidth. We use a single worker issuing random 4KB I/O read requests for 60 seconds performing raw (unbuffered) accesses over a logical disk device at a constant rate of 80 IOPS seconds, with an initial disk bandwidth allocation of 8 IOPS. The results of this experiment are shown in Figure 4. The results depict the bandwidth utilization of the worker in the case of a best-effort system without any resource management as well as in the case with the application protocol end-point sending disk bandwidth requests. The time period between disk bandwidth requests is varied from 1s to 5s in increments of 2s for the latter case. The utilization is measured at the storage manager protocol end-point.

A key observation is that the bandwidth allocation of the worker converges to the demand at the same rate when the time period between disk bandwidth requests is less than or equal to 3 seconds. However, the convergence is slower when the time period between bandwidth requests is 5 seconds. This indicates that all application protocol end-points must adopt very similar time periods to sample the application queues to achieve similar convergence rates to their demand. In addition, different sampling time periods may skew resources towards applications that sample their application queues more frequently in a resource-constrained environment.

While the best-effort system reaches its peak bandwidth more rapidly than with resource management, these results also suggest that a high initial disk bandwidth allocation (equal to or higher than 80 IOPS in this case) would result in behavior identical to the case of the best-effort system.

4.3. Latency

Next, we focus on managing latency for workloads with varying demands on the subsystem cache. In this case, the application protocol end-point is continuously monitoring the average latency in the application I/O stream(s). When the application protocol end-point determines that a significant latency shift has occurred it sends a request for latency reduction to the storage manager protocol end-point. The storage manager then follows the resource management algorithm for cache space as detailed in
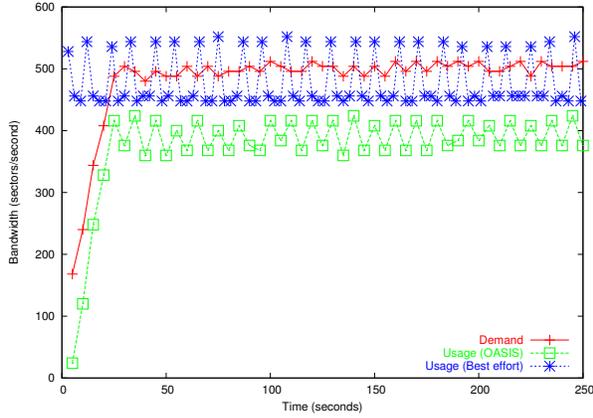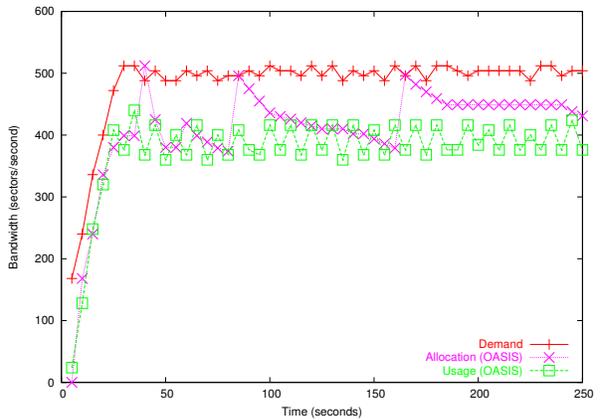
(a) Worker 1 (Sequential) B/W Usage: OASIS vs. Best-effort
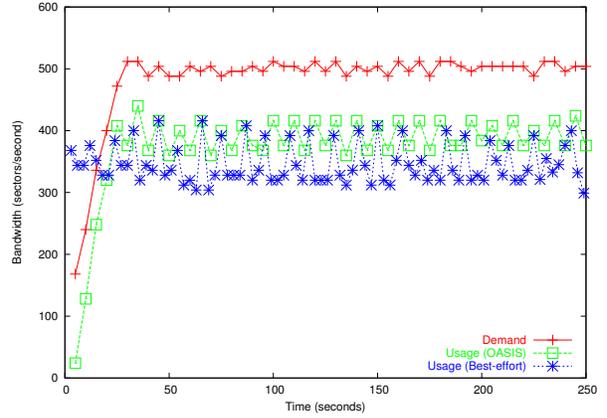
(b) Worker 2 (Random) B/W Usage: OASIS vs. Best-effort
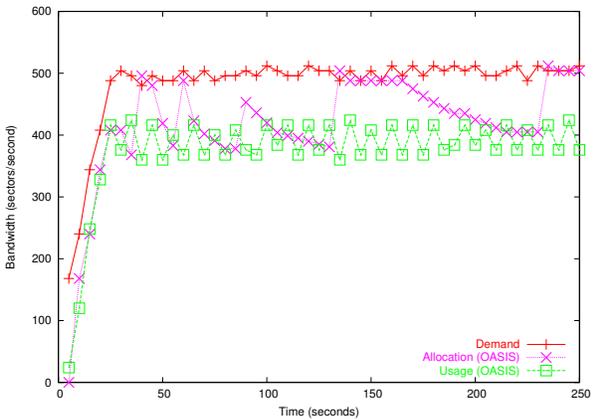
(c) Worker 1 (Sequential) B/W Allocation and Usage (OASIS)

