Autonomic Provisioning of Backend Databases in Dynamic Content Web Servers

Jin Chen
Department of Computer Science
University of Toronto
Toronto, Canada
Email: jinchen@cs.toronto.edu

Gokul Soundararajan, Cristiana Amza
Department of Electrical and Computer Engineering
University of Toronto
Toronto, Canada
Email: {gokul, amza}@eecg.toronto.edu

***Abstract*— In autonomic provisioning, a resource manager allocates resources to an application, on-demand, e.g., during load spikes. Modelling-based approaches have proved very successful for provisioning the web and application server tiers in dynamic content servers. On the other hand, accurately modelling the behavior of the back-end database server tier is a daunting task. Hence, automated provisioning of database replicas has received comparatively less attention. This paper introduces a novel pro-active scheme based on the classic K-nearest-neighbors (KNN) machine learning approach for adding database replicas to application allocations in dynamic content web server clusters. Our KNN algorithm uses lightweight monitoring of essential system and application metrics in order to decide how many databases it should allocate to a given workload. Our pro-active algorithm also incorporates awareness of system stabilization periods after adaptation in order to improve prediction accuracy and avoid system oscillations. We compare this pro-active self-configuring scheme for scaling the database tier with a reactive scheme. Our experiments using the industry-standard TPC-W e-commerce benchmark demonstrate that the pro-active scheme is effective in reducing both the frequency and peak level of SLA violations compared to the reactive scheme. Furthermore, by augmenting the pro-active approach with awareness and tracking of system stabilization periods induced by adaptation in our replicated system, we effectively avoid oscillations in resource allocation.**

I. Introduction

Autonomic management of large-scale dynamic content servers has recently received growing attention [1], [2], [3], [4] due to the excessive personnel costs involved in managing these complex systems. This paper introduces a new pro-active resource allocation technique for the database back-end of dynamic content web sites.

Dynamic content servers commonly use a three-tier architecture (see Figure 1) that consists of a front-end web server tier, an application server tier that implements the business logic, and a back-end database tier that stores the dynamic content of the site. Gross hardware over-provisioning for each workload's estimated peak load can become infeasible in the short to medium term, even for large sites. Hence, it is important to efficiently utilize available resources through dynamic resource allocation, i.e., on-demand provisioning for all active applications. One such approach, the Tivoli on-demand business solutions [3], implements dynamic provisioning of resources within the stateless web server and application server tiers. However, dynamic resource allocation among applications within the stateful database tier, which commonly becomes the bottleneck [5], [6], has received comparatively less attention.

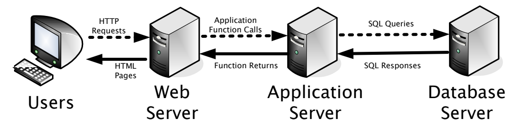


Fig. 1. Architecture of Dynamic Content Sites

Recent work suggests that fully-transparent, tier-independent provisioning solutions can be used in complex systems that contain persistent state such as the database tier as well [1], [4]. These solutions, similar to Tivoli, treat the system as a set of black boxes and simply add boxes to a workload's allocation based on queuing models [1], utility models [4], [7] or marketplace approaches [8]. In contrast, our insight in this paper is that for a stateful system, such as a database tier, off-line system training coupled with on-line system monitoring and tracking system stabilization after triggering an adaptation are key features for successful provisioning.

We build on our previous work [9] in the area of database provisioning. As in our previous work, our goal is to keep the average query latency for any particular workload under a predefined Service Level Agreement (SLA) value. Our previous work achieves this goal through a *reactive* solution [9], where a new database replica is allocated to a dynamic content workload in response to load or failure-induced SLA violations.

In this paper, we introduce a novel pro-active scheme that dynamically adds database replicas in advance of predicted need, while removing them in underload in order to optimize resource usage. Our pro-active scheme is based on a classic machine learning algorithm, K-nearest-neighbors (KNN), for predicting resource allocation needs for workloads. We use an adaptive filter to track load variations, and the KNN classifier to build a performance model of database clusters. The learning phase of KNN uses essential system and application metrics, such as, the average throughput, the average number

of active connections, the read to write query ratio, the CPU, I/O and memory usage system-level statistics. We train the performance model on these metrics during a variety of stable system states using different client loads and different numbers of database replicas. Correspondingly, our pro-active dynamic resource allocation mechanism uses active monitoring of the same database system and application metrics at run-time. Based on the predicted load information and the trained KNN classifier, the resource manager adapts on-line and allocates the number of databases that the application needs in the next time slot under varying load situations.

While pro-active provisioning of database replicas is appealing, it faces two inter-related challenges: 1) the unpredictable delay of adding replicas and 2) the instability of the system after triggering an adaptation. Adding a new database replica is a time-consuming operation because the database state of the new replica may be stale and must be brought up-to-date via data migration. In addition, the buffer cache at the replica needs to be warm before the replica can be used effectively. Thus, when adding a replica, the system metrics might show abnormal values during system stabilization e.g., due to the load imbalance between the old and newly added replicas. We show that a pro-active approach that disregards the needed period of system stabilization after adaptation induces system oscillations between rapidly adding and removing replicas. We incorporate awareness of system instability after adaptation into our allocation logic in order to avoid such oscillations. Some form of system instability detection based on simple on-line heuristics could be beneficial when incorporated even in a reactive provisioning technique [9]. On the other hand, our pro-active technique can detect and characterize periods of instability with high accuracy due to its system training approach. During training on a variety of system parameters, the system learns their normal ranges and the normal correlations between their values, resulting in more robust instability detection at run-time.

Our prototype implementation interposes an autonomic manager tier between the application server(s) and the database cluster. The autonomic manager tier consists of an autonomic manager component collaborating with a set of schedulers (one per application). Each scheduler is responsible for virtualizing the database cluster and for distributing the corresponding application's requests across the database servers within that workload's allocation.

We evaluate our pro-active versus reactive provisioning schemes with the shopping and browsing mix workloads of the TPC-W e-commerce benchmark, an industry-standard benchmark that models an online book store. Our results are as follows:

1) The pro-active scheme avoids most SLA violations under a variety of load scenarios.
2) By triggering adaptations earlier and by issuing several database additions in a batch, the pro-active scheme outperforms the reactive scheme, which adds databases incrementally and only upon SLA violations.
3) Our system is shown to be robust. First, our instability detection scheme based on learning avoids unnecessary oscillations in resource allocation. Second, our system adapts quite well to load variations when running a workload request mix different than the mix used during system training.

The rest of this paper is organized as follows. We first introduce the necessary background related to the KNN learning algorithm in section II. We then introduce our system architecture and give a brief overview of our dynamic replication environment in Section III. Then, we discuss our reactive and pro-active approaches in Section IV and Section V, respectively. Section VI describes our experimental testbed and benchmark. Section VII illustrates our results for applying the two approaches in the adaptation process of database clusters under different workload patterns. We compare our work to related work in Section VIII. Finally, we conclude the paper and discuss future work in Section IX.

