Outlier Detection for Fine-grained Load Balancing in Database Clusters

Jin Chen†, Gokul Soundararajan⋆, Madalin Mihailescu†, Cristiana Amza⋆
†Department of Computer Science
⋆Department of Electrical and Computer Engineering
University of Toronto
{jinchen,madalin}@cs.toronto.edu
{gokul,amza}@eecg.toronto.edu

Abstract

Recent industry trends towards reducing the costs of ownership in large data centers emphasize the need for database system techniques for both automatic performance tuning and efficient resource usage. The goal is to host several database applications on a shared server farm, including scheduling multiple applications on the same physical server or even within a single database engine, while meeting each application's service level agreement.

Automatic provisioning of database servers to applications and virtualization techniques, such as, live virtual machine migration have been proposed as useful tools to address this problem.

In this paper we argue that by allocating entire server boxes and migrating entire application stacks in cases of server overload, these solutions are too coarse-grained for many overload situations. Hence, they may result in resource usage inefficiency, performance penalties, or both. We introduce an outlier detection algorithm which zooms in to the fine-grained query contexts which are most affected by an environment change and/or where a perceived overload problem is likely to originate from. We show that isolating these query contexts through either memory quota enforcements or fine-grained load balancing across different database replicas of their respective applications allows us to alleviate resource interference in many cases of overload.

1 Introduction

In this paper, we investigate fine-grained selective tuning of resource allocations to applications in the database back-end of large scale dynamic content web sites. Dynamic content web sites commonly use a three-tier architecture, consisting of a front-end web server, an application server

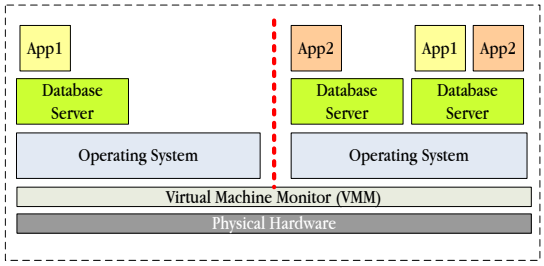


Figure 1. Multiple Services hosted using Xen

implementing the business logic of the site, and a back-end database server storing and querying the (dynamic) content of the site. The diverse nature and complex relationships among the components that make up such server systems and the multitude of tuning knobs for each component e.g., cache size, number of active connections, etc., make their performance-tuning difficult and time-consuming for humans.

Database self-management techniques in stand-alone database systems [6, 16], and provisioning of database servers to applications [22] have been recently introduced to address this problem. On the other hand, recent industry trends towards efficient resource usage through server consolidation e.g., for savings on power and cooling, have further increased the inherent difficulty of the resource allocation problem. In order to reduce the costs of ownership, large service providers now run many concurrent applications and efficiently multiplex their requests over existing hardware resources. As a result, server boxes may be dynamically allocated to applications from a shared resource pool and several concurrent applications may run on the same physical server. In these environments the use of Virtual Machine Monitors (VMMs), such as Xen [4], which virtualize the underlying hardware has recently increased in popularity. VMMs allow for security and failure isola-

tion between database applications placed in separate Virtual Machine containers (see Figure 1). Furthermore, live Virtual Machine (VM) migration [10] is a technique which allows online migration of entire application stacks from an overloaded server to alternate servers. In this paper we argue that previous database replica provisioning and live VM migration techniques are too coarse-grained for efficient load balancing of applications. Moreover, these existing coarse-grained provisioning solutions, even commercial ones such as IBM's Tivoli Intelligent Orchestrator, typically use very simple techniques, such as monitoring the CPU usage to trigger provisioning of server boxes.

With this paper, we take a first step towards accurate diagnosis and selective retuning for shared database clusters. We propose fine-grained adjustments of resource allocation to applications, which may be more appropriate for the most common load-balancing needs. We observe that the above mentioned shared hosting environment has a natural hierarchy in terms of both system and application structure. Each physical server can host several Virtual Machines (VM). A VM may host several database engines or a single database engine running multiple applications. Each database application in its turn typically has several logical structural levels. For example, a database application is formed of client interactions, which consist of database transactions. Finally, database transactions consist of queries. Coupled with this system and application hierarchy, the possible underlying causes of a perceived overload situation on any particular physical server may be at different levels of granularity. For example, the perceived overload at a particular server may be caused by i) a VM-level change, such as a change in the set of applications scheduled on the same VM or on the same physical server as a whole and the resulting resource interference, ii) an application-level change, such as an increase in the number of clients of a single application running within a VM container together with other applications or iii) a query-level change, such as an increase in the frequency of a particular resource-intensive query or a change in access patterns for a particular query. Examples of root causes for localized access pattern change are data evolution over time or an index change affecting only a few queries.

Load balancing through VM-migration is clearly overkill for most of these scenarios and may lead to lower resource usage, hence higher costs. Our contribution is a minimally intrusive approach to fine grained resource allocation taking into account both memory and CPU usage for addressing bottlenecks in shared database clusters.

