Dynamic Global Resource Allocation in Shared Data Centers and Clouds

Gokul Soundararajan[†‡], Saeed Ghanbari[†‡], Daniel Lupei[†], Jin Chen[⋆‡], and Cristiana Amza[†‡]
Department of Electrical and Computer Engineering[†]
Department of Computer Science[⋆]
University of Toronto
IBM Canada, CAS Research, Markham, Ontario, Canada[‡]

Abstract

We design, implement and evaluate a global resource allocator to provide end-to-end quality of service in shared data centers and Clouds. Global resource allocation involves performance modeling for proportioning several levels of storage cache, and the storage bandwidth between applications according to overall performance goals. The problem is challenging due to the interplay between different resources, e.g., changing the client cache quota affects the access pattern at the cache/disk levels below it in the storage hierarchy. We use a combination of on-line modeling and sampling to arrive at near-optimal configurations within minutes. The key idea is to incorporate access tracking and known resource dependencies e.g., due to cache replacement policies, into our performance model.

In our experimental evaluation, we use both micro-benchmarks and the industry standard benchmarks TPC-W and RUBiS. We show that our global resource allocation approach provides up to a factor of 1.4 better overall performance compared to a combination of state-of-the-art single resource controllers, at the same cost.

1 Introduction

With the emerging trend towards server consolidation in large data centers, techniques for dynamic resource allocation for performance isolation between applications become increasingly important. With server consolidation, operators multiplex several concurrent applications on each physical server of a server farm, connected to a shared network attached storage (as in Figure 1). Compared to traditional environments, where applications run in isolation on over-provisioned resources, the benefits of server consolidation are reduced costs of management, power and cooling. However, multiplexed applications are in competition for system resources, such as, CPU, memory and disk, especially during load bursts. Moreover, in this shared environment, the system is still required to meet per-application performance goals. This gives rise to a complex resource allocation and control problem.

Currently, resource allocation to applications in state-of-the-art platforms occurs through different performance optimization loops, run independently at different levels of the software stack, such as, at the database server, operating system and storage server, in the consolidated storage environment shown in Figure 1. Each local controller typically optimizes its own local goals, e.g., hit rate, disk throughput, etc., oblivious to application-level goals. This might lead to situations where local, per-controller, resource allocation optima do not lead to the global optimum; indeed local goals may conflict with each other, or with the per-application goals. Therefore, the main challenge in these modern enterprise environments is designing a *global* strategy which adopts a *holistic* view of system resources; this strategy should efficiently allocate all resources to applications, and enforce per-application quotas in order to meet overall optimization goals e.g., overall application performance or service provider revenue.

Unfortunately, the general problem of finding the globally optimum partitioning of all system resources, at all levels to a given set of applications is an NP-hard problem. Complicating the problem are interdependencies between the various resources. For example, let's assume the two tier system composed of database servers and consolidated

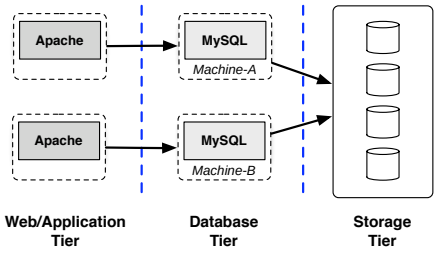


Figure 1: **Data Center Infrastructure:** We show a typical data-center architecture using consolidated storage

storage server as in Figure 1, and several applications running on each database server instance. For any given application, a particular cache quota setting in the buffer pool of the database system influences the number and type of accesses seen at the storage cache for that application. Partitioning the storage cache, in its turn, influences the access pattern seen at the disk. Hence, even deriving an off-line solution, assuming a stable set of applications, and available hardware e.g., through profiling, trial and error, etc., by the system administrator, is likely to be highly inaccurate, time consuming, or both.

Due to these problems, with a few exceptions [12, 17], previous work has eschewed dynamic resource partitioning policies, in favor of investigating mechanisms for enforcing performance isolation, under the assumption that per-application quotas, deadlines or priorities are pre-defined e.g., manually, for each given resource type. Examples of such mechanisms include CPU quota enforcement [2, 16], memory quota allocation based on priorities [3, 4], or I/O quota enforcement between workloads [9, 11, 13].

Moreover, typically, previous work investigated enforcing a given resource partitioning of a single resource, within a single software tier at a time. In our own previous work in the area of dynamic partitioning we have investigated either partitioning memory, through a simulation-based exhaustive search approach [25], or partitioning storage bandwidth, through a completely different, adaptive feedback-loop approach [24]. This paper builds and expands on our previous approach for addressing global resource allocation [26] with a more thorough investigation and explanation of the base performance models used, and a more extensive evaluation, including timing results and more complex application scenarios.

We focus on two types of resources, memory and storage bandwidth, located at two tiers, the database server and the storage server. Our goal is to dynamically proportion the client and storage caches, and the storage bandwidth between applications, according to overall performance goals.

To achieve this, we first build a simple performance model of the application and system in order to guide the search, by providing a good approximation of the overall solution.

The performance model provides a resource-to-performance mapping for each application, in all possible resource quota configurations. Our key idea is to incorporate readily available information about the application and system into the performance model. Specifically, we reuse and extend on-line models for workload characterization, i.e., the miss-ratio-curve (MRC) [36], as well as simplifications based on common assumptions about cache replacement policies. For example, we use the well known *inclusiveness* property of a LRU-based cache hierarchy in order to model a single-level cache for the purpose of searching the appropriate memory quotas to assign to applications. We further derive a disk latency model for a quanta-based disk scheduler [30] and we parametrize the model with metrics collected from the on-line system, instead of using theoretical value distributions, thus avoiding the fundamental source of inaccuracy in classic analytical models [10].

Finally, we refine the accuracy of the computed performance model through experimental sampling. We use statistical interpolation between computed and experimental sample points in order to re-approximate the per-application performance models, thus optimizing convergence towards near-optimal configurations. We experimentally show that, by using this method, convergence towards near-optimal configurations can be achieved in mere minutes, while an exhaustive exploration of the multi-dimensional search space, representing all possible partitioning configurations, would take weeks, or even months.

We implement our technique using commodity software and hardware components without any modifications to interfaces between components, and with minimal instrumentation. We use the MySQL database engine running a set of standard benchmarks, i.e., the TPC-W e-commerce benchmark, emulating an on-line bookstore like Amazon.com, the RUBiS on-line bidding benchmark, emulating an on-line auctions site, such as eBay.com, as well as several synthetic Zipf-like workloads with varying degrees of cache-ability. Our experimental testbed is a cluster of dual pro-

cessor servers connected to a commodity hardware RAID controller.

We show experiments for on-line convergence to a global partitioning solution for per-server database buffer pools, a storage cache, and the disk bandwidth shared by several applications. We compare our approach to two baseline approaches, which optimize either the memory partitioning or the disk partitioning, as well as combinations of these approaches without global coordination. We show that in non-trivial environments, our computed model effectively prunes most of the search space, even without any additional tuning through experimental sampling. At the same time, our global resource partitioning solution improves overall application performance by up to a factor of 1.4 compared to a combination of state-of-the-art single-resource controllers.

The remainder of this paper is structured as follows. Section 2 provides a background on existing techniques for server consolidation in modern data centers, highlighting the need for a global resource allocation solution. We describe our global resource partitioning algorithm with a focus on optimizing convergence time in Section 3. Section 4 describes our virtual storage prototype and sampling methodology in detail. Section 5 presents the algorithms we use for comparison, our benchmarks, and our experimental methodology, while Section 6 presents the results of our experiments on this platform. Section 7 discusses related work and Section 8 concludes the paper.