II. Background

Classic analytic performance models [1], can predict whether or not a system will violate the SLA given information on future load. These modelling approaches are, however, not ameanable to our problem. This is due to the typically time consuming derivation of an analytic model for modelling complex concurrency control mechanisms such as the one in a replicated database system. Furthermore, in our complex system, the average query latency is not only related to the query arrival rate but is also related to the semantics of the query and the particular query workload mix.

Instead, we use the *k-nearest-neighbor* (KNN) classifier, a machine learning approach which considers multiple features in the system. KNN is an instance-based learning algorithm and has been widely applied in many areas such as text classification [10]. In KNN, a classification decision is made by using a majority vote of k "nearest" neighbors based on a similarity measure, as follows:

- For each target data set to be predicted the algorithm finds the k nearest neighbors of the training data set. The distance between two data points is regarded as a measure of their similarity. The Euclidean distance is often used for computing the distance between numerical attributes, also called as features.
- The distance we use in this paper is *weighted* Euclidean distance given by the following formula: $Dist(X,Y) = \sqrt{\sum_{i=1}^{N} weight_i * (x_i - y_i)^2}$ Here, X and Y represent two different data points, N denotes the number of features of the data, x_i, y_i denote the ith feature of X and Y respectively and $weight_i$ denotes the weight of our i^{th} feature. Each weight reflects the importance of the corresponding feature.
- Find the majority vote of the k nearest neighbors. The similarities of testing data to the k nearest neighbors are aggregated according to the class of the neighbors, and the testing data is assigned to the most similar class.

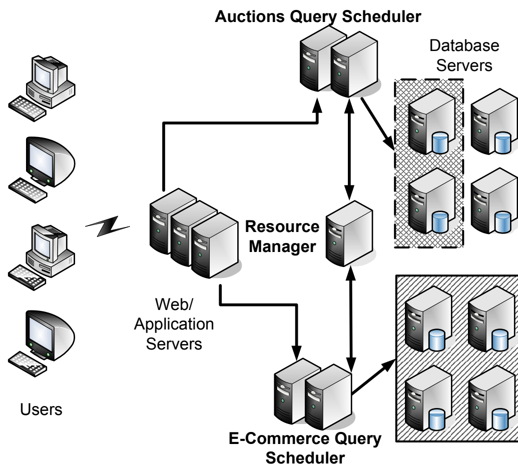


Fig. 2. Cluster Architecture

- Cross validation of the training data is an often used criterion that we also use to select the weights of the features and the number of "nearest" neighbors - K.

One of advantages of KNN is that it is well suited for problems with multi-modal classes (i.e., with objects whose independent variables have different characteristics for different subsets), and can lead to good accuracy for such problems. In KNN, trained models are implicitly defined by the stored training set and the observed attributes. Furthermore, KNN is robust to noisy training data and effective if the training data is sufficiently large. However, KNN's computation cost grows proportionately with the size of the training data, since we need to compute the distance of each target attribute to all training samples. Indexing techniques (e.g. K-D tree) can reduce this computational cost.

III. System Architecture

Figure 2 shows the architecture of our dynamic content server. In our system, a set of schedulers, one per application is interposed between the application and the database tiers. The scheduler tier distributes incoming requests to a cluster of database replicas. Each scheduler[1] 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 the application. The replica set is chosen by a resource manager that makes the replica allocation and mapping decisions across the different applications.

The scheduler uses our *Conflict-Aware* replication scheme [12] for achieving one-copy serializability [13] and scalability. With this scheme, each transaction explicitly declares the tables it is going to access and their access type. This information is used to detect conflicts between transactions and to assign the correct serialization order to these conflicting transactions. The transaction serialization order is expressed by the scheduler in terms of version numbers. The scheduler tags queries with the version numbers of the tables they need to read and sends them to the replicas. Each database replica keeps track of the local table versions as tables are updated. A query is held at each replica until the table versions match the versions tagged with the query. As an optimization, the scheduler also keeps track of versions of tables as they become available at each database replica and sends read-only queries to a single replica that already has the required versions. The scheduler communicates with a database proxy at each replica to implement replication. As a result, our implementation does not require changes to the application or the database tier.

[1]Each scheduler may itself be replicated for availability [11], [12].

Since database allocations to workloads can vary dynamically, each scheduler keeps track of the current *database set* allocated to its workload. The scheduler is also in charge of bringing a new replica up to date by a process we call *data migration* during which all missing updates are applied on that replica.

Our goals for resource management in our system are that the resource manager should be:

- ***Prompt.*** It should sense impending SLA violations accurately and quickly, and it should trigger resource allocation requests as soon as possible in order to deal with the expected load increase.
- ***Stable.*** It should avoid unnecessary oscillations between adding and removing database servers, because such oscillations waste resources.

A. Dynamic Replication

In this section, we provide an overview of the resource manager that implements dynamic replication and briefly introduce the replica addition, removal, mapping as well as data migration mechanisms in our system.

The resource manager makes the replica allocation and mapping decisions for each application based on its requirements and the current system state. The requirements are expressed in terms of a service level agreement (SLA) that consists of a latency requirement on the application's queries. The current system state includes the current performance of this application and the system capacity. The allocation decisions are communicated to the respective schedulers, which then allocate or remove replicas from their replica sets.

1) Replica Addition and Removal: The resource manager adds or removes a replica to/from an application allocation if it determines that the application is in overload or underload, respectively. Database replica removal needs to be performed conservatively because adding a database to a workload has high overheads. The replica addition process consists of two phases: data migration and system stabilization (see Figure 3). Data migration involves applying logs of missing updates on the new replica to bring it up-to-date. System stabilization involves load balancing and warmup of the buffer pool on the new replica. While some of these stages may overlap, replica addition can introduce a long period over which query latencies are high.

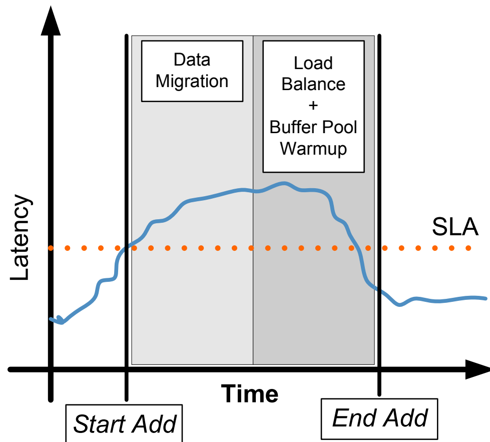


Fig. 3. Latency instability during replica addition.

2) Potential for Oscillations in Allocation: Oscillations in database allocations to workloads may occur during system instability induced by adaptations or rapidly fluctuating load. Assume an adaptation is necessary due to a burst in client traffic. Since our database scheduler cannot directly measure the number of clients, it infers the load by monitoring various system metrics instead. In the simplest case, the scheduler infers the need to adapt due to an actual latency SLA violation. However, *during* the adaptation phases, i.e., data migration, buffer pool warmup and load stabilization, the latency will be high or may even temporarily continue to increase as shown in Figure 3. Latency sampling during this potentially long time is thus not necessarily reflective of a continued increase in load, but of system instability after an adaptation is triggered. If the system takes further decisions based on sampling latency during the stabilization time, it may continue to add further replicas which are unnecessary, hence will need to be removed later. This is an oscillation in allocation which carries performance penalties for other applications running on the system due to potential interference.

