Reactive Provisioning of Backend Databases in Shared Dynamic Content Server Clusters

GOKUL SOUNDARARAJAN
University of Toronto
and
CRISTIANA AMZA
University of Toronto

This paper introduces a self-configuring architecture for on-demand resource allocation to applications in a shared database cluster. We use a unified approach to load and fault management based on data replication and reactive replica provisioning. While data replication provides scaling and high availability, reactive provisioning dynamically allocates additional replicas to applications in response to peak loads or failure conditions, thus providing per application performance. We design an efficient method for data migration when joining a new replica to a running application that allows for the quick addition of replicas with minimal disruption of transaction processing. Furthermore, by augmenting the adaptation feedback loop with awareness of the delay introduced by the data migration process in our replicated system, we avoid oscillations in resource allocation.

We investigate our transparent database provisioning mechanisms in the context of multitier dynamic content Web servers. We dynamically expand/contract the respective allocations within the database tier for two different applications, the TPC-W e-commerce benchmark and the RUBIS online auction benchmark. We demonstrate that our techniques provide quality of service under different load and failure scenarios.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems—*Query processing*; H.2.7 [**Database Management**]: *Database Administration—Logging and recovery*

General Terms: Algorithms, Experimentation, Management, Measurement, Performance, Reliability

Additional Key Words and Phrases: Autonomic systems, databases, query processing, transactions

1. INTRODUCTION

Autonomic management of large-scale dynamic content servers has recently received growing attention [IBM Corporation 2003b, 2003b; Tesauro et al. 2005; Bennani and Menascé 2005] due to the excessive personnel costs involved in managing these complex systems. This article introduces a novel technique for

Author's address: C. Amza, Edward Rogers Sr. Department of Electrical and Computer Engineering, University of Toronto, Canada; email: amza@eecg.toronto.edu.

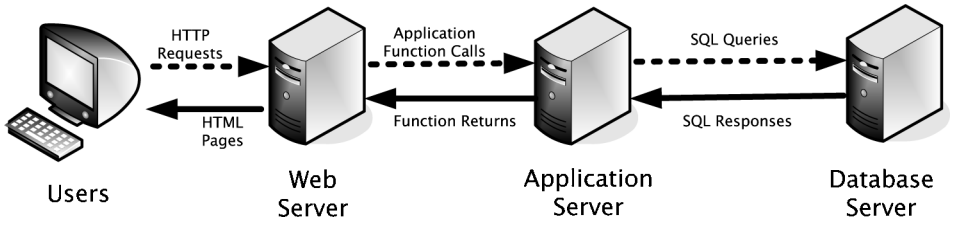


Fig. 1. Architecture of dynamic content sites.

on-demand resource allocation across multiple dynamic-content applications that share a cluster database tier.

Dynamic content servers commonly use a three-tier architecture (see Figure 1) that consists of a frontend Web server tier, an application server tier that implements the business logic, and a backend database tier that stores the dynamic content of the site. Large data centers may host multiple applications concurrently, such as e-commerce, auctions, news, and games. If services experience daily patterns with peak loads for each service type at a different time (e.g., daytime for e-commerce, evening for auctions, morning for news sites, night for gaming), there is opportunity for reassigning hardware resources from one service to another. Thus, instead of gross hardware overprovisioning for each application's estimated peak load, the Web service provider can efficiently multiplex data center resources across applications through dynamic resource allocation, that is, on-demand provisioning.

Several such approaches [Ranjan et al. 2002; DBL ; IBM Corporation 2003b; Bennani and Menascé 2005; Woodside et al. 2006; Menasce et al. 2001; Walsh et al. 2004; Tesauro et al. 2005] investigate dynamic provisioning of resources within the (mostly) stateless Web server and application server tiers. However, dynamic resource allocation among applications in the stateful database tier, which commonly becomes the bottleneck [Amza et al. 2002; slashdoteffect], has received comparatively less attention.

In this article, 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. Furthermore, our algorithm removes resources from an application's allocation when it is in underload. This approach provides the following benefits. First, it allows cost savings by efficient resource usage, for example, allocating resources to applications only when needed may allow powering down unnecessary cluster servers for extended periods of time, thus reducing energy consumption in large data centers. 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.

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 we allocate a *disjoint* set of machines to host the replicas of each application and we dynamically adjust each application's cluster partition. 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 for that application will be stale and must be brought up-to-date (or a new instance of that application may need to be installed) before it can be used through a process we call *data migration*. 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 available 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. This discussion shows that there is a trade-off between using disjoint and fully-overlapped replica allocation strategies. Disjoint allocation reduces interference and thus improves steady state performance. Fully-overlapped allocation avoids replica addition delay and thus can speed up the system's response to load spikes and failures.

Based on this trade-off, we design and implement an efficient method to integrate a database replica into an application allocation that we call application allocation with *partial overlap*. The partial overlap strategy is based on the observation that write queries in dynamic content applications are typically more lightweight and have a much lower memory footprint compared to read queries [Amza et al. 2003a]. 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 involve expensive database joins as a result of complex search criteria. Moreover, read queries are much more frequent than write queries. Hence executing writes of different applications (as opposed to their reads) on the same machine typically causes minimal resource interference. This leads us to a dynamic allocation solution where the application allocations are disjoint, but, for each application, we send batched updates periodically to a set of database machines outside of that application's allocation, thus keeping them partially up-to-date. Our partial overlap scheme limits application interference and keeps data migration time bounded whenever we need to expand an application's allocation.

Finally, another key feature of our dynamic allocation strategy is incorporating *delay awareness* into the adaptation process. Even with our partial overlap allocation technique, the replica addition process can be long (e.g., due to the need for buffer cache warm-up) and the application latency may remain high during this process. A naive reactive policy that measures application latency in order to add replicas, periodically can cause instability even during replica addition. Specifically, sampling high latency values during adaptation causes unnecessary further allocations for that application which later need to be removed, hence oscillation in application allocations and cross-application interference. To improve system stability, our dynamic allocation strategy suppresses

allocation decisions while replica addition is in progress. An additional heuristic for preventing oscillatory allocation behavior is to eagerly react to resource bottlenecks, but to use a conservative replica removal process as follows. Specifically, our system compensates for the delay during adaptation by using early latency indications on a newly added replica to trigger an additional replica allocation. In contrast, for replica removal, a two-step process is used. Initially, we stop read queries to a replica but continue to keep the replica up-to-date. If the application's performance is not significantly affected, then write queries are stopped to remove the replica from the application's allocation.

Our prototype implementation interposes an autonomic manager tier between the application server(s) and the database cluster in the dynamic content server. The autonomic manager tier consists of a resource manager collaborating with a set of schedulers, one per application, to virtualize the database cluster and the application allocations within the cluster so that each application server sees a single database.

Our experimental evaluation uses the TPC-W industry standard e-commerce benchmark [TPC] modeling an online bookstore and the RUBIS online bidding benchmark [Amza et al. 2002]. Our evaluation shows that our dynamic replication approach performs well for rapid variations in an application's resource requirements. We also show experimentally that the same approach can be used to handle failure scenarios. A comparison of various allocation techniques with and without overlap shows that the partial overlap allocation policy is most effective and requires fewer resources than the disjoint or full overlap allocation policies. The partial overlap allocation strategy provides over 90% latency compliance for both applications when they are run together under a range of load and failure scenarios.

The rest of the article is organized as follows. Section 2 describes the environment in which our dynamic adaptation techniques work and specifically provides the necessary background on scaling dynamic content applications and introduces our cluster architecture. Section 3 introduces our partial overlap resource allocation strategy and presents our data migration algorithm. Section 4 describes our delay-aware adaptation process. Section 5 presents our benchmarks and experimental platform. We investigate dynamic allocation of database resources to applications experimentally in Section 6. Section 7 discusses related work and Section 8 concludes the article.

2. ENVIRONMENT

This section describes the environment in which our autonomic adaptations are meant to work. In particular, we describe our underlying database replication algorithm and our cluster design.

In this work, we build on a replication technique called *conflict-aware replication* [Amza et al. 2003a, 2003b], which is among recent research prototypes [Lin et al. 2005, Kemme and Alonso 2000a, 2000b, Plattner and Alonso 2004] providing asynchronous replication with strong consistency guarantees. This combination has proven to be a difficult challenge with traditional replication techniques. Classic eager replication schemes [Weikum and Vossen 2002]

with synchronous updates applied to all database replicas trivially provide strong consistency but severely limit performance, mainly due to excessive conflict waits and deadlocks [Gray et al. 1996]. Classic lazy replication [Guy et al. 1999; Terry et al. 1995] with delayed propagation of modifications, hence asynchronous replication, provides better performance but writes can arrive out of order at different replicas and reads can access inconsistent data.

Both asynchronous execution and strong consistency are necessary in a replicated dynamic content server with autonomic adaptation. As mentioned, a synchronous system typically does not allow increasing application performance by adding new replicas, hence, it limits system adaptivity. Furthermore, integrating a stale replica into an application's allocation should ideally occur without stalling ongoing transactions on already active replicas for that application. This cannot be achieved without asynchronous execution for the set of replicas already active, on one hand, and the new replica that is being brought up to date on the other hand. Finally, strong consistency is desirable because it hides the replicated nature of the backend and its configuration adaptations from the application server which interacts with the replicated backend as with a single database.

In the following, we describe the consistency model we support, the programming model for the application business logic, the cluster architecture and the conflict-aware replication scheme.

