Database replication policies for dynamic content applications — Source link

Gokul Soundararajan, Cristiana Amza, Ashvin Goel

Institutions: University of Toronto

Published on: 18 Apr 2006 - European Conference on Computer Systems

Topics: Replication (computing), Distributed database, Fault tolerance, Server and Replica

Database Replication Policies for Dynamic Content Applications

Gokul Soundararajan, Cristiana Amza, Ashvin Goel
Department of Electrical and Computer Engineering
University of Toronto
Toronto, Canada

ABSTRACT

The database tier of dynamic content servers at large Internet sites is typically hosted on centralized and expensive hardware. Recently, research prototypes have proposed using database replication on commodity clusters as a more economical scaling solution. In this paper, we propose using database replication to support *multiple* applications on a shared cluster. Our system dynamically allocates replicas to applications in order to maintain application-level performance in response to either peak loads or failure conditions. This approach allows unifying load and fault management functionality. The main challenge in the design of our system is the time taken to add database replicas. We present replica allocation policies that take this time delay into account and also design an efficient replica addition method that has minimal impact on other applications.

We evaluate our dynamic replication system on a commodity cluster with two standard benchmarks: the TPC-W e-commerce benchmark and the RUBIS auction benchmark. Our evaluation shows that dynamic replication requires fewer resources than static partitioning or full overlap replication policies and provides over 90% latency compliance to each application under a range of load and failure scenarios.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.4.5 [**Reliability**]: Backup Procedures, Fault-tolerance; I.2.8 [**Problem solving, Control Methods, and Search**]: Control theory, Heuristic methods; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Measurement, Management, Performance, Reliability

Keywords

Database systems, Adaptation, Fault-tolerance, Cluster

1. INTRODUCTION

Today, dynamic content servers used by large Internet sites, such as Amazon and EBay, employ a three-tier architecture that consists of a front-end web server tier, an application server tier that implements the business logic of the site, and a back-end database tier that stores the content of the site (see Figure 1). The first two tiers, the web and the application server, typically use non-persistent data and are generally hosted on inexpensive clusters of machines. However, the database tier storing persistent data is centralized and hosted either on a high-end multiprocessor [15, 31] or on specialized and expensive devices such as a shared network disk [1].

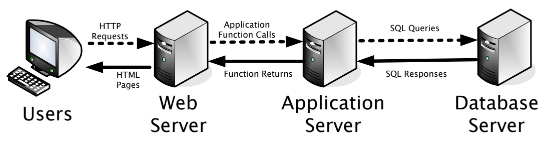


Figure 1: Architecture of dynamic content servers

Recently, several research prototypes have proposed using replicated databases built from commodity clusters as a more economical solution. These replicated databases, which have been used for running a single application, such as, an e-commerce benchmark [24, 6], have shown good performance scaling with increasing replication. For example, a read-heavy workload can scale linearly to dozens of machines [4, 6].

In this paper, we investigate using database replication on a commodity cluster to run multiple applications. Our system dynamically allocates machine replicas to each application to maintain application-level performance. In particular, we use a predefined latency bound to determine whether an application's requirements are being met. This approach provides several benefits. First, it allows efficient resource management by avoiding per-application over-provisioning, e.g., it allows reducing costs and/or energy consumption. Second, dynamic replication enables a unified approach to resource management as well as fault tolerance. Our system detects load spikes and failures as a resource bottleneck and adapts in the same manner by adjusting the number of replicas allocated to an application. Third, the previous scaling studies have shown that the optimal number of replicas allocated to an application depends on the type of workload mix. For example, a read-heavy workload scales to larger numbers of replicas than a write-heavy workload. A database cluster shared among multiple applications allows tuning the number of replicas for each application based on the workload mix.

While dynamic database replication is appealing, it raises two key issues: 1) the replica allocation policy and 2) the replica mapping policy. The replica allocation policy chooses the number of replicas to allocate to an application. The replica mapping policy maps an application's allocated replicas to a set of machines. Below, we first focus on the design of the mapping policy and then describe the allocation policy.

The performance of our system depends on the delay associated with adding a new database replica to an application, which can be a time-consuming operation. Consider the case where different applications are mapped to *disjoint* sets of machine replicas. When an application requires an additional machine replica, it must use an unallocated machine or a machine allocated to another application. In either case, the database state of the new replica will be stale and must be brought up-to-date, e.g., via data migration from an existing replica, before it can be used. In addition, the buffer cache at the new replica needs to be warm before the replica can be used effectively.

Replica addition delay can be avoided altogether with *fully-overlapped* replicas where all the database applications are replicated across all the cluster machines. In this case, there is no replica addition delay because replicas do not have to be added or removed. However, this approach causes interference due to resource sharing. For example, when multiple database applications run on the same machine, their performance can degrade due to buffer cache interference. Such interference is avoided if using disjoint replica sets across applications. This discussion shows that there is a trade-off between using disjoint and fully-overlapped replica mapping. Disjoint mapping reduces interference and thus improves steady-state performance. Fully-overlapped mapping avoids replica addition delay and thus can speed up the system's response to load spikes and failures.

In this paper, we propose using an intermediate mapping policy called *partial overlap* that provides fast response to overload and minimizes interference across applications. In this policy, we distinguish between an application's read and write replica sets, called *read set* and *write set* respectively. The read (write) set is the set of machine replicas from which an application reads (writes) data. The read set of an application is a subset of its write set. In our approach, the read sets are disjoint but the write sets of applications can overlap with each other. This policy allows fast response to overload conditions in the common case, because additional resources can be provided to an application by simply extending its read set within the boundaries of its write set. This operation is relatively fast because it does not require data migration, although expanding the read set may require warming the buffer cache e.g., if switching the allocation of a a read set replica from another application becomes necessary. Data migration is needed only if an application's write set needs to be expanded.

The partial overlap policy minimizes interference across applications because write queries in dynamic content applications are typically lightweight compared to read queries [4]. For instance, in e-commerce applications, an update query typically updates only the record pertaining to a particular customer or product, while read queries caused by browsing can involve expensive database joins as a result of complex search criteria. The memory footprint and complexity of read queries often exceed those of write queries. Moreover, read queries are much more frequent than write queries. Partitioning the read sets of applications across different machines minimizes interference, while overlapping write sets causes little interference and generally avoids replica addition delay.

In addition to the mapping policy, our replication system uses an allocation policy that is designed to meet the application's latency requirements while providing system stability. To meet the latency requirements, the policy should aggressively add machine replicas when the application's latency is close to its bound. However, a simple reactive allocation policy that periodically measures application latency to add replicas can cause instability. In particular, the replica addition process can be long (either due to data migration or buffer cache warm-up) and the application latency may remain high during this process. A reactive policy that measures application latency even during replica addition can therefore cause unnecessary allocation and additional cross-application interference as well as positive feedback and oscillation.

To meet the latency and stability goals, we use a delay-aware allocation policy that has three novel features. First, the policy typically suppresses allocation decisions while replica addition is in progress. Second, it compensates for the delay during replica addition by aggressively reacting to resource bottlenecks. Specifically, it uses early latency indications on the newly added replica to trigger aggressive allocation. Finally, it uses a two-step conservative replica removal process to avoid oscillatory allocation behavior. Initially, it stops read queries to a replica, hence removes the replica from the application's read set. If the application's performance is not significantly affected then write queries are stopped to remove the replica from the application's write set.

In our evaluation, we explore a number of replica mapping and allocation policies for dynamic database replication. We characterize the effects of the disjoint, full overlap as well as our partial overlap replica mapping policies. In addition, we compare the reactive and our delay-aware replica allocation policies. We evaluate our system with two benchmarks: 1) the industry-standard, shopping mix workload of the TPC-W e-commerce benchmark [30] that models an on-line book store (called TPC-W in the rest of the paper), and 2) the bidding mix workload of the RUBIS benchmark [3] that models an on-line bidding system (called RUBIS). Our evaluation shows that the partial overlap mapping policy is most effective and requires fewer resources than the disjoint or full overlap replication policies. The partial overlap mapping policy together with our replica allocation algorithm shows good scaling and provides over 90% latency compliance for both applications when they are run together under a range of load and failure scenarios.

The remainder of this paper is structured as follows. Section 2 provides the motivation for using dynamic replication, and Section 3 describes the architecture of our system and provides some background on our replication scheme. Section 4 describes our replica allocation and mapping policies. Section 5 presents our experimental platform and the results of our experiments. Section 6 discusses related work and Section 7 concludes the paper.