(d) Worker 2 (Random) B/W Allocation and Usage (OASIS)

Figure 3. Competing workloads: The figure shows the I/O bandwidth demand, allocation, and usage of two competing workers with equal bandwidth demand with the OASIS resource management protocol. The worker on the left (Worker 1) uses a sequential workload while the worker on the right (Worker 2) uses a random workload. The figure also shows the usage of the two competing workers under a best-effort scenario where no allocation management is used. Bandwidth, measured at the storage manager protocol end-point in disk sectors per second, is shown on the y-axis. Time in seconds is shown on the x-axis.

Section 3.4.2.

Competing Workloads. In this experiment we evaluate the OASIS resource management algorithm by considering competing workloads whose latency demands on the storage system are close to maximum capacity. We also compare the behavior of our resource management algorithm to a best-effort system.

The experimental setup in this case involves two workers that are identically configured to perform random I/O reads using 4KB blocks within a 100MB working set stored in a 36GB disk[4]. The rate of I/O issues from each worker

---

[4]Note that each worker accesses a separate data set stored in a separate disk to avoid interference in disk accesses.

is fixed at 15 IOPS for the entire duration of the experiment (15 minutes). The I/O path to the disk passes through the block level cache whose implementation is described in Section 4.1. Each worker's initial cache allocation granted by the storage manager is 80MB in the OASIS case. In order to mask the effects of capacity (cold) misses, each worker initially warms the cache by sequentially reading the entire working set. The second worker is started 2.5 minutes after the first worker to ensure that the application protocol end-point of the latter has reached a stable point in its latency measurements.

When the experiment is run with the OASIS resource management scheme, the results show a convergence to a fair distribution of cache resources between the two work-
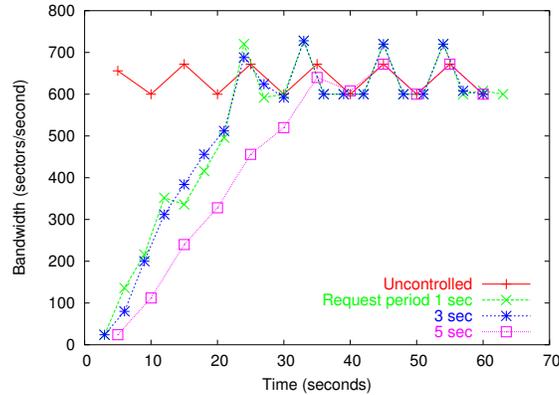
Figure 4. **Responsiveness to increasing bandwidth demands.** Disk bandwidth utilization in the absence and presence of the disk bandwidth resource management algorithm. In the latter case, the minimum period between requests for additional bandwidth varies between one, three, and five seconds. Bandwidth, measured at the storage manager protocol end-point in disk sectors per second, is shown on the y-axis. Time in seconds is shown on the x-axis.