A similar argument may hold for other system metrics measured during adaptation. Their values will not be indicative of any steady-state system configuration even if the load presented to the system remains unchanged. While rapid load fluctuations may induce similar behavior, simple smoothing or filtering techniques can offer some protection to very brief load spikes. While all schemes presented in this paper use some form of smoothing or filtering, which can dampen brief load fluctuations, our emphasys is on avoiding tuning of any system parameter, including smoothing coeficients. Instead we develop techniques for automatically avoiding all cases of allocation oscillation caused by system metric instability.

3) Replica Mapping: Dynamic replication presents an inherent trade-off between minimizing application interference by keeping replica sets disjoint versus speeding replica addition by allowing overlapping replica sets. In this paper, we use warm migration where partial overlap between replica sets of different applications is allowed. Each application is assigned a disjoint primary replica set. However, write queries of an application are also periodically sent to a second set of replicas. This second set may overlap with the primary replica set of other applications. The resource manager sends batched updates to the replicas in the secondary set to ensure that they are within a staleness bound, which is equal to the batch size or the number of queries in the batch. The secondary replicas are an overflow pool that allow adding replicas rapidly in response to temporary load spikes since data migrating onto these replicas is expected to be a fast operation.

4) Data Migration: In this section, we describe the implementation of data migration in our system. Our data migration algorithm is designed to bring the joining database replica up to date with minimal disruption of transaction processing on existing replicas in the workload's allocation.

Each scheduler maintains persistent logs for all write queries of past transactions in its serviced workload for the purposes of enabling dynamic data replication. The scheduler logs the queries corresponding to each update transaction and their version numbers at transaction commit. The write logs are maintained per table in order of the version numbers for the corresponding write queries.

During data migration for bringing a stale database replica up-to-date, the scheduler replays on it all missing updates from its on-disk update logs. The challenge for implementing an effective data migration is that new transactions continue to update the databases in the workload's allocation while data migration is taking place. Hence, the scheduler needs to add the new database replica to its workload's replica mapping *before* the end of data migration. Otherwise, the new replica would never catch up. Unfortunately, new updates cannot be directly applied to the (stale) replica before migration completes. Hence, new update queries are kept in the new replica's holding queues during migration. In order to control the size of these holding queues, the scheduler executes data migration in *stages*. In each stage, the scheduler reads a batch of old updates from its disk logs and transfers them to the new replica for replay without sending any new queries. Except for pathological cases, such as sudden write-intensive load spikes, this approach reduces the number of disk log updates to be sent after each stage, until the remaining log to be replayed falls below a threshold bound. During this last stage, the scheduler starts to send new updates to the replica being added, in parallel with the last batch of update queries from disk.

IV. Reactive Replica Provisioning

Figure 4, shows the replica allocation logic and the conditions for triggering an addition of a database replica to an application and for removing a replica from an application. In the following we describe these adaptations in detail.

A. Reactive Replica Addition

In the `Steady State`, the resource manager monitors the average latency received from each workload scheduler

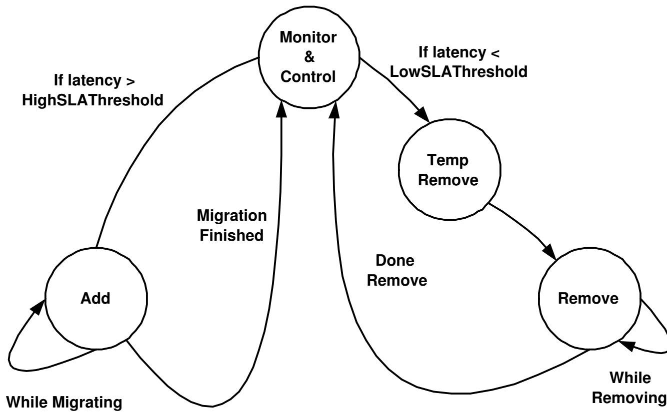


Fig. 4. Reactive Replication Provisioning Logic

during each sampling period. The resource manager uses a smoothened latency average computed as an exponentially weighted mean of the form $WL = \alpha \times L + (1 - \alpha) \times WL$, where L is the current query latency. The larger the value of the α parameter, the more responsive the average to current latency.

If the average latency over the past sampling interval for a particular workload exceeds the `HighSLAThreshold`, hence an SLA violation is imminent, the resource manager places a request to add a database to that workload's allocation.

B. Reactive Replica Removal

If the average latency is below a `LowSLAThreshold`, the resource manager triggers a replica removal. The right branch of Figure 4 shows that the removal path is conservative and involves a tentative remove state before the replica is finally removed from an application's allocation. The allocation algorithm enters the tentative remove state when the average latency is below the low threshold. In the tentative remove state, a replica continues to be updated, but is not used for load balancing read queries for that workload. If the application's average latency remains below the low threshold for a period of time, the replica is removed from the allocation for that workload. This two-step process avoids system instability by ensuring that the application is indeed in underload, since a mistake during removal would soon require replica addition, which is expensive. For a forced remove during overload, we skip the tentative removal state and go directly to the removal state. In either case, the database replica is removed from a application's replica set only when ongoing transactions finish at that replica.

C. Enhancement of Reactive Approach with System Instability Detection

The resource manager makes several modifications to this basic allocation algorithm in order to account for replica addition delay and protect against oscillations. First, it stops making allocation decisions based on sampling query latency until the completion of the replica addition process. Completion includes both data migration and system stabilization

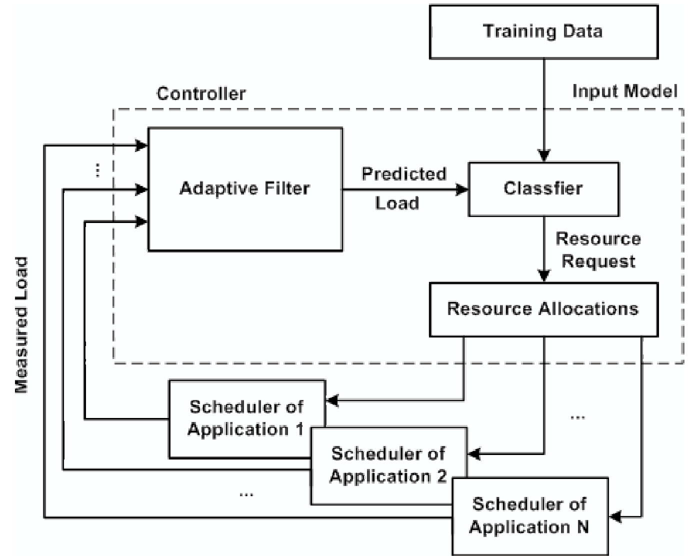


Fig. 5. Pro-active Replication Provisioning Scheme

phases. The scheduler uses a simple heuristic for determining when the system has stabilized after a replica addition. In particular, it waits, for a *bounded* period of time, for the average query latency at the new replica to become close (within a configurable `ImbalanceThreshold` value) to the average query latency at the old replicas. Finally, since this wait time can be long and can impact reaction to steep load bursts, the resource manager uses the query latency at the new replica in order to improve its responsiveness. Since this replica has little load when it is added, we use its average latency exceeding the high threshold as an early indication of a need for even more replicas for that application. The resource manager triggers an extra replica addition in this case.