Our hypothesis is that a localized change leading to an SLA violation typically produces outlier metric values, i.e., statistically remarkable points outside the normal stable state distribution. The rationale is that a change in a few queries can be so significant as to impact the SLA of one or more applications only if these queries clearly "stand-out" by being either: i) heavyweight in terms of some metrics, such as memory usage and with a moderate to large deviation from stable state metrics or ii) moderately heavyweight but showing a large deviation from stable state metrics.

Based on this observation, we use a fine-grained anomaly detection algorithm that allows us to pinpoint the overload problem to specific *query classes*, and to apply a focused solution to the problem. Within each DBMS, we perform lightweight on-line monitoring of several metrics, such as the buffer pool hit ratio, page accesses, etc., and we tie statistics collection to each query class. We use recorded statistics in order to determine query contexts which contain outlier measurements compared to the most recent *stable state* average values for the respective monitored metrics on the same server. A *stable state* record of average values for all metrics is made whenever the SLA is continuously met for an application during a measurement interval. For query classes which show outliers in terms of memory usage metrics, such as number of accessed pages or buffer pool misses, we investigate memory usage changes more precisely, by dynamically tracking the miss ratio curve (MRC) [24] in the buffer pool per query class. This allows us to focus our diagnosis on queries that might be the root cause of the problem by creating memory interference. Depending on our diagnosis of the outlier contexts, we either incrementally schedule problem query classes on a different replica or limit the memory quota allocation of these query classes while maintaining their placement.

If no outliers are detected or fine-grained resource allocation fails to solve the problem, we fall back on coarse-grained server provisioning and application isolation techniques. The rationale is that the cost of the more precise analysis of detailed metrics and placement reshuffling of *many* queries for near-optimal resource usage might outweigh any potential benefits from fine-grained load balancing; such algorithms would be more appropriate at initial application deployment or as periodic system maintenance.

Our technique does not imply any change to predefined interfaces or component structure and is transparent to the clients.

In our prototype implementation, we leverage our previous work [2, 22], on scheduler-based asynchronous replication schemes with strong consistency, which provides scaling and consistency in replicated database clusters.

A scheduling tier [22] can dynamically allocate disjoint server sets to applications and maintains consistency between the instances of an application allocated on different physical servers by a read-one-write-all technique. In this paper, we enhance our previous scheme by allowing multiple applications to be scheduled to execute on the same physical server or even within a single database engine. Furthermore, we implement and experiment with fine-grained

memory allocation and load balancing techniques based on outlier detection.

We present a preliminary evaluation of our techniques in different scenarios of dynamic change leading to SLA violations. In our experiments we use the industry-standard TPC-W e-commerce benchmark, which models an on-line book store, such as Amazon.com and the RUBiS on-line bidding benchmark modeled after eBay.com. In order to record the various metrics, we instrument the buffer pool management and other internal operations of the MySQL InnoDB database engine. We vary the load or access patterns of specific queries when running each application alone or both applications together. We show that investigating contexts with outlier measurements can lead to selective retuning solutions that inherently have a lower overhead and potentially better resource usage than coarse grained solutions.

The rest of this paper is organized as follows. Section 2 provides the necessary background on MRC-based dynamic memory allocation schemes. Section 3 introduces our fine-grained resource allocation and load balancing solution. Section 4 describes details of our prototype implementation, our benchmarks and methodology. Section 5 presents our results. Section 6 discusses related work. Section 7 concludes the paper.

2 Background: Miss Ratio Curve Tracking

The miss-ratio curve (MRC) of an application shows the page miss-ratios at various amounts of physical memory. This approach was first used in cache simulation and was recently proposed for dynamic memory management [24]. MRC reveals information about an application's memory demand, and can be used to predict the page miss ratio for an application given a particular amount of memory. The MRC can be computed dynamically through Mattson's Stack Algorithm[17]. The algorithm is based on the *inclusion property* that states that a memory of $k+1$ pages contains contents of a memory of k pages. The popular Least-Recently Used (LRU) algorithm exhibits the inclusion property thus allowing us to estimate the miss ratio for a given amount of memory m.

The LRU algorithm can be represented using a stack where the top of the stack contains the most recently accessed page and the bottom of the stack contains the least recently used page. A machine with physical memory of n pages would be represented by a stack of size n. In addition, we maintain an array of hit counters ($Hit[\,]$) which counts the number of page hits for each memory size. The data structures are updated as follows. For each memory reference (p) in a sequence of memory accesses, the page is searched in the stack. If the page is found at location (i) from the top of the stack, it indicates that a memory of size i would have kept page p in memory, and we increment the number of hits for memory size i by 1. If the page is not found, it indicates that either the page is accessed for the first time or a memory of n pages was not enough to keep this page in memory. In this case, we increment $Hit[\infty]$ by 1. We calculate the miss ratio using:

$$MR(m) \quad = \quad 1 - \frac{\sum_{i=1}^{m} Hit[i]}{\sum_{i=1}^{n} Hit[i] + Hit[\infty]} \qquad (1)$$

3 Design

We introduce a fine-grained resource allocation and load balancing algorithm in a replicated database system. We assume that several applications run concurrently on a shared database cluster. The optimizer's task is to dynamically provision replicas for applications and to schedule requests on those replicas in such a way to maintain the individual latency requirements of each application. Maintaining query latency under an average query latency bound is considered the service level agreement (SLA). We further assume that dynamic changes, such as load bursts, failures and query pattern changes can occur at any given time. However, we also assume that the frequency of dynamic changes is such that the system is able to achieve stable states where the SLA is continuously met for at least some of the supported applications at least during some measurement intervals.

