Dynamic Partitioning of the Cache Hierarchy in Shared Data Centers

Gokul Soundararajan, Jin Chen[†], Mohamed A. Sharaf and Cristiana Amza
Department of Electrical and Computer Engineering
Department of Computer Science[†]
University of Toronto

ABSTRACT

Due to the imperative need to reduce the management costs of large data centers, operators multiplex several concurrent database applications on a server farm connected to shared network attached storage. Determining and enforcing per-application resource quotas in the resulting cache hierarchy, on the fly, poses a complex resource allocation problem spanning the database server and the storage server tiers. This problem is further complicated by the need to provide strict Quality of Service (QoS) guarantees to hosted applications.

In this paper, we design and implement a novel coordinated partitioning technique of the database buffer pool and storage cache between applications for any given cache replacement policy and per-application access pattern. We use statistical regression to dynamically determine the mapping between cache quota settings and the resulting per-application QoS. A resource controller embedded within the database engine actuates the partitioning of the two-level cache, converging towards the configuration with maximum application utility, expressed as the service provider revenue in that configuration, based on a set of latency sample points.

Our experimental evaluation, using the MySQL database engine, a server farm with consolidated storage, and two e-commerce benchmarks, shows the effectiveness of our technique in enforcing application QoS, as well as maximizing the revenue of the service provider in shared server farms.

1. INTRODUCTION

The costs of management, power and cooling for large service providers hosting several applications are currently prohibitive, taking up more than 77% of the average company budget [30]. This is a major impediment on the efficiency of this industry, by limiting reinvestment, research and development. To achieve cost reductions, automated *server consolidation* techniques for better resource usage while providing differentiated Quality of Service (QoS) to applications become increasingly important. With server consolidation, several concurrent applications are multiplexed on each physical server of a server farm connected to consolidated network attached storage. The challenge lies in the complexity of the dynamic resource partitioning problem for avoiding application interference at multiple levels of this shared system. For example, the provider may service multiple applications on an infrastructure composed of web servers, database servers and storage servers (as in Figure 1). An especially important problem in these environments, which we focus on in this paper, is controlling application interference in the cache hierarchy across two tiers contributing directly to the performance of consolidated database applications, namely, 1) the database server tier, and 2) the storage server tier. Towards controlling this interference, we propose a dynamic global cache partitioning scheme that exploits the synergy between the cache at the database server i.e., the buffer pool, and the cache at the storage server.

Previous work in the area of dynamic resource partitioning has focused on controlling interference within a single tier at a time. For example, gold/silver/bronze priority classes within the buffer pool of a database system hosting several concurrent applications have been used to enforce memory priorities [4, 5]. Similarly, storage techniques for partitioning the I/O bandwidth between applications have been developed [31, 21, 13]. Additionally, enforcing per-application CPU quotas through resource virtualization techniques has been studied either at the operating system [3], or at the database system level [23, 24].

The previous approaches fall short of providing effective resource partitioning due to the following two reasons. The first reason is that application QoS is usually expressed as a Service Level Objective (SLO), not as per-resource quotas; there is currently no automatic mechanism to accurately assign resource quotas for applications corresponding to a given application metric. The second reason that prevents these approaches from providing effective resource partitioning is the absence of coordination between different resource controllers located within different tiers. This absence of coordination might lead to situations where local partitioning optima do not lead to the global optimum; indeed local goals may conflict with each other, or with the per-application goals. This resource allocation problem is further complicated when applications define different *utilities* (or *penalties*) for meeting (or missing) the specified SLOs. In such settings, the need is even stronger for an SLO-aware coordinated cache partitioning method which maximizes the system utility.

Coordination between the database buffer pool and stor-

PVLDB '08, August 23-28, 2008, Auckland, New Zealand
Copyright 2008 VLDB Endowment, ACM 978-1-60558-305-1/08/08

age cache has already been shown to be an effective mechanism in the context of cache replacement policies [19, 32]. However, coordinated cache replacement is an efficient mechanism for improving the performance of a single application, whereas in the presence of multiple applications, an orthogonal coordinated cache partitioning mechanism is still required. In this paper, we show that integrating our cache partitioning solution with current coordinated cache replacements policies provides further performance improvements that are not achievable using replacement policies alone.

Towards addressing the dynamic resource allocation problem in shared server environments, we introduce a novel technique for coordinated cache partitioning of the database server and storage caches. Our technique is independent of the cache replacement policy used at each level and it works with both coordinated and uncoordinated cache replacement policies. Our technique determines per-application resource quotas in each of the two caches on the fly, in a transparent manner, with minimal changes to the DBMS, and no changes to existing interfaces between components. To achieve this, we augment the DBMS with a resource controller in charge of partitioning *both* the buffer pool and the storage cache between applications. The target is to find a setting that maximizes the overall utility associated with the SLOs of a given set of applications. The resource controller maps the application specified SLO to a target *data access latency*, which is the average block access latency measured at the database buffer pool required to meet the SLO.

To decide the right partitioning, the cache controller explores the configuration space through an on-line simulation of the cache hierarchy. This allows us to converge faster towards an optimal partitioning solution. However, the cache controller actuates the cache partitioning settings periodically, to the current best configuration, and measures performance in the current configuration, in order to validate the simulation. The controller employs statistical regression to dynamically determine per-application performance/utility models as mapping functions between the cache quota settings of the two caches and the corresponding application latency/utility. It then uses these per-application models to answer "what-if" cache partitioning scenarios, for any given set of applications, hence to dynamically converge towards a partitioning that maximizes the perceived overall reward.

We implement our technique in a prototype of a two-level cache controller. In our experiments, we use the MySQL database engine and two applications: the TPC-W e-commerce benchmark, emulating an on-line bookstore, such as Amazon.com, and the RUBiS on-line bidding benchmark, emulating an on-line auctions site, such as eBay.com. We use our prototype in an experimental testbed, where instances of the two applications share physical servers as well as the storage server, to enforce cache quota allocations for different SLO and load scenarios, and different cache replacement policies. In terms of cache replacement policies, we integrate our coordinated, dynamic cache partitioning technique with i) classic uncoordinated LRU replacement at each cache level, as well as ii) coordinated cache replacement based on demote hints from the buffer pool to the storage cache [32].

We show that our coordinated dynamic partitioning technique provides compliance with the SLO requirement of applications with strict SLO's, while at the same time maintaining efficient resource usage. As a result, our dynamic cache partitioning technique minimizes penalties in overload and maximizes the revenue of the service provider in underload.

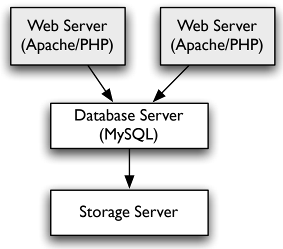


Figure 1: Data center architecture with shared DBMS and shared storage

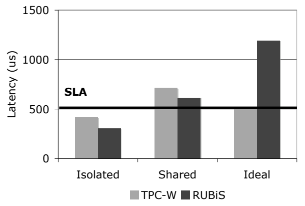
The remainder of this paper is structured as follows. Section 2 provides a background on server consolidation in modern data centers highlighting the detrimental effect of interference between two applications. We describe our coordinated cache partitioning algorithm in Section 3. Section 4 describes our virtual storage prototype. 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

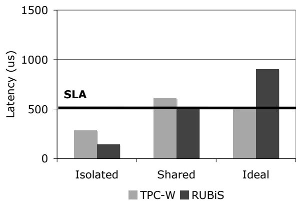
Modern enterprise systems consist of multiple software layers including web/application server front-ends, database servers running on top of the operating system, and storage servers at the lowest level. In order to reduce hardware and management costs in large data centers, the storage system is usually shared by a cluster farm, as shown in Figure 1. Since slow disk access is the bottleneck in this system, both the database servers and the shared storage server use memory to cache data blocks, resulting in a *two-tier* cache hierarchy.