2.1 Consistency Model

The consistency model we use for all our protocols is strong consistency or 1-copy serializability [Bernstein et al. 1987]. A replicated system provides 1-copy serializability if it makes the system look like one physical copy to the user. With 1-copy serializability, the execution of all transactions is equivalent to a serial execution, and that particular serial execution is the same on all replicas. Thus, conflicting operations of different transactions execute in the same order on all replicas. We illustrate the consistent conflict ordering concept previously introduced with two examples. Assume that transactions T_1 and T_2 have conflicting accesses on data items x and y. Then, with 1-copy serializability, T_1 is consistently ordered either before T_2 on all replicas (e.g., it cannot read any data that T_2 wrote) or after T_2 on all replicas (e.g., it reads data items versions that T_2 wrote). Given this, the following interleaving of operation executions of transactions T_1 and T_2 cannot occur with 1-copy serializability. In the operation interleaving T_1 r(x), T_2 w(x), T_2 w(y), T_1 r(y), T_1 cannot read the unmodified value of x but it can read the modified value of y. In the operation interleaving, T_1 r(x) T_2 r(y) T_1 w(y) T_2 w(x), both T_1 and T_2 cannot read the unmodified versions of x and y.

2.2 Programming Model

A single (client) Web interaction may include one or more transactions, and a single transaction may include one or more read or write queries. The application writer specifies where in the application code transactions begin and

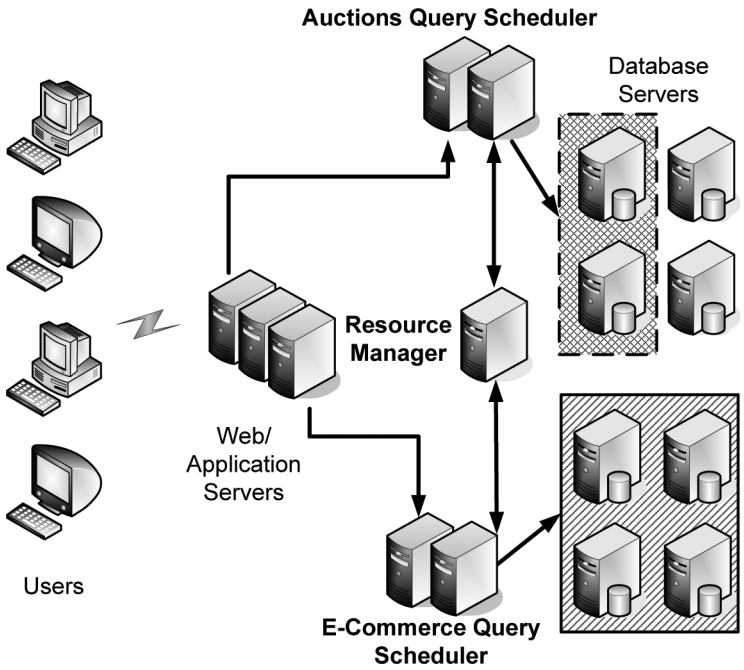


Fig. 2. Cluster architecture.

end. In the absence of transaction delimiters, each single query is considered a transaction and is automatically committed (so called auto-commit mode).

At the beginning of each transaction consisting of more than one query, the programmer inserts a predeclaration of the tables accessed in the transaction and their modes of access (read or write). The tables accessed by single-operation transactions do not need to be predeclared.

2.3 Cluster Design

In our adaptive database replication techniques, different applications share the same cluster of database backends. An application is defined as the entire set of Web interactions and their associated database transactions. All transactions of an application are run on an allocated set of replicas. Each application accesses a distinct set of database tables on their allocated set of machines which may or may not overlap with other application allocations depending on the resource allocation scheme.

Our dynamic-content cluster architecture is shown in Figure 2. A global *resource manager* arbitrates resource allocations between the different applications within the database cluster. A set of *query schedulers*, one per application, distribute incoming requests to the cluster of database replicas and deliver the responses back to the application server. Each per-application scheduler may itself be replicated for availability [Amza et al. 2003a, 2003b]. The presence of the scheduler is transparent to the application server and the database, both of which are unmodified. The application server interacts directly only with the

scheduler in charge of its corresponding application. These interactions are synchronous, so that, for each query, the application server blocks until it receives a response from the scheduler.

Each scheduler uses a set of database proxy processes, one at each database engine, to communicate with the databases. In addition, each scheduler uses a per-application sequencer that assigns a unique sequence number to each transaction. The sequence numbers are used to enforce a total ordering on conflicting update transactions at each database replica. Figure 2 shows the sequencer and resource manager together in one component since, in practice, they can be placed on the same machine or even inside the same process. In the following, we describe the per-application replication scheme that we build on, called conflict-aware replication.

2.4 Conflict-Aware Replication

The key idea in conflict-aware replication is to augment the scheduler that distributes queries on the database cluster with reliable state in such a way as to optimize performance based on asynchronous replication internally, while externally guaranteeing consistent results.

The scheduler uses a conservative concurrency control method to enforce a predefined, consistent ordering of conflicting transactions at all replicas. Specifically, it uses the sequencer to assign a unique sequence number to each transaction at the beginning of the transaction (i.e., at the table predeclaration). The scheduler sends each write-type query (i.e., `INSERT`, `UPDATE`, and `DELETE`) in a transaction to all database replicas and it sends each read-only query (i.e., `SELECT`) to a single replica. The scheduler returns the answer to a write-type query to the application server as soon as it receives a response from any database replica, thus achieving asynchronous query execution. On the other hand, we achieve 1-copy serializability by forcing transactions that have conflicting operations on a particular table to execute in the total order of the sequence numbers assigned to them.

2.4.1 *Scheduling Optimizations.* Due to the asynchrony of write-type transactions, at any given point, some replicas may be up-to-date with the application of writes, while other replicas may contain stale information. Sending a subsequent read query to any database would still result in a serializable execution if the sequence-number order is respected. However, the scheduler optimizes waiting time for each read by sending it only to an up-to-date replica. We call this optimization *conflict-aware* scheduling. Furthermore, since each query in a transaction is serviced synchronously from the time the first read in a transaction executes to the time a subsequent read arrives for scheduling, the load on the database replicas could have changed. The scheduler dynamically adjusts to changes in load by making fine-grained load-balancing decisions at the level of every query. In particular, the scheduler sends a read query to the least loaded replica from the set of up-to-date replicas. To implement these optimizations, the scheduler needs to maintain internal state for all outstanding replicated operations in each transaction as described in more detail in the next section.

2.4.2 *Implementation.* As the application executes the transaction, the application server sends read and write operations to the scheduler. Each table predeclaration is forwarded by the scheduler to the database proxies and treated as a conceptual lock for each declared table in the corresponding read or write mode. The sequencer assigns a unique sequence number to each transaction and thereby implicitly to each of its locks. Since the tables accessed are specified beforehand, all conflicts between two transactions are thus ordered consistently.

The scheduler tags each operation with the sequence number that was assigned to the transaction. Each database proxy regulates access to its database server by letting an operation proceed only if the database has already processed all conflicting operations that precede it in sequence-number order and all operations that precede it in the same transaction. If an operation is not ready to execute, it is queued at the proxy. Single operation read-only transactions execute at a single replica without the need for explicit sequence number assignment.

In the following, we describe the state maintained at the scheduler and at the database proxy to support failure-free execution. Additional state maintained for fault-tolerance purposes is described in Section 2.4.2.3.

2.4.2.1 *The Scheduler's State.* The scheduler maintains for each active transaction, its sequence number and the locks requested by that transaction. In addition, it maintains a record for each operation that is outstanding with one or more database proxies. A record is created when an operation is received from the application server and updated when it is sent to the database engines or when a reply is received from one of them. The record for a read operation is deleted as soon as the response is received and delivered to the application server. For every replicated operation (i.e., table lock request, write, commit, or abort), the corresponding record is deleted only when all database proxies have returned a response.

The scheduler also records the current load of each database in terms of the total estimated complexity of all active queries executing at that databases [Amza et al. 2003a]. This value is updated with new information included in each reply from a database proxy.

2.4.2.2 *The Database Proxy's State.* The database proxy maintains a reader-writer lock [Weikum and Vossen 2002] queue for each table. These lock queues are maintained in order of sequence numbers. Furthermore, the database proxy maintains an out-of-order queue for all operations that arrive out of sequence-number order.

2.4.2.3 *Fault Tolerance of Writes.* To enable fault tolerance and flexible extensions of an application's database allocations, the scheduler maintains persistent logs for all write-type queries (i.e., `INSERT`, `UPDATE`, and `DELETE`) of past transactions in its serviced application. These logs are maintained until all corresponding transactions either commit or abort at all databases in the available database pool. The write logs are maintained per table for the corresponding write queries in order to facilitate later replay. The scheduler

maintains a version number for each table which counts the write-type queries logged for that table. We maintain the log in a relational database, with one relational table per corresponding table in the application schema.

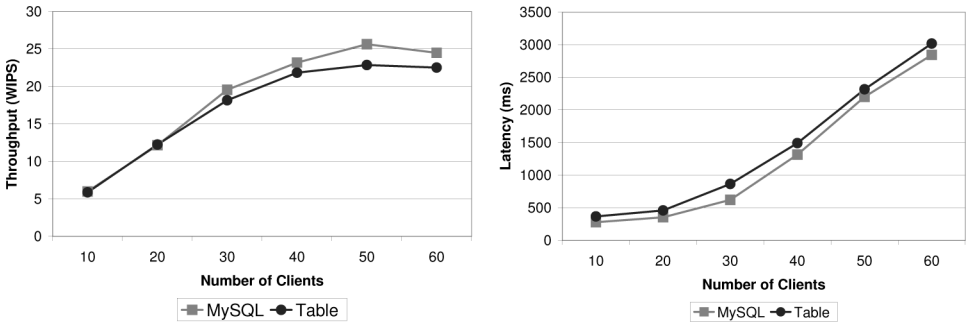
The scheduler maintains all per-table version numbers for its application in an array called *global version vector* ($\vec{G}$). In addition to the global version vector, the scheduler maintains a *replica version vector* ($\vec{R}$) for each database replica. This data structure keeps track of the number of updates applied by the replica. These data structures are used when bringing a new database replica up-to-date through data migration.