ers. As soon as the second worker starts, the storage manager revokes 10MB from the first worker in order to add it to the 70MB of available cache space and grant the second worker its initial allocation of 80MB (Figure 5(b)). Subsequent requests from the workers' application protocol endpoints for latency reductions result in progressively smaller allocations, satisfied through equal-sized revocations from the allocation of the other worker. In this way, the storage manager ensures that the variation in allocations diminishes significantly around the stable point of 75MB. Note that (unlike the case of bandwidth) the storage manager can easily detect that the system is in a resource-constrained state since the total cache size is limited to 150MB.

The behavior is very different in the best effort scenario. In this case, the first worker process is able to put the entire working set of 100 MB into memory because there are no allocation constraints. Subsequently, the second worker process starts and but has only 60 MB of cache space to hold its working set. As a result, there is a significant difference in performance between the two worker processes – this phenomenon is not observed during the OASIS case because of a fairer cache allocation between the two worker processes. Over time in the best-effort case, cache replacement starts to slowly neutralize the difference between the allocations to the two worker processes and asymptotically narrows the performance gap.

It is important to note that while convergence to a fair allocation of resources is a property of the OASIS scheme, the best-effort shows only *apparent* convergence. In other words, the behavior of the best-effort scheme is completely dependent on the I/O rates of the worker processes – there are situations where the best-effort scheme will not converge. For example, if the I/O rates of the first and second workers are 100 and 60 IOPS, the cache allocations would remain constant for the duration of the experiment

for the best-effort case. In the same scenario, the workers in the OASIS scheme would show behavior identical to that in Figure 5. The convergence in the OASIS scheme is independent of the I/O rates of the worker processes and is decided by the commodity demand for latency.

Another issue is whether the OASIS results are sensitive to the initial allocation of 80 MB to the two competing processes. A variation of initial allocations reveals a variation of convergence times to a stable allocation of resources but the converged allocation of cache resources is identical and fair.

Responsiveness. Another important observation from the latency-time and cache space-time graphs of Figure 5 is that system responsiveness in the face of growing demand for latency reduction is in the order of tens of seconds, which is in contrast with the typical system response in the case of bandwidth (Figure 4). A key factor that accounts for this difference lies in the speed with which the application protocol end-point detects a latency shift. Since latency is directly related to hit ratio and the latter is a slowly-changing statistical measure, a latency shift cannot be detected as rapidly as a change in the bandwidth demand.