3.1 Environment

Our dynamic content server architecture consists of the web/application and database server tiers (see Figure 2). Interposed between the application and the database tiers is a set of schedulers, one per application, that distribute incoming requests to a cluster of database replicas. Each machine in the database tier uses either Xen as a VMM and hosts one or more virtual machines or hosts one or more database systems directly on top of the native Linux OS. The scheduler tier contains a resource manager [22] responsible for dynamically allocating replicas for each application on physical servers.

The resource manager makes global replica allocation decisions across the different applications. Each scheduler is in charge of maintaining replica consistency between different replicas of a single application and for load balancing read-only queries among the set of replicas allocated for the corresponding application. We assume that the data of an application is fully replicated and kept consistent on all physical servers allocated to that application. Each scheduler, upon receiving a query from the application server, sends the query using a read-one, write-all replication scheme to the replica set allocated to its corresponding application.

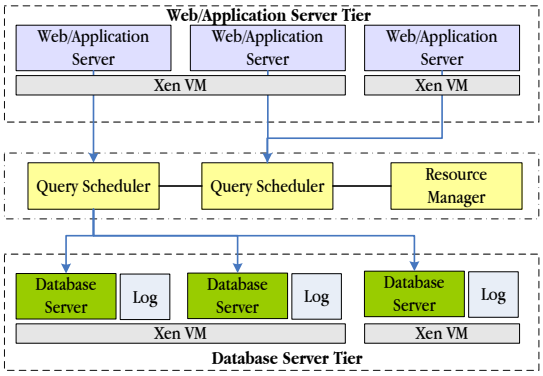


Figure 2. Cluster Architecture

The schedulers communicate with a set of decision managers, one per physical server. The decision managers in their turn use a set of log analyzers, one per database system running on their server to detect outlier contexts for applications running on their designated server.

3.2 Overview of Selective Retuning Algorithm

Our goal is to avoid coarse grained configuration retuning for entire applications in favor of selective fine-grained retuning. Our goal for selective retuning is to employ a potentially imprecise, but lightweight algorithm based on monitoring of a set of system and application counters.

Given an existing set of replicas for an application across a database cluster, scheduling and placement is done at the level of *query class* contexts. Our scheduling unit, a query class, consists of all query instances of an application with the same query template but different arguments. The scheduler determines the query templates of each application on the fly. Each query class is placed by the scheduler on a sub-set of replicas of its application and load balanced across these replicas.

For each physical server, we maintain a vector of metric averages measured during the last stable state for each query class of an application executing on that server. We differentiate *stable* versus *unstable* system behavior based on whether the SLA for the corresponding application as a whole was met (stable) or encountered one or more violations (unstable) during a measurement interval.

Upon an application-level SLA violation, we use a statistical approach to pinpoint the fine-grained application contexts most affected by the dynamic change. For each server where the application is running, we compare current measured metric values with stable metric values for each query class executing on the same server. We determine *outlier metrics* that are likely to be correlated with the high level problem by either having a wide variation compared to their stable state counterparts or by having a moderate variation in an otherwise heavyweight query for the particular metric. We define query contexts where outlier measurements occur as outlier contexts. We use outlier context detection as a guide for an incremental diagnosis and fine-grained resource allocation and load balancing solution as described next.

3.3 Statistics Collection

In order to guide system anomaly detection and correction, our system monitors several system, application-level and DBMS-level performance metrics. System metrics include CPU usage, I/O usage, and memory usage collected from `vmstat` on each physical server. Application-level metrics, such as average query latencies and average throughput, are tracked through the scheduler for SLA compliance checks. Within each DBMS, we perform lightweight on-line monitoring of several metrics, and we tie statistics collection to corresponding query class contexts. Specifically, we track the latency, throughput, buffer pool misses, the number of page accesses, the I/O block requests, the number of prefetch (read-ahead) requests, and a window of the most recent page accesses issued by the DBMS on behalf of the queries belonging to each specific query class.

Whenever a *stable* measurement interval occurs for an application, i.e., an interval when the SLA has been continuously met, we update the last stable value seen (as an average over the duration of the respective interval) for each metric on each server where the application is running. We maintain these average metrics in a data structure called a *stable state signature*; one such signature is maintained per query context.