2.5 Motivation for Database Replication with Dynamic Resource Allocation

In this section, we argue that the trade-offs that we make in our underlying replication technique are worthwhile due to common characteristics of dynamic content applications. A conflict-aware scheduler enforces 1-copy serializability by consistent transaction ordering at the table level, and it reduces conflict waiting time by directing reads to replicas where no conflicts exist. This design matches the characteristics of the database workloads that we have observed in dynamic content sites, namely, the high cost of reads relative to the cost of writes, high locality, and high conflict rates. In the following, we show how these characteristics favor our design. Then we present experimental results that motivate the use of dynamic database replication.

First, we have shown in an earlier workload characterization study [Amza et al. 2002] that the typical read-only transactions in dynamic content applications are much more frequent than their write-type transactions (usually by a ratio of 5 to 1 or higher) and much more complex than write-type transactions (usually by more than an order of magnitude). Hence, the critical conflict waits to optimize are read-write conflicts between long read-only queries and conflicting updates, while write-write conflicts between write-type transactions become a bottleneck only at large cluster sizes [Amza et al. 2003a]. The conflict-aware scheduler reduces read-write conflict waiting time by asynchronous execution of updates and by load balancing reads on conflict-free replicas. We have also previously shown [Amza et al. 2003b], that for write-intensive workloads, the overhead of conservative table locking can be partially alleviated by reducing the duration of the conservatively perceived conflicts through explicit marking of the last use for each table in a transaction.

Second, dynamic content applications exhibit significant locality in their access patterns. For example, in online shopping [TPC], bestsellers, promotional items, and new products are accessed with high frequency. Similarly, the stories of the day on a bulletin board or the active auctions on an auction site receive the most attention [Amza et al. 2002]. Due to locality in access patterns, for the usual range of application versus memory sizes (e.g., all industry-standard sizes of TPC-W and typical memory sizes of commodity servers), much of the working set can be captured in memory. As a result, disk I/O is limited [Amza et al. 2002]. This characteristic favors replication as a method for distributing the data compared to alternative methods such as data partitioning. Replication is suitable to relieve hot spots, while data partitioning is more suitable for



(a) Throughput (b) Latency

Fig. 3. Overhead of scheduler-based concurrency control versus MySQL row-level locking.

relieving high I/O demands [Boral et al. 1990]. Thus we choose to investigate full replication schemes instead of data partitioning or using partial replication schemes. As a bonus, full replication also brings with it high data availability.

Third, not only reads, but updates to the database exhibit locality as well. For instance, the probability that a bestseller book is bought and its item stock is updated, a comment is posted for the story of the day, or a bid is placed on one of the hot items, is higher than for other books, stories, and auctioned items, respectively. The locality in access patterns of dynamic content applications thus implies higher conflict rates between transactions relative to traditional applications. In other words, the probability that a table conflict is in fact a true row-level conflict is intuitively higher than for traditional OLTP applications (e.g., banking transactions) where most read and write access patterns are uniformly distributed onto the table data.

Finally, our conservative concurrency control algorithm brings with it two benefits (i) *deadlock prevention*, that is, given that the probability of deadlock increases approximately quadratically with the number of replicas [Gray et al. 1996], any concurrency control algorithm that allows deadlock is undesirable for large clusters. and (ii) *transparency*, meaning that no modifications are necessary in the application server or in the database to take advantage of a conflict-aware scheduler.

Next, we present experimental results that support the arguments just presented and motivate the use of our underlying replication algorithm and this article's exploration of a dynamic resource allocation strategy with partial overlap.

2.5.1 *Motivating Baseline Results.* Figure 3 shows the overhead introduced by the scheduler concurrency control and query indirection through the scheduler compared to the baseline row-level locking of the database. We use a single MySQL database with InnoDB tables running the standard shopping workload mix of the TPC-W benchmark in all our experiments. We use enough Web server machines to make sure that the Web server stage is not the bottleneck. We report the throughput, in Web interactions per second (WIPS) and latency at the client in milliseconds with an increasing number of clients. In one experiment (Table), queries are intercepted by the scheduler and forwarded

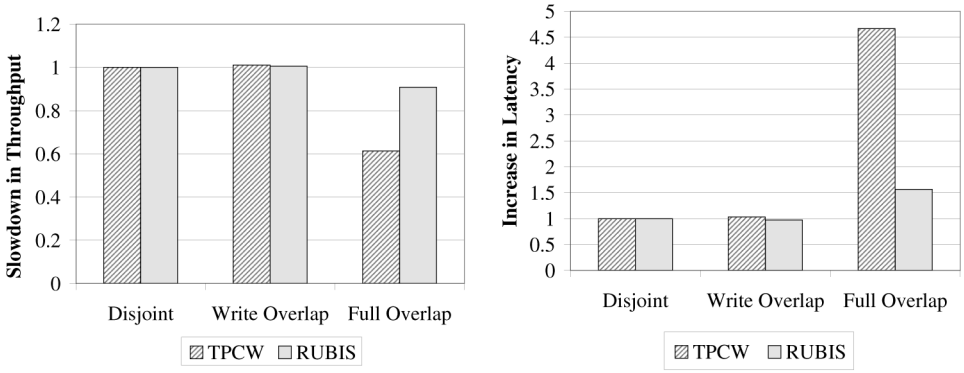
to the database through the database proxy, which implements per-table ordering. In the second experiment (MySQL), the scheduler is not present, and the application server sends queries directly to the MySQL database. In both cases, the MySQL database system is unmodified and uses row-level locking, its built-in concurrency control algorithm. As we can see from the figure, the scheduler overhead is minimal in terms of both throughput and latency.

Next, we present scaling results for our replication scheme from our previous work [Amza et al. 2003a]. Figures 5(a) and 5(b) 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, and a curve obtained through simulation for larger clusters of up to 60 database machines.

Our simulation with large numbers of replicas uses a simulator we introduced in our previous work [Amza et al. 2003a]. The simulator was calibrated using the real 8-node cluster. Figures 5(b) and 5(b) show that the simulated 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 read versus write transactions is 50 to 1 for TPC-W and 8 to 1 for RUBIS.

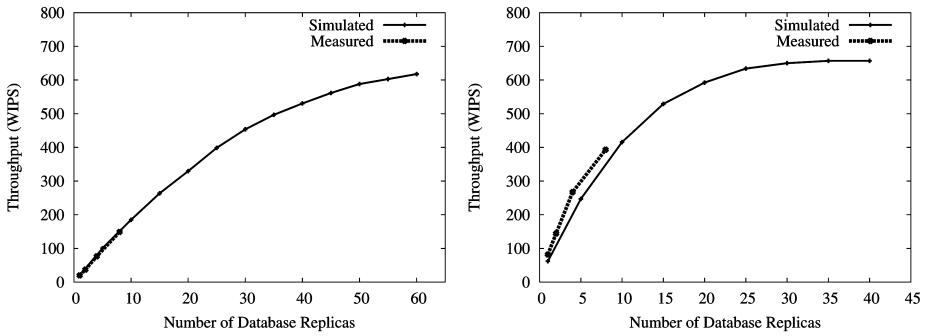
While database replication is appealing, it requires appropriate strategies for dynamic replica allocation. As described earlier, the two most significant and competing challenges in the design of these policies are the delay associated with replica addition and the buffer cache interference with overlapping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

We conduct an experiment with the TPC-W and RUBIS applications using three configurations on our experimental cluster. The applications are run (i) on separate machines (disjoint), (ii) load balanced on both machines together (full overlap) or (iii) reads are serviced from a separate machine for each individual application but both machines are maintained up-to-date for both applications (write overlap). Figure 4 shows the throughput and latency for TPC-W and RUBIS when run on a cluster of two database machines in these three configurations. In all configurations, the results reported are after the buffer pool has been warmed up. With disjoint allocation, TPC-W has a memory footprint that fits in available memory, while RUBIS's footprint exceeds total available memory. However, when the applications share each of the two machines, the RUBIS application 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. Based on the time each application spends waiting for I/O as reported by the `vmstat` Linux utility, the percent of total CPU time waiting for I/O increases from 0% to 44% for TPC-W and from 31% to 38% for RUBIS in the fully overlapped versus disjoint configurations. As a result, the TPC-W throughput is halved with full overlap allocation compared to disjoint allocation and its query latency is



(a) Throughput Slowdown (b) Latency Increase

Fig. 4. The effect of interference in the buffer pool.



(a) TPC-W (b) RUBIS

Fig. 5. Throughput scaling with increasing database cluster size for TPC-W and RUBIS.

more than doubled. 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. Finally, we see that overlapping only the writes of the two applications does not degrade performance significantly, hence the interference effect is not present in the write overlap allocation.

3. DYNAMIC REPLICA ALLOCATION WITH PARTIAL OVERLAP

We distinguish between an application's read and write replica sets, called *read set* and *write set*, respectively. The read set is the set of machine replicas from which an application reads. Likewise, the write set of an application is the set of replicas that are maintained fully up-to-date with the writes of the application through our underlying replication scheme. The transaction delimiters and write-type queries are sent to all databases in an application's write set. For a particular application, the read set and write set may be different. However, the read set of an application is always a subset of its write set.