2 Background and Motivation

We investigate techniques for determining and enforcing resource quotas, for two types of resources, memory and disk, at different software tiers, i.e., database server and storage server, per-application, on the fly. In this section, we present and evaluate the state-of-the-art in single resource partitioning. Specifically, we describe current dynamic memory partitioning techniques and disk schedulers for I/O performance isolation and we explain why these techniques are insufficient in themselves. Finally, we also present motivating experiments, which show that the performance of a global resource allocator can be better than that of memory-only or disk-only resource schedulers. However, a naive exhaustive search implementation of such a global resource allocator might come at a very prohibitive cost.

2.1 Single Resource Partitioning

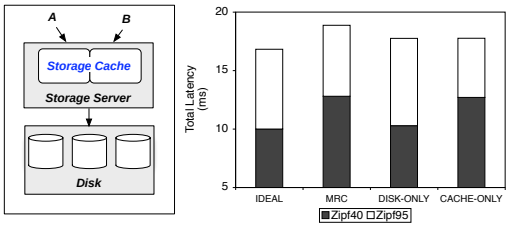
In this section, we describe previous work that either implement algorithms to allocate the storage bandwidth to several applications or allocate cache/memory to several applications.

Storage Bandwidth Partitioning: Several disk scheduling policies [30, 11, 32, 13] have been proposed in the literature. For the purpose of this paper, we focus on policies which are capable of enforcing acceptable levels of isolation between co-scheduled applications. Specifically, we have implemented and compared the performance provided by the following schedulers: (1) Quanta-based scheduling [30], (2) Start-time Fair Queuing(SFQ) [11], (3) Earliest Deadline First (EDF), (4) Lottery-based [32] and (5) Façade [13]. Our results show that the Quanta-based scheduler, where each workload is given a quantum of time during which it uses the disk in exclusive mode, offers the best isolation level when two or more competing applications are scheduled on the same disk [18]. This is because it allows the storage server to exploit the locality in I/O requests issued by an application during its assigned quantum, which in turn results in minimizing the effects of additional disk seeks due to application interference.

However, the algorithms discussed above assume that the I/O deadlines or disk bandwidth proportions are given *a priori*. In this paper, we study how to dynamically determine the bandwidth proportions at runtime. Once the bandwidth proportions are determined, we use Quanta-based scheduling to enforce the allocations, since it provides the strongest isolation guarantees.

Memory/Cache Partitioning: Dynamic memory partitioning between applications is typically performed using MRC-based algorithms. MRC-based cache partitioning leverages the miss-ratio curve (MRC) [36], to dynamically partition the cache to multiple applications, in such a way to optimize the aggregate miss rate. The MRC represents the page miss rate versus memory size curve and can be computed dynamically through Mattson's Stack Algorithm [14]. The algorithm assigns memory increments iteratively to the application with the highest predicted miss rate benefit.

For multi-level caches, in our previous work [25] we have shown the feasibility of an exhaustive search approach for memory partitioning between applications [25]. As we have shown, *cache simulation* can be used to make the exhaustive search feasible in practice. However, since the cache setting affects the I/O behavior for each application,



(a) Setup (b) Results

Figure 2: Comparison of aggregate latency motivates multi-resource controllers.

finding the optimal cache setting should ideally be coordinated with finding the optimal storage bandwidth setting. Unfortunately, our previous cache simulation approach is inappropriate for finding that optimal bandwidth setting. This is due to the fact that modern storage servers control disk arrays and use complex, proprietary scheduling firmware; this makes disk simulation, typically based on a simple elevator scheduling assumption, highly unreliable [23, 33, 29].

In the following, we show that using an existing cache allocator separately from a disk scheduler does not guarantee the global optimum for combined cache and disk partitioning.

2.2 Motivating Experiment

In this section, we present a simple motivating experiment to show the need for global multi-resource allocation. To simplify the presentation, in this experiment, we only consider two resources, namely, the storage cache and storage bandwidth. We use two synthetically generated Zipf-like workloads: Zipf95 and Zipf40, where the data access pattern is more skewed in Zipf95 compared to Zipf40. The details on our workloads are presented in Section 5.

The partitioning schemes we use are as follows: MRC and CACHE-ONLY, are an MRC-based and an exhaustive cache simulation-based search for the cache partitioning, respectively [36, 25], DISK-ONLY is an experimental exhaustive search for quanta allocations for disk bandwidth partitioning using a Quanta-based scheduler, and IDEAL is an experimental exhaustive search for the best combined cache and disk partitioning configuration. In all schemes, we evaluate the combined application latencies (by simple summation).

We choose this simple metric for the sake of fairness of comparison. In particular, state-of-the-art resource controllers e.g., MRC-based, are agnostic of Service Level Objectives (SLOs), whereas exhaustive search approaches are flexible enough to accommodate any performance goals. Hence, while all techniques contribute towards reducing application latency, an SLO-based IDEAL technique can perform arbitrarily better for specific SLO settings.

We present our comparison results in Figure 2. As we can see from the figure, even for this simple scenario, a global resource allocator (IDEAL) can improve performance compared to single resource allocators. However, convergence time of the IDEAL scheduler was 5 hours, compared to 15 minutes for MRC, 30 minutes for CACHE-ONLY and 5 minutes for DISK-ONLY. This makes an exhaustive search approach, which iterates through all possible configurations and takes experimental samples, clearly infeasible for any non-trivial environment. To further illustrate this point, let's consider a buffer pool and storage cache, as well as disk bandwidth sharing scenario. Let's say we can have 32 possible quota settings for each cache and 8 for the disk bandwidth quanta, for each application. Overall, in order to estimate an application's performance for all possible cache and storage quota configurations, we need to gather performance samples for $32 \times 32 \times 8 = 8192$ configurations. Each sample point measurement may take 16 minutes, on average, to ensure statistical significance e.g., due to cache warmup effects. Therefore, in order to compute an accurate performance model for just one application, we will need 8192×16 minutes, i.e., 2184 hours (approximately 3 months)!

These experiments and observations thus motivate us to design and implement a global resource partitioning algorithm based on an approximate system and application model, which we introduce next.

3 Dynamic Global Resource Allocation

In this section, we describe our approach to providing effective global resource partitioning in shared data centers. Our main objective is to meet an overall performance goal, e.g., minimize the overall latency when running a set of applications concurrently in shared data centers. In order to achieve this, we use the following:

1. A performance model based on minimal statistics collection in order to approximate a

near-optimal allocation of resources to applications according to our overall goal, and

2. An experimental sampling and statistical interpolation in order to refine the initial model.

In the following, we first introduce the problem statement, and an overview of our approach. Then, we introduce our performance model and its sampling-based on-line fine-tuning in detail.

3.1 Problem Statement

We study dynamic resource allocation to multiple applications in dynamic content servers with shared storage. In the most general case, let's assume that the system contains m resources and is hosting n applications. Our goal is to find the optimal configuration for partitioning the m resources among the n applications. Let's denote with $r_1, r_2, \ldots, r_n$ the data access times of the n applications hosted by the service provider. For the purposes of this paper, we assume that the goal of the service provider is to minimize the sum of all data access latencies for all applications, i.e. $\mathcal{U} = min \sum_{i=1}^{n} r_i$.

However, our approach does not depend on the particular goal we set. For example, let's assume that we know the *utility* e.g., monetary reward, that the service provider obtains for any given data access latency r_i, of a hosted application i, expressed as a utility function, $f(r_i)$. In this case, the goal would be to optimize the combined application utilities $\sum_{i=1}^{n} f(r_i)$. Alternatively, we can optimize the aggregate deviation of all applications from their contracted performance target. Whichever goal we set, we assume that our algorithm is aware of that goal and can monitor application performance in order to compute the total benefit obtained for all applications, in any resource quota configuration.