We also maintain the parameters of the MRC curves for each query class in the *stable state* record for their corresponding application. The MRC is determined when a query class is first scheduled on the system and is not recomputed unless an SLA violation occurs and memory related counters show outlier measurements for that particular query class. We define two parameters for the MRC curve of a query class context: *total memory* needed and *acceptable memory* needed. The total memory needed is the smallest of the following two numbers: the maximum amount of memory on a physical server and the memory size for which the miss ratio is estimated to be 0. The miss ratio associated with the *total memory* needed is called the *ideal miss ratio* of the corresponding context. The *acceptable memory* needed is the estimated memory for which the miss ratio is within a fixed threshold of the *ideal miss ratio*. The *acceptable miss ratio* is the miss ratio associated with the *acceptable memory* needed.

3.3.1 Outlier Context Detection

The goal of outlier context detection upon an SLA violation is to determine query class contexts on each server running that application which show deviation from stable state behavior and are likely to have impacted application performance significantly.

We divide the current measured metrics by their last recorded average stable value. Then we multiply the result with the weight of the corresponding query in the application for this metric to obtain the weighted metric showing that metric's impact, called *metric impact* value. Weights are assigned *per metric* by normalizing each metric value to the least value across all queries for the same metric. For example, a query is assigned a high weight for the page access metric if its contribution to the total number of page accesses for all queries is high. Next, we use a classic outlier detection algorithm to find out the outliers for the weighted values of metrics. For each weighted metric, we determine the range of all current measured values for all query classes, and we define $Q1$ and $Q3$ to be the first and third quartile of this range. We define IQR to be the interquartile range $(Q3 - Q1)$, the range $[Q1 - 1.5 * IQR, Q3 + 1.5 * IQR]$ is defined as the inner fence, and the range $[Q1 - 3 * IQR, Q3 + 3 * IQR]$ is defined as the outer fence.

If a *metric impact* value is outside of the inner fence, it will be labeled a mild outlier; if it is outside the outer fence, it will be labeled an extreme outlier. We then consider as outlier query contexts, the query contexts which contain outlier *metric impact* values.

3.3.2 Memory Interference Detection and Alleviation

For each query class which shows outlier measurements in memory related counters e.g., miss ratio and page access counts, we classify the query class as a problem query class. We investigate further by triggering the recomputation of the MRC curve for each problem query class. If the parameters of the MRC curve show a significantly higher total memory need for the query class, then this query class is likely associated with memory interference, hence it continues to be suspect. This case includes new query classes that have been scheduled on the system and for which we have not computed the MRC curves before. We compute the MRC curves of these new queries and we classify them as problem query classes.

We have two options for addressing problem query classes that are suspected to cause memory interference: The first option is to schedule a suspect query class on a different replica. The second option is to limit the amount of buffer pool that the problem query class is allocated, by enforcing a fixed quota allocation for the respective query class, while maintaining the placement of the query on the same replica as before.

There are trade-offs between these two solutions. By placing the problem query class QC on a different replica, we incur the overheads of possibly having to create or update that replica, and of warming up the buffer pool for the respective query class on that replica. On the other hand, allocating a fixed memory quota for the buffer pool used by a given query class may decrease memory utilization on that machine. For example, the queries within the respective query class may not occur in the workload mix for a period of time or may have phases when they leave most of the QC quota underutilized. Fully exploring these trade-offs is beyond the scope of this paper. In this paper, we use a simple heuristic solution. For each physical server where changes in MRC have occurred, we determine if the current placement of query contexts can meet the *total memory need* of all query contexts. If this is not the case, we perform an iterative algorithm to determine if we can maintain the placement of each problem query class QC on that replica. The algorithm attempts to find a memory quota for each QC such that the miss ratios for all QC and the rest of the application queries scheduled on the same physical server, are predicted to be their respective *acceptable miss ratios* at the respective memory quotas by the MRC algorithm. If appropriate quotas can be found, we limit the memory usage for each problem query class while maintaining close to ideal performance. Otherwise, we reschedule the problem query class on an another replica. The same algorithm as above is used to determine the appropriate replica for placement.

If no outlier query contexts can be determined, we use similar algorithms as above on the top-k heavyweight queries in terms of memory metrics. Finally, if our fine-grained retuning action is ineffective, we fall back on the coarse grained allocation solutions. We allocate new replicas and isolate applications on them until all applications meet their SLAs.

3.3.3 Other Resource Interference Scenarios

For other cases of resource interference, such as CPU and I/O, we do not currently maintain a sufficient number of per-context metrics to enable accurate problem diagnosis through outlier detection. If CPU saturation on a server is detected as the root cause of an SLA violation, the scheduler reactively provisions more servers from the available pool for the set of query classes running on the overloaded server. In cases of I/O interference, a simple heuristic is to remove query contexts from the physical server where I/O interference occurs in decreasing order of their I/O rate until the perceived problem on that server is normalized.