In this paper, we propose methods for controlling interference among applications in this cache hierarchy. Our techniques are applicable to situations where the working set of concurrent applications does not fit into the cache hierarchy. These situations are, and will remain common in the foreseeable future due to the following reasons. First, while both the buffer pool and storage server cache sizes are increasing, so do the memory demands of applications e.g., scientific and commercial very large databases. Second, efficiently using the combined caching capabilities of database server and storage server is challenging even for a single application. Indeed, the potential for double caching of blocks, and the typically poor temporal locality of accesses that miss in the buffer pool lead to poor cache utilization in the storage level cache [8, 20]. Finally, running several applications on a cluster with consolidated storage, and/or on the same physical server exacerbates the above problems due to application interference for memory, hence the increased potential for capacity misses in the cache hierarchy.

The synergy between buffer pool and storage cache has been exploited through replacement policies for improving cache hierarchy effectiveness for a single application [11, 14, 15, 19, 22, 32, 33]. Specifically, recent work has shown that



(a) LRU/LRU



(b) LRU/DEMOTE

Figure 2: We experiment with two cache configurations: LRU/LRU and LRU/DEMOTE. The results show significant room for improvement.

communication between caches is essential [19, 32, 33] for effective use of multi-tier caches. For example, the Demote [32] scheme sends block eviction hints or explicit demote operations from the client cache e.g., the database buffer pool, to the storage cache with the goal to maintain *exclusiveness* between the two caches. When the client cache is about to evict a clean block, it sends the clean block to the storage cache using a special Demote operation. The storage cache places the demoted block in its cache, ejecting another block if necessary. The storage cache moves the blocks read by the client to the LRU (least-recently-used) position such that they will be evicted before the demoted blocks. Li et al. [19] and Yadgar et al. [33] extend the Demote idea using DBMS specific information. Their work has shown that these techniques increase the effectiveness of the combined buffer pool and storage caches and are essential to the performance of the database system. However, in the next section, we show that the detrimental effect of application interference in the cache hierarchy offsets the gains obtained from the above advanced replacement policies.

2.1 Motivating Example

We present a motivating example to highlight the need for better management of shared multi-tier caches. We use two applications: TPC-W, considered *strict SLO*, and RUBiS, considered *best effort*, and schedule the applications such that they share a single DBMS instance, as well as the storage server, as shown in Figure 1. We require that the average TPC-W query *data access latency* be less than 500μs; in practice, some pre-defined margin of error may be acceptable. We also assume that any reductions in the latency of the *best effort* application, RUBiS, compared to the worst case scenario are rewarded e.g., through revenue increases for the provider; we consider the worst case scenario for RUBiS to correspond to its incurring the full disk latency on each query data access.

We run the two applications using a single MySQL/InnoDB database engine and a consolidated storage server. Since MySQL/InnoDB does not provide an easily *partitionable* buffer pool, we replace its buffer pool with our own implementation. We use a 1GB buffer pool and a 1GB storage cache and we experiment with two cache replacement policies in the two shared caches. We denote a scheme as LRU/LRU a scheme where the classic LRU replacement policy is used in both the buffer pool and the storage cache (Figure 2(a)). We denote as LRU/DEMOTE a scheme where a LRU replacement is used at the buffer pool modified to support the Demote cache block eviction hints for the storage cache (Figure 2(b)). We provide the details of our storage platform in Section 4. We compare two schemes: (1) SHARED where the applications share both the DBMS buffer pool and the storage cache, with no quota enforcement and (2) IDEAL where we experimentally iterate through all possible partitioning configurations of both caches and choose the optimal setting where we meet the SLO for TPC-W, while minimizing the latency for RUBiS.

Figure 2 shows the performance of each benchmark under the above schemes, in addition to the performance for ISOLATED, which corresponds to running each benchmark in isolation, using a 1GB buffer pool at the DBMS and a 1GB storage cache. Figure 2(a) shows that under LRU/LRU, the average latency of TPC-W, in isolation, is 420μs, while for RUBiS the isolation latency is 304μs. This experiment shows that our storage infrastructure is capable of meeting the SLO for TPC-W. Next, we run both TPC-W and RUBiS allowing them to share the buffer pool and the storage cache. In this case, there is no SLO enforcement resulting in TPC-W consistently violating its SLO with an average 715μs data access latency. This scenario would result in hefty penalties for the service provider. By partitioning the caches, the IDEAL partitioning scheme finds a cache setting that maintains TPC-W within the SLO. This scheme shows the best possible resource usage scenario and the highest revenue for the service provider.

We repeat our experiments with the LRU/DEMOTE cache replacement policy (Figure 2(b)). Since the Demote scheme results in a better utilization of the overall cache hierarchy, both TPC-W and RUBiS obtain lower latencies when in isolation, compared to the LRU/LRU case. The average data access latency is 284μs for TPC-W, while for RUBiS is 143μs. While the LRU/DEMOTE policy provides better cache utilization, using the SHARED scheme still results in a SLO violation, since the average data access latency for TPC-W is 617μs. Similar to the results in the LRU/LRU case, the IDEAL scheme maintains the TPC-W latency within the SLO for LRU/DEMOTE as well.

The above results show that the performance of a *strict SLO* application can severely degrade when two database applications are co-located within the same DBMS instance. These experiments thus motivate coordination in terms of both cache partitioning and replacement policy between the two caches. However, the problem of finding the globally optimum partitioning of the two caches to a given set of applications is an NP-hard problem. Let's consider the time needed to find the IDEAL cache partitioning setting. For example, say we can have 32 possible quota settings for each cache. Then, in order to estimate an application's performance for all possible cache and storage quota configurations, we need to gather performance samples for $32 \times 32 = 1024$ configurations. Each sample point measurement may take 16 minutes, on average, to ensure statistical significance e.g., due to cache warmup effects and the need to measure latency several times in each configuration. Therefore, in order to compute an accurate performance model for just *one* application, we will need 1024×16 minutes, i.e., 273 hours (approximately 11 days)! In our experiment for obtaining the IDEAL cache setting, we reduce this time significantly, by iterating from larger cache quotas to smaller cache quotas for the two caches, for each application, thus amortizing the warm-up time of the larger cache quota configurations for the smaller cache quota configurations. This still results in a total running time for the two applications on the order of days, which is unacceptable for on-line adaptation.

In the rest of this paper, we describe the design and implementation of a novel approximate algorithm that partitions both the database buffer pool and the storage cache *on-line* for any cache replacement policy and any per-application access pattern.

3. CACHE PARTITIONING ALGORITHM

In this section, we describe our approach to providing effective coordinated cache partitioning in two-level caches. Our main objective is to maximize the utility i.e., reward or revenue, derived by the server provider from running a set of applications concurrently on a shared cache hierarchy. Towards this, we use a novel technique, called *utility-aware iterative learning* to determine the size of cache quotas at different levels in the system, i.e., the DBMS and the storage server. The key idea is to dynamically determine, through a statistical regression method, the mapping between a cache partitioning setting for a given set of applications and its corresponding overall utility for the service provider.

In the following, we first introduce the problem statement, and an overview of our approach. Then, we introduce our utility-aware iterative learning approach along with details of its main components.

3.1 Problem Statement

We study dynamic cache allocation to multiple applications with pre-defined QoS requirements in the cache hierarchy of server farms with network attached storage.