The primary challenge for end-to-end resource partitioning is to accurately determine application performance for different resource quota configurations, on-the-fly. For example, in our infrastructure, depicted in Figure 1, each application's performance is determined by the following resource quotas: (1) amount of buffer pool (ρ_c), (2) amount of storage cache (ρ_s), and (3) amount of disk bandwidth (ρ_D).

Finding a practical solution to this problem is difficult, because the optimal resource allocation depends on many factors, including the (dynamic) access patterns of the applications, and how the inner mechanisms of each system component e.g., cache replacement policies, affect interdependencies between system resources.

3.2 Overview of Approach

Our technique determines per-application resource quotas in the database and storage caches, on the fly, in a transparent manner, with minimal changes to the DBMS, and no changes to existing interfaces between components. Towards this objective, we use an online performance estimation algorithm to dynamically determine the mapping between any given resource configuration setting and the corresponding application latency. While designing and implementing a performance model for guiding the resource partitioning search is non-trivial, our key insight is to design a model with sufficient expressiveness to incorporate i) tracking of dynamic access patterns, and ii) sufficiently generic assumptions about the inner mechanisms of the system components and the system as a whole.

For this purpose we collect a trace of I/O accesses at the DBMS buffer pool level and we use periodic sampling of the average disk latency for each application in a baseline configuration, where the application is given all the disk bandwidth. We feed the access trace and baseline disk latency for each application into a performance model, which computes the latency estimates for that application for all possible resource configurations. We thus obtain a set of resource-to-performance mapping functions, i.e., performance models, one for each application. Next, we enhance the accuracy of each performance model through experimental sampling. We use statistical regression to reapproximate the performance model by interpolating between the pre-computed and experimentally gathered sample points.

We then use the corresponding per-application performance models to determine the *near*-optimal allocation of resources to applications according to our overall goal. Specifically, we leverage the derived performance model of each application, and use *hill climbing* [20] to converge towards a partitioning setting that minimizes the combined application latencies. In the following subsection, we describe our model that estimates the performance of an application using multi-level caches and a shared disk.

3.3 Per-Application Performance Model

We use two key insights about the inner workings of the system, as explained next, to derive a close performance approximation, while at the same time reducing the complexity of the model as much as possible.

3.3.1 Key Assumptions

The key assumptions we use about the system are i) that the cache replacement policy used in the cache hierarchy is known to be either one of LRU or DEMOTE [35], and ii) that the server is a closed-loop system i.e., it is interactive and the number of users is constant during periods of stable load. Both of these assumptions match our target system well, leading to a performance model with sufficient accuracy to find a near-optimal solution, as we will show in Section 6.

3.3.2 Modeling the Cache Hierarchy

Our key idea is to replace the search space of a cache hierarchy with the simpler search space of a single level of cache, in order to obtain a close performance estimation, at much higher speed. Towards this goal, we use the experimental observation that the network latency is negligible compared to the disk access latency. This affords us the simplification that the contribution of a cache hit in any level of cache to overall application performance is roughly the same. Therefore, we approximate any cache hierarchy with the model of a single cache. We derive different models according to the cache replacement policy used in the cache hierarchy. We use the two most commonly deployed or proposed cache replacement policies, LRU and DEMOTE [35], in deriving our respective cache performance models.

In a cache hierarchy using the LRU replacement policy at all levels, if an application is given a certain cache quota q_i at a level of cache i, any cache quotas q_j given at any lower level of cache j, with $q_j < q_i$ will be mostly wasteful. This is because of the cache hierarchy *inclusiveness* property, where any cache miss from q_i will result in bringing the needed block into all lower levels of the cache hierarchy, before providing the requested block to cache i. In contrast, in a cache hierarchy using DEMOTE, when a block is fetched from disk, it is not kept in any lower cache levels. Lower levels of caches only cache blocks only when the block is evicted from a higher cache level. Through this, cache *exclusiveness* is maintained leading the application to benefit from the combined quotas.

Hence, we make the following approximations in our performance model:

In an LRU cache hierarchy, only the maximum size quota given at any level of cache matters; therefore, we approximate the MRC of a two level cache, consisting of a buffer pool (c) and a storage cache (s) by the following formula:

$$\mathcal{M}(\rho_c, \rho_s) \quad \approx \quad \mathcal{M}(max[\rho_c, \rho_s]) \tag{1}$$

In a DEMOTE cache hierarchy, the combined cache quotas given to the application at all levels of cache has the same effect as giving the total quota in a single level of cache. Therefore, we use the following formula to approximate the MRC of a two-level cache:

$$\mathcal{M}(\rho_c, \rho_s) \quad \approx \quad \mathcal{M}(\rho_c + \rho_s) \tag{2}$$

3.3.3 Modeling the Disk Latency

For modeling the disk latency, we observe that the typical server system is an *interactive*, closed-loop system. This means that, even if incoming load may vary over time, at any given point in time, the rate of serviced requests is roughly equal to the incoming request rate. According to the *interactive response time law* [10]:

$$l_d = \frac{N}{X} - z \tag{3}$$

where l_d is the response time of the storage server, including both I/O request scheduling and the disk access latency, N is the number of application threads, X is the throughput, and z is the think time of each application thread issuing requests to the disk.

We then use this formula to derive the average disk access latency for each application, when given a certain quanta of the disk bandwidth. We assume that think time per thread is negligible compared to request processing time, i.e., we assume that I/O requests are arriving relatively frequently, and disk access time is significant. If this is not the case, the I/O component of a workload is likely not going to impact overall application performance. However, if necessary, more precision can be easily afforded e.g., by a *context tracking* approach, which allows the storage server

to distinguish requests from different application threads [27], hence infer the average think time.

We further observe that the throughput of an application varies proportionally to the disk quanta that the application is given. Since disk saturation is unlikely in interactive environments with a limited number of I/O threads, this is very intuitive, but also verified through extensive validation experiments using a quanta-based scheduler and a variety of workloads.

Through a simple derivation, we arrive at the following formula:

$$l_d = \frac{l_{d(\rho_d=1)}}{\rho_d} \quad (4)$$

where $l_{d(\rho_d=1)}$ is the *baseline disk latency* for an application, when the whole bandwidth is allocated to that application. This formula is intuitive. For example, if the entire disk was given to the application, i.e., $\rho_d = 1$, then the storage access latency is equal to the underlying disk access latency. On the other hand, if the application is given a small quanta, i.e, $\rho_d \approx 0$, then the storage access latency is very high (approaches ∞).

3.3.4 Overall Performance Model

We use the single-level MRC-based model of the cache hierarchy together with the disk latency model to derive the overall application performance as follows.

$$L_{avg}(\rho_c, \rho_s, \rho_d) = l_c\mathcal{H}(\rho_c, \rho_s) + l_d\mathcal{M}(\rho_c, \rho_s) \quad (5)$$

where L_{avg} is the average application latency given quotas ρ_c, ρ_s and ρ_d of the buffer pool cache, storage cache and disk bandwidth, respectively, $\mathcal{H}(\rho_c, \rho_s)$ and $\mathcal{M}(\rho_c, \rho_s)$ are the hit/miss rates, respectively for the cache hierarchy, l_c is the cache hit latency (assumed to be similar regardless of cache level) and l_d is the average latency to fetch data from the storage server. If we plug in the cache and disk latency formulas derived previously, for a (commonly used) LRU cache hierarchy, the formula translates into:

$$L_{avg}(\rho_c, \rho_s, \rho_d) = l_c\mathcal{H}_c(\rho_c) + l_d\mathcal{M}_c(max[\rho_c, \rho_s]) \quad (6)$$

where we use the hit/miss rates experimentally derived through access trace collection and computing the MRC at the buffer pool level (denoted by c in the formula) and l_d is the storage access latency given by Equation 4. As discussed, this is a close approximation of a close-loop system using an LRU-based two-tier cache.