4 Implementation Details and Methodology

We use two applications in our experimental evaluation: TPC-W and RUBiS. The TPC-W benchmark from the Transaction Processing Council (TPC) [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 TPC-W shopping mix workload with 20% writes which is considered the most representative e-commerce workload by the TPC.

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 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 have implemented the two applications on a dynamic content infrastructure consisting of the Apache web server, the PHP application server and the MySQL InnoDB (ver. 5.0.24) database engine. We run our applications use on a cluster of Dell PowerEdge servers with 4 Intel Xeon processors, running at 3.00 GHz. All servers use the Ubuntu Linux operating system with Linux kernel 2.6.27. We perform lightweight logging of different events by instrumenting MySQL. We assume an SLA in terms of average query latency per server of 1 second for all applications.

The fully automated parts of our prototype include provisioning for CPU saturation, scheduling based on query placement, lightweight logging of metrics, recording of metric values corresponding to stable states, outlier analysis and determination of outlier query contexts. For statistics collection, we instrumented the MySQL InnoDB database engine to record query level metrics with little or no overhead. To avoid locking overhead, we create a private logging buffer per thread. We log the specified counts, statistics and unique page accesses per query class. Finally, we flush the logs to disk only when the buffer is full or if the thread is being shutdown. While collecting the traces, there was no noticeable decrease in the throughput of the applications.

Other aspects of our prototype are automated only through off-line trace analysis although they could be used on-line as well. These include determination of MRC curves for query classes. Yet other aspects, such as detecting I/O interference, are poorly automated and still depend on our direct observation and interpretation of statistic collection.

5 Results

5.1 Overview

In this section, we present different performance degradation scenarios in our dynamic content server architecture and how they can be effectively mitigated using our fine-grained resource allocation and load balancing technique. We show a variety of resource interference scenarios which we believe representative of real-world shared platform scenarios. We also present the diagnosis and reaction mechanisms applicable in each case ranging from a fully automated reaction to CPU saturation to an administrator viewpoint observation in the I/O interference case.

In the first scenario, we show a workload change situation leading to automatic reconfiguration when only one application is running on the system. Second, we show the effect of interference in the buffer pool by an access pattern change in one query class. Third we show the memory interference due to buffer pool sharing by multiple applications within the same database system. Finally, we show the effect of I/O interference when two applications run within separate virtual machines on the same physical server. In all of these scenarios, we show the fine grained outlier detection and fine-grained resource allocation actions for alleviating the problem.

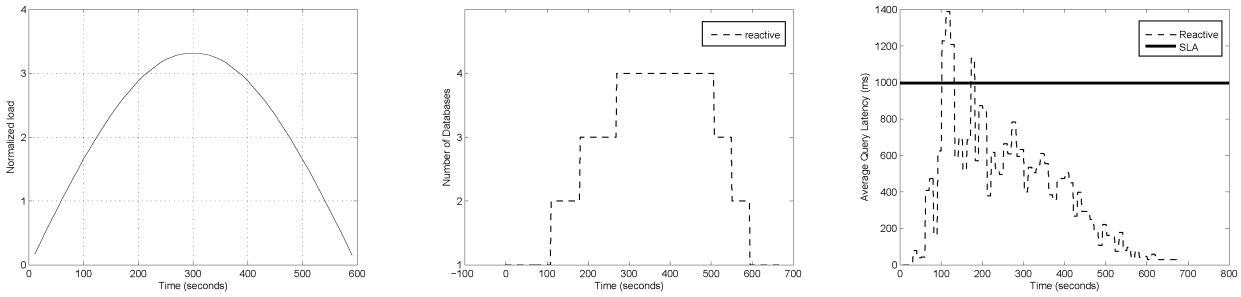
5.2 Alleviation of CPU Saturation

We use our TPC-W client emulator to emulate a sinusoid load function, shown in Figure 3(a). This function is in terms of the number of clients presented to the web server. In addition, the emulator adds some random noise on top of the load function by randomly varying the session time and thinking time of clients. When CPU usage is saturated, our reactive provisioning algorithm is triggered. The dynamic machine allocation is shown in Figure 3(b). All query classes of TPC-W are load balanced on an increasing set of replicas. Figure 3(c) shows that the average query latency drops back below the SLA after provisioning of a sufficient number of replicas.

5.3 Alleviation of Memory Interference due to Index Mis-configuration

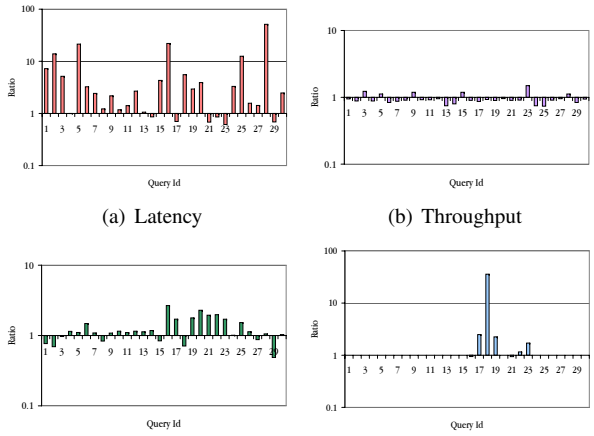
To show the effect of a localized change, we drop the `O_DATE` index from the TPC-W database configuration. This index is used only in the execution of the *BestSeller* query, which also uses two other indexes. After removing the index, the response time of TPC-W rises significantly from an average of 600ms to 2 seconds.