2. MOTIVATION

Our previous work [4, 5, 6] has shown that database replication on commodity clusters scales well and is a viable alternative to using expensive multiprocessor and/or network storage hardware configurations. Next, we present some results from this work to motivate our dynamic replication approach. Figures 2 and 3 show the performance scaling graphs for the TPC-W and the RUBIS benchmarks. Each graph contains two curves, a scaling curve obtained experimentally on a cluster of 8 database machines[1], and a curve obtained through simulation for larger clusters of up to 60 database machines.

Our simulation with large numbers of replicas was calibrated using the real 8-node cluster. Figures 2 and 3 show that the simulated

[1]The machines are AMD Athlon MP 2600+ computers with 512MB of RAM, using RedHat Fedora Linux.

performance scaling graphs for TPC-W and RUBIS are within 12% of the experimental numbers for both applications. The simulations show that TCP-W scales better than RUBIS. The reason is that replication allows scaling the throughput of read queries, and the scaling limit depends on the ratio between the average complexity of read and write queries. The time spent in executing reads versus writes is 50 to 1 for TPC-W and 8 to 1 for RUBIS. These graphs show that the knee of the scaling curve depends on the workload and cannot be determined a priori. With a dynamic replication approach, each application could be assigned replicas based on its scaling curve.

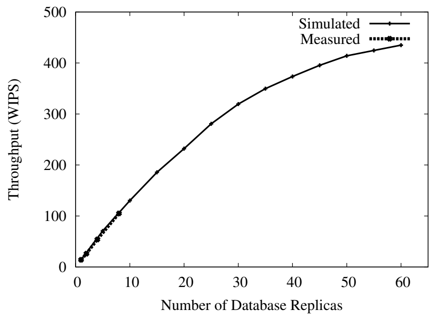


Figure 2: TPC-W throughput scaling.

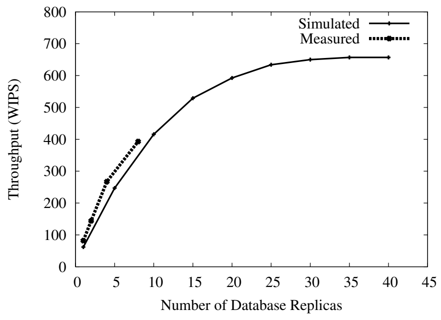


Figure 3: RUBIS throughput scaling.

While database replication is appealing, it requires policies for replica allocation and mapping. As described earlier, the two most significant and competing challenges in the design of these policies are the delay associated with replica addition and the buffer cache interference with overlapping replicas. While it should be intuitively clear that adding a database replica can be slow, below we show that buffer cache interference can also be a significant problem, which led us to investigate the disjoint and the partial overlap mapping policies.

We conducted an experiment with the TPC-W and RUBIS workloads using two configurations on our experimental cluster. The workloads are either run on two separate machines (Disjoint Mapping) or on both machines together (Overlap Mapping). When running together, we load balanced both the workloads on both the machines. Table 1 shows the percent of total CPU time spent waiting for I/O as reported by the `vmstat` Linux utility. In both configurations, the results reported are after the buffer pool has been warmed. With Disjoint Mapping, TPC-W experiences no wait time because it has a small memory footprint that fits in available memory, while RUBIS waits for 31% time because its footprint exceeds total available memory. However, when the workloads execute on both the machines, the wait time for TPC-W increases dramatically. The RUBIS workload evicts TPC-W's pages from the shared buffer pool. This causes TPC-W to issue more I/O requests, which degrades its performance severely. As a result, the TPC-W throughput is halved with Overlap Mapping compared to Disjoint Mapping and the query latency is doubled (numbers not shown here). While it is possible to reduce interference effects by various tuning methods, such as by increasing available memory to the buffer pool and by tuning per-application buffer pool sizes, this approach quickly becomes infeasible as the number of overlapping applications is increased.

	Disjoint Mapping	**Overlap Mapping**
TPC-W	0	44
RUBIS	31	38

Table 1: Percent of total CPU time waiting for I/O

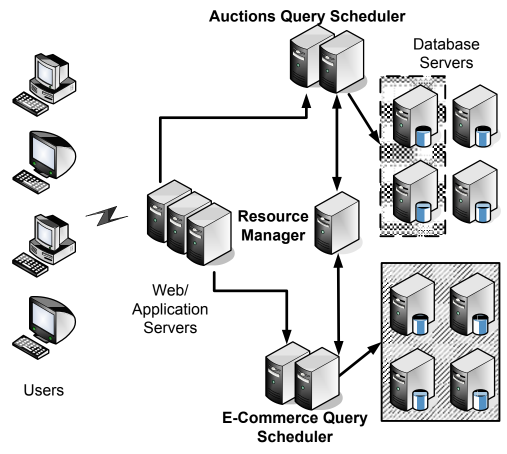


Figure 4: Cluster architecture

3. REPLICATION ARCHITECTURE

The dynamic content server architecture consists of the web, application and the database server tiers. In our system, 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. Figure 4 shows the architecture of our system. In our system, the web and the application tiers are combined but separating these tiers would have no effect on our dynamic replication architecture. Each scheduler[2] 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 resource manager is described in the next section.

The application scheduler provides consistent replication, i.e., one-copy serializability [8], by assigning a global serialization order to all transactions and ensuring that transactions execute in this order at all the replicas allocated to its workload.

For scalability, the scheduler uses our *Conflict-Aware* replication scheme [5, 6]. With this scheme, each transaction explicitly declares the tables it is going to access and their access type. Conflict-aware replication uses this information to detect conflicts

[2]Each scheduler may itself be replicated for availability [4, 5].

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.

4. DYNAMIC REPLICATION

In this section, we provide an overview of the resource manager that implements dynamic replication and then present the replica allocation and the mapping policies used by the manager. Finally, we present an efficient data migration algorithm that is used during replica addition and that has minimal impact on transaction processing.

4.1 Overview

The resource manager makes the replica allocation and mapping decisions for each application based on the performance needs and the current performance level of the application. The performance needs are expressed in terms of a service level agreement (SLA) that consists of a latency requirement on the application's queries. The current performance level is measured in terms of the average query latency observed at the application. This latency is maintained at the application scheduler and periodically sent to the resource manager. The resource manager uses the average latency and the application's latency requirement to make the allocation decisions periodically. This period is the same for all applications. The allocation decisions are communicated to the respective schedulers, which then allocate or remove replicas from their replica sets.

The resource manager operates in two modes, underload and overload. During underload, the number of replicas is sufficient to handle the overall demand and allocation decisions per application are made independently. During overload, the total demand exceeds the capacity of the cluster. In the latter case, the system uses a simple fairness scheme that allocates an equal number of replicas to each application.

4.2 Replica Allocation Policy

The resource manager uses average query latency to make replica allocation decisions. The averaging is performed at every query using an exponentially weighted mean [35] 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 is to current latency. The resource manager uses a threshold scheme for replica allocation. The manager periodically compares the average latency to a high threshold value (HighSLAThreshold) to detect imminent SLA violations and adds a replica when the average latency is above the high threshold. Similarly, when the average latency is below a low threshold (LowSLAThreshold), it detects underload and removes a replica. This basic technique is similar to overload adaptation in stateless services [35], where simple smoothing of latency and thresholding have been reported to give acceptable stability.

Unfortunately, this technique by itself does not provide satisfactory performance in our system due to the delay associated with

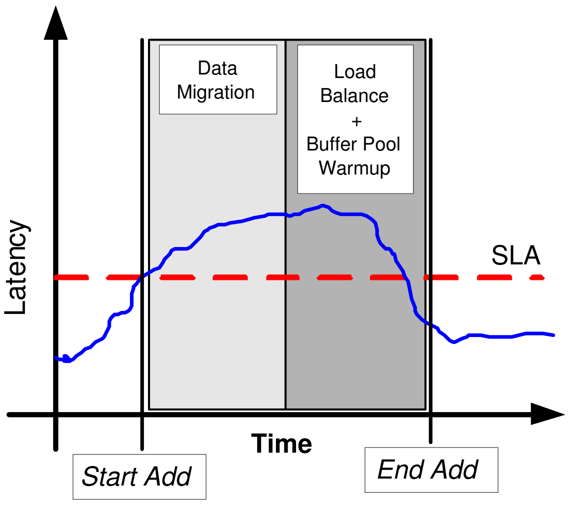


Figure 5: Typical replica addition process

adding database replicas. Below, we describe modifications to the technique to improve the stability and the performance of the system during replica allocation. These modifications are discussed as part of the replica addition and the removal process. Figure 6, shows our replica allocation logic and the conditions under which a database replica is added to or removed from an application allocation.