In our model, we assume that the system is hosting n applications, where each application runs on only one database engine. Each engine has its own buffer pool cache. Additionally, the system has a storage cache which is shared among all applications. Finally, we assume that each application is associated with a pre-specified *utility*, i.e., benefit as a function of the data access latency perceived by the given application. Thus, the cache partitioning problem translates into allocating each application a buffer pool quota and a storage cache quota in such a way to maximize the service provider's revenue. Specifically, let's denote with $r_1, r_2, \ldots, r_n$ the data access latencies of the n applications hosted by the service provider and let $U_i(r_i)$ represent the utility function for the i^{th} application. The goal of the service provider is to maximize the sum of all application utilities i.e.,:

$$max \sum_{i=1}^{n} U_i(r_i) \quad (1)$$

Finding a practical solution to this problem is difficult, because of the following three reasons:

First, as we have shown in Section 2, exhaustively evaluating the application performance for all possible configurations experimentally is infeasible.

Second, effective utilization of the caches depends on several factors, including the (dynamic) access patterns of the applications, the (dynamic) number of applications scheduled on the server farm, and the cache replacement policy used in each cache. Due to the unpredictable impact of these cache and application parameters, implementing an analytical model of performance for guiding the cache partitioning search becomes a daunting task.

Third, accurately evaluating the *utility*, i.e., benefit gained from an application's use of its total memory quota within a system component, such as, the database or storage server is non-trivial. Using common cache metrics, such as monitoring the *hit rate* in each of the two caches is impractical because: i) the hit rate at the storage cache depends on the behavior of the upper level cache, i.e. the size of the buffer pool and its replacement policy and ii) their respective access times differ. In more detail, increasing the allocation for an application in the buffer pool usually affects the block accesses seen, hence the hit rate measured, at the storage cache. Moreover, different cache replacement algorithms e.g., LRU versus DEMOTE influence the number and type of accesses seen at the storage cache. Finally, a buffer pool hit is usually more valuable to the application than a storage cache hit, because the storage cache access usually incurs the additional network delay to the storage server. Therefore, simply combining the two cache hit rates for each application does not provide a meaningful overall utility value for memory usage.

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 system components. For this purpose, we introduce a novel algorithm, called *utility-aware iterative learning*, which iteratively performs the following two inter-related operations:

1. We build approximate performance models, called *application surfaces* for mapping cache configuration quotas to the application latency, and its corresponding utility, for each application, on the fly, and

2. We use the per-application performance models to answer "what-if" cache partitioning scenarios, for any given set of applications, as part of an efficient automatic search for the optimal two-tier cache partitioning solution.

Specifically, we employ statistical *support vector machine regression* (SVR) [12] for approximating the per-application performance models, based on a set of sample points. Each sample point consists of the application latency for a given cache quota configuration. As sample points are incrementally collected, our algorithm iterates through successive refinement steps, re-approximating the per-application performance models and the optimal solution, until convergence of both the models and the overall optimum occurs.

To achieve this, we augment the DBMS with a resource controller in charge of partitioning *both* caches between applications. The DBMS cache controller runs our *utility-aware iterative learning* algorithm to dynamically converge towards a partitioning setting that maximizes the combined application utilities. For each application, the DBMS collects a set of sample points recording the average data access latency, and its corresponding calculated utility in each cache configuration. In order to speed up convergence, the controller gathers sample points by cache simulation, instead of experimentally. However, the cache controller actuates cache partitioning to better configurations periodically, in order to reduce the penalties incurred by the service provider, whenever the latencies of applications exceed the SLO. When actuating, the controller samples the latency in that cache quota configuration, hence can validate/adjust the respective simulation-based sample point.

In the following subsections, we first describe our performance models in more detail, then introduce our utility-aware iterative learning algorithm.

3.3 Performance Models

We dynamically build a per-application *latency model*, which maps cache partitioning quotas for the respective application to the expected latency. Each per-application model is thus a 3D-surface, of the form $r_i(q_{i,c}, q_{i,s})$, or simply $r_i(q_c, q_s)$, where q_c and q_s are the quotas allocated to the application in the buffer pool and storage cache, respectively. Each sample point $r_i(q_c, q_s)$ is the data access latency of the application, as estimated by an on-line simulation of the cache hierarchy. Selective experimental points are also collected to validate or adjust the simulation points.

For the purposes of approximating each function, r_i, based on a set of sample points for that application, we use *support vector machine regression* (SVR) [12]. Then, for each approximated latency model, r_i, we compute the corresponding *utility model* as $U_i(r_i)$, which is another 3D-surface, we call *application surface*. The application surface represents the service provider's revenue for hosting application i for the corresponding application latency function r_i, obtained for different cache configurations (q_c, q_s).

3.4 Utility-Aware Iterative Learning Algorithm

For a given set of applications and a cache hierarchy, our goal is to find the cache configuration maximizing the combined application utilities. Towards this, we propose our *utility-aware iterative learning algorithm*.

In each iteration of our algorithm, we enhance the quality of the latency models described above. Specifically, we employ statistical regression to approximate the per-application latency models, as a set of functions r_i, one such function for each application i, based on a set of latency sample points, $r_i(q_c, q_s)$, collected for different cache configurations (q_c, q_s). Given the enhanced latency models, we calculate the corresponding application surface for each application. Finally, our goal is to pick from each 3D application surface a single point (i.e., a cache configuration), which:

1. provides $max \sum_{i=1}^{n} U_i(r_i)$, and
2. respects the constraint that the sum of the cache quotas (proportions) allocated to applications for each level of cache must be equal to 1.

In order to expedite this search process, we perform a *hill-climbing* search for the cache configuration settings over the n application surfaces given the requirements above.

Our proposed learning algorithm iterates through successive refinement steps, as more sample points are incrementally added, re-approximating the per-application performance models, as a set of functions r_i, as well as the optimal solution, using statistical regression, and hill-climbing, respectively, until convergence of both the models and the overall optimum occurs. The learning algorithm converges when either one of the following conditions occurs: i) adding more sampling points does not increase the accuracy of the regression function i.e., the per-application surfaces vary only within a predefined deviation bound across iterations, or ii) the maximum value of the combined utilities for all applications does not change anymore across iterations, even with increasing the resolution of the regression functions.

Algorithm 1 Iterative learning for searching the optimal cache partitioning configuration Q^*

```
1: Initialize: $\forall i$, sample set $S_i$ of application $i$, $S_i = \emptyset$
2: repeat
3:     for $i = 1$ to $n$ do
4:         1) Add k new samples to sample set $S_i$
5:         2) Use SVR to learn the function $r_i$ using sample set $S_i$
6:     end for
7:     3) Map data access latencies $r_i$ to utility values
8:     4) Find MCU = max $\sum_{i=1}^{n} U_i(r_i(q_c, q_s))$ for all valid configurations.
9:     5) Actuate to current best configuration $Q^*$ which generates MCU.
10: until Regression error is below a threshold or the MCU value is stable
```

Algorithm 1 shows the pseudo-code for our iterative learning process. At a given iteration of the algorithm, each application i has a sample set, denoted by S_i, initialized to empty (line 1). In each iteration step, for each application i, we generate a new set of sample points to expand the current sample set (line 4); we then learn the regression functions r_i, based on the current sample set. Based on the regression functions for all applications, we convert application performance metrics i.e., average data access latency to the respective utility values (line 7). Next, we employ *hill climbing* to find the maximum combined utility (MCU) value for all valid configurations in the resulting search space (line 8). Next, we actuate to the optimal cache partitioning configuration Q^*, which is a set of pairs of cache configurations per application, that is, $Q^* = (q_{1,c}, q_{1,s}), (q_{2,c}, q_{2,s}), ..., (q_{n,c}, q_{n,s})$ (line 9), and we proceed to check for convergence (line 10).