In Figure 4, we plot the ratios of the measured values of 4 metrics divided by their respective stable state average values. The problem has caused the overall latency to increase,



(a) Sine load function (b) Machine allocation (c) Average query latency

Figure 3. Alleviation of CPU Contention



(a) Latency (b) Throughput

(c) Misses (d) ReadAhead

Figure 4. Dropping the `O_DATE` Index

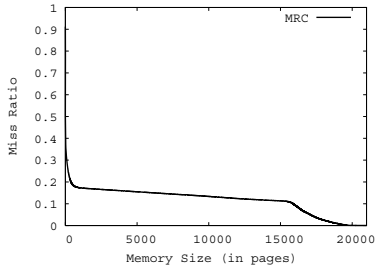


Figure 5. Miss Ratio Curve of *BestSeller* under Normal Configuration

the throughput to decrease, and the buffer pool misses to increase; but only a few queries have a sharp increase in read aheads. Indeed, this is expected as result of the increased

Hit Ratio (%)	**BestSeller**	**Non-BestSeller**
Shared Buffer	95.5	96.2
Partitioned Buffer	95.7	99.5
Exclusive Buffer	96.1	99.9

Table 1. Hit Ratio of Different Buffer Pool Management Algorithms

resource interference that the unoptimized *BestSeller* query is causing for all other queries on many metrics.

Outlier detection on the memory related counters, such as page accesses, page misses and read-ahead determines that 6 distinct query classes are mild outliers, including the *NewProducts*(#19)) and *BestSeller*(#18). To further narrow down the cause of the problem, we re-compute the MRC curves of these problem queries based on recent page accesses.

We find that only the *BestSeller* query class shows significant change in the total memory and acceptable memory. The MRC curve of the *BestSeller* without index has a longer tail than its index-based counterpart shown in Figure 5.3 as expected. We thus suspect that this query class is likely associated with the root cause of memory interference.

The new *BestSeller* query class has a flatter MRC curve, and thus the memory quota that it needs to meet its acceptable miss ratios is 3695 pages, less than the original 6982 pages for the index-based query class. We enforce a fixed quota allocation for this problem query class, while maintaining the placement of the query on the same replica as before.

We use a simulator of buffer pool management driven by traces of page accesses per query class to demonstrate the benefits of our algorithm. The buffer pool is divided into two dedicated partitions: one partition for servicing the *BestSeller* query class and the other partition for all other queries of the application.

We show the hit ratio in the buffer pool before buffer partitioning, after buffer partitioning and if each of the BestSellers and the non-BestSellers queries, respectively, were allocated the whole buffer pool (exclusively used buffer pool) in Table 1. The hit ratio under the exclusive use of the buffer pool are the ideal hit ratio that each set of queries can reach, respectively. They would also correspond to the hit ratio if we isolated the BestSeller query class on a different replica. For the partitioned buffer pool case, the *BestSeller* has almost the same hit ratio (95.7%) as running in a shared buffer pool (95.5%) or running in an exclusively used buffer pool (96.1%). With a partitioned buffer pool, the hit ratio for Non-BestSeller queries improves from 96.2% to 99.5%, reaching close to the ideal hit ratio of 99.9%. These results show that the partitioned buffer pool on a single replica has similar performance while using fewer physical machines than isolating the BestSeller and non-BestSeller parts of the application on two different replicas.

This example shows that our algorithm has the potential to automatically narrow the problem down to the one query which causes the performance degradation. We also show that fine grained targeted actions can minimize the performance impact, while maintaining good resource usage.

5.4 Alleviation of memory contention in a shared buffer pool

Placement		Latency	Throughput
		(s)	(WIPS)
TPC-W	IDLE	0.45	7.10
TPC-W	RUBiS	5.42	4.29
TPC-W	RUBiS*	1.27	6.44

Table 2. Effect of memory contention in a shared buffer pool. RUBiS* is the RUBiS workload with the *SearchItemsByRegion* scheduled on a different machine.

In this section, we study a scenario representative of an environment change where initially only TPC-W is running within a DBMS, but then the RUBiS workload is also started within the same DBMS in a shared buffer pool configuration.

Table 2 shows the baseline scenario where TPC-W is allocated the entire buffer pool. It also shows the case where TPC-W and RUBiS compete for space in the buffer pool. In all cases, the database instance is given 128MB buffer pool space, which corresponds to 8192 memory pages. With a shared buffer pool, TPC-W's throughput drops to 4.29 WIPS and its latency increases ten-fold to 5.42s compared to running alone.