Motivated by the experimental results shown in the previous section, we keep application read sets disjoint in order to avoid application interference. The write set of each application is the same as its read set except during adaptation for expanding an application's resource allocation. Finally, in our algorithm, in addition to the replicas within an application's write set, we maintain a set of additional replicas within a staleness bound. These replicas are an overflow pool used for rapid adaptation to temporary load spikes since data migration onto them is expected to be relatively fast. All overflow replicas are updated through batched updates whenever they violate the staleness bound. They may belong to the read and/or write sets of other applications. The overlap region between applications is configurable in our system. To simplify algorithm analysis for the purposes of this article, we will henceforth assume that, for a given application, the overflow replica set consists of all other replicas available in the system outside the application's write set partition. A smaller overlap region may become necessary if a large number of applications share the cluster, hence execution of updates on behalf of all applications on all database machines becomes prohibitive.

In the following section, we introduce our data migration algorithm for incorporating a new replica into the write set of an application.

3.1 Data Migration

Due to the stateful nature of databases, the allocation of a database to an application requires the transfer of data to bring the newly added replica up-to-date. Our data migration algorithm is designed to transfer the current state to the joining database with minimal disruption of transaction processing. To bring a stale database replica up-to-date, the scheduler has to send it all missing updates from the on-disk update logs to it. When the update log is replayed on the new replica, data migration is complete. The challenge for an effective data migration implementation is that new transactions continue to update the databases in the application's allocation, while data migration is taking place. Hence, the scheduler needs to add the new database replica to its application's replica write set before the end of data migration, otherwise the new replica would never catch up. Unfortunately, any updates made after the start of migration cannot be directly applied to the stale replica. New update queries are thus queued at the new replica and applied only after data migration is complete. Since the migration time can be long, the in-memory queue of updates at that replica may grow without bound, overflowing the available memory space for that replica.

In order to prevent this problem, in our implementation, 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. This approach reduces the number of logged 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 is able to concurrently send both old queries from disk logs and new updates to the replica that is being added.

Algorithm 1 Data Migration

Require: Database replicas $DB_1 \dots DB_{n_R}$ from which $DB_1 \dots DB_i$ are currently used, *global version vector* $\vec{G}$, *replica version vectors* $\vec{R}_1 \dots \vec{R}_{n_R}$, *batching threshold* BT, database schema containing n_T tables $\{T_1 \dots T_{n_T}\}$
Ensure: Database replica DB_j is added to the *write set*
1: **While** $(\vec{G} - \vec{R}_j) > BT$ **do**
2: *Set the migration checkpoint vector*: $\vec{M} \leftarrow \vec{G}$
3: **for all** table t in $\{T_1 \dots T_{n_T}\}$ **do**
4: **for** $v = \vec{R}_j[t] + 1$ to $\vec{M}[t]$ **do**
5: Get update U with version v for table t from the LOG
6: Send update U to replica DB_j
7: **end for**
8: **end for**
9: **end while**
10: Initialize *trap version vector* $\vec{TV}$
11: Send `BEGIN_MIGRATION` message to replica DB_j
12: **for all** table t in $\{T_1 \dots T_{n_T}\}$ **do**
13: **for** $v = \vec{R}_j[t] + 1$ to $min(\vec{G}[t], \vec{TV}[t] - 1)$ **do**
14: Get update U with version v for table t from the LOG
15: Send update U to replica DB_j
16: **end for**
17: **end for**
18: Send `END_MIGRATION` message to replica DB_j

Algorithm 1 shows the pseudocode for the data migration process. Assume that the overall database pool contains n_R replicas $DB_1 \cdots DB_{n_R}$. The replicas use a schema that contains n_T tables $\{T_1 \cdots T_{n_T}\}$. The scheduler maintains two version vectors (1) *global version vector* $(\vec{G})$ and (2) a *replica version vector* $(\vec{R}_j)$ for each database replica DB_j where $0 \leq j \leq n_R$. The version vectors $\vec{G}$ and $\vec{R}_j$ store n_T values while $\vec{G}[t]$ indicates how many writes have been applied to table t, and $\vec{R}_j[t]$ indicates how many writes have been applied to table t by database replica DB_j.

As shown in line 1 of the algorithm, before every stage, the scheduler checks whether the difference between $\vec{G}$ and the replica's $\vec{R}_j$ for each table is manageable (i.e., less than some bound). If not, then the scheduler records the current global version vector $\vec{G}$ into a *migration checkpoint vector* $(\vec{M})$ (line 2). The *migration checkpoint vector* records the highest version of each table corresponding to all committed transactions at the start of the current stage. The scheduler then transfers a batch of old logged updates with versions between $\vec{R}_j[t]$ and $\vec{M}[t]$ for each table t, respectively, to the replica without sending any new queries (lines 3–8). This reduces the number of logged updates to be sent after each stage until, during the last stage, when the scheduler is able to concurrently send new updates to the replica that is being added. Since during this last phase of migration, the new queries that are sent are also logged by the scheduler and the global version vector keeps increasing, the scheduler uses an additional mechanism to avoid sending duplicate queries to the new replica. In particular, the scheduler records the version number of the first new query sent to each table on the replica under migration into a *trap version vector* $(\vec{TV})$

(line 10). The scheduler then sends old queries from the log up to the minimum of $\vec{G}[t]$ and $\vec{T}V[t] - 1$ for each table t (lines 13–16). When data migration is finished, the scheduler sends an `END_MIGRATION` message to the database proxy (line 18). After receiving this message, the proxy begins the processing of the queued write queries.

3.1.1 *Warm Migration.* Data migration time can be long if the database replica that is to be incorporated has not been updated in a long time. Hence, in order to ensure predictable adaptation costs, as mentioned before, we keep the data migration time within a tight bound by sending batched updates periodically to databases outside of an application's allocation. We call data migration with periodic batch updates *warm migration*. Updates are batched until a batching threshold (BT in Algorithm (1)) is reached. Once the batching threshold is reached a short duration data migration is initiated for just the updates in the batch for all database replicas that we intend to keep warm. The batch threshold is set based on a desired upper bound for the delay introduced by the application of the maximum size batch. There is a trade-off between the data migration time (i.e., the system reactivity) and the potential interference caused by frequent batch updates for other applications. As we have seen from our baseline experiments, the interference effect of writes is minimal with two applications. Hence, this trade-off cannot be fully explored unless several concurrent applications share the cluster. Other considerations, such as the possibility of aggregating queries inside a batch for more efficient application of writes (e.g., several `INSERT` queries into a single `INSERT` query), potential energy savings from extended periods of idleness, or the need to frequently garbage collect log updates may also play a role in selecting the optimal batch size. Exploring these considerations is beyond the scope of this article. We select a batching threshold of 1,000 updates, which allows us to add a new database in roughly 2.5 minutes.

3.1.2 *Database Checkpoints.* Even with update batching, the size of the database update log may grow without bound in the case where the update batch is not applied on all replicas (e.g., during database failure). To minimize the reintegration time of very stale databases, for example, after a long-term failure, and to facilitate periodic garbage collection of old update log entries maintained by the scheduler, we also implement database *checkpoints* at regular intervals. Each database proxy performs periodic checkpoints of its corresponding database together with the current state in terms of the version numbers of its database tables. To make a checkpoint, the database proxy stops all write operations going out to the database engine, and when all pending write operations have finished, it takes a snapshot of the database (i.e., copies all tables) and adds the new state to the checkpoint, that is, the *replica version vector*. If any tables have remained unchanged since the last checkpoint, they do not need to be included in the new checkpoint.

In the case of adding a new database, which is either too stale or does not have a database image for the required application, we import an existing

checkpointed database image including its table version numbers from one of the existing replicas. We then use the previously described data migration algorithm to bring this replica fully up-to-date. A full checkpoint currently takes on the order of minutes to perform for a 4GB database. Although we implement a basic checkpointing scheme in our prototype which we do not evaluate in this article, well-known techniques for minimizing the time for taking file snapshots exist [Hutchinson et al. 1999].

4. REACTIVE PROVISIONING WITH DELAY AWARENESS

In this section, we describe the reactive control loop of our autonomic resource manager, which allocates replicas to applications in order to meet predefined per-application service level agreements (SLA).

We define the SLA as maintaining the *average query latency* for a particular application under a given value. Specifically, we determine the fraction of the total end-to-end latency that the query latency represents on average and derive a conservative upper bound for the query latency such that the end-to-end latency is met with a high probability.

The schedulers keep track of average query performance metrics and communicate performance monitoring information periodically to the resource manager. All schedulers use the same sampling interval for the purposes of maintaining these performance averages and communicating them to the resource manager. The resource manager, based on its global knowledge of each application's SLA requirements and their perceived performance, makes database allocation decisions for all applications. The decisions are communicated to the respective application schedulers, which act accordingly by including or excluding databases from their database read and/or write sets for their corresponding application. The resource manager increases an application's allocation if the respective application is perceived to be in overload as long as the total resources available are not exceeded. When the overall system is in overload, we revert to a fairness scheme that allocates an equal share of the total database resources to each application.

Our reactive provisioning algorithm has two key components: (1) per-application performance monitoring and (2) adaptation delay awareness through a state machine approach. Performance monitoring is used to trigger adaptations in response to impending SLA violations for any particular application. At the same time, the resource manager uses a state machine approach to track the system state during adaptations in order to trigger any subsequent adaptations only after the changes of all previous adaptations have become visible. This closes the feedback loop and avoids unnecessary overreaction and oscillation in the system. We explain the two key ingredients of our main protocol in detail in the next sections.