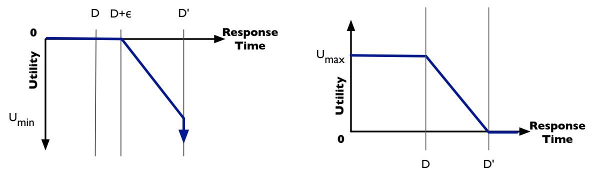
In the following, we describe the main operations in our utility-aware iterative learning algorithm. In particular, we provide the details of steps 1–4 in Algorithm 1 above.

Step 1: Sampling

We experiment with two methods for generating sample points: (1) *random* sampling, and (2) *greedy* sampling. In *random* sampling, a sample is selected randomly from all possible cache partitioning configurations; every possible sample has an equal chance of being selected. Random sampling is not goal oriented, hence can lead to relatively slow convergence. *Greedy* sampling optimistically predicts that the current optimum found at a given iteration is close to the global optimum. It thus preferentially adds sample points within a gradually increasing radius of the current optimum, seeking rapid convergence. A variant of our *greedy* algorithm is to add sample points along vectors with the highest gradient in the current search space.

Step 2: Statistical Regression

For the purposes of approximating each function, r_i, based on a set of sample points for that application, we use *support vector machine regression* (SVR) [12]. SVR is a non-linear regression algorithm that is tolerant to measurement errors



(a) High Priority (b) Best Effort

Figure 3: Utility Functions

(small noise) in the sample set, as well as generalizing for the regions that are not sampled, unlike other machine learning techniques, such as multi-linear-regression [17]. SVR maps the regression problem to a quadratic optimization, finding the optimum solution.

Given a set of training points $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)\}$, SVR finds a function $\overline{f}(\mathbf{x})$ that has a small deviation (ϵ) from the targets y_i for all training data points. The estimated function $\overline{f}(\mathbf{x})$ takes the form:

$$\overline{f}(\mathbf{x}) = \sum_{i=1}^{m} \alpha_i y_i K(\mathbf{x}_i, \mathbf{x}) \qquad (2)$$

To build our latency model per-application, each training point i represents one of the sample points, where $\mathbf{x_i}$ is the cache configuration for that point (i.e., (q_c, q_s)) and y_i is the latency corresponding to that configuration. Each training point $\mathbf{x_i}$ is associated with a variable α_i that represents the strength with which the training point is embedded in the final function. The points which lie closest to the hyperplane, denoting $\overline{f}(\mathbf{x})$, are called the *support vectors*. $K(\mathbf{x}_i, \mathbf{x})$ is a kernel function which maps the input into a high dimensional space, called feature space, where linear support vector regression is applied. We use radial basis functions (RBFs) as our kernel functions.

Step 3: Mapping Latency to Utility

The utility function corresponding to the performance of any given application (e.g., [18, 6]) varies since it depends on the contract between the service provider and the client and the costs for the service provider to host the application. Our algorithm does not depend on the exact specification of the utility function. Thus, without loss of generality, for the purposes of this paper, we classify applications in two categories: *strict SLO* (or *high priority*) applications and *best effort* applications.

Figure 3(a) depicts the utility function we use for strict SLO applications. For this application class, the provider pays a penalty whenever the application's SLO i.e., its average data access latency (denoted as response time in the Figure), is violated beyond a small margin of error called *slack*. On the other hand, the provider has no benefits for providing service better than the pre-agreed SLO for the application. As shown in the Figure, as long as the application's response time is less than a deadline D with some slack ϵ, the utility is constant at zero. Beyond this value, the provider starts paying penalties for SLO violations, proportional to the magnitude of the violation, until another threshold D' considered to be unacceptable to the customer.

Figure 3(b) shows the utility function for the *best effort* application class. The provider pays no penalties, regardless of the level of service for an application in this class. Hence, the baseline level of performance with response time beyond D' has the utility value zero. This baseline level would correspond to the application performance for 100% cache miss rates for any level of cache in our case. However, we assume that performance above the baseline carries a reward for the service provider, which increases proportionally to the level of service until reaching a maximum performance level, after which no more benefits accrue.

Step 4: Finding the Maximum Combined Utility

In order to achieve a near-optimal performance, we need to select from each 3D application surface a cache configuration so that the total application utilization is maximized. This results in a combinatorial search space where finding the optimal solution is not feasible. Hence, we use the greedy *hill climbing* algorithm with random restarts to find the point where the combined utility is the maximum.

3.5 On-line Adaptation to Dynamic Changes

After our *utility-aware iterative learning* algorithm converges, we obtain accurate per-application surfaces, and the optimal Q^* cache partitioning configuration, for the current set of applications running on the infrastructure. Depending on the type of dynamic change, the entire algorithm, or selected parts of it, may need to be re-executed. For example, if a new application is co-scheduled on the same infrastructure, we need to sample the latency and compute the *application surface*, only for the new application. Then, we re-compute the new optimum, Q^*, cache partitioning configuration by *hill climbing*, based on the new set of *application surfaces*. If the access pattern of a given application changes e.g., as detected by significant changes in its miss ratio curve monitored on the trace collected on-line for simulation purposes, we need to build a new application surface from scratch for the given application, and recompute the global optimum configuration. For any other type of dynamic change where the application pattern does not change, hence the cache behavior is stable, e.g., if the number of clients of any given set of applications increases, the per-application surfaces remain accurate, hence we simply need to recompute the optimum configuration in order to minimize losses.

4. PROTOTYPE IMPLEMENTATION

We implement our dynamic cache partitioning algorithm within MySQL and in our Linux-based virtual storage prototype, Gemini. We run a database server using a networked storage server. The architecture, shown in Figure 4, includes a two-level cache hierarchy, consisting of a buffer pool and a storage cache.

MySQL communicates with the virtual storage device through standard Linux system calls and drivers, either iSCSI or NBD (network block device), as shown in the Figure 4. 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 Gemini modules on the I/O communication path.

In the following, we first describe the interfaces and communication between the core modules, then describe the role of each core module in more detail. Finally, we describe the dual role of the Gemini prototype as an on-line cache simulator, where the same modules which service an I/O request are used concurrently to explore the configuration space faster and with minimal overhead.

4.1 Virtual Storage System

Gemini is a modular virtual storage system which can be deployed over commodity storage firmware. It supports data accesses to multiple virtual volumes and it can interface through Linux with either a storage controller for a RAID system or a single hard disk. Finally, we design a database system plug-in to enable coordination between the database system and the storage server.

Storage clients, such as MySQL, use NBD for reading and writing logical blocks. For example, as shown in Figure 4, MySQL/InnoDB mounts the NBD device (`/dev/nbd1`) on `/dev/raw/raw1`. The Linux virtual disk driver uses the NBD protocol to communicate with the storage server. An I/O request from the client takes the form `<type,offset,length>` where `type` is a *read* or *write*. The I/O request is passed by the OS to the NBD kernel driver on the client, which transfers the request over the network to the NBD protocol module running on the storage server.

The storage server is built using different modules. Each module consists of several threads processing requests. The modules are interconnected through in-memory buffers. The modular design allows us to build many storage configurations by simply connecting different modules together.

Disk module: The disk module sits at the lowest level of the module hierarchy. It provides the interface with the underlying physical disk by translating application I/O requests to the virtual disk into `pread()`/ `pwrite()` system calls, reading/writing the underlying physical data. We disable the operating system buffer cache by using direct I/O i.e., the I/O `O_DIRECT` flag in Linux.