4.4. Database Workload

The goal of this experiment is to validate the previous experimental results by evaluating a case representative of a real-life application scenario involving multiple application processes contending for resources in a shared storage environment. The real-life application scenario examines situations where both the number and access patterns of application threads vary over time. Furthermore, the dominant access pattern of application threads is either random or sequential. We examined several such scenarios that are
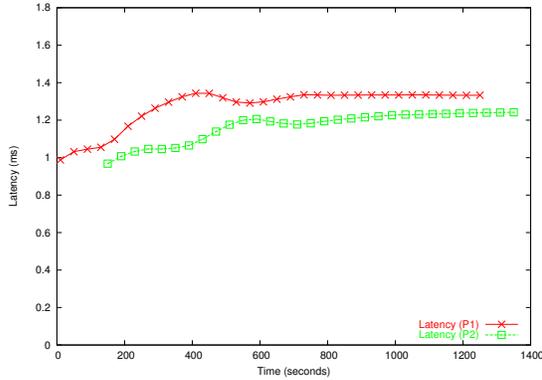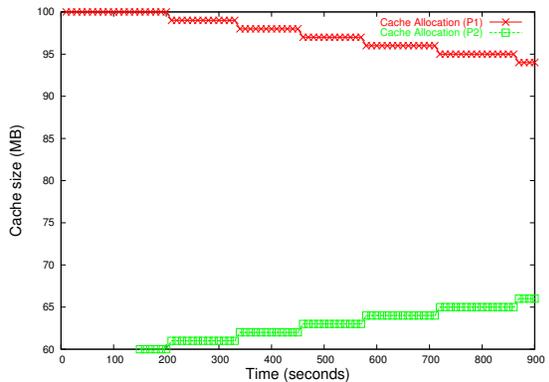
(a) Average I/O Latency in OASIS
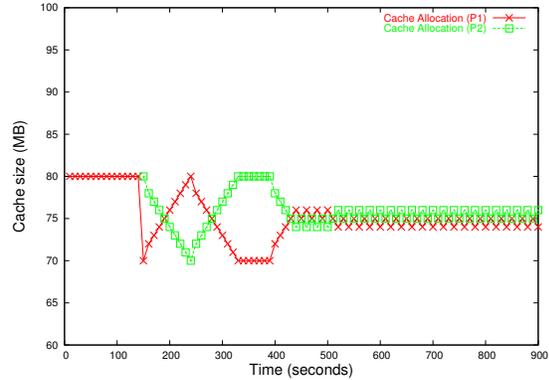
(b) Cache Allocation in OASIS
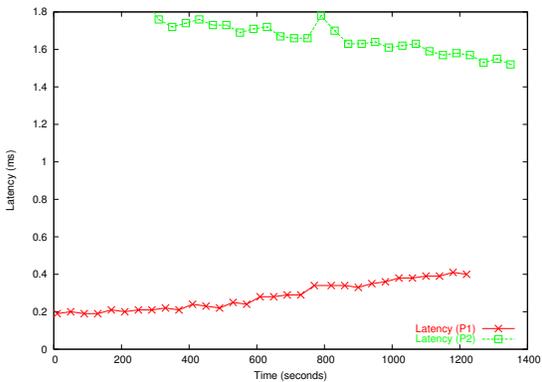
(c) Average I/O Latency in Best-effort

(d) Cache Allocation in Best-effort

Figure 5. Competing workloads: Two processes (P1, P2) compete for cache space over a duration of 15 minutes in both OASIS and best-effort modes. The graphs on the right show that OASIS is able to generate a fairer allocation of resources than a best-effort approach. The graphs on the left indicate the effect on latency.

problematic in shared storage environments and chose one that is important, yet easy to replicate. In this scenario, a transaction-processing workload runs concurrently on a set of database tables with an aggregation workload. The transaction-processing workload gathers operational data from a collection of remote agents and applies it to the database. The aggregation workload runs periodically and generates statistical reports on the contents of the data populated by the transactional workload for review by management. In typical shared storage environments, the aggregation workload is sequential in nature and tends to overwhelm the random-natured transaction processing workload. Both the aggregation and transaction workloads are highly parallelized with multiple threads of control. Further details about the workloads can be seen in Table 6.

There is minimal lock contention between the transaction and aggregation workloads. Row-level locking is used with a transaction isolation level of *read-committed* that al-

lows readers and writers to proceed in parallel without strict serialization. The choice was made by the application architects to improve performance and minimize lock contention.

In this experiment, the workloads were identical to those used in the real-life scenario, whereas the experimental setup is a close approximation to the actual environment where the real-life scenario was observed. We chose the open-source PostgreSql for hosting the database tables. The version of the database used was 7.4.3 and was *not* modified for the purpose of this experiment. The database was large with about 8 million records spread over multiple tablespaces. The aggregator workload ran concurrently with the transaction-processing workload on an overlapping set of tables. The database was hosted on the same experimental setup used in the previous experiments.

The offered load in the transaction processing workload was progressively scaled (by up to a factor of 8, in multi-

Workload	Transaction	Aggregation
Nature	Random	Sequential
Average Inter-arrival time	0.121 ms	0.049 ms
Average I/O Size	0.81 KB	3.95 KB
Percentage of Reads	6%	99.9%

Figure 6. **Workload Characteristics:** The table shows the characteristics of the transaction and aggregation workload with respect to sequentiality, inter-arrival time, I/O size and read-write ratio.