4.1 Per-Application Performance Monitoring

To prevent triggering adaptations upon short and sudden spikes, our resource manager uses a smoothened response time in its adaptation decisions. Smoothing is achieved through a commonly-used [Welsh and Culler 2003]

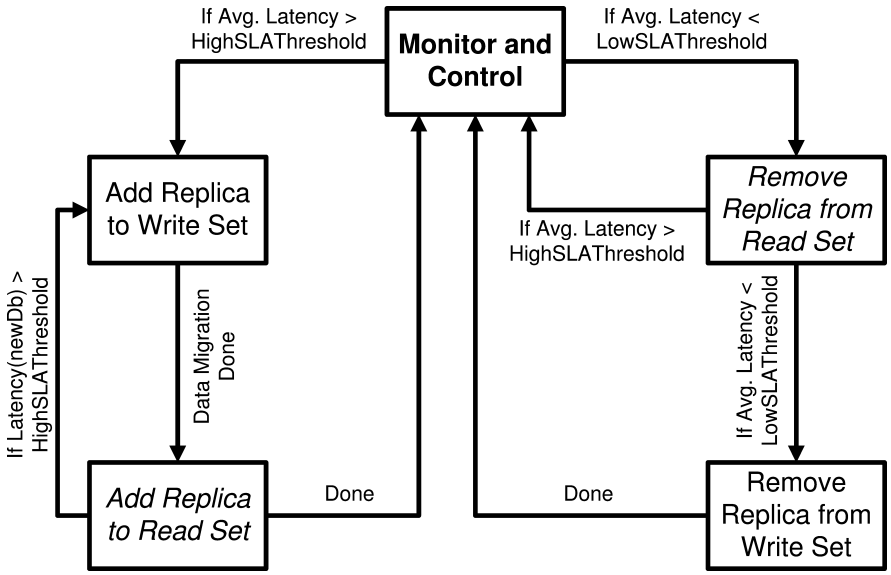


Fig. 6. Replica allocation logic.

exponentially-weighted running latency average of the form $WL = \alpha \times L + (1 - \alpha) \times WL$, where L is the current query latency. A larger value of the α parameter makes the average more responsive to current changes in the latency. The resource manager uses $\alpha = 0.25$. Our results are, however, relatively independent of the precise value of the α parameter. Specifically, we did not need to choose a different set of parameters for each benchmark nor did we choose different parameters for smoothing for the purposes of adding and removing databases to/from an application's allocation.

4.2 Feedback Loop with Adaptation Delay Awareness

Figure 6 presents the main states used in the feedback loop of the resource manager. We explain the state transitions for adding and removing databases, respectively, in more detail in the following two sections.

4.2.1 *Adding Databases to an Application's Allocation.* The resource manager starts in the initial steady state, called `MonitorAndControl`, where the resource manager monitors the average latency received from each application scheduler during each sampling period. If the average latency over the past sampling interval for a particular application exceeds the `HighSLAThreshold` and hence an SLA violation is imminent, the resource manager places a request to add a database to that application's allocation. This request may involve data migration to bring up a new database for that application. Since the request for adding a new database may not be fulfilled immediately, latency sampling is suspended at the resource manager until the request has been fulfilled and the result of the change can be observed. This implies waiting in the `AddReplicatoWriteSet` until (i) state a free machine can be found and allocated

to the overloaded application and (ii) while the newly added database replica is brought up-to date, if necessary.

The replica is then added to the application's read set, and the scheduler starts sending read queries of the application to the new replica. However, during replica addition, the system as a whole may continue to register high latencies until system stabilization. We use a heuristic in order to detect system stabilization and thus avoid oscillating between eagerly adding and removing databases for a particular application. Thus, the `AddReplicatoReadSet` state includes waiting for system stabilization which involves load balancing and warmup of the buffer pool on the new replica. 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. 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 application and we transition directly into adding another replica in this case. Otherwise, we first wait for system stabilization before resuming overall latency sampling and potentially adding another database to the application if the SLA is still not met at that time.

4.2.2 *Removing a Database from an Application's Allocation.* The resource manager removes a database from an application allocation in either of the following two scenarios: (i) the application is in underload for a sufficient period of time and does not need the database (voluntary removed) or (ii) the system as a whole is in overload and fairness between allocations needs to be enforced (forced removed).

In the former case, as we can see from the right branch of the finite state machine diagram in Figure 6, the removal path is conservative with an extra temporary remove state on the way to final removal of a database from an application's allocation. This is another measure to avoid system instability by making sure that an application is indeed in underload and remains quiescent in a safe load region even if one of its databases is tentatively removed.

In our current system, this wait is achieved by tentatively removing a database from an application's read set (but not from its write set) if its average latency has been under the `LowerSLAThreshold` for the last sampling interval. After the replica is removed from the application's read set, we loop inside this state for a configurable number of confidence intervals, `RemoveConfidenceNumber`, before transitioning into the `RemoveReplicafromWriteSet` state. Each confidence interval is of the same length as the estimated duration of warm data migration for the particular application. Hence, we base our observation period on the data migration penalty and leave some leeway for making even more conservative decisions by looping in this state more than once. Subsequently, the feedback loop initiates the final removing procedure of the database from the application's write set and tracks this process until complete.

4.3 Alternative Dynamic and Static Allocation Algorithms

In this section, we introduce a number of other scheduling algorithms for comparison with our main dynamic partial-overlap allocation algorithm with warm migration. By using alternates for some of the features of our main algorithm, we are able to demonstrate what aspects contribute to its overall performance.

Specifically, we consider various degrees of allocation overlap from disjoint to full overlap of application allocations and both static and dynamic allocation algorithms. We also consider an algorithm without delay-awareness in the adaptation control loop. Hence, we distinguish the following alternative dynamic allocation algorithms using different database read and write sets and either delay-aware or delay-oblivious adaptation.

4.3.1 *Dynamic Disjoint Allocation with Cold Migration.* In this scheme, each application is assigned a current partition of the database cluster. We keep up-to-date only the databases in the particular application's allocation and any database within the partition can be selected to service a read of the particular application. The protocol uses our finite state machine approach in a similar way as our main protocol to dynamically adjust partition allocations. If we need to extend an application's partition, data migration time can be long if the database replica to incorporate has not been updated in a long time. The protocol's potential benefit is ensuring zero interference between applications during periods of stable load, thus increasing the probability that each individual working set fits in each database buffer cache.

4.3.2 *Dynamic Partial-Overlap Allocation with Hot Migration.* Writes of all applications are sent to all databases, while the read set of each application is allocated a specific partition. Logging of write queries is unnecessary in this protocol. Since all databases are up-to-date, we can quickly add many databases to an application's read set allocation. On the other hand, this protocol does not extend to the general case of large clusters running many concurrent applications where writes can become a bottleneck. In addition, since persistent logs are not kept, the protocol needs to special case the treatment of database failures. In this case, reintegrating a failed replica always copies an existing database snapshot from another replica.

4.3.3 *Dynamic Partial-Overlap Allocation with Stateless Scheduling.* Any of the dynamic allocation protocols with warm, hot or cold migration can be combined with a stateless resource manager. Instead of following our state machine approach, the resource manager simply reacts to any reported above high or below-low threshold smoothened average query latency during a particular sampling interval by increasing or decreasing the application allocation, respectively. This technique is similar to overload adaptation in stateless services [Welsh and Culler 2003] where simple smoothing of the average latency has been reported to give acceptable stability to short load spikes.

4.3.4 *Fully-Overlapped Allocation.* In this approach, the writes of all applications are sent to all databases. Each read query of any application can be

sent to any replica at any given point in time. This protocol does not use our finite state machine transitions since the read sets and write sets of all applications contain all machines and are never changed. The protocol offers the advantage of fine-grain multiplexing of resources, hence high overall resource usage. Furthermore the protocol offers the flexibility of opportunistic usage of underloaded databases for either application under small load fluctuations. On the downside, both reads and writes of all applications share the buffer cache on each node, hence poor performance can occur if the buffer cache capacity is exceeded.

4.3.5 *Static Partitioning.* This is a standard static partitioning protocol where each application is assigned a fixed, predefined partition of the database cluster. The read set of each application is the same as the write set. Both contain the machines within the fixed application partition and never change.

5. EXPERIMENTAL SETUP

5.1 Platform and Methodology

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 [Apache] and the MySQL 4.0.16 database server with InnoDB tables [MySQL]. 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. Furthermore, the machine running the client emulator is not the bottleneck in any experiment. The MySQL databases are run on 8 machines. For all experiments, we run each individual experiment repeatedly and present either measurement averages or confidence intervals in the corresponding graphs.

To generate the input load function for each application in our experiments, we start with a number of clients for which one database is sufficient to service the requests for that application (which is considered a level 1 load). We then change the number of clients used by the client emulator for that application dynamically during the experiment and represent the load as the number of clients normalized to the initial (level 1) client load. We select a batching threshold of 1,000 updates for all experiments involving warm migration which allows us to add a new database to an application's write set in roughly 2.5 minutes.

5.2 Benchmarks

5.2.1 *TPC-W E-Commerce Benchmark.* The TPC-W benchmark from the Transaction Processing Council [TPC] is a transactional Web benchmark designed for evaluating e-commerce systems. Several interactions are used to

simulate the activity of a retail store. The database size is determined by the number of items in the inventory and the size of the customer population. We use 100K items and 2.8 million customers which results in a database of about 4GB.

The inventory images, totaling 1.8GB, 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. The complexity of the interactions varies widely, with interactions taking between 20ms and 1s on an unloaded machine. Read-only interactions consist mostly of complex read queries in autocommit mode, up to 30 times more heavyweight than read-write interactions containing transactions. The weight of a particular query (and interaction) is largely independent of its arguments.

We are using the TPC-W shopping mix workload with 20% writes which is considered the most representative e-commerce workload by the Transactional Processing Council.