V. Pro-active Replication Provisioning

In our pro-active approach, the controller predicts performance metrics and takes actions in advance of need to add databases such that the SLA is not violated while resources are used close to optimally.

A. Overview

Our approach can be summarized as follows. *We predict the status of the application in the next time interval and classify it into two categories: SLA violation or within SLA for a given number of database servers*. By iterating through all possible database set configurations, we decide the minimum size database set that is predicted to have no SLA violations.

In more detail, Figure 5 shows the main process of our pro-active provisioning scheme. The scheduler of each application works as the application performance monitor and is responsible for collecting various system load metrics and reporting these measured data to the global resource manager (controller). The controller consists of three main components. First, the adaptive filter predicts the future load based on the current measured load information. Next, the classifier finds

out whether the SLA is broken given the predicted load and the number of active databases. The classifier directs the resource allocation component to adjust the number of databases to the proper number of databases for this application according to its prediction. The resource allocation component decides how to map this request onto the real database servers by considering the requests of all applications and the available system capacity.

B. Enhancement of Pro-active Approach with System Instability Detection

Our classifier is trained under stable states; our training data is gathered for several constant loads. When the system triggers an adaptation to add a new database or remove a database from a workload's allocation, the system goes through a transitional (unstable) state. At this time, the system metric values are quite different from the ones measured during stable system states that we use as the training data. As a result many wrong decisions could be made if the classifier uses the system metrics sampled during a period of instability. This could in its turn lead to oscillations between rapidly adding and removing database replicas, hence unnecessary adaptation overheads and resource waste.

To overcome this unstable phenomenon, we enhance our pro-active provisioning algorithm to be aware of the instability after adaptation. The system automatically detects unstable states by using two indicators: the load imbalance ratio and the ratio of average query throughput versus the average number of active connection. During our training process, we record the normal range of variations of the load imbalance ratio among different databases. If the measured ratio is beyond the normal range, we decide that the system is in an unstable state. Second, we select the two features assigned the highest weights during our training phase, which, as we will show, are the throughput and number of connections. If we observe that the ratio of the two metrics is beyond the normal range, we also decide that the system is in an unstable state.

In our approach, if the system is detected to be in an unstable state, we suppress taking any decisions until the system is stable.

C. Implementation Details

There are various filter and classifier algorithms that can fit into our pro-active replication provisioning scheme. We take the ease of the implementation and their promptness as our selection principles.

1) Filter: We use a critically damped g-h filter [14] to track the variations of our load metrics. This filter minimizes the sum of the weighted errors with the decreasing weights as the data gets older; i.e., the older the error, the less it matters. It does this by weighting the most recent error by unity, the next most recent error by a factor Θ (where $\Theta < 1$), the next error by Θ^2, the next error by Θ^3, and so on.

The filtering equations are as follows:

$$\dot{x}^*_{n+1,n} = \dot{x}^*_{n,n-1} + \frac{h_n}{T}(y_n - x^*_{n,n-1}),$$

$$x^*_{n+1,n} = x^*_{n,n-1} + T\dot{x}^*_{n+1,n} + g_n(y_n - x^*_{n,n-1}),$$

The equations provide an update estimate of the predicted gradient $\dot{x}^*_{n+1,n}$ and predicted value $x^*_{n+1,n}$. T denotes the sampling interval, x is the metric we are interested in, y is its corresponding measured value and $\dot{x}$ denotes the gradient of x. The first subscript is used to indicate the estimated time and the second subscript is used to indicate the last measurement time. Thus, $x^*_{n+1,n}$ denotes an estimate of x during the next time slot $n+1$ based on the measurements made at current time n and before.

The parameters g and h are related to Θ by the formulas $g = 1-\Theta^2, h = (1-\Theta)^2$, where Θ is decided by the standard deviation of the measurement error and the desired prediction error.

This filter uses a simple recursive equation calculation for the constants g and h. Thus it is fast in tracking speed compared to more advanced adaptive filters, such as extended Kalman Filters [14].

2) KNN Classifier: We select several application metrics and system metrics as the features used by our KNN classifier in order to enhance our confidence about the automatically inferred load information. These system metric are readily available from our environment. They are as follows:

1) Average query throughput - denotes the number of queries completed during a measurement interval.
2) Average number of active connections - counts the average number of active connections as detected by our event-driven scheduler within its select loop. An active connection is a connection used to deliver one or more queries from application servers to the scheduler during a measurement interval.
3) Read write ratio - shows the ratio of read queries versus write queries during a measurement interval. This feature reflects the workload mix.
4) Lock ratio - shows the ratio of locks held versus total queries during a measurement interval.
5) CPU, Memory and I/O usage reported by `vmstat`.

Metrics 1 to 4 are gathered by the scheduler of each application. The traditional system metrics in 5 are measured on each database server.

Although a single metric may reflect the load information to some degree, basing decisions on a single metric could be seriously skewed e.g, if the composition of queries in the mix changes or if the system states are not fully reproducible. We use cross validation techniques in KNN to identify the importance (i.e., weight) of each metric. The weights are automatically determined and they reflect the usefulness of features. Not all features are always useful during on-line load situations. Their usefulness depends on the characteristics of the current workload mix. For example, for a CPU-bound workload mix, I/O metrics will be irrelevant for the purposes of load estimation.

During our training phase, we run 10-fold cross validation for weight combinations varying within a finite range, and pick the weight setting whose accuracy is higher than our

target accuracy (95%) or achieves the highest accuracy in the given range. We can continue to expand the search space by gradient methods until we achieve a good accuracy. This training process assigns weights for all features off-line. Less important features automatically get lower weights.

VI. Experimental Setup

To evaluate our system, we use the same hardware for all machines running the client emulator, the web servers, the schedulers and the database engines. Each is a dual AMD Athlon MP 2600+ computer with 512MB of RAM and 2.1GHz CPU. All the machines use the RedHat Fedora Linux operating system. All nodes are connected through 100Mbps Ethernet LAN.

We run the TPC-W benchmark that is described in more detail below. It is implemented using three popular open source software packages: the Apache web server [15], the PHP web-scripting/application development language [16] and the MySQL database server [17]. We use Apache 1.3.31 web-servers that run the PHP implementation of the business logic of the TPC-W benchmark. We use MySQL 4.0 with InnoDB tables as the database backend.

All experimental numbers are obtained running an implementation of our dynamic content server on a cluster of 8 database server machines. We use a number of web server machines sufficient for the web server stage not to be the bottleneck. The largest number of web server machines used for any experiment is 3. We use one scheduler and one resource manager. The thresholds we use in the reactive experiments are a HighSLAThreshold of 600ms and a LowSLAThreshold of 200ms. The SLA threshold used in our pro-active approach is 600ms. The SLA threshold was chosen conservatively to guarantee an end-to-end latency at the client of at most 1 second for the TPC-W workload. We use a latency sampling interval of 10 seconds for the scheduler.