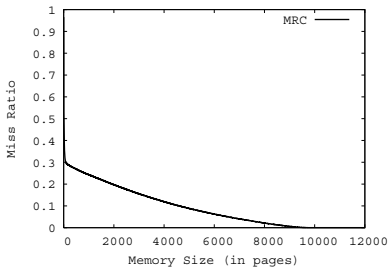


Figure 6. Miss Ratio Curve of RUBiS *SearchItemsByRegion* Query

In the absence of CPU saturation, we check memory related counters for TPC-W query classes, including page misses, page accesses and read-ahead requests. Five distinct TPC-W query classes are detected as mild outliers. However, after we recompute the MRC curve for each problem query class, we find that their total memory and acceptable memory parameters show no change. Hence these query classes themselves are not the cause of the performance anomaly.

We compute the MRC curves for the newly added RUBiS queries, considering them as potential problem query classes. Following our heuristic memory alleviation algorithm, we determine whether we can find a memory quota for each problem query class QC such that the miss ratios for both QC and the rest of the application queries on the same server are predicted to meet their respective acceptable ratios. The top ranking problem query is the RUBiS *SearchItemsByRegion*. Its MRC curve is shown in Figure 6, and its acceptable memory needed is around 7906 pages. It is clear that this query cannot be co-located with the TPC-W application in a shared 8192 pages buffer pool, since only the *BestSeller* of TPC-W needs at least 6982 pages.

Therefore, we schedule *SearchItemsByRegion* on a different replica. We show the result of this fine-grained load balancing change in the last row of Table 2. We see that after changing the placement of the *SearchItemsByRegion* query class, TPC-W recovers to a latency of 1.27 sec, and a throughput of 6.44 WIPS.

5.5 Alleviation of I/O contention among VM domains

While virtual machines provide strong isolation guarantees among different domains in terms of faults, security and privacy, they do not provide performance isolation. This is evident, for example, when the applications are I/O intensive. We perform an experiment, where two RUBiS in-

stances are placed on a machine running the Xen virtual machine monitor. We use two virtual machine domains and place a RUBiS instance in each domain.

Placement		Latency	Throughput
Domain-1	**Domain-2**	(s)	(WIPS)
RUBiS	IDLE	1.50	97.15
RUBiS	RUBiS	4.78	30.47
RUBiS	RUBiS*	1.50	95.37

Table 3. Effect of I/O contention among different domains.

We experiment with two different placements shown in the first two rows of Table 3: (1) a baseline where RUBiS is placed in domain-1 and domain-2 is kept idle and (2) a configuration with RUBiS on both domain-1 and domain-2. In this case each RUBiS application is running on its own separate data (as if two distinct applications were running on the system).

In both cases, we set the virtual memory of each domain to 256MB and set the MySQL buffer pool to 128M. We run 200 clients on each RUBiS instance. As Table 3 shows in our baseline case, the latency of RUBiS is 1.5 seconds and the throughput is 97 web interactions per second (WIPS). If two RUBiS instances are placed on the VMM, there is a significant drop in throughput (to 30 WIPS) and a sharp increase in latency (to 4.8s).

Our current techniques do not allow us to automate the diagnosis of this case. However, by analyzing the logs of statistics for the respective runs we observe that i) the CPU usage of each domain is low, ii) by plotting and analyzing the MRC curves of the two applications we determine that there is no memory interference and iii) RUBiS is an I/O intensive application. Thus, we suspect that the problem is due to I/O contention on domain-0 (the Xen controller) due to the I/O intensive nature of the two applications.

A naive solution is to schedule the virtual domains of the two applications on two different physical machine. However, our heuristic I/O interference alleviation algorithm can potentially be used in this case to reschedule only a subset of the queries on a different machine. Indeed, if we remove queries in decreasing order of their I/O rate, performance returns to normal. Specifically, the query associated with the *SearchItemsByRegion* web interaction in RUBiS contribute a large majority (87%) of the I/O accesses, and thus is the first query to remove. The third row of Table 3 shows the benefits with RUBiS on domain-1 and RUBiS* *(with SearchItemsByRegion removed)* on domain-2. We can see that after removing the heaviest query, the latency is 1.5s and the throughput is 95 WIPS, which is almost the same as the baseline case.

6 Related Work

The management and performance optimization of database systems by humans is becoming increasingly costly and time consuming [19]. Therefore, many related research efforts are studying methods for automating problem diagnosis through statistical machine learning approaches to find the root cause of performance problems [9, 11], to deal with software upgrades and operator mistakes [19], and to detect failures [8].

In order to track the cause of perceived problems and to understand the interactions between components in complex systems, several recent methods propose tagging requests as they pass through different layers [7] while others determine a causal order using a temporal join of various logs [3]. [11, 12] correlate various system metrics to service level objective (SLO) violations. They use a Tree-augmented Bayesian Network to determine a subset of metrics that correlate to a SLO violation. Furthermore, the authors generate signature of different metric combinations [12]. Using the signatures, they determine if the current situation is similar to a previously diagnosed problem. They show that determining a causal order between events can then be used to detect anomalies. Other papers [8] use statistical analysis of interactions between components to detect failures and system evolution.