Finally, all we need to do in order to have a complete resource-to-performance model is to vary the quota allocations for the two caches and the disk bandwidth for the application, to all possible combinations *in the model*, and *compute* all corresponding latencies by the above formula. This can be performed very fast, *on-line*, since it involves no measurements from the system during periods of stable load. The model needs periodic recalibration, by taking a new sampling of the baseline disk latency for each application in order to account for load variations. However, this assumes a short (less than a minute) actuation and measurement for each application and the recomputation of the recalibrated model within milliseconds afterwards. Similarly, if the application pattern changes, a new application trace needs to be collected and the new MRC recomputed.

3.4 Model Fine-tuning

In order to fine-tune our performance model at run time, hence adaptively correct any inaccuracies, we use more expensive sampling-based approaches which we describe next.

3.4.1 Sampling and Interpolation

Our performance model provides us with a set of pre-computed sample points based on minimal statistics collection (for the MRC and baseline disk bandwidth). Some of these points may be, however, inaccurate.

We collect experimental samples of application latency in various resource partitioning configurations, and use statistical regression i.e., *support vector machine regression* (SVR) [8], to reapproximate the resource-to-performance mapping function without sampling the search space exhaustively. SVR allows us to estimate the performance for configuration settings we haven't actuated, through interpolation between a given set of sample points.

We iteratively collect a set of k randomly selected sample points. Each sample represents the average application latency *measured* in a given configuration. We replace the respective points in our performance model with the new set of experimentally collected samples. Using *all* sample points, consisting of both computed and ex-

perimentally collected samples, we retrain the regression model. We also cross-validate the model by training the regression model on a sub-set of all samples and comparing with the regression function obtained using the remaining samples. If during cross-validation, we determine that the regression-based performance model is stable [8], then we conclude that we do not need to collect any more samples, and we have achieved a highly accurate performance model for the respective application. Otherwise, we iterate through the above process until convergence is achieved.

Hence, the more accurate the initial performance model, the less sampling is needed to fine-tune the model. Conversely, if the initial performance model is highly inaccurate e.g., because completely different cache replacement policies than built into the performance model are deployed, then exhaustive sampling will be eventually performed to replace all initial model-generated sample points.

3.5 Finding the Optimal Configuration

Based on the per-application performance models derived as above, we find the resource partitioning setting which gives the global optimum i.e., lowest combined latency in our case, by using *hill climbing* with random-restarts [20]. The *hill climbing* algorithm is an iterative search algorithm that moves towards the direction of increasing value of combined utility value for all valid configurations at each iteration. To avoid reaching a local maximum, we conduct several searches from several points chosen randomly until each search reaches a maximum. We use the best result obtained from all searches.

4 Prototype Implementation

We extend an existing infrastructure supporting exhaustive search for finding the optimal solution for memory-only resource partitioning between applications [27, 25], to implement our new performance model introduced in Section 3, as well as for supporting sampling and resource partitioning of *multiple resources* at several levels.

Our infrastructure consists of a virtual storage system prototype designed to run on commodity hardware. It supports data accesses to multiple virtual volumes for any storage client, such as, database servers and file systems. It uses the Network Block Device (NBD) driver packaged with Linux to read and write logical blocks from the virtual storage system, as shown in Figure 3. NBD is a standard storage access protocol similar to iSCSI, supported by Linux. It provides a method to communicate with a storage server over the network. We modified existing *client* and *server* NBD protocol processing modules for the storage client and server, respectively, in order to interpose our storage cache and disk controller modules on the I/O communication path, as shown in the figure.

In addition, we provide interfaces for creating/destroying new virtual volumes and setting resource quantas per virtual volume. Our infrastructure supports a global resource controller in charge of partitioning multiple levels of storage cache hierarchy and the storage bandwidth. The controller determines per-application resource quotas on the fly, in a transparent manner, with minimal changes to the DBMS i.e., to collect access traces at the level of the buffer pool, and to replace its cache module with our own, and no changes to existing interfaces between components.

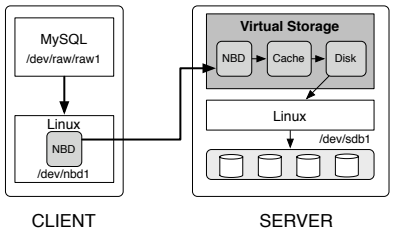


Figure 3: **Virtual Storage Architecture**: We show one client connected to a storage server using NBD. MySQL mounts the NBD device (`/dev/nbd1`) on `/dev/raw/raw1`. The storage server virtualizes the physical partition `/dev/sdb`.

4.1 Sampling Methodology

For each hosted application, and given configuration, in order to collect a sample point, we record the average and standard deviation of the data access latency, for the corresponding application in that configuration. For each sample point where we change the cache configuration, we wait for cache warmup, until the application miss rate is stable (which takes approximately 15 minutes on average in our experiments). Once the cache is stable, we monitor and record the application latency several times in order to reduce the noise in measurement. Once measured, sample points for an application

can also be stored as an *application surface* on disk and later retrieved.

4.1.1 Efficient Sampling for Exhaustive Search

For the purpose of exhaustive sampling i.e., for the implementation of the IDEAL algorithm used in the experiments presented in Section 2, the controller iteratively sets the desired resource quotas and measures the application latency during each sampling period. We use the following rules of thumb in order to speed up the exhaustive sampling process:

Cost-aware Iteration: We sort resources in descending order of re-partitioning cost i.e., cache repartitioning has higher re-partitioning sampling cost compared to the disk due to the need to wait for cache warm-up in each new configuration. Therefore, we go through all cache partitioning possibilities as the outermost loop of our iterative exhaustive search; for each cache setting we go through all possible disk bandwidth settings in an inner loop, thus making fewer changes to stateful resources overall. This optimization decreased the time of the experiment mentioned above from 52 hours to 10.5 hours.

Order Reversal: The time to acquire a sample can be further reduced by iterating from larger cache quotas to smaller cache quotas i.e., from 1024MB to 32MB in a 1024MB cache. In this case, the cache warmup of the largest cache quota will be amortized over the sampling for all cache quotas for the application. This additional optimization decreased the time of the experiment further, from 10.5 hours to a 5 hours convergence time, which is the convergence time we reported in Section 2.

5 Evaluation

In this section, we describe several resource partitioning algorithms we use in our evaluation. In addition, we describe the benchmarks and methodology we use.

5.1 Algorithms used in Experiments

We compare our GLOBAL resource partitioning scheme, where we combine performance estimation and experimental sampling, with the following resource partitioning schemes.

- IDEAL: Finds the configuration with best overall latency by exhaustive search for all possible cache and disk partitioning configurations. In order to speed up the search (which would experimentally take weeks), we have experimentally verified that MRC-based cache miss rates and cache simulation are highly accurate for our workloads. We then compute all possible cache miss ratios for all possible cache quota configurations based on access traces, thus deriving $\mathcal{M}(\rho_c, \rho_a)$, and use the cache simulator to estimate the cache latency in each actuated cache configuration. We then perform manually selected experimental measurements, hence avoiding the need to wait for cache warmup in all configurations.