A. TPC-W E-Commerce Benchmark

The TPC-W benchmark from the Transaction Processing Council [18] is a transactional web benchmark designed for evaluating e-commerce systems. Several interactions are used to simulate the activity of a retail store such as Amazon. 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.

The inventory images, totaling 1.8 GB, are resident on the web server. We implemented the 14 different interactions specified in the TPC-W benchmark specification. Of the 14 scripts, 6 are read-only, while 8 cause the database to be updated. Read-write interactions include user registration, updates of the shopping cart, two order-placement interactions, two involving order inquiry and display, and two involving administrative tasks. We use the same distribution of script execution as specified in TPC-W. In particular, we use the TPC-W shopping mix workload with 20% writes which is considered the most representative e-commerce workload by the Transactional Processing Council. The complexity of the interactions varies widely, with interactions taking between 20 ms and 1 second on an unloaded machine. Read-only interactions consist mostly of complex read queries in auto-commit mode. These queries are up to 30 times more heavyweight than read-write transactions.

B. Client Emulator

To induce load on the system, we have implemented a session emulator for the TPC-W benchmark. A session is a sequence of interactions for the same customer. For each customer session, the client emulator opens a persistent HTTP connection to the web server and closes it at the end of the session. Each emulated client waits for a certain think time before initiating the next interaction. The next interaction is determined by a state transition matrix that specifies the probability of going from one interaction to another. The session time and think time are generated from a random distribution with a given mean. For each experiment, we use a load function according to which we vary the number of clients over time. However, the number of active clients at any given point in time may be different from the actual load function value at that time, due to the random distribution of per-client think time and session length. For ease of representing load functions, in our experiments, we plot the input load function normalized to a baseline load.

VII. Experimental Results

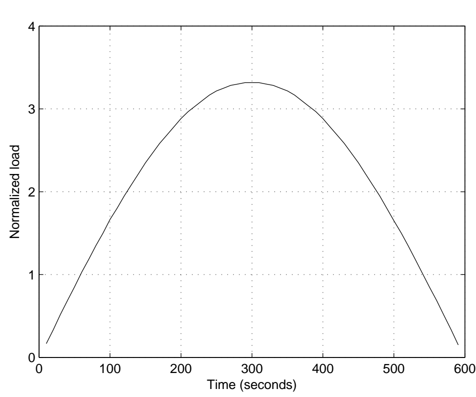
A. System Training

In this section, we describe our training phase and its effect on the assigned weights for our pre-selected system features.

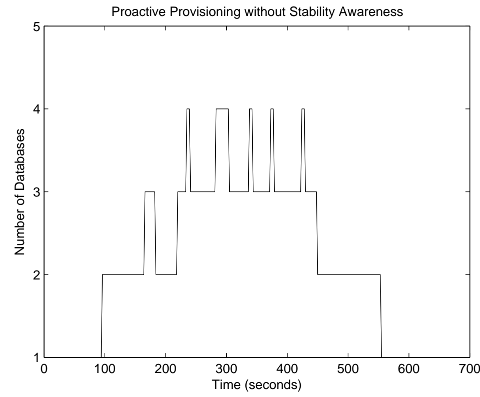
We train our system on the TPC-W shopping mix with database configurations of 1 through 8 replicas and client loads from 30 to 220 clients under stable states. The weights of features in the TPC-W shopping mix obtained from the training phase on this mix are listed here in the order of importance: the average number of active connections, the average query throughput, the read/write ratio, the CPU usage, the Lock ratio, the memory usage and the I/O usage. The TPC-W shopping mix has significant locality of access. Hence, it is not an I/O intense workload. This explains the low relevance of I/O usage. Furthermore, the MySQL database management system does not free the memory pages for TPC-W even if it is in under-load, so memory usage also has low relevance for inferring the load level. On the other hand, contrary to our intuition, the lock ratio does not show a high association with the load level. The lock ratio could, however, show higher relevance for larger cluster configurations.

B. Pro-active Approach without Stability Awareness

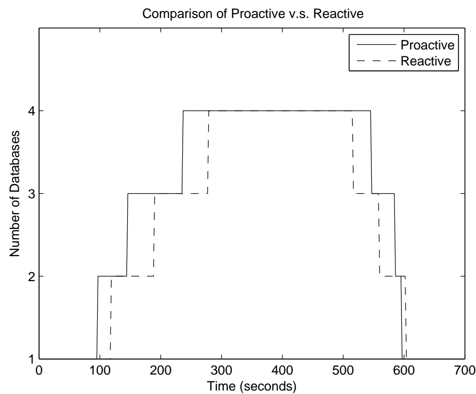
In this section, we show the influence of system metric instability during adaptation. Figure 7(b) shows an example of such oscillations that happen under the continuous load function shown in Figure 7(a). The oscillations happen due to (incorrect) adaptation decisions taken during system instability



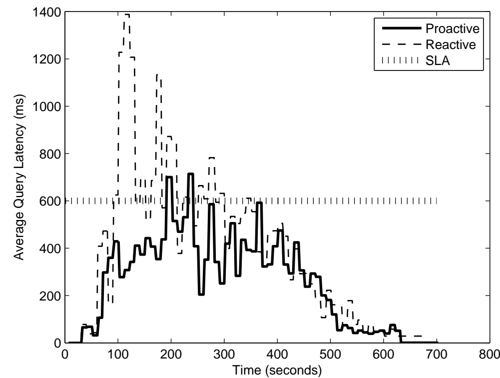
(a) Sine load function



(b) Average query latency for Pro-active without Stability



(c) Machine allocation



(d) Average query latency for Pro-active versus Reactive

Fig. 7. Scenario with sine load

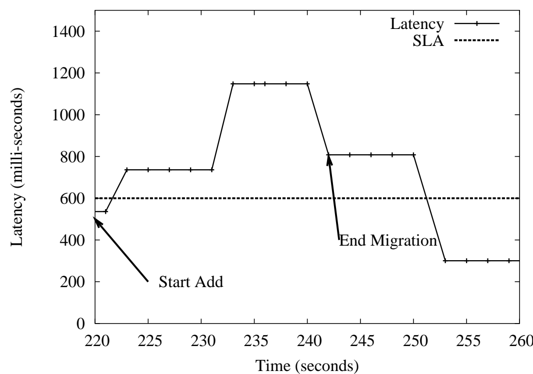


Fig. 6. Latency variation while the system adapts from 2 to 3 databases.

TABLE I
AN ILLUSTRATION OF INSTABILITY

No. of db (time)	Throughput	Avg. Active Conn.
Just before addition		
2db (t1)	80.80	118
After addition		
3db (t2)	142.20	105
3db (t3)	167.10	89
3db (t4)	211.40	70
In stable state		
3db	170	70

when system metrics are varying wildly immediately after an adaptation.

In order to explain the oscillations, Figure 6 zooms into a small time period of the previous adaptation graph (between times 220 and 260 seconds of the experiment) to illustrate the variation of latency during an adaptation. It shows the latency pattern when the system adapts from a configuration of 2 databases to a configuration of 3 databases i.e., one database addition. We can see that the latency initially increases during the data migration phase, until End Migration, then stays flat while the buffer pool warms up on the new replica and finally decreases and stabilizes thereafter.