Mode	Best-effort				Oasis			
Trans Workload Scaling	Trans Perf (TPS)	Trans Queue Len	Aggr Perf (TPS)	Aggr Queue Len	Trans Perf (TPS)	Trans Queue Len	Aggr Perf (TPS)	Aggr Queue Len
1	49	7.2	26780	0	163	0	20472	0
2	82	15.8	25178	0	360	0	17290	0
4	142	19.5	22933	0	803	0	15400	0
8	235	31.8	20787	0	1029	0	13584	0

Figure 7. **Database Workload:** The table shows the performance of the transaction and aggregation workloads in both best-effort and Oasis mode with bandwidth resource management. The performance is reported in transactions per second (TPS). The transaction workload was scaled from 1 to 8 in multiples of 2 and is reported as Transaction Workload Scaling on the leftmost column. The average queue length for the transaction and aggregation workloads is shown for both modes of resource management.

ples of 2) relative to a base workload. Whereas in real life the transaction processing workload is slowly varying, for experimental analysis we decided to fix the offered load to an average rate during a run of the experiment.

The experiment was run in two modes: in the first *best-effort* mode, the environment mimics a traditional shared storage environment where no attempt is made to regulate the allocation of resources to the workloads. In the second Oasis mode, we used the bandwidth reservation scheme described in Sections 3 and 4.1. While the workloads were not modified, we implemented an application protocol end-point for both the workloads that monitored the database statistics and generated commodity requests for bandwidth based on the observed statistics. The statistics observed are not specific to PostgreSql and are also available in other databases such as MySql, Derby (Cloudscape), Oracle, DB2 and SqlServer.

Table 7 reports the performance of the transaction and aggregation workloads in both best-effort and Oasis mode with bandwidth resource management. The performance is reported in transactions per second (TPS). The transaction workload is reported as Transaction Workload Scaling on the leftmost column. The average queue length for the transaction and aggregation workloads is shown for both modes of resource management.

Several factors can be eliminated in interpreting the results in Table 7. First, the same block level scheduler is used in both modes removing any scheduling effects from the comparison of the results. Second, the cache profiles of both the transaction and aggregation workload are small compared to the overall read and write cache sizes ($< 5\%$) and can be ruled out as a factor. Finally, as discussed before, lock contention between the two workloads is minimal by design and not a factor in determining the performance difference.

The first thing to note from the results in Table 7 is that the Oasis bandwidth resource management scheme is able to remove the queue build ups in *both* the transaction and aggregation workloads, while the best-effort case shows a queue-build up in the transaction workload. A very important corollary to this result is that the Oasis resource management algorithm does not favor one workload over the other and attempts to maximize the resource utilization for both workloads. Hypothetically, it is possible to influence the priority of workloads to favor one over another but that is not within the scope of this paper.

Another observation from Table 7 is that the Oasis bandwidth resource management scheme provides as much as a factor of 5 improvement in the performance of the transaction workload. At the same time, the aggregation workload was impacted by about 30-40%. This was due to the increase in the transaction workload that was random in nature and affected the sequentiality of accesses in the aggregation workload.

In this experiment, the total disk bandwidth is not constant in all these experiments. This is due to the mix of sequential and random accesses in the aggregation and transaction workloads. This further demonstrates that the bandwidth resource management algorithm is able to adapt to variations in total allocated bandwidth.

5. Conclusions and Future Work

The paper introduces a new storage management architecture named OASIS that allows applications and the storage environment to negotiate resource allocations using a communication protocol without any human intervention. To achieve this goal, OASIS relies on several key features:

- Continually inspects the application's I/O behavior, and determines the application's storage requirements automatically.

- Communicates this information from the applications to a storage manager by means of a communication protocol that isolates the complexity of applications and storage from each other.

- Uses algorithms that dynamically and continuously map application requirements into appropriate low-level resource allocations in order to maximize the storage utilization of all resources subject to fairness in the allocation of resources to applications.