4.2.1 *Replica Addition*

Figure 5 shows a typical replica addition process, which consists of two phases: data migration and system stabilization. Data migration involves applying logs of missing updates to 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 these stages may overlap, replica addition can introduce a long period over which query latencies are high. The simple allocation policy described above based on periodically measuring violations of the latency requirement can thus trigger unnecessary replica allocation as well as interference with other applications.

The resource manager makes two modifications to the basic allocation algorithm to account for replica addition delay. First, it stops making allocation decisions until the completion of the replica addition process. Second, 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 to improve its responsiveness. The state transitions on the left side of Figure 6 show the replica addition logic in detail. Specifically, in the `Steady State`, the resource manager monitors the average latency received from each workload scheduler during each sampling period. 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. The resource manager tracks the replica addition process until the request has been fulfilled and the result of the change can be observed. This implies potential waiting in two states, corresponding to adding the new replica to the write set and the read set of the application, respectively. When adding a replica to the write set of the application, data migration to bring up a new database for that workload may be necessary. We transition out of the corresponding state only when data migration is finished.

The second state corresponds to adding the replica to the read set

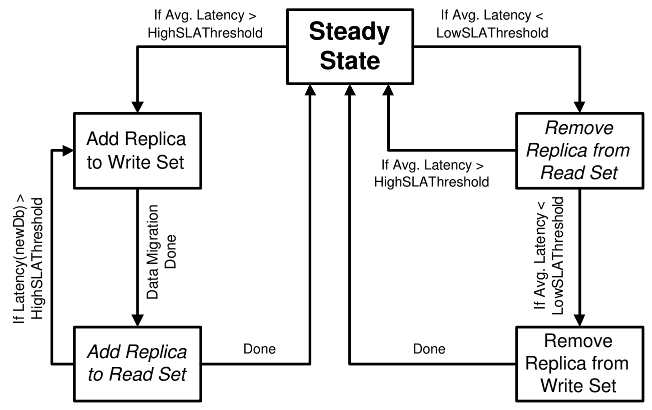

If Avg. Latency > HighSLAThreshold
Steady State
If Avg. Latency < LowSLAThreshold
Add Replica to Write Set
Remove Replica from Read Set
If Avg. Latency > HighSLAThreshold
If Latency(newDb) > HighSLAThreshold
Data Migration Done
If Avg. Latency < LowSLAThreshold
Add Replica to Read Set
Done
Done
Remove Replica from Write Set

Figure 6: Replica allocation logic

of the application, which includes waiting for system stabilization, i.e., load balancing and buffer pool warmup. The resource manager compares average statistics collected by the scheduler from the old read replica set and the new replica in order to determine when system stabilization is complete. Since this wait may be long and will impact system reactivity to steep load bursts, we optimize waiting time by using the individual average latency generated at the newly added database as a heuristic. Since this database has no load when added, we use its latency exceeding the SLA as an early indication of a need for even more databases for that workload and we transition directly into adding another replica in this case.

4.2.2 *Replica Removal*

The resource manager removes a database from an application's allocation in two cases. First, the application is in underload for a sufficient period of time and does not need a replica (voluntary remove). Second, the system as a whole is in overload and fairness between allocations needs to be enforced (forced remove).

In the former case, the right branch of Figure 6 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 is removed from an application's read set but not from its write set. If the application's average latency remains below the low threshold for a period of time, the replica is removed from the write set also. 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 write set only when ongoing transactions finish at that replica.

4.3 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. Below, we describe three replica mapping schemes that we analyze in this paper: disjoint, full overlap and partial overlap.

4.3.1 *Disjoint*

In this scheme, each application is assigned a disjoint replica set from the machines in the database cluster. An application updates only the replicas in its replica set and any replica within the replica set can be selected to service a read. The benefit of this approach is that it has minimal cross-application interference during periods of stable load. However, when adding a replica, data migration can take a while if the new replica has not been updated for a long time. We call this migration to a potentially stale replica with a cold buffer cache a *cold migration*.

4.3.2 *Full Overlap*

With full overlap, writes of all applications are sent to all the databases in the cluster. Each read query of any application can be sent to any replica and the read sets and the write sets of all applications are never changed. This approach allows maximum sharing of resources across applications and obviates the need for replica addition (or data migration) or deletion. However, the reads and writes of all applications share the buffer-cache at each replica which can cause cross-application interference and poor performance when the buffer-cache capacity is exceeded. Furthermore, with large clusters, the large write set can exceed the scaling limit for some applications (e.g., see Figure 3). Finally, if a large number of applications are sharing the cluster, the execution of all writes on all nodes may ultimately cause interference.

4.3.3 *Partial Overlap*

The partial overlap scheme lies in between the disjoint and the full overlap mapping schemes. Each application is assigned a disjoint primary replica set. However, write queries of an application are also periodically sent to a secondary set of replicas. This secondary set overlaps 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, where the bound is equal to the batch size or the number of queries in the batch. Although the batched updates to the secondary set can cause cross-application interference, we expect this interference to be small because dynamic content applications are typically read-heavy and reads are not sent to the secondary replicas.

The secondary replicas are an overflow pool that allow adding replicas rapidly in response to temporary load spikes since migrating data to them involves sending no more than a batch size of updates and is expected to be a fast operation. We call this data migration strategy *warm migration*. A special case of warm migration occurs when the batch size is one. In this case, which we call *hot migration*, update queries are sent immediately to all the replicas in the primary and the secondary set. Then a replica is added (i.e., a replica is moved from the secondary set to the primary set) by simply issuing read queries to the secondary replica.

The secondary replica set is configurable in our system. However, to simplify analysis, we will henceforth assume that the secondary set of an application consists of all replicas in the system outside the application's primary set.

4.4 Data Migration Algorithm

In this section, we describe our data migration algorithm that is designed to bring a new replica up-to-date while minimally disrupting transaction processing on the current replicas in the application's replica set. With this goal, the migration is performed using data stored at the scheduler rather than from an existing replica. The scheduler maintains persistent logs of write queries and their version numbers per table and updates the log at each transaction commit point. These logs are replayed during data migration to update a new replica and the version numbers at the new replica are used to detect the set of missed updates. For efficiency, we allow transaction processing to occur at the current replicas while data is being migrated to the new replica. The challenge with this approach is that the updates made by these new transactions need to be incorporated at the new replica. If transactions arrive continually, the data migration process, which itself takes time, would lag behind the new transactions and the new replica would never be up-to-date. To avoid this problem, data migration is performed in a batched fashion until the remaining logs reach below a threshold size. At this point, the new replica is considered added. Then the remaining logs and the new transaction updates are sent to the new replica using the same method as for existing replicas. The replica orders the log entries and the updates and applies them in version number order. At this point, the replica is up-to-date. To limit the number of log updates that need to be sent, the threshold size should be large. However, it should be chosen so that the updates, which are stored in memory, do not exceed the memory available at the replica machine.

The logs for a transaction are maintained at the scheduler at least until the transaction either commits or aborts at all databases in the replica set. In addtion, the logs are garbage collected after their size exceeds a certain bound. Occasionally, replica addition may require migrating updates that have been garbage collected. In this case, data migration consists of installing a snapshot of the entire database for that application from an existing replica.

5. EVALUATION

In this section, we evaluate the performance of our system to show that dynamic replication enables handling rapid variations in an application's resource requirements while maintaining quality of service across applications. Our evaluation consists of four different sets of experiments. First, we use a single application to clearly illustrate the impact of replica-addition delay on dynamic database replication. Second, we evaluate the performance of our system under heavy and varying load by using two benchmark applications, TPC-W and RUBIS, that are described in detail below. Third, we evaluate the effect of database faults on our system. Finally, we perform sensitivity analysis and show that the system is robust and does not require careful parameter tuning to achieve good performance. Below, we first describe the benchmarks and the experimental setup and then present our evaluation.

5.1 Benchmarks

The TPC-W and the RUBIS benchmarks used in our experiments are implemented using three popular open source software packages: the Apache web server, the PHP web-scripting/application development language [23] that implements the business logic of the benchmarks, and the MySQL database server with InnoDB tables [22].