In addition, table I shows the variation of our highest weighted system metrics, the number of active incoming connections and the throughput just before and just after adding the new database (during 4 time steps). The table also shows the corresponding stable state values in the target configuration. During stable states, the average number of active connections at the scheduler is closely correlated with the total load on the system induced by active clients. In contrast, as we can see from the table, the number of active connections might register a sudden decrease immediately after adding a new database even while the throughput increases. These effects are due to the various factors at play during system stabilization. For example, as the new replica gets more requests, the overload on existing replicas starts to normalize and many client requests that were delayed due to overload finally complete. As a result, a larger than usual number of clients may get their responses at this time. These clients will be in the thinking state during the next interval explaining the lower number of active incoming conections after adaptation.

These abnormal system metric variations after an adaptation may induce wrong load and latency estimates or predictions. For example, our KNN predictor might interpret the low number of active incoming connections as underload. Wrong decisions, such as removing a database immediately after adding it or vice versa may result. In the rest of the experiments, our KNN-based prediction algorithm is enhanced to suppress taking decisions during intervals when system instability is detected.

C. Performance Comparison of the Pro-active and Reactive Approaches

In the following, we evaluate the two autonomic provisioning approaches: reactive and pro-active. We first consider a scenario with continuously changing load, where the load variation follows a sinusoid (sine) function. Then we consider a sudden change scenario with a large load spike.

1) Load with Continuous Change Scenario: We use our client emulator to emulate a sinusoid load function, shown in Figure 7(a). As we can see from Figure 7(c), the pro-active approach triggers replica adding actions earlier than the reactive approach, because it performs future load prediction. In contrast, the pro-active removal is slightly slower than the reactive database removal because we use a conservative decision regarding when the system is sufficiently stable to accurately decide on removal. Figure 7(d) shows the comparison of average query latency for these two approaches. As a result of the earlier resource adaptations of the pro-active provisioning, this approach registers fewer SLA violations than the reactive approach. Furthermore, the degree of SLA violations, reflected in the average latency peaks, is also reduced compared to the reactive approach.

2) Sudden Load Spike Scenario: We use our client emulator to emulate a load function with a sudden spike, shown in Figure 8(a).

From Figure 8(c), we see that neither scheme can avoid SLA violations when the load jump happens, since the change is too abrupt to predict. However, the pro-active provisioning has a lower SLA violation peak and duration than the reactive provisioning approach. Specifically, the pro-active approach reaches a query latency peak of 2 seconds, and the SLA violations last around 50 seconds while the reactive approach reaches a query latency peak of 5 seconds, and its corresponding SLA violations last more than 2 minutes.

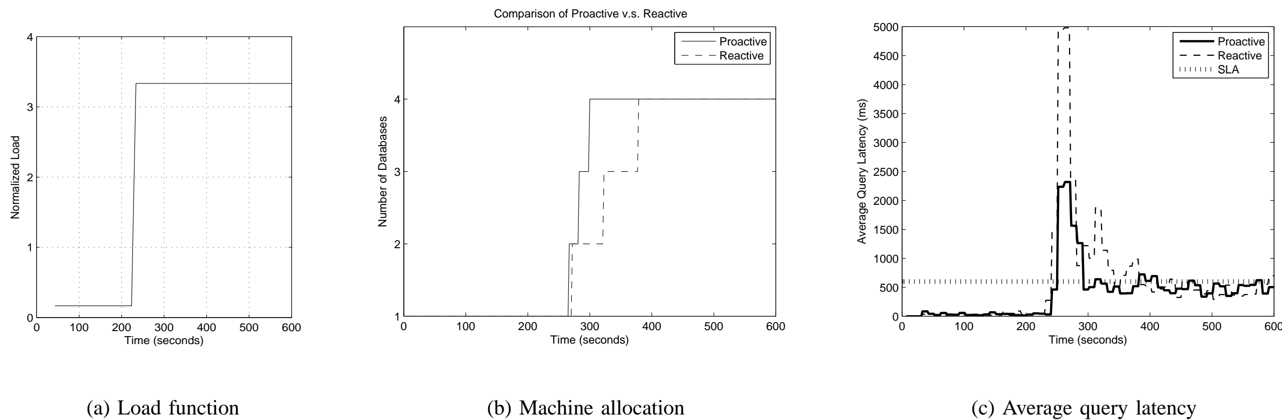
The reactive provisioning is much slower in its adaptation because it needs to obtain feedback from the system after each database addition. It does not know how many databases it should add, so it has to add the databases one at a time. In contrast, the pro-active approach can predict how many databases the system needs for the current and predicted load and it is able to trigger several simultaneous additions in advance of need. Figure 8(b) shows that the pro-active scheme adds 3 databases in a batch by requesting 3 databases simultaneously, while the reactive approach needs to add the 3 databases sequentially.

D. Robustness of the Pro-active Approach

In this section, we show that our pro-active scheme is relatively robust to some degree of on-line variation in workload mix and different load patterns given a fixed training data set.

Figures 9(a) and 9(b) show how our pro-active approach adapts on-line under a workload request mix different than the one it has been trained with. In particular, we train the system on a data set corresponding to stable load scenarios for the TPC-W shopping mix as before. We then show on-line adaptations for running TPC-W with the browsing mix. The browsing mix contains a different query mix composition than the shopping mix (with 5% versus 20% write queries). We use the same sine load function as before. We can see that our pro-active scheme adapts quite well to load increases, minimizing the number of SLA violations. It adapts less well to load decreases, however, by retaining more databases than strictly necessary. This effect is most obvious towards the end of the run.

Figure 10 shows the robustness of our learning-based approach under a step load function. Figure 10(a) shows the step function and the evolution of the instantaneous number of active client connections (as opposed to thinking clients) as measured at the emulator induced by this load function. Figure 10(b) shows that the allocation of the pro-active scheme is stable while the reactive scheme may register some oscillations in allocation if the thresholds it uses are not tuned. We run the reactive scheme in two configurations, pre-tuned and untuned by using to different values for the ImbalanceThreshold, which governs the scheme's heuristic instability detection. We use a threshold of 10% load imbalance with a time-out of 2 minutes for a pre-tuned reactive approach and a random value of these parameters for the other reactive graph shown in the Figure. We can see that the reactive scheme registers allocation oscillations which also incur latency SLA violations for both parameter settings with more oscillations for the untuned configuration. More sensitivity analysis results and oscillations induced by different parameters for the reactive scheme, e.g.,



(a) Load function

(b) Machine allocation

(c) Average query latency

Fig. 8. A scenario with sudden load change

LowSLAThreshold values are shown elsewhere [9]. Since our pro-active scheme learns the normal imbalance range and uses highly relevant system metrics according to the learning phase to determine instability, it is inherently more robust and has no allocation oscillations.

Finally, although the step function makes it slightly harder to predict the load trend compared to the sine load function, for the pro-active approach latency SLA violations are mostly avoided in this load scenario as well.