Cache module: The cache module allows data to be cached in memory for faster access times. The cache module is portable to different environments by providing a simple hashtable-like interface (modelled after *Berkeley DB*) supporting `get()`, `put()`, `delete()` and `flush()` operations. It supports different block sizes, dynamic resizing, asynchronous I/O, several cache replacement algorithms and several prefetching policies. For the purposes of this paper, the cache maintains data as a collection of *blocks*, implements two cache replacement policies, either LRU or DEMOTE, and manages accesses from concurrent threads. Since MySQL/InnoDB does not support buffer pool partitioning, we embed our caching library into MySQL, replacing MySQL's buffer pool manager. The server cache is located on the same physical node as the storage controller. The two instances of the cache module create a *two-tier* cache hierarchy.

NBD Protocol module: We modify the original NBD processing module on the server side, used in Linux for virtual disk access, to convert the NBD packets into our own internal protocol packets, i.e., into calls to our Gemini server cache module.

4.2 Cache Simulator

The Gemini system has the capability to double-up as a simulator in any runtime configuration in order to estimate settings with the best performance, on-line. In on-line simulation mode, it explores the search space of cache partitioning settings for any given cache replacement and data distribution technique employed by our prototype. The modifications to the operations are minimal and consist of simulating: i) disk accesses and network communication by recording their corresponding delays on a virtual clock and ii) data touches based on a real access trace. We replace the NBD processing modules with a trace module. The trace module replays the most recent accesses in the trace collected at the level of the MySQL buffer pool. The same cache code, as described for the caching module, runs for the two caches in simulation as in the real implementation.

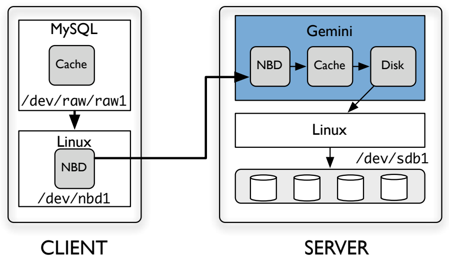


Figure 4: Gemini Storage Architecture: We show one client connected to a storage server using NBD.

5. EVALUATION

In this section, we describe several cache partitioning algorithms we use in our evaluation, as well as the benchmarks and our evaluation platform.

5.1 Algorithms used in Experiments

We implemented a prototype of our utility-aware iterative learning algorithm (Section 3), which we will call DYNAMIC and compared it to the following schemes:

CONSERVATIVE: We take the conservative approach and allocate both the buffer pool and the storage cache to the high-priority application. To the low-priority application, we allocate only a minimum cache quota, such that its data accesses can still occur i.e., 32MB in our implementation, and dedicate the rest of the cache space to the high-priority application.

PROFILE: We profile each application off-line to determine the amount of buffer pool it needs in order to meets its SLO. We assign each application the respective amount of buffer pool cache, whereas the storage cache is shared among all applications with no-quota enforcement. Hence, this scheme is SLO-aware, however, it is oblivious to the presence of the second-level cache.

MRC: A miss-ratio curve (MRC) estimates the page miss ratio for an application given a particular amount of memory. It has been applied to effectively allocate memory to several applications [34]. In this paper, we extend MRC for the purpose of partitioning a two-level cache hierarchy. Specifically, at the buffer pool level, we partition the buffer pool using the MRC computed for each application. Similarly, at the storage cache level, we partition the storage cache by building an MRC for each application using its *missed* data accesses (i.e., accesses that are not satisfied by the buffer pool cache).

IDEAL: We perform off-line experiments iterating through all possible partitioning configurations of the two caches and choose the setting which maximizes the revenue.

SHARED: We allow applications to share both the DBMS buffer pool and the storage cache with no quota enforcement.

DYNAMIC: This is our cache partitioning scheme described in Section 3.

5.2 Benchmarks

We use two industry-standard benchmarks, TPC-W and RUBiS, to evaluate our proposed algorithm.

TPC-W^{10}: 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 create TPC-W^{10} by running 10 TPC-W instances in parallel creating a database of 40 GB.

RUBiS^{10}: 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 distinguish between three kinds of user sessions: visitor, buyer, and seller. For a visitor session, users are only allowed to browse. During a buyer session, in addition to the functionality provided during the visitor sessions, 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 [27]. We create a scaled workload, RUBiS^{10} by running 10 RUBiS instances in parallel.

5.3 Evaluation Platform

We run our Web based applications on a dynamic content infrastructure consisting of the Apache web server, the PHP application server and the MySQL/InnoDB (version 5.0.24) database storage engine. We run the Apache Web server and MySQL on Dell PowerEdge SC1450 with dual Intel Xeon processors running at 3.0 Ghz with 2GB of memory. MySQL connects to the raw device hosted by the NBD server. We run the NBD server on a Dell PowerEdge PE1950 with 8 Intel Xeon processors running at 2.8 Ghz with 3GB of memory. The storage uses a direct-attached SAS enclosure with 15 10K RPM 250GB hard disks configured to use RAID-0. We install Ubuntu 6.06 on both the client and server machines with Linux kernel version 2.6.18-smp. We configure our caching library to use 16KB block size to match the MySQL/InnoDB block size.

6. RESULTS

In this section, we present an experimental evaluation of our multi-tier cache allocation technique. We conduct experiments on our prototype storage system to evaluate the performance of our approach. We use two applications: TPC-W as the strict SLO (high-priority) application, and RUBiS as the best-effort application. We express the SLO in terms of average data access latency. A data access latency SLO of less than 500μs provides an average *query response time* below 500ms for both our benchmarks, which closely approximates values used as QoS for the two e-commerce applications in previous studies [28]. Thus, in our utility function, we set $D = 500\mu$s, $D' = 3500\mu$s (the average disk access time), $U_{min} = -100$ and $U_{max} = 100$. In addition, we experiment with relaxing the SLO by varying the slack (ϵ) from 10% ($D + \epsilon \leq 550\mu$s) to 100% ($D + \epsilon \leq 1000\mu$s). We explore the effects of different application access patterns and the effect of different replacement policies on the optimal cache partitioning. We use 1 database server and 1 storage server, each configured with a 1GB cache.

6.1 Latency Surfaces

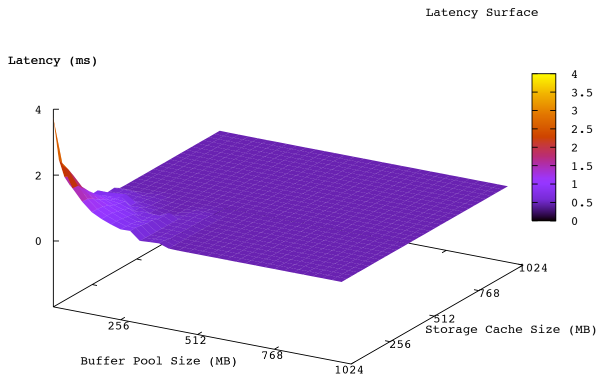
In Figure 5, we show the latency surface of two applications: (1) TPC-W, and (2) RUBiS. The figure shows the data access latency for different settings of the buffer pool size and the storage cache size. The light gray areas indicate configurations with high data access latency, whereas the dark gray areas indicate configurations with low data access latency. These applications have varying working sets. TPC-W, having a small working set, obtains low data access latencies even with small allocations of cache space. RUBiS, with a larger working set, requires more cache space in the cache hierarchy to obtain low data access latencies.