5.1.1 TPC-W E-Commerce Benchmark

The TPC-W benchmark from the Transaction Processing Council [30] 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 to the shopping cart, two order-placement interactions, two interactions that involve order inquiry and display, and two that involve 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 50 times more heavyweight than read-write transactions.

5.1.2 RUBIS Auction Benchmark

We use the RUBIS Auction Benchmark to simulate a bidding workload similar to EBay. The benchmark implements the core functionality of an auction site: selling, browsing, and bidding. We do not implement complementary services like instant messaging, or newsgroups. We distinguish between three kinds of user sessions: visitor, buyer, and seller. For a visitor session, users need not register but are allowed to browse only. Buyer and seller sessions require registration. In addition to the functionality provided during the visitor sessions, during a buyer session, users can bid on items and consult a summary of their current bid, rating, and comments left by other users. We use the default RUBIS bidding workload that contains 15% writes. This mix is considered the most representative of an auction site workload according to an earlier study of EBay workloads [27].

5.2 Experimental Setup

Our experimental setup consists of web servers, schedulers (one per application), the resource manager, database engines and client emulators that simulate load on the system. All these components use the same hardware. Each machine is a dual AMD Athlon MP 2600+ (2.1GHz CPU) computer with 512MB of RAM. We use the Apache 1.3.31 web server [2] and the MySQL 4.0.16 database server with InnoDB tables [22]. All the machines use the RedHat Fedora 3 Linux operating system with the 2.6 kernel. All nodes are connected via 100Mbps Ethernet LAN.

To demonstrate the scaling and the performance behavior of the database backend, the Apache web/application servers are run on a sufficient number of machines so that these servers do not become a bottleneck for either application. The MySQL databases are run on 8 machines.

5.2.1 Client Emulator

We have implemented a session emulator for the TPC-W and the RUBIS applications to induce load on the system. A session is a sequence of interactions by 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.

The load induced by the client emulator depends on the number of clients emulated and the application. To ease representing this load for both the TPC-W and the RUBIS applications on the same graph, we normalize the input load to a baseline load. The baseline load is the number of clients that saturate a single machine. In our setup, the baseline load was roughly 25 clients for TPC-W and 150 clients for RUBIS.

5.2.2 *Experimental Parameters*

Our experiments use 600 ms for the `HighSLAThreshold` and 200 ms for the `LowSLAThreshold` parameters. The `High-SLAThreshold` parameter value was chosen conservatively to guarantee an end-to-end latency at the client of at most one second for each of the two workloads. The low threshold parameter is chosen to be less than 50% of the high threshold parameter, which provides stability in small database configurations (i.e., when adapting from 1 to 2 databases). We use a latency sampling interval of 10 seconds for the schedulers. This value does not require careful tuning because the replica allocation policy accounts for the delay during replica addition or deletion. As a result, the sampling interval can be relatively short and the schedulers can respond rapidly to changes in load. The value of the smoothing parameter α, which affects the system response, is set to 0.25. Section 5.6 shows that these parameters do not require extensive tuning.

5.3 Single Application Workload

In this section, we use a single TPC-W benchmark application to show the effect of replica-addition delay on both the replica allocation and the mapping policies.

5.3.1 *Replica Allocation*

Figure 7 shows the results of using two replica allocation policies. The input load function is shown in Figure 7(a). The first policy uses continuous latency sampling and triggers replica addition or deletion when the average latency rises above the `High-SLAThreshold` parameter or falls below the `LowSLAThresh-old` parameter. The second policy is delay-aware and implements the replica allocation policy described in Section 4.2.

In this experiment, we use partial overlap with hot migration as the replica mapping scheme. This scheme ensures that replica addition is a relatively fast operation. Even so, Figure 7(b) shows that oscillations occur when the replica-addition delay is not taken into account by the allocation policy. These oscillations occur because the latency does not become normal until the queries that caused the spike in latency finish executing. During this period, this policy overallocates replicas, which subsequently causes the latency to dip below the `LowSLAThreshold`. As a result, the resource manager then deletes replicas. This situation would be even worse when the replica-addition delay is longer, such as with warm or cold migration. Our delay-aware policy avoids these oscillations as shown in Figure 7(c). This figure shows that the resource manager adds databases to meet demand without overallocation.

5.3.2 *Replica Mapping*

Figure 8 compares the results of using the cold and warm replica mapping policies. We initially subject the system to a load that requires 3 databases to satisfy the SLA. After 2 hours (7200 seconds) of elapsed time, we increase the load to 7.

Figure 8(a) shows the load function, while Figure 8(b) shows the latency spikes caused by the two policies. Both the intensity and the duration of the spike is smaller for warm migration compared to cold migration because the replicas are maintained relatively up-to-date through periodic batched updates. Figure 8(c) shows the allocation of replicas during cold migration. The width of each adaptation step widens with each replica addition because, in addition to the application of the two hour log of missed updates, the amount of data and the number of queries to be transferred and executed on the new replicas accumulates with the incoming transactions from the new clients. Hence, the system has a difficult time catching up. Figure 8(d) shows that warm migration is able to quickly adapt to the spike in load.

5.4 Multiple Application Workload

In this section, we use both the TPC-W and the RUBIS benchmark applications to evaluate our replication system. Initially, we consider a simpler scenario, where the load for only the TPC-W workload is varied, while the RUBIS load is constant throughout the experiment. Then we consider scenarios when both loads vary dynamically.

5.4.1 *TPC-W Workload Adaptation*

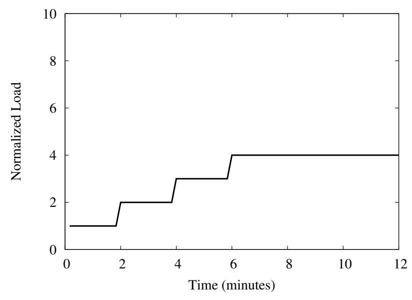
Section 5.3 showed that delay-aware allocation reduces oscillatory allocation behavior and warm migration outperforms cold migration. Our system uses this combination together with the partial overlap mapping policy described in Section 4.3. Below, we compare our system against two alternatives, static partitioning that uses disjoint mapping and the full overlap mapping policies. These schemes represent opposite end points of the mapping schemes. Static partitioning assigns a fixed, disjoint and fair-share partition of the database cluster to each workload, while full overlap mapping allows both the applications to operate on all the machines in the system. These schemes, unlike partial overlap mapping, require no dynamic allocation or migration, and serve as good baseline cases for comparison.

Figure 9 shows the results of running the two benchmarks. Figure 9(a) shows that the load function for TPC-W changes over time while the RUBIS load is kept constant. The TPC-W normalized load function varies from one to seven, while the RUBIS load is kept at one. Note that load steps are roughly 2.5 minutes wide, so a sharp seven-fold load increase occurs within a short period of 15 minutes. The number of replicas allocated to TPC-W under the partial overlap policy is shown in Figure 9(b). Note that the read and write sets of the workloads do not change for the other policies. The three graphs at the bottom of Figure 9 show the query latency with the three mapping policies.

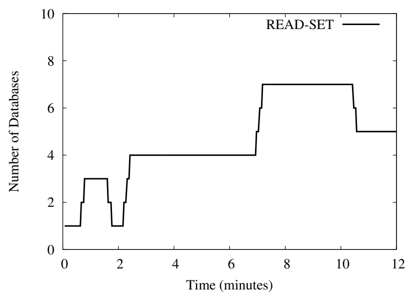
The latency results show that partial overlap mapping substantially outperforms both the static partitioning and the full overlap mapping schemes which exhibit sustained and much higher latency SLA violations. The poor performance of the static partitioning scheme occurs as a result of insufficient resources allocated to TPC-W since this scheme splits resources fairly across workloads (4 machines per workload). Full overlap performs poorly due to the interference caused by the overlapping read sets of the two workloads in the buffer cache of the database, since requests from either application can be scheduled on any database replica at any point in time.

Figure 9(c) shows that the latency in our system briefly exceeds the TPC-W SLA of 600 ms as the system receives additional load while adapting to previous increase in load. However, the system catches up quickly and the latency target is met immediately after the last load step. Figure 9(b) shows that machines are gradually removed from the TPC-W allocation as load decreases in the last part of the experiment. The write-set removal lags behind the read-set removal because of our two-step removal process (see Section 4.2) where replicas in the write set are removed more conservatively than replicas in the read set.