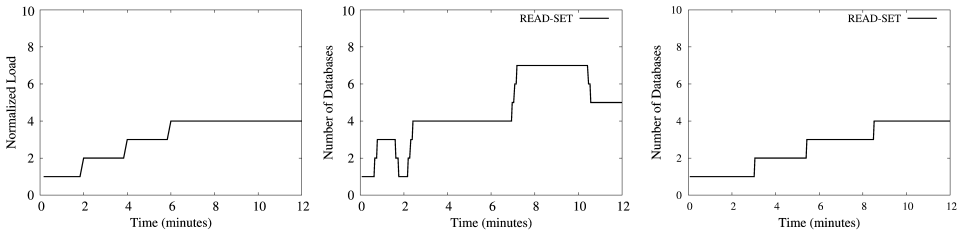
5.2.2 *RUBIS Auction Benchmark.* We use the RUBIS Auction Benchmark to simulate a bidding workload similar to e-Bay. The benchmark implements the core functionality of an auction site: selling, browsing, and bidding. We do not implement complementary services like instant messaging or newsgroups. We distinguish between three kinds of user sessions: visitor, buyer, and seller. For a visitor session, users need not register but are only allowed to browse. Buyer and seller sessions require registration. In addition to the functionality provided during the visitor sessions, during a buyer session, users can bid on items and consult a summary of their current bid, rating, and comments left by other users.

We are using the default RUBIS bidding workload containing 15% writes, considered the most representative of an auction site workload according to an earlier study of e-Bay workloads [Shen et al. 2001].

5.3 Client Emulator

We implemented a client browser emulator. 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 given 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 specified mean.

The client emulator is a lightweight implementation in `C/C++` which consumes very little of the system's resources. Through experimentation, we notice that we can support more than 500 clients on a single machine. We can also run multiple instantiations of the client emulator concurrently on different machines.



(a) Load function (b) Basic allocation policy (c) Delay-aware allocation policy

Fig. 7. Comparison of replica allocation techniques with and without delay awareness.

6. EXPERIMENTAL RESULTS

In this section, we present our experimental results. First, we show the adaptation process using a single application. Second, we show how we adapt to satisfy the demands of multiple applications. In addition, we show our fairness scheme when the system is in overload. Third, we present how our architecture can be used to recover from failures in the backend. Finally, we present sensitivity results showing the performance of the system in response to changes in system parameters.

6.1 Single Application Adaptation

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

6.1.1 *Impact of Adaptation Delay Awareness.* Figure 7 shows the results of using two replica allocation techniques. The input load function is shown in Figure 7(a). The first technique is delay oblivious and uses continuous latency sampling to trigger replica addition or deletion when the average latency rises above the `HighSLAThreshold` parameter or falls below the `LowSLAThreshold` parameter, respectively. The second technique is delay-aware and implements the replica allocation technique described in Section 4.

In this experiment, we use partial overlap with hot migration as the replica allocation 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. During the instability period after replica addition, 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 the demand without overallocation.

6.1.2 *Reactivity of Replica Allocation Techniques.* Figure 8 compares the results of using the cold and warm replica allocation techniques with partial overlap. We initially subject the system to a load that requires 3 databases to

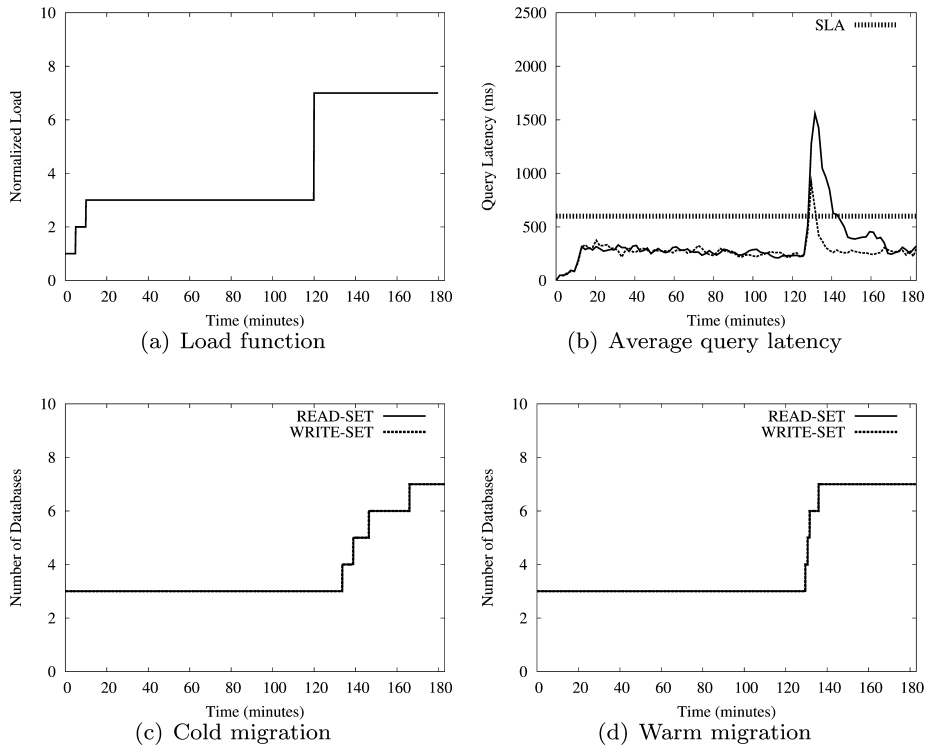


Fig. 8. Comparison of replica allocation techniques with cold and warm data migration.

satisfy the SLA. After 2 hours (7,200 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 techniques. 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.

6.2 Multiple Application Adaptations

In this section, we use both the TPC-W and the RUBIS benchmark applications to evaluate our adaptive 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.

Table I. Percent Compliance and Number of Allocated Replicas

Scheme	% Compliance	Allocated Replicas
Partial overlap	92%	5.2
Static partitioning	36%	4
Full overlap	31%	8

6.2.1 *TPC-W Adaptation.* Section 6.1 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 allocation policy described in Section 4. We now compare our system against two alternatives, static partitioning that uses disjoint allocation and the full-overlap allocation techniques. These schemes represent opposite end points of the allocation schemes. Static partitioning assigns a fixed, disjoint and fair-share partition of the database cluster to each application, while full-overlap allocation allows both the applications to operate on all the machines in the system. These schemes, unlike partial-overlap allocation, 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(c). Note that the read and write sets of the applications do not change for the other techniques. The three graphs at the bottom of Figure 9 show the query latency with the three allocation techniques.

The latency results show that partial-overlap allocation substantially outperforms both the static partitioning and the full-overlap allocation 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 applications (4 machines per application). Full overlap performs poorly due to the interference caused by the overlapping read sets of the two applications 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(d) shows that the latency in our system briefly exceeds the TPC-W SLA of 600ms as the system receives additional load while adapting to previous increases in load. However, the system catches up quickly and the latency target is met immediately after the last load step. Figure 9(c) 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 I shows the percentage compliance and the average number of replicas used by the three schemes in this experiment. To calculate compliance,

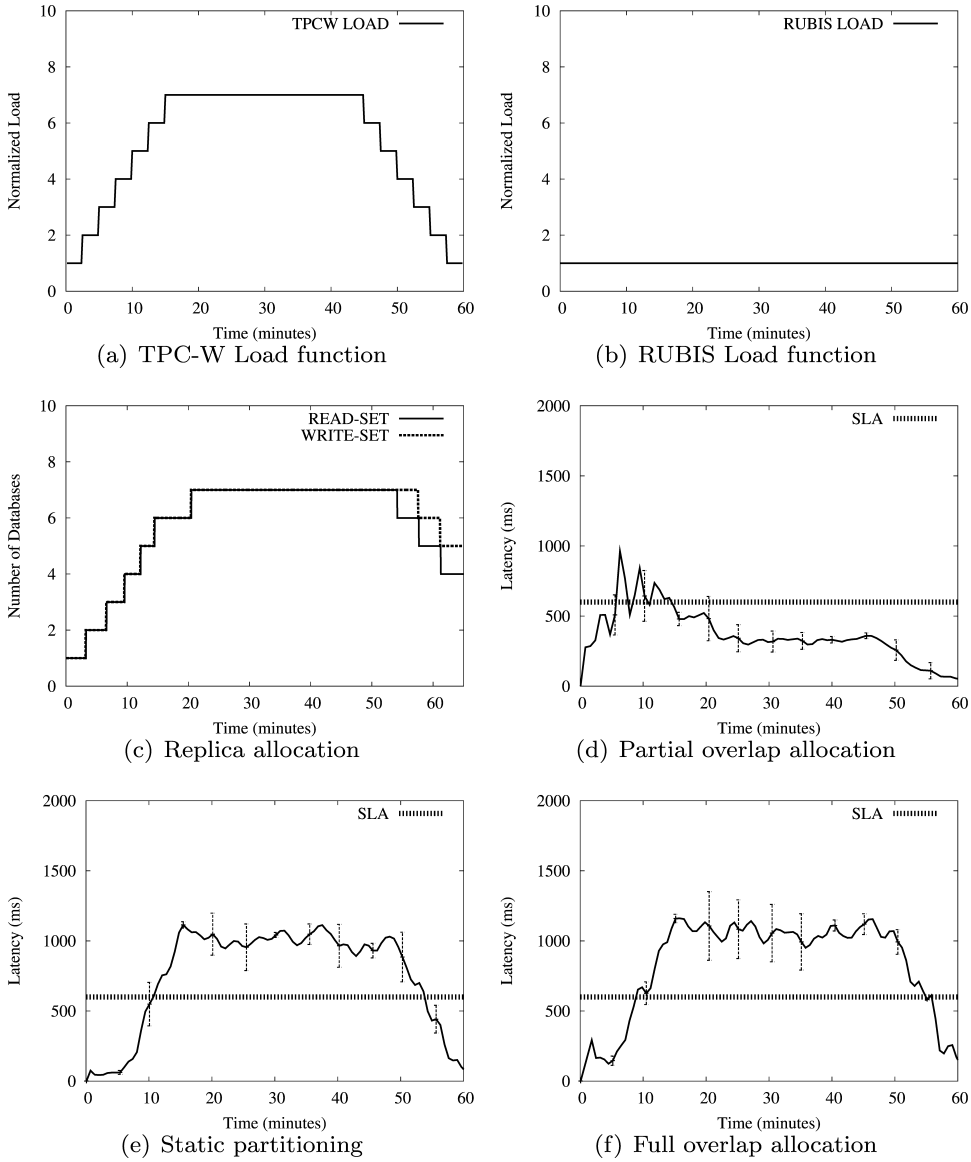
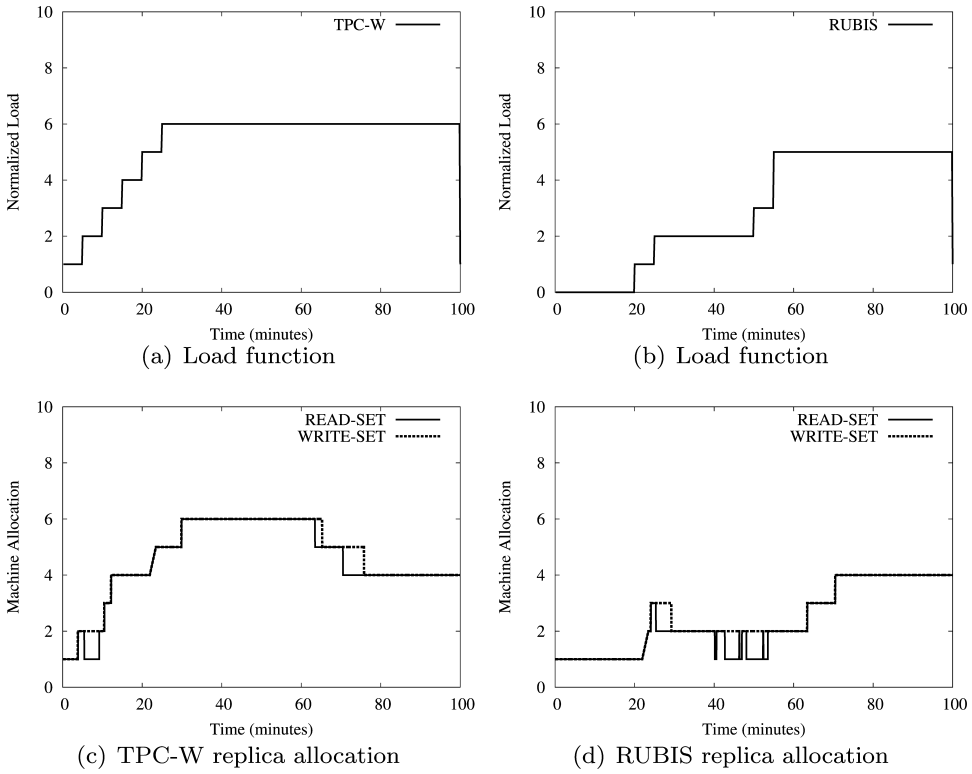