6.2 Latency under LRU/LRU

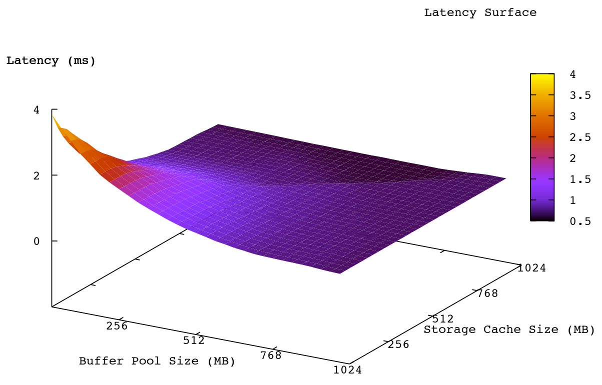
In Figure 6, we compare the performance of the different cache partitioning schemes when both the database buffer pool and the storage cache use the LRU cache replacement policy. Figure 6(a) shows that, under the SHARED and the MRC schemes, the SLO of our high-priority application (i.e., TPC-W) is violated. For instance, the average data access latency of TPC-W under SHARED is 715μs, as opposed to the pre-specified SLO of 500μs. This is mainly because both schemes are oblivious to the SLO requirements.

On the other hand, both the CONSERVATIVE and the PROFILE schemes satisfy the SLO requirements of TPC-W. However, both schemes over-allocate cache resources to TPC-W to the detriment of the best-effort application (RUBiS). Between the two schemes, RUBiS performs worse under CONSERVATIVE, which allocates all the available cache to TPC-W. In contrast, under PROFILE, RUBiS achieves a better performance, since PROFILE allows RUBiS to share the storage cache with TPC-W.

The IDEAL scheme, and similarly our DYNAMIC scheme, are both able to strike a fine balance between satisfying the TPC-W SLO requirement while providing an acceptable performance to RUBiS. This is simply due to the fact that under LRU/LRU, the storage cache typically includes blocks already contained in the database buffer pool. Thus, there is no additional benefit for an application, if its partition in the storage cache is smaller than its partition in the buffer pool. Since both CONSERVATIVE and PROFILE are oblivious to this *inclusiveness* property exhibited by LRU/LRU, they might allocate storage cache to TPC-W. This is wasteful, since TPC-W does not derive any additional benefit from the storage cache allocation, while the respective allocation of storage cache could have been of a significant benefit if allocated to RUBiS. Our DYNAMIC scheme dynamically recognizes this trade-off and it detects that more revenue is achievable if the storage cache is allocated to RUBiS. Thus, by accurately computing the overall utility function, DYNAMIC chooses a near-optimal cache partitioning setting, where most of the database buffer pool is allocated to TPC-

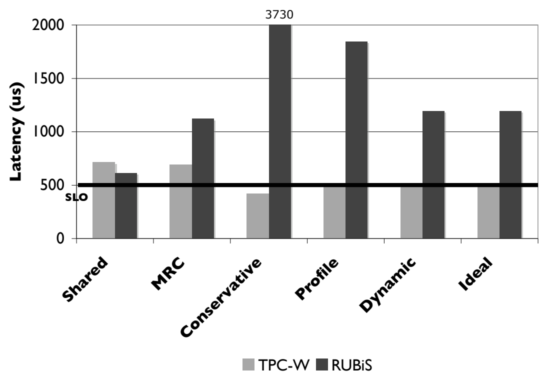


(a) TPC-W (*small working set*)

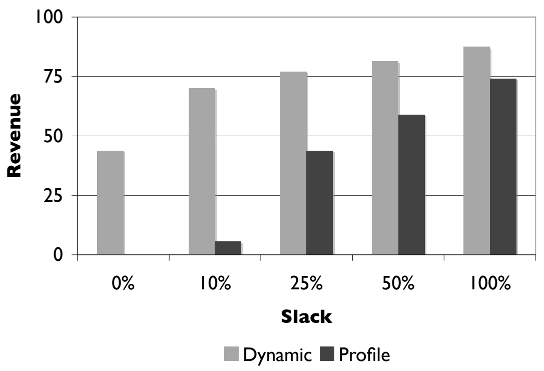


(b) RUBiS (*large working set*)

Figure 5: Latency Surfaces: Data access latency for different partitionings of buffer pool and storage cache.



(a) Strict SLO



(b) Profit with varying Slack

Figure 6: LRU/LRU

W (the high-priority application) and most of the storage cache is allocated to RUBiS (the best-effort application). With near-optimal settings, using DYNAMIC, we reduce the latency of RUBiS to 1193μs (versus 1844μs for PROFILE).

6.3 Revenue under LRU/LRU

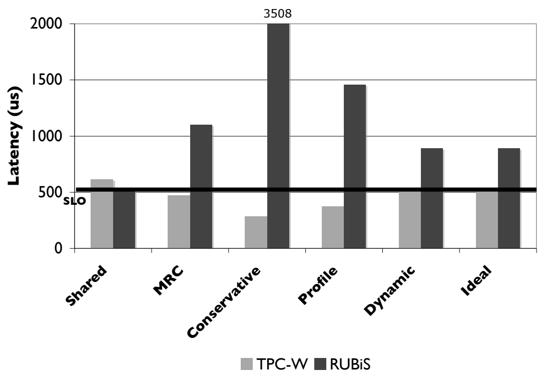
The gains provided by our DYNAMIC scheme are even more prominent when the provided latencies are mapped to the corresponding revenues, as shown in Figure 6(b). The figure also shows that with larger *slack*, we are able to further increase revenue. For instance, DYNAMIC increases the revenue from 43 (with 0% slack) to 87 (with 100% slack). This increase is achieved by reducing the RUBiS data access latency from 1193μs to 612μs. On the other hand, the PROFILE scheme is unable to take advantage of the slack to the same degree. For example, there is no additional revenue generated when the slack is 0%, and the revenue generated with larger slacks is significantly lower than the revenue generated using the DYNAMIC scheme.

6.4 Latency under LRU/DEMOTE

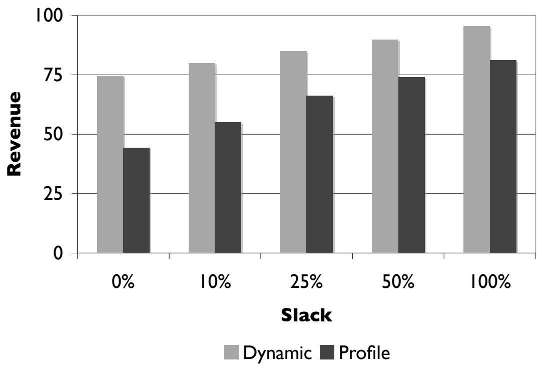
In Figure 7, we repeat the previous experiment using the LRU/DEMOTE scheme, where the database buffer pool informs the storage cache of block evictions, and the storage cache uses the DEMOTE cache replacement policy. The DEMOTE policy maintains *exclusiveness* between the database buffer pool and the storage cache. Thus, the DEMOTE scheme results in a better utilization of the overall cache hierarchy, leading to both TPC-W and RUBiS obtaining lower latencies even when in isolation, compared to the LRU/LRU case.

Under SHARED, both applications compete for the cache space, causing TPC-W to incur higher cache misses at both the buffer pool and the storage cache; this in turn leads to an average data access latency of 615μs for TPC-W, which is 23% higher than the pre-specified SLO. The fact that the best effort RUBiS is doing well under this scheme does not matter since the provider incurs substantial loss for violating TPC-W SLO.