VIII. Related Work

This paper addresses the hard problem of autonomic resource provisioning within the database tier, advancing the research area of autonomic computing [2]. Autonomic computing is the application of technology to manage technology, materialized into the development of automated self-regulating system mechanisms. This is a very promising approach to dealing with the management of large scale systems, hence reducing the need for costly human intervention.

Resource prediction is important to adjust parameters of utility functions or decide the mapping between SLA and the amount of resources in autonomic resource provisioning. A related paper [7] uses a table-driven approach that stores response time values from offline experiments with different workload density and different numbers of servers. Interpolation is used to obtain values not in the table. This method is simple, but it may become infeasible for highly variable workload where it is hard to collect sufficient data if the number of available resources is large. Different queuing models [1] [19] are presented as analytic performance models of web servers and demonstrate good accuracy in simulations.

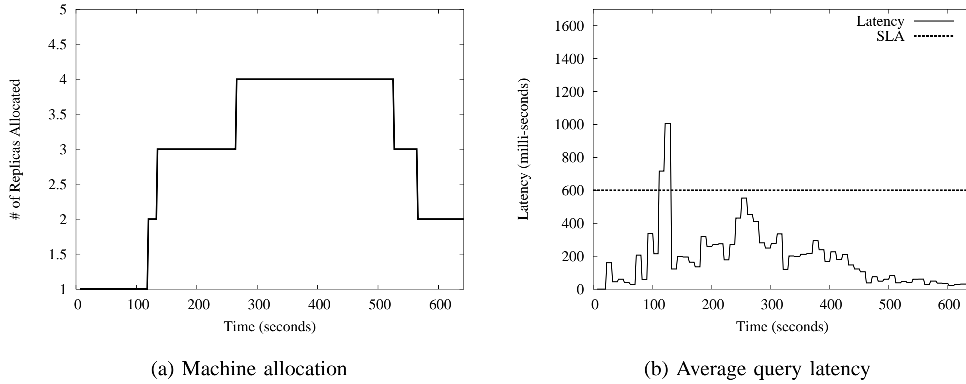
To the best of our knowledge, current performance prediction techniques [7] [1] [19] for large data centers assume a single database server as back-end, hence their performance is unknown for database clusters. The applicability of generic queuing models to database applications needs further investigation to understand modelling accuracy for the complex concurrency control situations in database clusters.

Various scheduling policies for proportional share resource allocation can be found in the literature, such as STFQ [20]. Steere et al. [21] describe a feedback-based real-time scheduler that provides reservations to applications based on dynamic feedback, eliminating the need to reserve resources a priori. In other related papers discussing controllers [22], [23] the algorithms use models by selecting various parameters to fit a theoretical curve to experimental data. These approaches are not generic and need cumbersome profiling in systems running many workloads. An example is tuning various parameters in a PI controller [22]. The parameters are only valid for the tuned workload and not applicable for controlling other workloads. In addition, none of these controllers incorporate the fact that the effects of control actions may not be seen immediately and the fact that the system may be instable immediately after adaptation.

Cohen et al. [24] propose using a tree-augmented Bayesian network (TAN) to discover correlations between system metrics and service level objectives (SLO). Through training, the TAN discovers the subset of system metrics that lead to SLO violations. While this approach predicts violations and compliances with good accuracy, it does not provide any information on how to adapt to avoid SLO violations. In contrast, our prediction scheme determines how many replicas must be added to maintain SLO compliance.

Our study builds on recently proposed transparent scaling through content-aware scheduling in replicated database clusters [25], [26], [27]. On the other hand, these systems do not investigate database replication in the context of dynamic provisioning. While Kemme et al. [25] proposes algorithms for database cluster reconfiguration, the algorithms are not evaluated in practice. Our paper studies efficient methods for dynamically integrating a new database replica into a running system and provides a thorough system evaluation using realistic benchmarks.

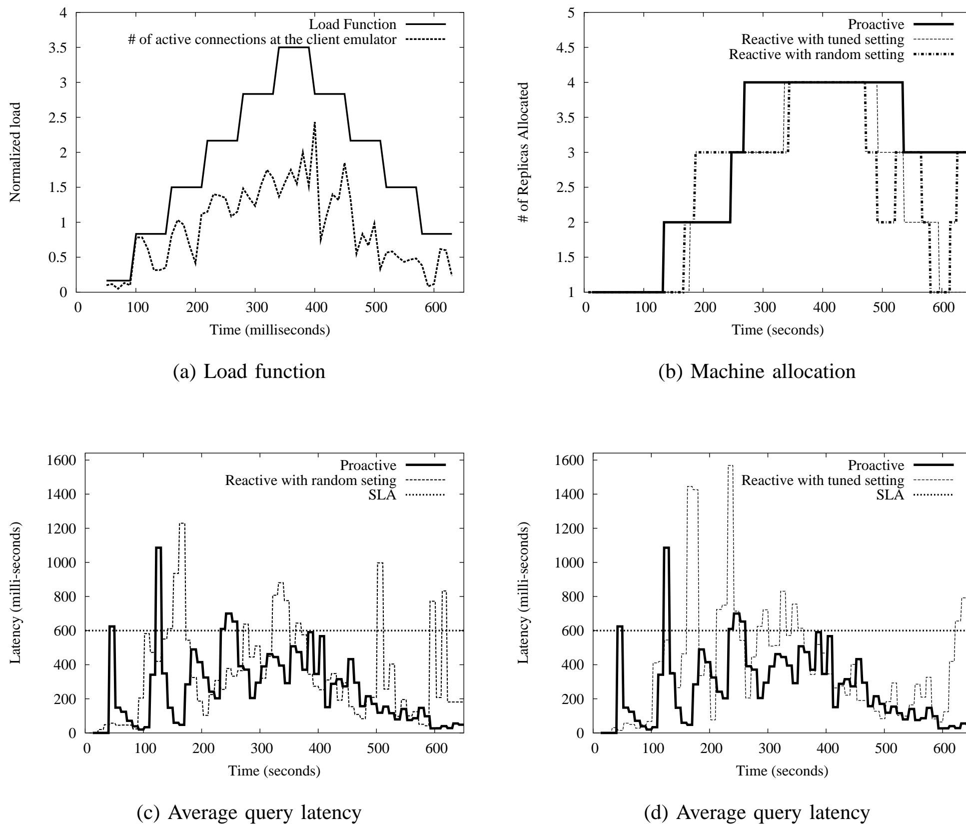
Finally, our work is related but orthogonal to ongoing projects in the areas of self-managing databases that can self-



(a) Machine allocation

(b) Average query latency

Fig. 9. A browsing workload scenario with sine load



(a) Load function

(b) Machine allocation

(c) Average query latency

(d) Average query latency

Fig. 10. A scenario with a step load function

optimize based on query statistics [28], and to recent work in automatically reconfigurable static content web servers [29] and application servers [30].

IX. Conclusions and Future Work

In this paper, we address autonomic provisioning in the context of dynamic content database clusters. We introduce a novel pro-active scheme based on the classic K-nearest-neighbors (KNN) machine learning approach for adding database replicas to application allocations based on: i) load predictions, ii) extensive off-line measurements of system and application metrics for stable system states and iii) lightweight on-line monitoring that does not interfere with system scaling.