Fig. 9. Multiple application load, TPC-W load adaptation.

we divide the experiment into 10 second intervals and consider an interval as noncompliant if the latency rises above the `HighSLAThreshold` value even once in the interval. While static partitioning uses fewer machines, it has only 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 provides 92% SLA compliance.



(a) Load function (b) Load function

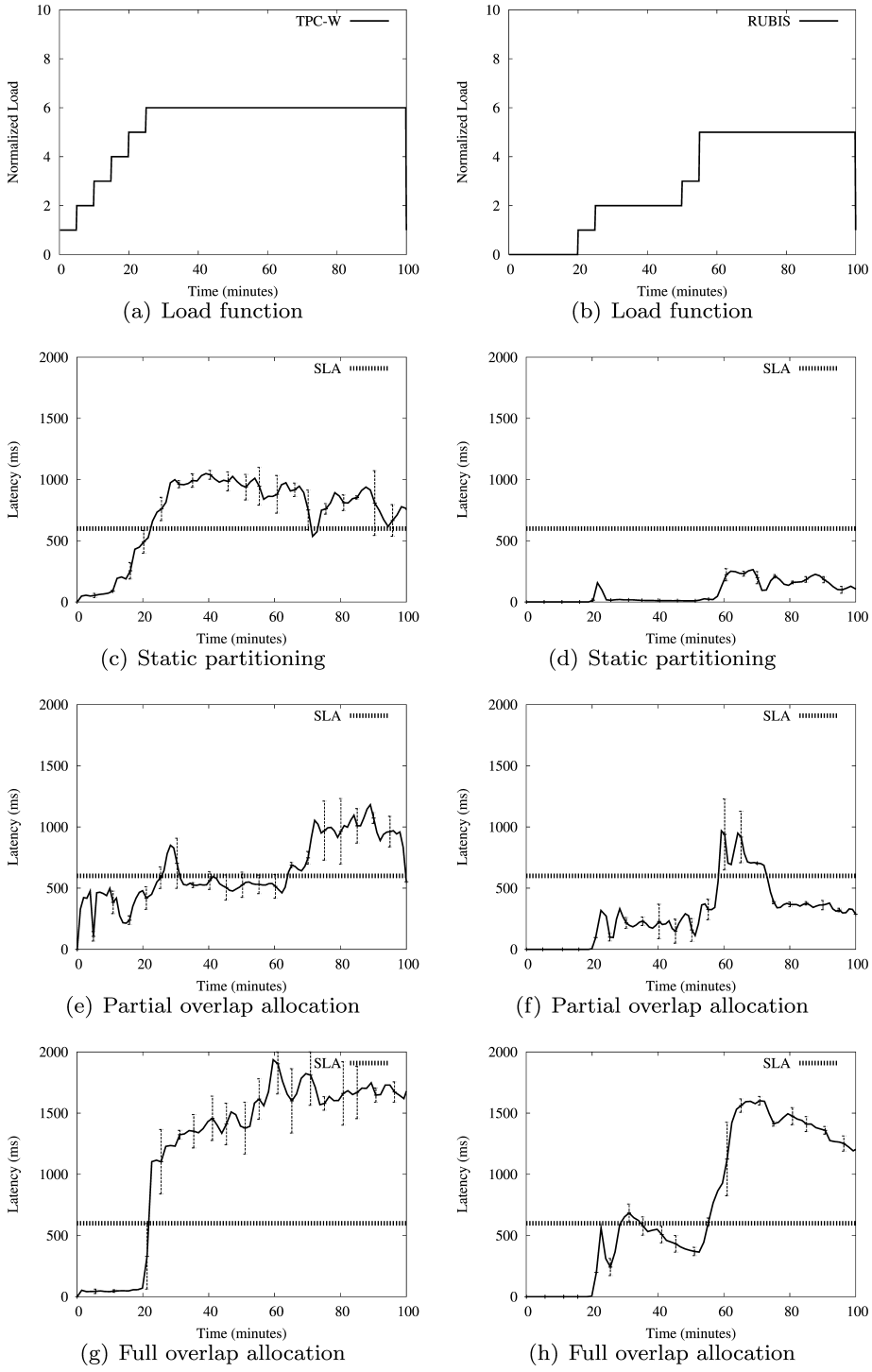
(c) TPC-W replica allocation (d) RUBIS replica allocation

Fig. 10. TPC-W and RUBIS adaptation.

6.2.2 *TPC-W and RUBIS 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 11 shows the complete set of results when running the two varying load benchmarks. Figure 10(a) shows the input load function for TPC-W, and Figure 10(b) shows the load function for RUBIS. 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 total available capacity of 8 database machines.

Figures 10(c) and 10(d) show the number of replicas allocated to the TPC-W and the RUBIS benchmarks by our partial-overlap allocation 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 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 applications. In this case, as Figure 10(c) shows, there are two consecutive forced replica removals from TPC-W so that TPC-W



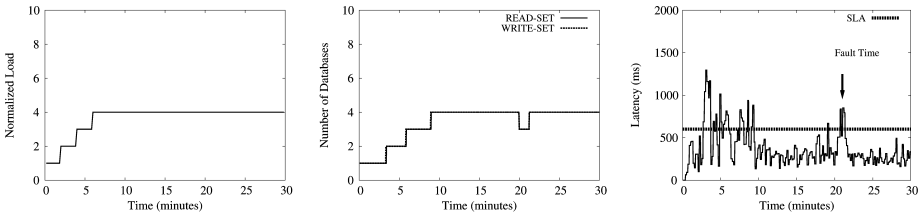
(a) Load function (b) Load function

(c) Static partitioning (d) Static partitioning

(e) Partial overlap allocation (f) Partial overlap allocation

(g) Full overlap allocation (h) Full overlap allocation

Fig. 11. TPC-W and RUBIS adaptation.



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

Fig. 12. Adaptation to replica failure.

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 TPC-W 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 11 show the query latency for TPC-W and RUBIS for the three allocation techniques. Figures 11(e) and 11(f) 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 of varying load on the two static allocation techniques, 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, regardless of their needs. Full overlap performs poorly for both applications. The high latency is caused by buffer cache interference, especially during overload.

6.3 Adaptation to Failures

Our dynamic replication system adapts replica allocation to meet application requirements and uses partial-overlap allocation 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 increasing events and adds new replicas to meet current demand.

Figure 12 demonstrates the fault-tolerant behavior of our system with a simple, single application experiment. Figure 12(a) shows the input load function for the TPC-W benchmark. Figure 12(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 300ms which is lower than the SLA, and therefore the resource manager does not take any action. However, Figure 12(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

Table II. Number of Allocated Replicas and Average Latency vs. the LowSLAThreshold Parameter

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

manager adapts its allocation by adding another replica. At this point, the latency drops to prefault levels.