- MRC: Uses MRC to perform cache partitioning *independently* at the buffer pool and the storage cache, based on access traces seen at that level. The disk bandwidth is equally divided among all applications.

- MRC+DISK: Uses the cache configurations produced by the MRC scheme and then explores all the possible configurations for partitioning the disk bandwidth.

- DISK-ONLY: Assigns equal portions of the cache to all applications at each level and explores all the possible configurations at the disk level.

- CACHE-ONLY: Explores all the possible settings for cache sizes at both cache levels, while dividing the disk bandwidth equally between all applications.

Out of these schemes, IDEAL and CACHE-ONLY are the only two schemes that require derivation of multi-level, multi-configuration MRC curves, i.e., $\mathcal{M}(\rho_c, \rho_a)$. Deriving the multi-level, multi-configuration MRC curves takes approximately 36 hours for one application.

5.2 Benchmarks

We use four workloads: two workloads based on the Zipf distribution, and two industry-standard benchmarks, TPC-W and RUBiS, to evaluate our end-to-end cache management algorithm.

Zipf95/Zipf40: We generate two workloads using a Zipf-like distribution [35]. In these workloads, few blocks are frequently accessed while others are accessed much less often. The probability of the i^{th} to be accessed is proportional to

$1/i^{\alpha}$, where α is a user-defined parameter. We generate two workloads using the Zipf distributions: (1) *Zipf95* where we set $\alpha = 0.95$ and (2) *Zipf40* where we set $\alpha = 0.4$. Since the accesses are more *skewed* when $\alpha \to 1$, the *Zipf95* workload uses a smaller working set.

TPC-W: The TPC-W benchmark from the Transaction Processing Council [1] is a transactional web benchmark designed for evaluating e-commerce systems. Several web interactions are used to simulate the activity of a retail store. The database size is determined by the number of items in the inventory and the size of the customer population. We use 100K items and 2.8 million customers which results in a database of about 4 GB. We use the *shopping* workload that consists of 20% writes. To fully stress our architecture, we run 10 TPC-W instances in parallel creating a database of 40 GB.

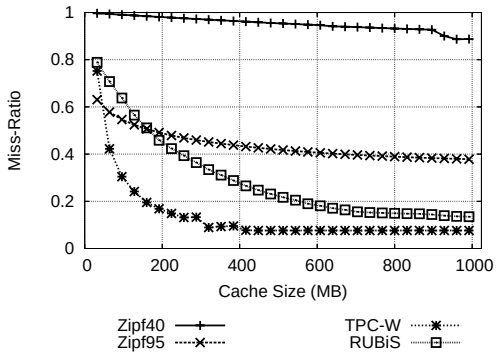
RUBiS: We use the RUBiS Auction Benchmark to simulate a bidding workload similar to e-Bay. The benchmark implements the core functionality of an auction site: selling, browsing, and bidding. We do not implement complementary services like instant messaging, or newsgroups. We distinguish between three kinds of user sessions: visitor, buyer, and seller. For a visitor session, users need not register but are only allowed to browse. Buyer and seller sessions require registration. In addition to the functionality provided during the visitor sessions, during a buyer session, users can bid on items and consult a summary of their current bid, rating, and comments left by other users. We are using the default RUBiS bidding workload containing 15% writes, considered the most representative of an auction site workload according to an earlier study of e-Bay workloads [21]. We create a scaled workload, we run 10 RUBiS instances in parallel.

6 Results

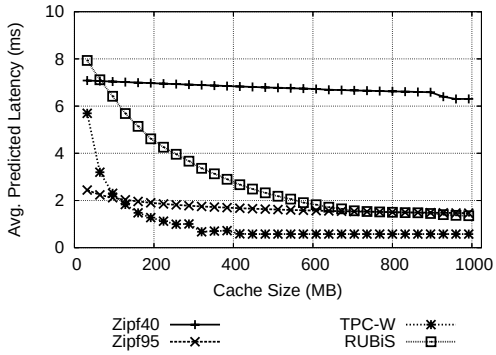
In this section, we evaluate the performance of our resource allocation algorithms by comparing their performance and speed of convergence. We also experimentally illustrate the *accuracy* of our performance model.

6.1 Preliminary Results

Figure 4-(a) shows the MRC curves at the buffer pool for all applications. We can see that TPC-W and RUBiS are more cacheable (have steeper MRC curves) than Zipf40 and Zipf95. The cut-off point for caching benefits is approximately 100 MB for



(a) MRC Curves



(b) Latency Curves

Figure 4: **MRC/Latency:** We show (a) the MRC curves for all four workloads and (b) the average predicted latencies.

Zipf95, 360MB for TPC-W, 700MB for RUBiS, and 32MB (minimum) for Zipf40. From the average latency as a function of total cache size, as predicted by our performance model for our benchmarks, shown in Figure 4-(b), we see that Zipf40, while benefitting little from cache allocations, has a high average latency, hence would benefit from larger disk quanta allocations. Overall, TPC-W has the lowest average latency, hence lowest disk bandwidth requirements once its basic cache needs are met.

6.2 Performance Results

We evaluate our model using our four applications on a setup consisting of two client (database server) machines with two applications sharing each of them, and one storage server shared among all four. We use a 512MB buffer pool on each client and a 1GB storage cache. All caches use LRU. Cache quotas are allocated in 32MB increments, with a minimum of 32MB. The disk bandwidth is allo-

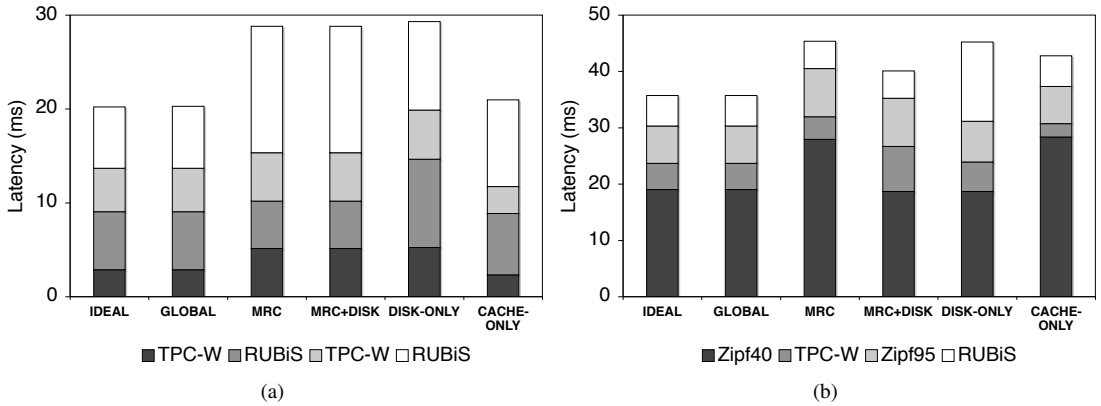


Figure 5: **Latency:** We show the aggregate latency for two workload combinations: (a) TPC-W/RUBiS + TPC-W/RUBiS and (b) Zipf40/TPC-W + Zipf95/RUBiS.