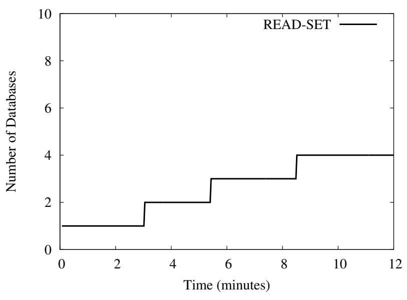
Table 2 shows the percentage compliance and the average number of replicas used by the three schemes in this experiment. To calculate compliance, we divide the experiment into 10 second intervals and consider an interval as non-compliant if the latency rises above the `HighSLAThreshold` value even once in the interval. While static partitioning uses fewer machines, it only has 36% compliance. Similarly, the full overlap scheme has 31% compliance although it uses 8 machines. On the other hand, our partial overlap, warm migration scheme uses 5.2 machines on average and



(a) Load function

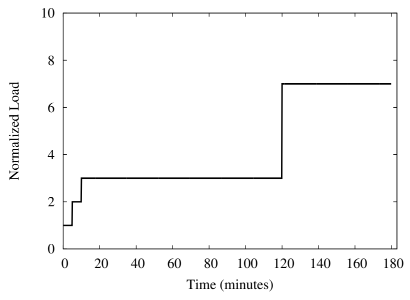


(b) Basic allocation policy

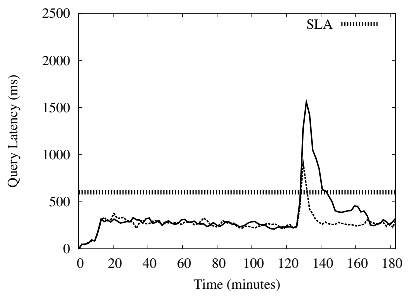


(c) Delay-aware allocation policy

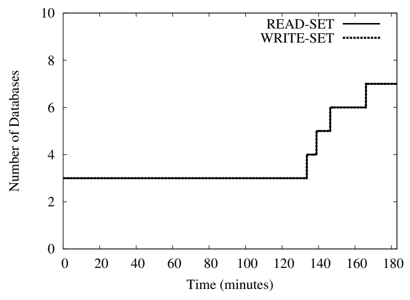
Figure 7: Comparison of replica allocation policies



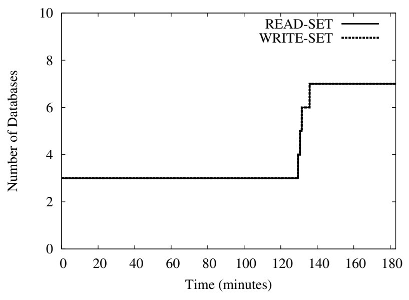
(a) Load function



(b) Average query latency



(c) Cold migration



(d) Warm migration

Figure 8: Comparison of replica mapping policies

Scheme	**% compliance**	**Allocated replicas**
Partial overlap	92%	5.2
Static partitioning	36%	4
Full overlap	31%	8

Table 2: Percent compliance and number of allocated replicas

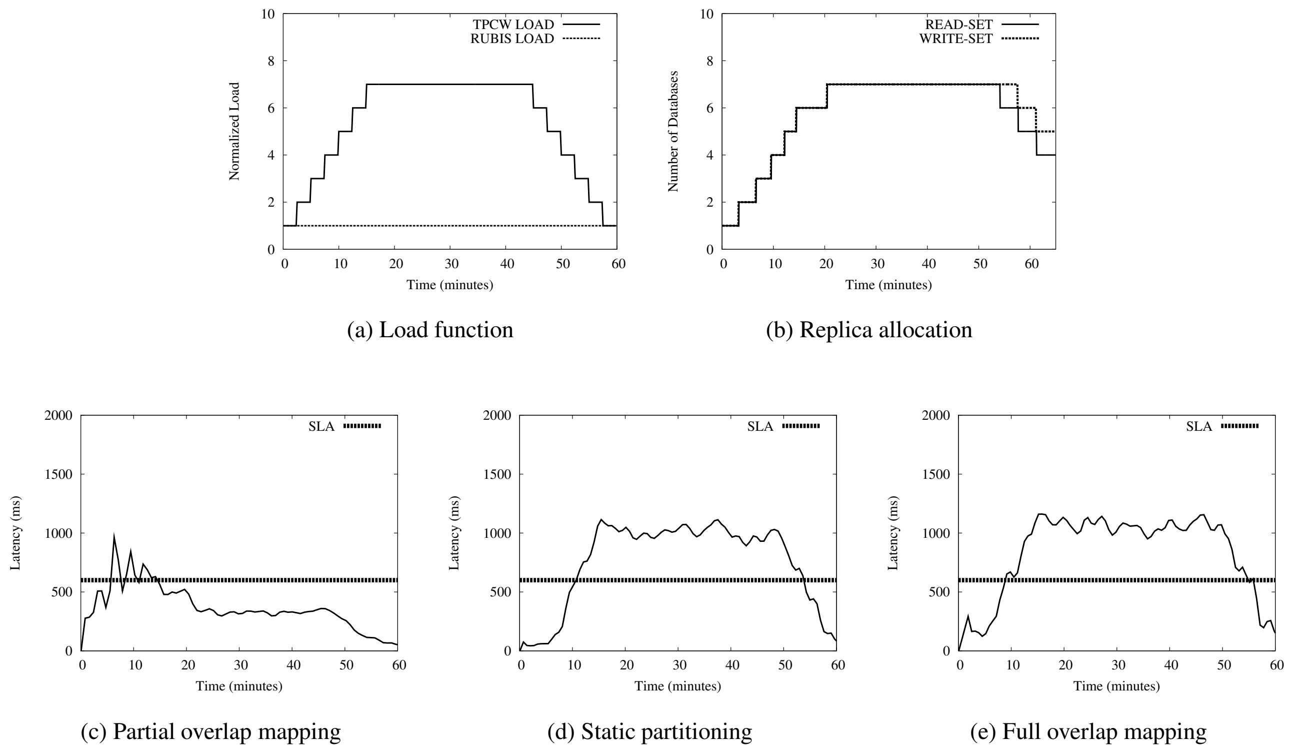
provides 92% SLA compliance.

5.4.2 *TPC-W and RUBIS Workload Adaptation*

In this section, we show the robustness of our system when both the TPC-W and the RUBIS workloads vary dynamically. These experiments also show how our resource manager handles the underload and overload conditions.

Figure 10 shows the complete set of results when running the two varying load benchmarks. Figure 10(a) shows the input load function for the two benchmarks. These loads vary so that the system is in underload initially but becomes overloaded roughly 50-60 minutes into the experiment when the total number of machines needed by the two benchmarks is approximately 11 (load levels 6 and 5) which exceeds the allocated capacity of 8 database machines.

Figures 10(b) and 10(c) show the number of replicas allocated to the TPC-W and the RUBIS benchmarks by our partial overlap mapping scheme. These figures show that the allocations closely follow the load increase during underload. However, the lightweight and irregular nature of the RUBIS workload leads to some oscillation



(a) Load function (b) Replica allocation

(c) Partial overlap mapping (d) Static partitioning (e) Full overlap mapping

Figure 9: Multiple application load, TPC-W load adaptation

in allocation between one and three replicas (mostly in the RUBIS read set) when two replicas appear to be sufficient for RUBIS. Once the system is in overload (roughly after 60 minutes), the system enforces fairness in replica allocation across workloads. In this case, Figure 10(b) shows there are two consecutive forced replica removals from TPC-W so that TPC-W eventually has 4 replicas allocated to it. The TPC-W write set lags behind the read set because ongoing update transactions need to finish on the removed replicas. The two machines removed from the TPC-W workload are added to RUBIS as they become available, and as a result, our dynamic partitioning scheme behaves similar to static partitioning during overload.

The rest of the graphs in Figure 10 show the query latency for TPC-W and RUBIS for the three mapping policies. Figures 10(d) and 10(g) show that our system keeps the query latency under the SLA for almost the entire underload period. During overload, our scheme performs comparably with static partitioning. However, the two consecutive spikes in RUBIS latency during this period are due to misses in the buffer cache in the two machines that were previously running TPC-W. This penalty occurs as a result of a real load change in the system. However, it shows that any unnecessary oscillation in replica allocation is expensive for database replication.