A number of independent database replication solutions exist that provide both scaling and strong consistency in replicated database clusters [23, 15, 20, 14, 18]. With a few notable exceptions [14] [18] [13], these systems either do not investigate database replication in the context of dynamic adaptation or do not consider fine grain load balancing or allocation.

Our fine-grained memory allocation algorithm is related to commercial solutions for dynamic memory allocation, such as in DB2 9 (Viper) and to previous work [5] on partition sizing within the buffer pool to accommodate the working sets and ensure per-class response time goals for queries from multiple applications.

7 Conclusions and Future Work

Recent industry trends emphasize efficient resource usage through server consolidation for reducing the costs of ownership of large data centers. As a result, multiple applications are commonly multiplexed on the same hardware. Resource multiplexing may involve hosting multiple database systems within different virtual machines running on the same physical server or even hosting multiple database applications on a single database system. These complex environments make achieving differentiated service level agreements for applications challenging due to

the many potential sources of dynamic change. For example, the set of applications scheduled on any physical server, the database system configuration, the application workload mix, the client load of an application, or an application's access patterns may change over time. If the cause of performance degradation cannot be identified, the reaction to the problem will likely be inaccurate. For example, in order to address a perceived overload problem on a particular server, the system might overreach and isolate applications on different physical servers, which is wasteful in terms of resource usage.

We have designed and prototyped a technique, called outlier detection, for detecting fine-grained application contexts that are likely to contain either the cause of the problem, or be affected most by it. We have augmented a previous scheduling technique on replicated database clusters to isolate fine-grained application contexts pinpointed by the outlier detection technique. We have further used an existing memory usage tracking technique at the level of individual query classes to validate the outlier findings in terms of memory usage anomalies. We show that fine-grained memory allocation and load balancing based on outlier detection are effective methods for alleviating performance problems due to dynamic change within an application or resource interference between applications. Our technique is transparent to clients and has negligible overhead.

Finally, outlier detection is a promising approach for narrowing down the search for other system or application anomalies, such as invoking a query with the wrong arguments, lock contention or deadlock situations. We are planning to study such applications in our future work.

References

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

[2] C. Amza, A. L. Cox, and W. Zwaenepoel. A comparative evaluation of transparent scaling techniques for dynamic content servers. In *ICDE*, 2005.

[3] P. T. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *OSDI*, 2004.

[4] 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.

[5] K. P. Brown, M. J. Carey, and M. Livny. Goal-oriented buffer management revisited. In *SIGMOD*, 1996.

[6] N. Bruno and S. Chaudhuri. Automatic physical database tuning: A relaxation-based approach. In *SIGMOD*, 2005.

[7] A. Chanda, K. Elmeleegy, A. L. Cox, and W. Zwaenepoel. Causeway: Support for controlling and analyzing the execution of multi-tier applications. In *Middleware*, 2005.

[8] M. Y. Chen, A. Accardi, E. Kiciman, D. A. Patterson, A. Fox, and E. A. Brewer. Path-based failure and evolution management. In *NSDI*, 2004.

[9] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. A. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *DSN*, 2002.

[10] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *NSDI*, 2005.

[11] I. Cohen, J. S. Chase, M. Goldszmidt, T. Kelly, and J. Symons. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *OSDI*, 2004.

[12] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, indexing, clustering, and retrieving system history. In *SOSP*, 2005.

[13] S. Elnikety, S. Dropsho, and W. Zwaenepoel. Tashkent+: Memory-aware load balancing and update filtering in replicated databases. In *EuroSys'07(to appear)*, 2007.

[14] B. Kemme, A. Bartoli, and Ö. Babaoglu. Online reconfiguration in replicated databases based on group communication. In *DSN*, 2001.

[15] Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *SIGMOD*, 2005.

[16] V. Markl, G. M. Lohman, and V. Raman. Leo: An autonomic query optimizer for DB2. *IBM Systems Journal*, 42(1), 2003.

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

[18] J. M. Milán-Franco, R. Jiménez-Peris, M. Patiño-Martínez, and B. Kemme. Adaptive middleware for data replication. In *Middleware*, 2004.

[19] K. Nagaraja, F. Oliveira, R. Bianchini, R. P. Martin, and T. D. Nguyen. Understanding and dealing with operator mistakes in internet services. In *OSDI*, 2004.

[20] C. Plattner and G. Alonso. Ganymed: Scalable replication for transactional web applications. In *Middleware*, 2004.

[21] 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.

[22] G. Soundararajan, C. Amza, and A. Goel. Database replication policies for dynamic content applications. In *EuroSys'06*, 2006.

[23] S. Wu and B. Kemme. Postgres-r(si): Combining replica control with concurrency control based on snapshot isolation. In *ICDE*, 2005.

[24] 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-XI*, 2004.