Scheme	**Buffer Pool** (MB)				**Storage Cache** (MB)				**Disk Quanta** (%)				**Time**
	TPCW	RUBiS	TPCW	RUBiS	TPCW	RUBiS	TPCW	RUBiS	TPCW	RUBiS	TPCW	RUBiS	(mins)
IDEAL	64	480	448	64	320	32	32	640	25	38	12	25	days
GLOBAL	32	480	480	32	320	32	32	640	25	38	12	25	16
MRC	192	320	192	320	32	928	32	32	25	25	25	25	16
MRC+DISK	192	320	192	320	32	928	32	32	25	25	25	25	21
DISK-ONLY	256	256	256	256	256	256	256	256	12	38	12	38	5
CACHE-ONLY	448	64	32	480	32	640	320	32	25	25	25	25	2160

(a) TPC-W/RUBiS + TPC-W/RUBiS

Scheme	**Buffer Pool** (MB)				**Storage Cache** (MB)				**Disk Quanta** (%)				**Time**
	Zipf40	TPCW	Zipf95	RUBiS	Zipf40	TPCW	Zipf95	RUBiS	Zipf40	TPCW	Zipf95	RUBiS	(mins)
IDEAL	64	448	480	32	32	32	32	928	38	12	25	25	days
GLOBAL	64	448	480	32	32	32	32	928	38	12	25	25	16
MRC	256	256	96	416	32	32	32	928	25	25	25	25	16
MRC+DISK	256	256	96	416	32	32	32	928	38	12	25	25	21
DISK-ONLY	256	256	256	256	256	256	256	256	38	12	25	25	5
CACHE-ONLY	96	416	480	32	32	32	928	32	25	25	25	25	2160

(b) Zipf40/TPC-W + Zipf95/RUBiS

Table 1: **Detailed Results:** We show detailed resource partitioning and convergence results for our two combinations (a) TPC-W/RUBiS + TPC-W/RUBiS and (b) Zipf40/TPC-W + Zipf95/RUBiS.

cated in 12.5% slices. In all experiments, the results shown for GLOBAL are based on model-based performance estimation, with sampling and interpolation turned off. We show more detailed results for our combined estimation and sampling model in section 6.4.

We run two experiments, with different application combinations, shown in Figure 5, and we compare the performance of all algorithms in terms of aggregate latency. We also present the resource partitioning configurations at convergence, and convergence times for each algorithm in Table 1. Overall, we see from both Figure 5 and Table 1, that our GLOBAL resource allocator arrives dynamically at a resource partitioning solution close to that of IDEAL, and offers the same performance as IDEAL. This performance is a factor of 1.42 and 1.12 better than the MRC+DISK scheme, which combines single resource allocators, respectively, for the two experiments. At the same time, its convergence time is similar to that of single-resource allocators (on the order of minutes), while the exhaustive search in IDEAL takes days (or in fact weeks, if we didn't guide the

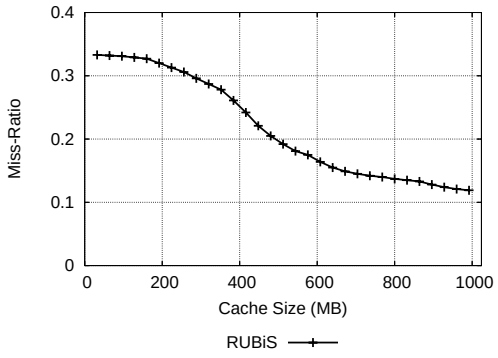


Figure 6: **MRC:** We show the MRC for RUBiS at the storage cache when the buffer pool quota is 320MB.

search manually). Finally, we observe that both the IDEAL and GLOBAL schemes allocate cache space to each application only at one level of the hierarchy, while usually the minimum space at the other. This is due to the inclusiveness property of LRU caches, as discussed before. We next show more insights into the performance of our allocator through a more in-depth analysis of these two experiments.

6.3 Detailed Analysis

For our first experiment, presented in the previous section, we run a TPC-W workload and a RUBiS workload on each client machine.

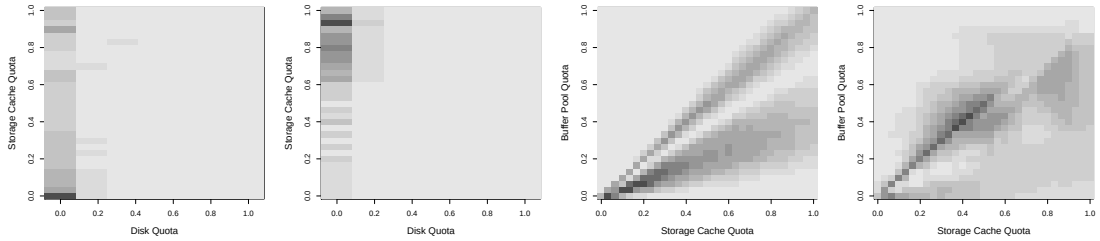
First, we can see that the configurations achieved in GLOBAL (and IDEAL) are non-trivial. For example, one TPC-W workload copes with a relatively small cache allocation (32MB,320MB) compensated by a higher disk bandwidth quota (25%), while the opposite holds for the TPC-W instance running at the second client, which gets a larger cache allocation (480MB,32MB), but a smaller disk quota (12%). Similar decisions are taken for the RUBiS workloads as well.

Second, we observe that the MRC-based cache allocation suffers penalties from pursuing local optimization goals in each level of cache. Achieving the optimal buffer pool hit rates for TPC-W and RUBiS (192MB/320MB) in both client caches is not necessarily conducive to the global optimum. Since the TPC-W workloads are easily satisfied with just the buffer pool cache, the two RUBiS workloads are left competing for the storage cache. In this case, we need to look at the MRC at the storage cache (shown in Figure 6) to explain the behavior. Due to the inclusiveness property, giving low quotas of storage cache to RUBiS is wasteful, up until this quota exceeds its buffer pool quota, translating into the initial flat portion of the MRC at the storage cache. Since the two RUBiS workloads have exactly the same behavior, one of them achieves the cross-over point into the steep area of the curve, hence proceeds to win all (928MB) of the storage cache quotas.

Finally, since the TPC-W and RUBiS workloads are highly cacheable, the CACHE-ONLY scheme achieves a solution close to IDEAL for this case.

For our second experiment, we run a load mix consisting of Zipf40 and TPC-W on one client and Zipf95 and RUBiS on the other client. We notice that the partitions received by each application in the GLOBAL and IDEAL schemes, as presented in the Table 1, follow the characteristics of the workloads explained before. Specifically, Zipf40 receives only 64MB at the first level of cache but the highest quota from the disk bandwidth (37.5%). The remaining buffer pool from that machine is given to TPC-W, who uses it to reduce its disk requirements to a point where it can perform well with a small disk quota (12.5%). The second level of cache can thus be entirely used to satisfy the higher cache needs of Zipf95 and RUBiS. Since RUBiS can benefit more from a larger cache, it gets the entire second level cache while Zipf95 meets its basic requirements with the smaller buffer pool on the client machine.

As before, the independent MRC-based allocation misses opportunities to exploit the synergy of the two-level cache. In this scenario, while TPC-W and Zipf40 have sufficient cache space in their shared buffer pool, the pairing of Zipf95 and RUBiS creates high cache pressure in the other buffer pool. Since RUBiS has a steeper MRC gradient than Zipf95, it gets most of the shared buffer pool (416MB) on that machine. The same goes for the storage cache, where RUBiS gets all the available space, 928MB. Therefore, due to inclusiveness, a large part of the combined cache is wasted. A better allocation would have been to allocate all buffer pool to the Zipf95 workload. However, we can see that MRC+DISK is able to offset some of the negative effects of the poor cache partitioning, by correctly partitioning the disk bandwidth, giving more to the most disk intensive application Zipf40 and less to TPC-W. This allocation provides a significant improvement compared to the MRC only scheme.



(a) Zipf95 (Disk) (b) RUBiS (Disk) (c) Zipf95 (Cache) (d) RUBiS (Cache)

Figure 8: **Accuracy:** We show a *heatmap* of the accuracy of our disk latency approximation and our two-level MRC approximation. The *darker* regions show regions with error and the *lighter* regions show regions with no error.