The remaining latency graphs show the impact on varying load on the two static mapping policies, static partitioning and full overlap. Static partitioning performs worse than our scheme in underload for TPC-W because this policy allocates resources equally to both applications immaterial of their needs. Full overlap performs poorly for both applications. The high latency is caused by buffer-cache interference, especially during overload.

5.5 Adaptation to Failures

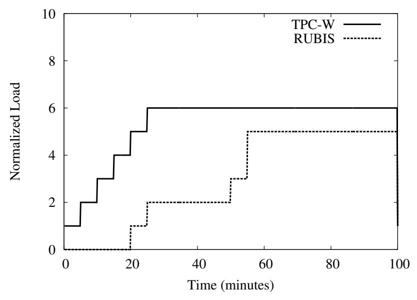
Our dynamic replication system adapts replica allocation to meet application requirements and it uses partial overlap mapping together with warm migration to speed the replica addition process. This approach enables handling database failures as well. In particular, our system treats failures simply as load-inducing events and adds new replicas to meet current demand.

Figure 11 demonstrates the fault-tolerant behavior of our system with a simple, single workload experiment. Figure 11(a) shows the input load function for the TPC-W benchmark. Figure 11(b) shows that the replica allocation matches the input load until 20 minutes into the experiment when a fault is injected into one of the TPC-W replicas. At this point, the TPC-W latency is approximately 300 ms, which is lower than the SLA and therefore the resource manager does not take any action. However, Figure 11(c) shows that the TPC-W latency increases rapidly (as a result of the fault) until it violates the SLA at roughly 22 minutes into the experiment. When the SLA is violated, the resource manager adapts its allocation by adding another replica. At this point, the latency drops to pre-fault levels.

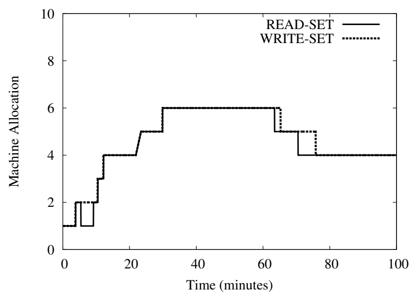
5.6 Sensitivity Analysis

This section shows that our system is robust and does not require careful hand tuning of parameters to achieve good performance. The main parameters in our system are the low and high SLA thresholds and the smoothing parameter α. The high SLA threshold is specified by the application. Below, we show the effects of varying the other two parameters.

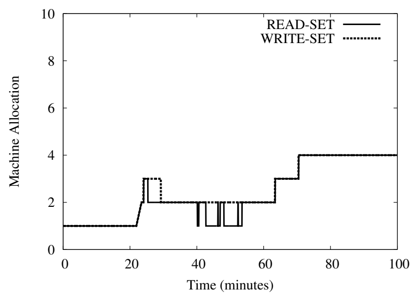
For this study, we use the TPC-W workload, and we designed an input load function that simulates various workload scenarios



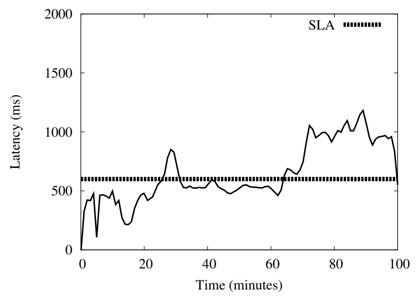
(a) Load function



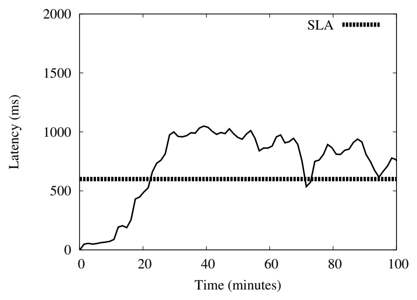
(b) TPC-W replica allocation



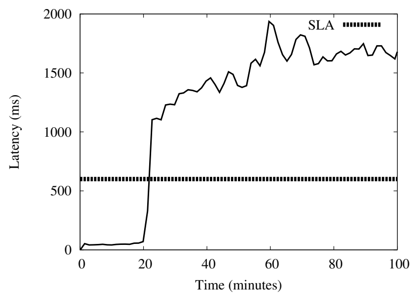
(c) RUBIS replica allocation



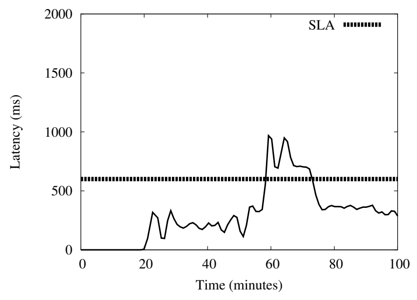
(d) Partial overlap mapping



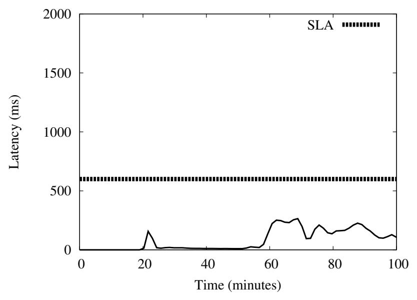
(e) Static partitioning



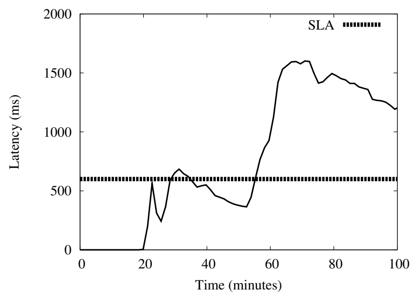
(f) Full overlap mapping



(g) Partial overlap mapping



(h) Static partitioning



(i) Full overlap mapping

Figure 10: TPC-W and RUBIS workload adaptation

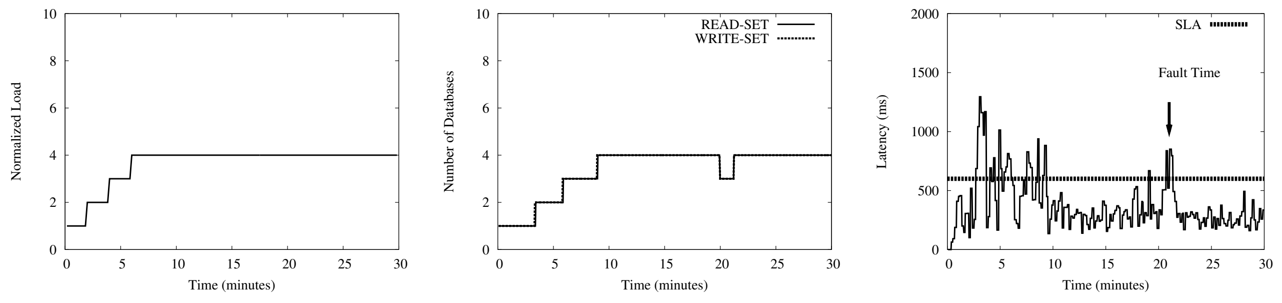
including changes in load, transient spikes, and regions of constant load. This load function, which stresses the system with frequent and high amplitude changes in load, is shown in Figure 12(a).

5.6.1 Variation in the LowSLAThreshold parameter

Figure 12 shows the output of our replica allocation scheme for the TPC-W workload as the `LowSLAThreshold` parameter is varied from $0.75 \times SLA$ to 0. Intuitively, higher values of `LowSLAThreshold` cause more aggressive replica removal and thus the number of machines allocated to a workload will be more precisely matched with the number of machines needed to meet the SLA. However, aggressive removal can lead to oscillatory allocation which is expensive because of buffer-cache interference. To reduce this problem, the allocation policy described in Section 4.2 separates replica removal for the read set and the write set of an application. Removing a replica from the write set can be much more expensive because a later replica addition will require data migration, and hence this removal is performed more conservatively than removing a replica from the read set.

Figures 12(b) through 12(f) show the number of replicas allocated to the read and the write sets of the application. These figures show that higher values of `LowSLAThreshold` cause more responsive read-set allocation (thin lines in the figures), but the more expensive write-set allocation is stable (thick lines in the figures). Small values of `LowSLAThreshold` cause read-set allocation to become less responsive and eventually replicas are never removed unless the system is in overload and a removal is forced to ensure fair allocation.