For fairness of comparison, we modify the MRC algorithm to support DEMOTE policy and we use *our* modified MRC algorithm to allocate the memory to applications. Specifically, the MRC algorithm analytically derives the miss-ratio curve by tracking cache contents using an LRU stack. Upon a read/write request, it moves the accessed block to the



(a) Strict SLO



(b) Profit with varying Slack

Figure 7: LRU/DEMOTE

top of the LRU stack. In the presence of Demotes, to model the policy correctly, we modify MRC so that to place blocks referenced in a Demote request to the top of the LRU stack instead of blocks referenced in I/O reads. Finally, I/O writes are handled the same for both LRU and Demote cache policies. Under our modified MRC, the average data access latency for TPC-W is within the SLO, while the RUBiS latency is still higher than our DYNAMIC scheme (1095μs under MRC vs. 903μs under DYNAMIC).

While, the CONSERVATIVE and the PROFILE algorithms maintain TPC-W's latency within the SLO, they over-provision the cache resources, leading to high latencies for RUBiS, 3508μs for the CONSERVATIVE scheme and 1476μs for the PROFILE scheme. The PROFILE scheme allocates the buffer pool to TPC-W assuming that the storage cache provides no additional benefit. While this assumption is true for the LRU/LRU layout, it is false for the LRU/DEMOTE layout, where by using the Demote algorithm, the storage cache provides a significant benefit to TPC-W. Hence, the PROFILE scheme provides a data access latency of 376μs, even while profiling to meet the 500μs SLO.

By accurately modelling the effect of two-tier caching, our DYNAMIC scheme selects a near-optimal partitioning setting, where TPC-W is allocated enough buffer pool space such that it meets its SLO. With this allocation, the TPC-W latency is within the SLO and the RUBiS latency is 903μs.

6.5 Revenue under LRU/DEMOTE

With larger slack, as shown in Figure 7(b), we can further reduce RUBiS latency, from 903μs to 284μs, thereby increasing the revenue from 74 (with 0% slack) to 95 (with 100% slack). The PROFILE scheme also generates higher revenue compared to the LRU/LRU layout, due to higher utilization of the storage cache. However, the DYNAMIC scheme provides a higher revenue than the PROFILE scheme. Hence, integrating our cache partitioning DYNAMIC scheme with the DEMOTE coordinated cache replacement policy provides further revenue improvements that are not achievable using DEMOTE with other comparison schemes.

6.6 Performance under Overload Scenario

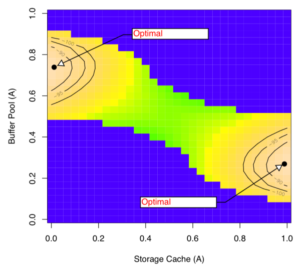
To better understand the improvement in performance achieved by DYNAMIC, we experiment with an overload scenario where two high-priority applications are scheduled. Specifically, we use two instances of the high-priority TPC-W application (denoted A and B) sharing the database and storage cache, thus creating an *overload* case, where the available resources are not sufficient to meet the SLO, given no slack. In this case, no additional revenue can be generated, and all schemes simply strive to minimize the losses. With two equally high priority applications, the CONSERVATIVE, PROFILE, and MRC schemes divide the database buffer pool and the storage cache equally (50/50) among the two TPC-W instances. Under the LRU/LRU layout, this leads to an average data access latency of 833μs, while our DYNAMIC scheme matches the IDEAL by obtaining an average data access latency of 743μs. DYNAMIC achieves this improvement by dynamically selecting an optimal cache configuration, which exploits the *inclusiveness* in LRU/LRU.

To provide an insight into the optimal partitioning, in Figure 8, we show the revenue function. The *x-axis* shows the fraction of the storage cache given to application A and the *y-axis* shows the fraction of the buffer pool given to application A. Since only two applications are running, Application B is given the remaining cache space. Figure 8 shows the *revenue* for different cache partitioning settings. The "low" revenue settings are shown in dark colors, and the "high" revenue is shown in light colors. The contour lines highlight the near-optimal settings. For example, as shown in Figure 8, the LRU/LRU layout has two optimal configurations. One optimal setting (top-left of the figure) is when Application A is given most of the buffer pool and a small fraction in the storage cache. The other optimal setting (bottom-right of the figure) is when Application B is given most of the buffer pool and very little of the storage cache.

If LRU/LRU is used, then the storage cache only provides a marginal benefit to the application given a large proportion of the database buffer pool. Thus, the plot shows that the optimal setting is achieved when the buffer pool is allocated to one application (A or B), and the storage cache allocated to the other (B or A). On the other hand, using the LRU/DEMOTE scheme, the storage cache benefits both applications equally leading to an optimal partitioning of 50/50 (Figure 8(b)).

6.7 Sampling Convergence

In Figure 9, we compare the speed of convergence of two sampling strategies: (1) greedy sampling and (2) random


(a) LRU/LRU

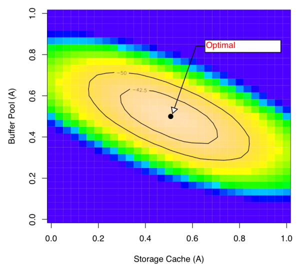

(b) LRU/DEMOTE

Figure 8: Overload: We show the total revenue for TPC-W/TPC-W for several configurations with the light regions showing "high" revenue and the dark regions showing "low" revenue. We also highlight the optimal cache partitioning settings.

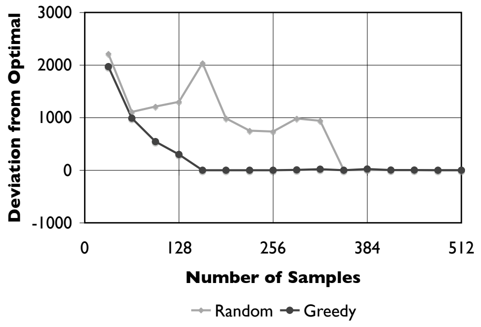


Figure 9: Comparison of Sampling Methods

sampling. In greedy sampling, we gather samples near the currently found optimal configuration. In random sampling, we select a set of random samples at each iteration. The benefit of greedy sampling is intuitively in potentially faster convergence towards the optimal cache configuration.

In Figure 9, the *x-axis* shows the number of samples selected for our statistical regression, and the *y-axis* shows the deviation from optimal. The *deviation from optimal* is the difference in revenue by using the estimated optimal partitioning, as opposed to the revenue generated using ideal cache partitioning. Initially, with a small number of samples, both the greedy approach and the random approach are far from optimal. However, after 64 samples, the greedy approach starts converging to the optimal, reached with only 160 samples (on average). On the other hand, the unguided random sampling converges only after 352 samples. Our sampling approach is efficient; we can collect 350 samples in simulation within 30 minutes. During this period of time, on average, two actuations take place, hence two experimental latency points are also collected.

6.8 Simulation Accuracy

We also evaluate the accuracy of our simulations by comparing the predicted latency with the measured latency when running in a specific configuration. In this experiment, we ran several configurations from small caches (64MB) to large caches (1GB). For each configuration, we compared the predicted latency obtained from simulation and the measured latency by running our prototype system. In all configurations the predicted latency is within 5% of the measured latency.

7. RELATED WORK

This section discusses previous work exploring different techniques for improving caching efficiency in the storage cache hierarchy.

The general area of adaptive cache management based on application patterns, or query classes has been extensively studied in database systems. For example, the DBMIN algorithm [10] uses the knowledge of the various patterns of queries to allocate buffer pool memory efficiently. The LRU-k [22], and its variant 2Q [15] cache replacement algorithms prevent useful buffer pages from being evicted due to sequential scans running concurrently. Brown et al. [4, 5] study schemes to ensure per-class response time goals in a system executing queries of multiple classes by sizing the different memory regions. Recently, IBM DB2 added the self-tuning memory manager (STMM) to size different memory regions [29]. However, the above works target only the memory regions within the DBMS. In our study, we have shown that optimally partitioning multi-tier caches results in significant performance gains.