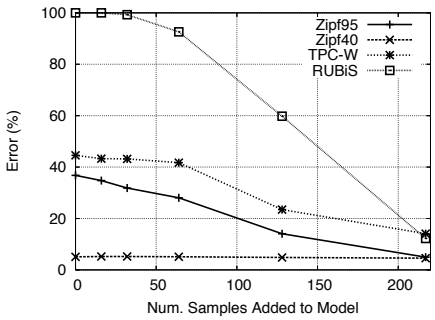


Figure 7: **Online Sampling:** We show that by adding performance samples collected at runtime, we improve the accuracy of our performance model.

6.4 Performance Model Accuracy

We present results highlighting the accuracy of our performance model approximations used in our multi-resource GLOBAL controller.

First, we evaluate the accuracy of our disk latency approximation, l_d, when using a quanta-based scheduler (Equation 4). We plot the accuracy of the performance model prediction versus the measured disk latency, for each application, when fixing the buffer pool size and varying only the storage cache size, and the storage bandwidth quanta given to the respective application. Figure 8a and Figure 8b present our results. On the *x* and *y* axes we show the proportion of the disk versus the proportion of the storage cache as a number between 0.0 and 1.0. We exhaustively sample the performance for all configurations, and we plot the discrepancy between the measured and predicted latencies.

The plot highlights areas with higher error as a *heatmap*, where higher error values are shown in darker colors. We plot results only for RUBiS and Zipf95, but all results show similar patterns. The average errors are 4.3% for Zipf40, 36% for Zipf95, 43% for TPC-W and 102% for RUBiS. As expected, we see the highest errors for the applications with the highest average "think time" per I/O thread (TPC-W and RUBiS). This is because we have ignored the effects of think times in our model. However, the high think times in these applications are due to the fact that they are highly cacheable, hence not disk intensive. Therefore disk latency inaccuracies do not substantially affect the accuracy of the overall latency model, which is 3.28% for TPC-W and 15.92% for RUBiS, on average.

Moreover, from the plot, we can see that most of the error is for the smallest disk quanta. This is a very localized area of the total search space, where inaccuracies may not matter, or can be improved by experimental sampling. Figure 7 shows the accuracy improvement through online performance sampling. In the *x*-axis we show the number of samples added to our performance model experimentally, and on the *y*-axis we show the error between the predicted and the actual latencies. For both TPC-W and RUBiS, adding samples by online sampling significantly reduces the error rate from 102% to 12% for RUBiS and from 43% to 14% for TPC-W.

Next, we present the accuracy of our MRC heuristic for two-level caches, given in Equation 1, which assumes cache inclusiveness in LRU caches. Figures 8c and 8d present the error in our MRC approximation. On the *x* and *y* axes we show the proportion of the storage cache, versus buffer pool given to the workload.

We exhaustively sample experimentally the performance for all cache configurations and we plot

the difference in the computed and predicted miss-ratios. Similar to the disk latency plot, we show a *heatmap* where higher error values are shown in darker colors and the areas of low error are shown in lighter colors. The sources of inaccuracy in our model are mainly due to two effects. First, the accesses seen at the storage cache are not the same as the trace we are using (collected at the buffer pool), since the buffer pool caches a fraction of accesses. This leads to small errors in the LRU approximation i.e., in the LRU stack algorithm, when computing the MRC. Second, if a dirty block is evicted from the buffer pool (top level cache), it needs to be written back to disk before eviction. Hence, it will likely still be stored in the storage (second level) cache, while our model assumes otherwise.

These inaccuracies are most pronounced when $\rho_s \geq \rho_c$, i.e., near to and under the diagonal in the plots, and for workloads with a higher fraction of I/O writes, such as RUBiS, and minimal otherwise. However, even for RUBiS, all errors are less than 2%, and these effects are not seen in Zipf40 and TPC-W (not shown), which are generally satisfied with much lower cache sizes, just from the buffer pool. For both of these applications, errors are close to zero, except for very localized areas of the map. Overall, the difference between the predicted and the computed miss-ratios is less than 4% on average.

7 Related Work

Previous related work has focused on dynamic allocation and/or controlling either memory allocation or disk bandwidth partitioning among competing workloads.

Dynamic Memory Partitioning: Dynamic memory allocation algorithms have been studied in the VMWare ESX server [31]. The algorithm estimates the *working-set* sizes of each VM and periodically adjusts each VM's memory allocation such that performance goals are met. Adaptive cache management based on application patterns or query classes has been extensively studied in database systems. For example, the DBMIN algorithm [7] uses the knowledge of the various patterns of queries to allocate buffer pool memory efficiently. In addition, many cache replacement algorithms have been studied e.g., LRU-k [15] in the presence of concurrent workloads. LRU-k prevents useful buffer pages from being evicted due to sequential scans running concurrently. Brown et al. [3, 4] study schemes to ensure per-class response time goals in a system executing queries of multiple classes by sizing the different memory regions. Finally, recently, IBM DB2 added the self-tuning memory manager (STMM) to size different memory regions [28].

Disk Bandwidth Partitioning: Dynamic allocation of the disk bandwidth has been studied to provide QoS at the storage server. Just like in our prototype, SLEDS [6], Façade [13], SFQ [11], and Argon [30] place a scheduling tier above the existing disk scheduler in order to control the I/Os issued to the underlying disk. However, these techniques assume that proportions are known e.g., set manually. However, more recent techniques, e.g., Cello [22], YFQ [5] and Fahrrad [19] build QoS-aware disk schedulers, which make low-level scheduling decisions that strive to minimize seek times as well as maintain quality of service.

Multi-resource Partitioning: Multi-resource partitioning is an emerging area of research where multiple resources are partitioned to provide isolation and QoS for several competing applications. Wachs et al. [30] show the benefit of considering both cache allocation and disk bandwidth allocation to improve the performance in shared storage servers. However, the resource allocation is done after modelling applications through extensive profiling. Wang et al. [34] extend the SFQ [11] algorithm to several storage servers. Padala et al. [17] study methods to allocate memory and CPU to several virtual machines located within the same physical server. However, these papers focus on either i) dynamic partitioning and/or quota enforcement of a single resource on multiple machines [34] or ii) allocation of multiple resources within a single machine [17, 30]. In our study, we have shown that global resource partitioning of multiple resources located at different tiers results in significant performance gains.

8 Conclusions

Resource allocation to applications on the fly is increasingly desirable in shared data centers with server consolidation. While many techniques for enforcing a known allocation exist, dynamically finding the appropriate per-resource application quotas has received less attention. The challenge is the exponential growth of the search space for the optimal solution with the number of applications and resources. Hence, exhaustively evaluating application performance for all possible configurations experimentally is infeasible.

Our contribution is an effective global resource allocation technique based on a unified resource-

to-performance model incorporating i) pre-existing generic knowledge about the system and interdependencies between system resources e.g., due to cache replacement policies and ii) application access tracking and baseline system metrics captured on-line.

We implement our global resource allocator within a virtual storage prototype with a two-tier cache hierarchy. We leverage the Network Block Device (NBD) to integrate our prototype within existing commodity software and hardware environments i.e., the MySQL database engine and a commodity Dell storage controller, with no changes to interfaces between components and minimal DBMS instrumentation.

We show through experiments using several standard e-commerce benchmarks and synthetic workloads that our performance model is sufficiently accurate in order to converge towards a near-optimal global partitioning solution within minutes. At the same time, our performance model effectively optimizes high-level performance goals providing up to a factor of 1.4 improvements compared to state-of-the-art single-resource controllers and their ad-hoc combination.