Table 3 shows the average number of replicas allocated to the read and the write sets and the average TPC-W latency. We see that the average number of replicas grows minimally with decreasing `LowSLAThreshold` and the average latency rises



(a) Load function (b) Replica allocation (c) Query latency

Figure 11: Adaptation to replica failure

LowSLAThreshold	Read Set	Write Set	Latency
$0.0 \times SLA$	2.51	2.51	213 ms
$0.1 \times SLA$	1.74	2.14	251 ms
$0.4 \times SLA$	1.56	2.09	277 ms
$0.5 \times SLA$	1.55	2.08	309 ms
$0.75 \times SLA$	1.41	2.08	303 ms

Table 3: Number of allocated replicas and average latency vs. the LowSLAThreshold parameter

Smoothing parameter α	% Compliance
1	93
0.5	91
0.25	96
0.125	92
0.0625	89

Table 4: Percent compliance vs. the smoothing parameter α

slowly with increasing LowSLAThreshold. Finally, it should be clear that when two or fewer replicas are allocated to the read set then a LowSLAThreshold value greater than $0.5 \times SLA$ causes unnecessary oscillation. As a result, any non-zero value for LowSLAThreshold that is below $0.5 \times SLA$ will yield reasonable performance.

5.6.2 Variation in the Smoothing Parameter α

The smoothing parameter α controls the response of the system. Higher values of α cause the system to react faster to the current value of latency, while lower values of α give more weight to the latency history. While a larger value of α speeds replica addition which helps maintain the SLA, aggressive replica removal can cause oscillatory and expensive re-allocation.

Table 4 demonstrates this trade-off. It presents the percentage compliance for the input load shown in Figure 12(a) as the α parameter is varied. The best compliance is achieved when $\alpha = 0.25$. However, the table also shows that compliance does not vary significantly and is over 90% for any value of the smoothing parameter.

6. RELATED WORK

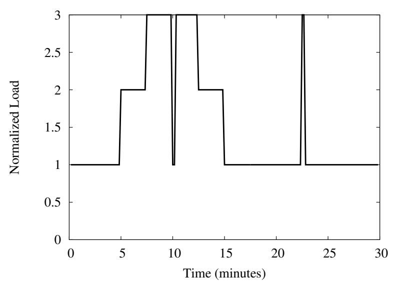
This paper addresses the hard problem of resource allocation and scaling within the database tier of dynamic content applications. This work builds on recently proposed techniques for transparent scaling via content-aware scheduling in replicated database clusters [16, 9, 21, 26], and in particular, uses the infrastructure from our previous work on asynchronous replication with conflict-aware scheduling [4, 5].

The main contribution of this paper is the exploration of database replication schemes in the context of dynamic provisioning. While Kemme et al. [16] propose algorithms for database cluster reconfiguration, the algorithms are not evaluated in detail. This paper proposes and evaluates efficient methods for dynamically integrating a new database replica into a running system and provides a detailed evaluation using realistic benchmarks.

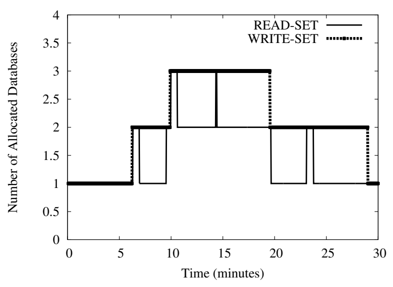
Similar to our work, the Middle-R [21] replicated database system targets adaptation to changing load. Middle-R is a primary-copy replicated system. The adaptation is done at two levels: local adaptation and global adaptation. The local adaptation controls the transaction concurrency level at each database replica using the throughput as a feedback mechanism. The global adaptation varies the assignment of object sets to replicas to reduce the load variance across servers. The main difference compared to our work is that the authors assume that all replicas are always up-to-date, hence the system incurs little or no adaptation delay. Furthermore, the Middle-R system is used with a single application and workload mix. Hence, interference between applications for resources is not considered in the design of the adaptations.

Modeling-based approaches have proved very successful for provisioning the web and application server tiers in dynamic content servers. These existing approaches treat the system as a set of black boxes and simply add boxes to a workload's allocation based on queuing models [7, 36], utility models [29, 34] or marketplace approaches [10].

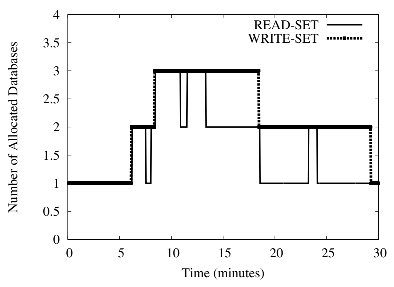
These analytic performance models have been shown to have good accuracy for dynamic resource allocation of web and application servers [7, 36]. On the other hand, to the best of our knowledge, most of these techniques [7, 34, 36] assume a single database back-end for the data center. The exception is a recent study by Urgaonkar et al. [32], which models generic multi-tier data centers using the G/G/1 queuing model. This model captures any arrival distribution and arbitrary service times at each server. Using this basic block, a dynamic resource allocation algorithm determines the number of servers needed at each tier in order to handle the arriving load while satisfying the response time service level agreement. While this work studies allocating web and application servers on demand, it does not study the dynamic allocation of database servers, which require state consistency maintenance. In



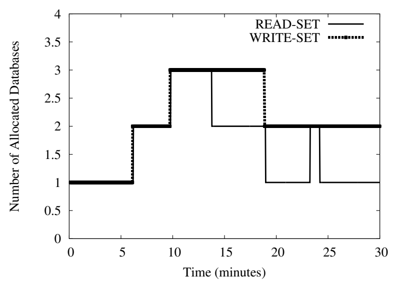
(a) Load function



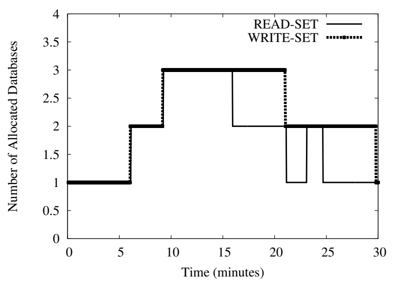
(b) `LowSLAThreshold` = 0.75 SLA



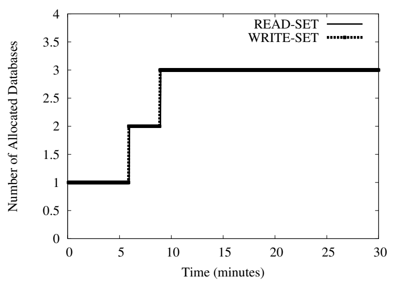
(c) `LowSLAThreshold` = 0.5 SLA



(d) `LowSLAThreshold` = 0.4 SLA



(e) `LowSLAThreshold` = 0.1 SLA



(f) `LowSLAThreshold` = 0.0 SLA

Figure 12: Replica allocation with different `LowSLAThreshold` parameters

contrast, our work focuses on aspects specific to dynamic allocation of database replicas, including data migration to bring a stale replica up to date and awareness of the lag between the adaptation decision time and the time when the results of the decision are seen.

There is a large body of literature related to dynamic resource allocation. Scheduling policies for proportional share resource allocation include Lottery scheduling [33] and STFQ [14]. Steere et al. [28] describe a feedback-based real-time scheduler that provides reservations to applications based on dynamic feedback, eliminating the need to reserve resources a priori. Control-based resource allocation algorithms [12, 18] use models and select various parameters to fit a theoretical curve to experimental data. These approaches are not generic and need cumbersome profiling in systems that run many workloads. For example, the PI controller parameters can be tuned [12], but these parameters are only valid for the tuned workload and not applicable for controlling other workloads. In comparison, our system does not require careful tuning to achieve good performance across applications with widely varying resource needs. Furthermore, unlike most control systems, our system must deal with long delays in the control loop.

Our work is related but orthogonal to ongoing projects in the areas of self-managing databases [20, 13, 19] and automatically reconfigurable static content web servers [25] and application servers [17].

For scaling, an alternative to replication is data partitioning. However, partitioning complicates reconfiguration because it requires global data reshuffling [11] which prohibits processing of common queries such as table joins during reorganization. In contrast, replication allows transparent reconfiguration since replicas can be added without significantly disrupting transaction processing at existing replicas.

7. CONCLUSIONS

In this paper, we propose using database replication to support multiple dynamic content applications on a commodity cluster. Our system dynamically allocates replicas to each application to maintain per-application performance. This dynamic replication approach enables a unified approach to load management as well as fault tolerance.