We use a full prototype implementation of both our pro-active approach and a previous reactive approach to dynamic provisioning of database replicas. Overall, our experimental results show that our KNN-based pro-active approach is a

promising approach for autonomic resource provisioning in dynamic content servers. Compared to the previous reactive approach, the pro-active approach has the advantage of promptness in sensing the load trend and its ability to trigger several database additions in advance of SLA violations. By and large our pro-active approach avoids SLA violations under load variations. For unpredictable situations of very sudden load spikes, the pro-active approach can alleviate the SLA violations faster than the reactive approach even if SLA violations do occur in this case. Finally, off-line training on system-level information is also useful for recognizing periods of instability after triggering adaptations. Detecting unstable system states reduces the prediction errors of the pro-active approach in such cases.

The assumption of our prediction algorithm (KNN) is that the distribution of real data is similar to our training data. Hence, if the incoming traffic and our training set differ greatly, our scheme is unable to make informed decisions. In our future work, we will explore advanced machine learning algorithms which can effectively do outlier detection, hence can identify whether the current workload is similar with our training set. If no match is detected, our resource manager will disable the pro-active scheme, and revert to a more conservative reactive scheme. On the other hand, we would like to explore online training algorithms that automatically use the most recent workload features as training data. In addition, we will explore advanced adaptive filters to smooth measurement errors and transient load variations.

Acknowledgements

We thank the anonymous reviewers for their detailed comments and suggestions for improvement on the earlier version of this paper. We further acknowledge the generous support of IBM Toronto Lab through a student and faculty fellowship with the IBM Centre for Advanced Studies (CAS), IBM Research for a faculty award, the Natural Sciences and Engineering Research Council of Canada (NSERC), Communications and Information Technology Ontario (CITO) and Canadian Foundation for Innovation (CFI) through discovery and infrastructure grants.

References

[1] M. N. Bennani and D. A. Menasce, "Resource allocation for autonomic data centers using analytic performance models," in *In Proceedings of the 2nd International Conference on Autonomic Computing (ICAC)*, 2005.

[2] IBM, "Autonomic Computing Manifesto," http://www.research.ibm.com/ autonomic/manifesto, 2003.

[3] IBM Corporation, "Automated provisioning of resources for data center environments," http://www-306.ibm.com/software/tivoli/solutions/provisioning/, 2003.

[4] G. Tesauro, R. Das, W. E. Walsh, and J. O. Kephart, "Utility-function-driven resource allocation in autonomic systems ." in *ICAC*, 2005, pp. 70–77.

[5] C. Amza, E. Cecchet, A. Chanda, A. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel, "Specification and implementation of dynamic web site benchmarks," in *5th IEEE Workshop on Workload Characterization*, Nov. 2002.

[6] "The "Slashdot effect": Handling the Loads on 9/11," http://slashdot.org/articles/01/09/13/154222.shtml.

[7] W. E. Walsh, G. Tesauro, J. O. Kephart, and R. Das, "Utility functions in autonomic systems," in *In Proceedings of the 1st International Conference on Autonomic Computing (ICAC)*, 2004.

[8] K. Coleman, J. Norris, G. Candea, and A. Fox, "Oncall: Defeating spikes with a free-market server cluster," in *In Proceedings of the 1st International Conference on Autonomic Computing (ICAC)*, 2004.

[9] G. Soundararajan, C. Amza, and A. Goel., "Database replication policies for dynamic content applications," in *In Proceedings of the First ACM SIGOPS EuroSys*, 2006.

[10] E.-H. S. Han, G. Karypis, and V. Kumar, "Text categorization using weight adjusted k -nearest neighbor classification," in *Lecture Notes in Computer Science*, vol. 2035, 2001.

[11] C. Amza, A. Cox, and W. Zwaenepoel, "Conflict-aware scheduling for dynamic content applications," in *Proceedings of the Fifth USENIX Symposium on Internet Technologies and Systems*, Mar. 2003, pp. 71–84.

[12] C. Amza, A. L. Cox, and W. Zwaenepoel, "Distributed versioning: Consistent replication for scaling back-end databases of dynamic content web sites." in *Middleware*, ser. Lecture Notes in Computer Science, M. Endler and D. C. Schmidt, Eds., vol. 2672. Springer, 2003, pp. 282–304.

[13] P. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[14] E. Brookner, *Tracking and Kalman Filtering Made Easy*. Wiley Interscience.

[15] "The Apache Software Foundation," http://www.apache.org/.

[16] "PHP Hypertext Preprocessor," http://www.php.net.

[17] "MySQL," http://www.mysql.com.

[18] "Transaction Processing Council," http://www.tpc.org/.

[19] T. Zheng, J. Yang, M. Woodside, M. Litoiu, and G. Iszlai, "Tracking time-varying parameters in software systems with ex-tended Kalman filters," in *Proceedings of CASCON*, 2005.

[20] P. Goyal, X. Guo, and H. M. Vin, "A Hierarchical CPU Scheduler for Multimedia Operating System," in *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, Seattle, WA, Oct. 1996.

[21] D. C. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole, "A Feedback-driven Proportion Allocator for Real-Rate Scheduling," in *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, Feb. 1999.

[22] Y. Diao, J. L. Hellerstein, and S. Parekh, "Optimizing quality of service using fuzzy control," in *DSOM '02: Proceedings of the 13th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*. Springer-Verlag, 2002, pp. 42–53.

[23] B. Li and K. Nahrstedt, "A control-based middleware framework for quality of service adaptations," *IEEE JSAC*, 1999.

[24] 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, pp. 231–244.

[25] B. Kemme, A. Bartoli, and Ö. Babaoglu, "Online reconfiguration in replicated databases based on group communication." in *DSN*. IEEE Computer Society, 2001, pp. 117–130.

[26] E. Cecchet, J. Marguerite, and W. Zwaenepoel, "C-JDBC: Flexible database clustering middleware," in *Proceedings of the USENIX 2004 Annual Technical Conference*, Jun 2004.

[27] J. M. Milan-Franco, R. Jimenez-Peris, M. Patio-Martnez, and B. Kemme, "Adaptive middleware for data replication," in *Proceedings of the 5th ACM/IFIP/USENIX International Middleware Conference*, Oct. 2004.

[28] P. Martin, W. Powley, H. Li, and K. Romanufa, "Managing database server performance to meet Qos requirements in electronic commerce systems," *International Journal on Digital Libraries*, vol. 3, pp. 316–324, 2002.

[29] S. Ranjan, J. Rolia, H. Fu, and E. Knightly, "QoS-Driven Server Migration for Internet Data Centers," in *10th International Workshop on Quality of Service*, May 2002.

[30] E. Lassettre, D. W. Coleman, Y. Diao, S. Froehlich, J. L. Hellerstein, L. Hsiung, T. Mummert, M. Raghavachari, G. Parker, L. Russell, M. Surendra, V. Tseng, N. Wadia, and P. Ye, "Dynamic surge protection: An approach to handling unexpected workload surges with resource actions that have lead times." in *DSOM*, ser. Lecture Notes in Computer Science, M. Brunner and A. Keller, Eds., vol. 2867. Springer, 2003, pp. 82–92.