References

[1] Transaction processing council. http://www.tpc.org.

[2] Paul T. Barham, Boris Dragovic, Keir Fraser, Steven Hand, Timothy L. Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP*, pages 164–177, 2003.

[3] Kurt P. Brown, Michael J. Carey, and Miron Livny. Managing memory to meet multiclass workload response time goals. In Rakesh Agrawal, Seán Baker, and David A. Bell, editors, *VLDB*, pages 328–341. Morgan Kaufmann, 1993.

[4] Kurt P. Brown, Michael J. Carey, and Miron Livny. Goal-oriented buffer management revisited. In H. V. Jagadish and Inderpal Singh Mumick, editors, *SIGMOD Conference*, pages 353–364. ACM Press, 1996.

[5] John L. Bruno, José Carlos Brustoloni, Eran Gabber, Banu Özden, and Abraham Silberschatz. Disk scheduling with quality of service guarantees. In *ICMCS, Vol. 2*, pages 400–405, 1999.

[6] David D. Chambliss, Guillermo A. Alvarez, Prashant Pandey, Divyesh Jadav, Jian Xu, Ram Menon, and Tzongyu P. Lee. Performance virtualization for large-scale storage systems. In *SRDS*, pages 109–118, 2003.

[7] Hong-Tai Chou and David J. DeWitt. An Evaluation of Buffer Management Strategies for Relational Database Systems. In *Proceedings of 11th International Conference on Very Large Data Bases*, pages 127–141, Stockholm, Sweden, August 1985.

[8] Harris Drucker, Christopher J. C. Burges, Linda Kaufman, Alex J. Smola, and Vladimir Vapnik. Support vector regression machines. In *Neural Information Processing Systems (NIPS)*, pages 155–161, 1996.

[9] Ajay Gulati, Arif Merchant, and Peter J. Varman. pclock: an arrival curve based approach for qos guarantees in shared storage systems. *SIGMETRICS Perform. Eval. Rev.*, 35(1):13–24, 2007.

[10] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation and Modelling* . John Wiley & Sons, New York, 1991.

[11] Wei Jin, Jeffrey S. Chase, and Jasleen Kaur. Interposed proportional sharing for a storage service utility. In *SIGMETRICS*, pages 37–48, 2004.

[12] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, , and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *High-Performance Computer Architecture*, Salt Lake City, UT, 2008.

[13] Christopher R. Lumb, Arif Merchant, and Guillermo A. Alvarez. Façade: Virtual storage devices with performance guarantees. In *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 131–144, Berkeley, CA, USA, 2003. USENIX Association.

[14] R.L. Mattson, J. Gecsei, D.R. Slutz, and I.L. Traiger. Evaluation techniques for storage hierarchies. In *IBM System Journal*, pages 78–117, 1970.

[15] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993*, pages 297–306. ACM Press, 1993.

[16] Oguzhan Ozmen, Kenneth Salem, Mustafa Uysal, and M. Hossein Sheikh Attar. Storage workload estimation for database management systems. In Chee Yong Chan, Beng Chin Ooi, and Aoying Zhou, editors, *SIGMOD Conference*, pages 377–388. ACM, 2007.

[17] Pradeep Padala, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, Arif Merchant, and Kenneth Salem. Adaptive control of virtualized resources in utility computing environments. In *EuroSys*, pages 289–302. ACM, 2007.

[18] Adrian Popescu and Saeed Ghanbari. A study on performance isolation approaches for consolidated storage. *Technical Report, University of Toronto*, May 2008.

[19] Anna Povzner, Tim Kaldewey, Scott Brandt, Richard Golding, Theodore M. Wong, and Carlos Maltzahn. Efficient guaranteed disk request scheduling with fahrrad. *SIGOPS Oper. Syst. Rev.*, 42(4):13–25, 2008.

[20] Stuart J. Russell, Peter Norvig, John F. Candy, Jitendra M. Malik, and Douglas D. Edwards. *Artificial intelligence: a modern approach*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[21] Kai Shen, Tao Yang, Lingkun Chu, JoAnne Holliday, Douglas A. Kuschner, and Huican Zhu. Neptune: Scalable Replication Management and Programming Support for Cluster-based Network Services. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems, USITS*, pages 197–208, San Francisco, California, USA, March 2001.

[22] Prashant J. Shenoy and Harrick M. Vin. Cello: a disk scheduling framework for next generation operating systems. *SIGMETRICS Perform. Eval. Rev.*, 26(1):44–55, 1998.

[23] Elizabeth Shriver, Arif Merchant, and John Wilkes. An analytic behavior model for disk drives with readahead caches and request reordering. *SIGMETRICS Perform. Eval. Rev.*, 26(1):182–191, 1998.

[24] Gokul Soundararajan and Cristiana Amza. Towards end-to-end quality of service: Controlling i/o interference in shared storage servers. *Submitted to Middleware*, 2008.

[25] Gokul Soundararajan, Jin Chen, Mohamed Sharaf, and Cristiana Amza. Dynamic partitioning of the cache hierarchy in shared data centers. *Submitted to VLDB*, 2008.

[26] Gokul Soundararajan, Daniel Lupei, Saeed Ghanbari, Adrian Daniel Popescu, Jin Chen, and Cristiana Amza. Dynamic Resource Allocation for Database Servers Running on Virtual Storage. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST'09)*, pages 71–84, 2009.

[27] Gokul Soundararajan, Madalin Mihailescu, and Cristiana Amza. Context aware block prefetching at the storage server. In *USENIX*, 2008.

[28] Adam J. Storm, Christian Garcia-Arellano, Sam Lightstone, Yixin Diao, and Maheswaran Surendra. Adaptive self-tuning memory in db2. In *VLDB*, pages 1081–1092, 2006.

[29] Mustafa Uysal, Guillermo A. Alvarez, and Arif Merchant. A modular, analytical throughput model for modern disk arrays. In *MASCOTS '01: Proceedings of the Ninth International Symposium in Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'01)*, page 183, Washington, DC, USA, 2001. IEEE Computer Society.

[30] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R. Ganger. Argon: performance insulation for shared storage servers. In *FAST'07: Proceedings of the 5th conference on USENIX Conference on File and Storage Technologies*, pages 5–5, Berkeley, CA, USA, 2007. USENIX Association.

[31] Carl A. Waldspurger. Memory resource management in vmware esx server. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI '02)*, pages 181–194, 2002.

[32] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI '94)*, pages 1–11, 1994.

[33] Mengzhi Wang, Kinman Au, Anastassia Ailamaki, Anthony Brockwell, Christos Faloutsos, and Gregory R. Ganger. Storage device performance prediction with cart models. In *SIGMETRICS '04/Performance '04: Proceedings of the joint international conference on Measurement and modeling of computer systems*, pages 412–413, New York, NY, USA, 2004. ACM.

[34] Yin Wang and Arif Merchant. Proportional-share scheduling for distributed storage systems. In *FAST '07: Proceedings of the 5th USENIX conference on File and Storage Technologies*, pages 4–4, Berkeley, CA, USA, 2007. USENIX Association.

[35] Theodore M. Wong and John Wilkes. My cache or yours? making storage more exclusive. In *ATEC '02: Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, pages 161–175, Berkeley, CA, USA, 2002. USENIX Association.

[36] Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou, and Sanjeev Kumar. Dynamic tracking of page miss ratio curve for memory management. In Shubu Mukherjee and Kathryn S. McKinley, editors, *ASPLOS*, pages 177–188. ACM, 2004.