We implemented a prototype of the OASIS architecture and performed experiments on a set of competing workloads derived from traces. Our results show that OASIS is able to detect the bandwidth and latency requirements of the competing workloads and generate a fairer allocation of storage resources than a best-effort approach. Moreover, experience with a real-life database scenario, shows that OASIS is able to satisfy the bandwidth requirements of competing multi-threaded workloads without any storage administrator input. In particular, OASIS is able to identify an under-performing workload and ensure it receives a fair share of the overall storage system resources, resulting in a performance increase by as much as a factor of five for that workload over best-effort resource allocation.

A key direction of future work would be to analyze the behavior of OASIS in more complex storage environments. One challenge in such environments is that an application requirement could map to multiple resources requiring an additional decision making process in selecting the right set of resources to allocate to the application. In addition, it would be interesting to explore the complex interaction between these different resources.

References

[1] M. Aguilera, J. Mogul, J. Weiner, P. Reynolds, and A. Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *Proc. of 19th Symposium on Operating System Principles*, October 2003.

[2] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. Hippodrome: Running Cycles Around Storage Administration. In *Proceedings of USENIX Conference on File and Storage Technologies*, January 2002.

[3] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz. Disk Scheduling with Quality of Service Guarantees. In *Proc. of the IEEE International Conference on Multimedia Computing and Systems*, 1999.

[4] D. Chambliss, G. Alvarez, P. Pandey, D. Jadav, J. Xu, J. Menon, and T. Lee. Performance Virtualization for Large Scale Storage Systems. In *Proceedings of the Symposium on Reliable Distributed Systems (SRDS)*, October 2003.

[5] Chris Chatfield. *The Analysis of Time Series: An Introduction*. Chapman and Hall, 2003.

[6] D. Feitelson and M. Naaman. Self-tuning Systems. *IEEE Software*, 16(2):52–60, March 1999.

[7] G. Gibson and R. van Meter. Network Attached Storage Architecture. *Communications of the ACM*, pages 37–45, November 2000.

[8] P. Goyal, D. Jadav, D. Modha, and R. Tewari. CacheCOW: QoS for Storage System Caches. In *Proceedings of 11th International Workshop on Quality of Service, Berkeley, CA*, June 2003.

[9] W. Hsu and A. J. Smith. Characteristics of I/O Traffic in Personal Computer and Server Workloads. *IBM Systems Journal*, 42(2), 2003.

[10] M. Karlsson, C. Karamanolis, and X. Zhu. Triage: Performance Isolation and Differentiation for Storage Systems. In *Proceedings of the International Workshop on Quality of Service (IWQoS 2004), Montreal, Canada*, pages 67–74, June 2004.

[11] F. Kelly, A. Maulloo, and D. Tan. Rate Control for Communication Networks: Shadow Prices, Proportional Fairness and Stability. *Journal of the Operational Research Society*, 49:237–252, 1998.

[12] C. Lumb, A. Merchant, and G. Alvarez. Facade: Virtual Storage Devices with Performance Guarantees. In *Proceedings of USENIX Conference on File and Storage Technologies*, March 2003.

[13] K. Magoutis, S. Addetia, A. Fedorova, M. Seltzer, J. S. Chase, D. Gallatin, R. Kisley, R. Wickremesinghe, and E. Gabber. Structure and Performance of the Direct Access File System. In *Proc. of USENIX Technical Conference, Monterey, CA*, June 2002.

[14] P. J. Shenoy and H. M. Vin. Cello: a disk scheduling framework for next generation operating systems. In *Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 44–55. ACM Press, 1998.

[15] J. Strunk and G. Ganger. A human organ analogy for self-* systems. In *First Workshop on Algorithms and Architectures for Self-managed Systems*, June 2003.

[16] D. Sullivan. *Using Probabilistic Reasoning to Automate Software-tuning*. PhD thesis, Harvard University, July 2003.

[17] D. Zilio, S. Lightstone, K. Lyons, and G. Lohman. Self Managing Technology in the IBM DB2 Database. In *Proc. of the 2001 ACM International Conference on Information and Knowledge Management*, November 2001.