We use the shopping workload mix of the TPC-W benchmark and the RUBIS on-line auction benchmark to evaluate the response and the stability of our dynamic replication protocol. The evaluation compares alternate replica allocation and mapping policies in detail and shows that our system can handle rapid variations in an application's resource requirements while maintaining quality of service across applications. This is achieved by using 1) a partial overlap replica mapping scheme that minimizes cross-application interference since the read sets of applications are disjoint, and 2) a warm migration algorithm that reduces the replica addition delay. We also show that our replica allocation algorithm avoids expensive replica addition and removal oscillations by taking the replica addition delay into account, and it does not require careful tuning of any parameters. Finally, we show that warm migration works well for scenarios both with and without faults.

Our system avoids modifications to the web server, the application scripts and the database engine and uses software platforms in common use. As a result, our techniques can be easily applied to real web sites.

8. ACKNOWLEDGEMENTS

We thank the anonymous reviewers and the systems group at University of Toronto for their detailed comments and suggestions for improvement on the earlier versions of this paper. We further acknowledge the generous support of IBM Centre for Advanced Studies (CAS) Toronto, IBM Research, the Natural Sciences and Engineering Research Council of Canada (NSERC), Communications and Information Technology Ontario (CITO) and Canadian Foundation for Innovation (CFI).

9. REFERENCES

[1] Oracle Real Application Clusters (Oracle RAC) 10g. `http://www.oracle.com/technology/products/database/clustering/`.

[2] The Apache Software Foundation. http://www.apache.org/.

[3] AMZA, C., CECCHET, E., CHANDA, A., COX, A., ELNIKETY, S., GIL, R., MARGUERITE, J., RAJAMANI, K., AND ZWAENEPOEL, W. Specification and implementation of dynamic web site benchmarks. In *5th IEEE Workshop on Workload Characterization* (Nov. 2002).

[4] AMZA, C., COX, A., AND ZWAENEPOEL, W. Conflict-aware scheduling for dynamic content applications. In *Proceedings of the Fifth USENIX Symposium on Internet Technologies and Systems* (Mar. 2003), pp. 71–84.

[5] AMZA, C., COX, A., AND ZWAENEPOEL, W. Distributed versioning: Consistent replication for scaling back-end databases of dynamic content web sites. In *4th ACM/IFIP/Usenix International Middleware Conference* (June 2003).

[6] AMZA, C., COX, A., AND ZWAENEPOEL, W. A Comparative Evaluation of Transparent Scaling Techniques for Dynamic Content Servers. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)* (April 2005).

[7] BENNANI, M. N., AND MENASCE, D. A. Resource allocation for autonomic data centers using analytic performance models. In *Proceedings of the 2nd International Conference on Autonomic Computing (ICAC)* (2005).

[8] BERNSTEIN, P., HADZILACOS, V., AND GOODMAN, N. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[9] CECCHET, E., MARGUERITE, J., AND ZWAENEPOEL, W. C-JDBC: Flexible database clustering middleware. In *Proceedings of the USENIX 2004 Annual Technical Conference* (Jun 2004).

[10] COLEMAN, K., NORRIS, J., CANDEA, G., AND FOX, A. Oncall: Defeating spikes with a free-market server cluster. In *In Proceedings of the 1st International Conference on Autonomic Computing (ICAC)* (2004).

[11] COPELAND, G., ALEXANDER, W., BOUGHTER, E., AND KELLER, T. Data placement in Bubba. In *Proceedings of ACM SIGMOD* (June 1988), pp. 99–108.

[12] DIAO, Y., HELLERSTEIN, J. L., AND PAREKH, S. Optimizing quality of service using fuzzy control. In *DSOM '02: Proceedings of the 13th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management* (2002), Springer-Verlag, pp. 42–53.

[13] ELNAFFAR, S., MARTIN, P., AND HORMAN, R. Automatically Classifying Database Workloads. In *Proceedings of the ACM Conference on Information and Knowledge Management* (Nov. 2002).

[14] GOYAL, P., GUO, X., AND VIN, H. M. 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).

[15] JHINGRAN, A. Anatomy of a real e-commerce system. In *Proceedings of the ACM SIGMOD* (May 2000).

[16] KEMME, B., BARTOLI, A., AND BABAOGLU, Ö. Online reconfiguration in replicated databases based on group communication. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)* (2001), IEEE Computer Society, pp. 117–130.

[17] LASSETTRE, E., COLEMAN, D. W., DIAO, Y., FROEHLICH, S., HELLERSTEIN, J. L., HSIUNG, L., MUMMERT, T., RAGHAVACHARI, M., PARKER, G., RUSSELL, L., SURENDRA, M., TSENG, V., WADIA, N., AND YE, P. Dynamic surge protection: An approach to handling unexpected workload surges with resource actions that have lead times. In *DSOM* (2003), M. Brunner and A. Keller, Eds., vol. 2867 of *Lecture Notes in Computer Science*, Springer, pp. 82–92.

[18] LI, B., AND NAHRSTEDT, K. A control-based middleware framework for quality of service adaptations. *IEEE Journal on Selected Areas in Communications* (1999).

[19] MARTIN, P., POWLEY, W., LI, H., AND ROMANUFA, K. Managing database server performance to meet qos requirements in electronic commerce systems. *International Journal on Digital Libraries 3* (2002), 316–324.

[20] MICROSOFT RESEARCH. AutoAdmin: Self-Tuning and Self-Administering Databases. http://www.research.microsoft.com/research/dmx/AutoAdmin, 2003.

[21] MILAN-FRANCO, J. M., JIMENEZ-PERIS, R., PATIO-MARTNEZ, M., AND KEMME, B. Adaptive middleware for data replication. In *Proceedings of the 5th ACM/IFIP/USENIX International Middleware Conference*,pp. 175–194, (Oct. 2004),

[22] MySQL. http://www.mysql.com.

[23] PHP Hypertext Preprocessor. http://www.php.net.

[24] PLATTNER, C., AND ALONSO, G. Ganymed: Scalable Replication for Transactional Web Applications. In *Proceedings of the 5th ACM/IFIP/Usenix International Middleware Conference* (Oct. 2004).

[25] RANJAN, S., ROLIA, J., FU, H., AND KNIGHTLY, E. QoS-Driven Server Migration for Internet Data Centers. In *10th International Workshop on Quality of Service* (May 2002).

[26] RÖHM, U., BÖHM, K., SCHEK, H.-J., AND SCHULDT, H. FAS - a freshness-sensitive coordination middleware for a cluster of olap components. In *Proceedings of the 28th International Conference on Very Large Databases* (Aug. 2002), pp. 134–143.

[27] SHEN, K., YANG, T., CHU, L., HOLLIDAY, J. L., KUSCHNER, D., AND ZHU, H. Neptune: Scalable replica management and programming support for cluster-based network services. In *Proceedings of the Third USENIX Symposium on Internet Technologies and Systems* (Mar. 2001), pp. 207–216.

[28] STEERE, D. C., GOEL, A., GRUENBERG, J., MCNAMEE, D., PU, C., AND WALPOLE, J. A Feedback-driven Proportion Allocator for Real-Rate Scheduling. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation* (Feb. 1999).

[29] TESAURO, G., DAS, R., WALSH, W. E., AND KEPHART, J. O. Utility-function-driven resource allocation in autonomic systems. In *Proceedings of the 2nd International Conference on Autonomic Computing (ICAC)* (2005), pp. 70–77.

[30] Transaction Processing Council. http://www.tpc.org/.

[31] Reported configurations from industry for running the TPC-W e-commerce benchmark. http://www.tpc.org/.

[32] URGAONKAR, B., AND CHANDRA, A. Dynamic provisioning of multi-tier internet applications. In *Proceedings of the 2nd International Conference on Autonomic Computing (ICAC)* (June, 2005).

[33] WALDSPURGER, C. A., AND WEIHL, W. E. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation* (November, 1994).

[34] WALSH, W. E., TESAURO, G., KEPHART, J. O., AND DAS, R. Utility functions in autonomic systems. In *In Proceedings of the 1st International Conference on Autonomic Computing (ICAC)* (2004).

[35] WELSH, M., AND CULLER, D. Adaptive overload control for busy internet servers. In *Proceedings of the Fifth USENIX Symposium on Internet Technologies and Systems* (March 2003).

[36] ZHENG, T., YANG, J., WOODSIDE, M., LITOIU, M., AND ISZLAI, G. Tracking time-varying parameters in software systems with ex-tended kalman filters. In *Proceedings of the International Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)* (2005).