Several works pass explicit hints from the client cache to the storage cache [25, 19, 7]. For example, these hints can indicate the reason behind a write block request to storage, and whether a block is about to be evicted from the client cache and should be cached at the storage level [19], explicit demotions of blocks from the storage client to server cache [32], or the relative importance of requested blocks [8]. These techniques modify the interface between the storage client and server, by requiring that an additional identifier be passed to the storage server. As opposed to our work, these techniques need thorough understanding of the application internals and changes to the kernel API and the storage protocol. For example, Li et al. [19] require the understanding of database system internals to distinguish the context surrounding each block I/O request. Similarly, Wong et. al [32] require the addition of a DEMOTE command to the SCSI protocol.

Transparent and gray-box techniques for storage cache optimization include inferring access patterns of the upper tier by observing characteristics of I/O requests [2, 9, 16], or using meta-data available at the file system layer. Schindler et al. investigate ways for providing the DBMS with more knowledge of the underlying storage characteristics [26]. The drawback of these techniques is that they are DBMS-specific or specific to the storage hardware. This may not be feasible in a data center.

8. CONCLUSIONS

In order to reduce the costs of management, power and cooling in large data centers, operators co-schedule several applications on each physical server of a server farm connected to a shared network attached storage. Determining and enforcing per-application resource quotas on the fly in this context poses a complex and challenging resource allocation and control problem due to i) the strict Quality of Service (QoS) requirements of many database applications,

ii) the unpredictable resource needs and/or access patterns of applications in modern environments with dynamic application co-scheduling and iii) the existing interdependency between tiers, such as the effects of cache replacement policies on application patterns at different levels.

Our contribution in this paper is to introduce a novel approach for controlling application interference in the cache hierarchy of shared server farms. Specifically, we design and implement a technique for partitioning the buffer pool and storage caches adaptively, online; a cache controller embedded into the DBMS actuates the partitioning of the two caches with the goal to dynamically converge towards a partitioning setting that minimizes the perceived application penalties. At the same time, the controller allocates any spare resources to best effort applications in order to maximize the revenue for the service provider.

Our method is implemented in a Linux-based prototype, called Gemini, which requires minimal DBMS instrumentation, and no changes to existing interfaces between commodity software and hardware components. Our experimental evaluation shows the effectiveness of our technique in enforcing application SLOs as well as maximizing the revenue of the service provider in shared server farms. In contrast, all other techniques we evaluated suffer from either violations of the SLO requirement of strict SLO applications, missed revenue opportunities, or both.

Acknowledgments

We would like to thank Daniel Lupei for helping with the experimental evaluation and Saeed Ghanbari for the discussions on statistical regression. Finally, we would like to thank the anonymous reviewers for their insightful feedback.

9. REFERENCES

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

[2] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Information and Control in Gray-Box Systems. In *SOSP*, 2001.

[3] P. T. Barham, B. Dragovic, K. Fraser, S. Hand, T. L. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP*, 2003.

[4] K. P. Brown, M. J. Carey, and M. Livny. Managing Memory to Meet Multiclass Workload Response Time Goals. In *VLDB*, 1993.

[5] K. P. Brown, M. J. Carey, and M. Livny. Goal-Oriented Buffer Management Revisited. In *SIGMOD*, 1996.

[6] D. Carney, U. Çetintemel, A. Rasin, S. B. Zdonik, M. Cherniack, and M. Stonebraker. Operator scheduling in a data stream manager. In *VLDB*, 2003.

[7] F. W. Chang and G. A. Gibson. Automatic I/O Hint Generation Through Speculative Execution. In *OSDI*, 1999.

[8] Z. Chen, Y. Zhang, Y. Zhou, H. Scott, and B. Schiefer. Empirical evaluation of multi-level buffer cache collaboration for storage systems. In *SIGMETRICS*, 2005.

[9] Z. Chen, Y. Zhou, and K. Li. Eviction-based Cache Placement for Storage Caches. In *USENIX Annual Technical Conference, General Track*, 2003.

[10] H.-T. Chou and D. J. DeWitt. An Evaluation of Buffer Management Strategies for Relational Database Systems. In *VLDB*, 1985.

[11] F. J. Corbat. A Paging Experiment with the Multics System. *MIT Press*, 1969.

[12] H. Drucker, C. J. C. Burges, L. Kaufman, A. J. Smola, and V. Vapnik. Support Vector Regression Machines. In *NIPS*, 1996.

[13] A. Gulati, A. Merchant, and P. J. Varman. pClock: an arrival curve based approach for QoS guarantees in shared storage systems. In *SIGMETRICS*, 2007.

[14] S. Jiang and X. Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *SIGMETRICS*, 2002.

[15] T. Johnson and D. Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *VLDB*, 1994.

[16] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Geiger: monitoring the buffer cache in a virtual machine environment. In *ASPLOS*, 2006.

[17] D. G. Kleinbaum, L. L. Kupper, A. Nizam, and K. E. Muller. *Applied Regression Analysis and Multivariable Methods (4th Edition)*. Duxbury Press, 2007.

[18] A. Labrinidis, H. Qu, and J. Xu. Quality contracts for real-time enterprises. In *BIRTE*, 2006.

[19] X. Li, A. Aboulnaga, K. Salem, A. Sachedina, and S. Gao. Second-Tier Cache Management Using Write Hints. In *FAST*, 2005.

[20] S. Liang, S. Jiang, and X. Zhang. STEP: Sequentiality and Thrashing Detection Based Prefetching to Improve Performance of Networked Storage Servers. In *ICDCS*, 2007.

[21] C. R. Lumb, A. Merchant, and G. A. Alvarez. Façade: Virtual Storage Devices with Performance Guarantees. In *FAST*, 2003.

[22] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K Page Replacement Algorithm For Database Disk Buffering. In *SIGMOD*, 1993.

[23] O. Ozmen, K. Salem, M. Uysal, and M. H. S. Attar. Storage workload estimation for database management systems. In *SIGMOD*, 2007.

[24] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive control of virtualized resources in utility computing environments. In *EuroSys*, 2007.

[25] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed Prefetching and Caching. In *SOSP*, 1995.

[26] J. Schindler, J. L. Griffin, C. R. Lumb, and G. R. Ganger. Track-aligned Extents: Matching Access Patterns to Disk Drive Characteristics. In *FAST*, 2002.

[27] K. Shen, T. Yang, L. Chu, J. Holliday, D. A. Kuschner, and H. Zhu. Neptune: Scalable Replication Management and Programming Support for Cluster-based Network Services. In *USITS*, 2001.

[28] G. Soundararajan, M. Mihailescu, and C. Amza. Context aware block prefetching at the storage server. In *USENIX*, 2008.

[29] A. J. Storm, C. Garcia-Arellano, S. Lightstone, Y. Diao, and M. Surendra. Adaptive Self-tuning Memory in DB2. In *VLDB*, 2006.

[30] A. Trossman. Virtualization capabilities of IBM Tivoli. In *1st Workshop on Virtualization and the Management of Information Services*, 2007.

[31] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger. Argon: performance insulation for shared storage servers. In *FAST*, 2007.

[32] T. M. Wong and J. Wilkes. My Cache or Yours? Making Storage More Exclusive. In *USENIX Annual Technical Conference, General Track*, 2002.

[33] G. Yadgar, M. Factor, and A. Schuster. Karma: know-it-all replacement for a multilevel cache. In *FAST*, 2007.

[34] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic tracking of page miss ratio curve for memory management. In *ASPLOS*, 2004.