6.4 Sensitivity Analysis

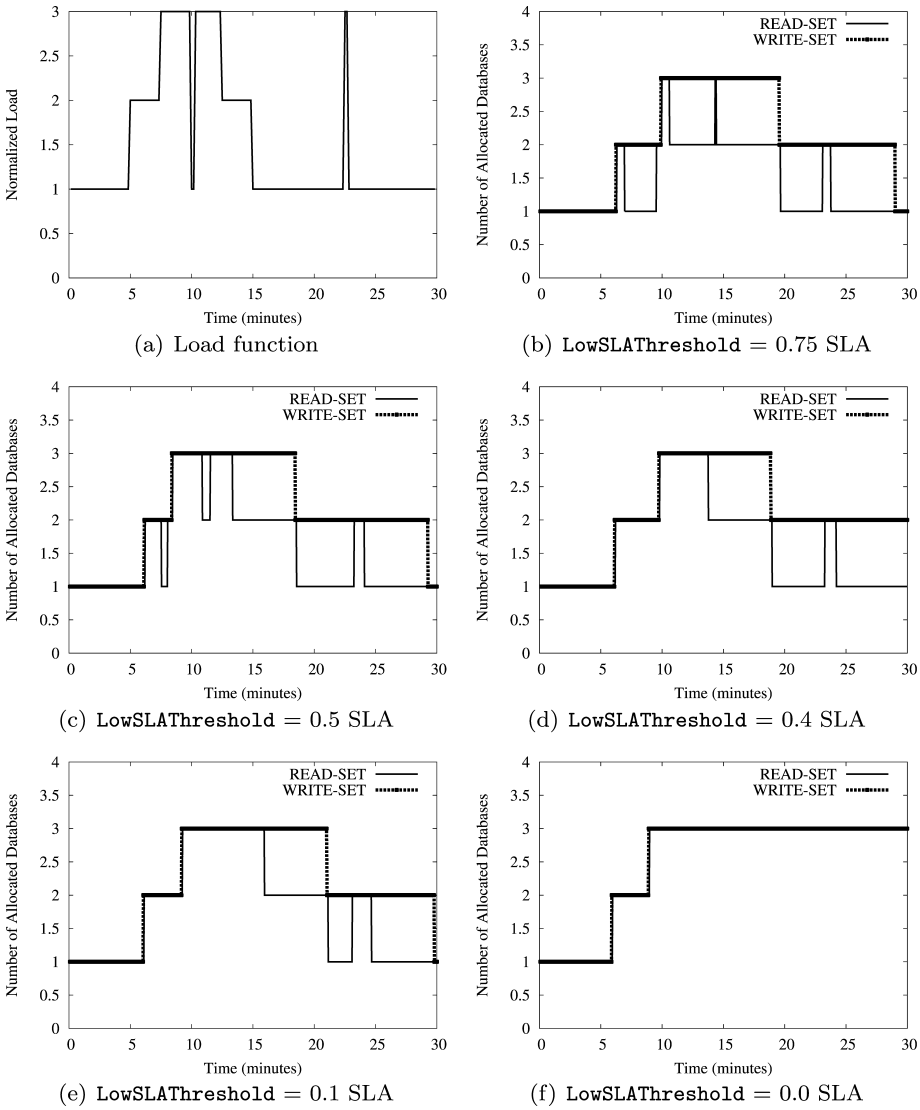
This section shows that our system is robust and does not require careful hand-tuning of parameters to achieve good performance. The main parameters in our system are the low and high SLA thresholds and the smoothing parameter α. The high SLA threshold is the same as the SLA specified by the application in all our experiments. Following, we show the effects of varying the other two main parameters, of fine-tuning the RemoveConfidenceNumber parameter and the effects of varying the speed of load change.

For this study, we use the TPC-W application, and we designed an input load function that simulates various workload scenarios 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 13(a).

6.4.1 *Variation in the LowSLAThreshold Parameter.* Figure 13 shows the output of our replica allocation scheme for the TPC-W application as the LowSLAThreshold parameter is varied from $0.75 \times SLA$ to 0. Higher values of LowSLAThreshold cause more aggressive replica removal, and thus the number of machines allocated to an application 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 13(b) through 13(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 II 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



(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

Fig. 13. Replica allocation with different `LowSLAThreshold` parameters.

of replicas grows minimally with decreasing `LowSLAThreshold`, and the average latency rises 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 nonzero value for `LowSLAThreshold` that is below $0.5 \times SLA$ will yield reasonable performance.

6.4.2 *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

Table III. Percent Compliance vs. the amoothing Parameter α

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

Table IV. The Effect on Performance by Varying `RemoveConfidenceNumber`

`RemoveConfidenceNumber`	Compliance %
0	91
0.5	90
1	92
3	93
5	92

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 reallocation.

Table III demonstrates this trade-off. It presents the percentage compliance for the input load shown in Figure 13(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.

6.4.3 *Variation of* `RemoveConfidenceNumber`. To determine the sensitivity of our algorithm to the parameter `RemoveConfidenceNumber`, we vary the parameter and measure the percentage of compliance (see Table IV). Similar to the smoothing parameter α, higher values of `RemoveConfidenceNumber` make the removal of databases more conservative. As Table IV shows, the performance does not vary much, and SLA compliance is high for all values of this parameter.

6.4.4 *Sensitivity to Speed of Load Change.* In this section, we show the effect of load change on adaptation time and the query latency. In previous sections, we have shown good performance in adapting to load functions that vary the load every 2 minutes or every 5 minutes. In this section, we use two load functions which vary the load faster, that is, every 1 minute and every 30 seconds, respectively. These experiments are designed to test the limits of our system to fast load changes.

Each load function is a ramp that starts with a normalized load of 1 and steps up to a load of 4 in the middle of the experiment, and then steps back down to 1. We show the average (smoothened) latency when using load steps either every minute as shown in Figure 14(a) or every 30 seconds as shown in Figure 14(b). As Figure 14(e) shows, we are able to adapt to load changes while maintaining the latency mostly underneath the SLA for the 1-minute step function, but

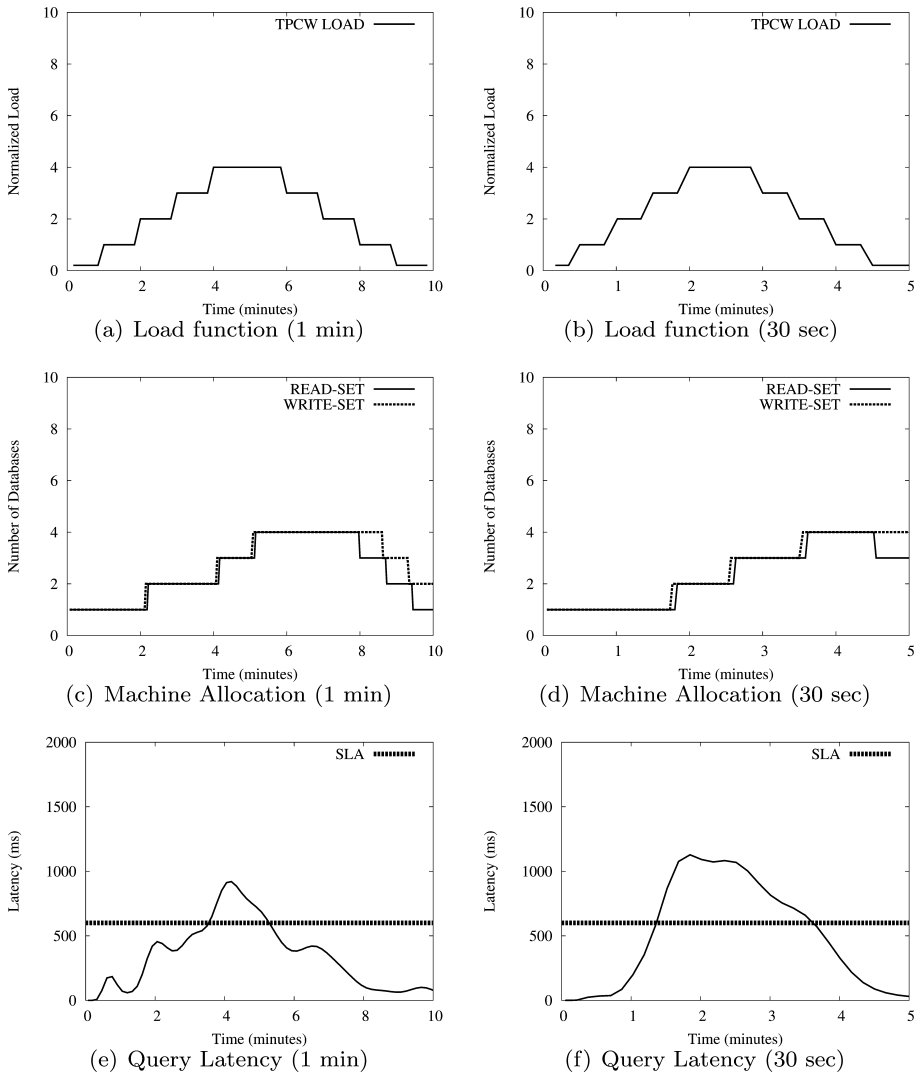


Fig. 14. Replica allocation with 1 minute and 30-second load step changes.

not for the 30-second step function. The cut-off depends on our update batch frequency for maintaining overflow replicas. In order to guarantee a desired reactivity level, the staleness limit for overflow replicas, in terms of the time estimate to bring a replica up to date which is 2.5 minutes in our experiments, should be set roughly equal to the desired system reaction time. Finally, even for frequent load changes of 1 minute or 30 seconds, our algorithm does not register any oscillations in resource allocation.

7. RELATED WORK

We address the hard problem of dynamic resource allocation within the database tier, advancing the research area of autonomic computing [IBM

Corporation 2003a]. 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. Specifically, in this article, we expand on our previous work on database replication techniques in single [Amza et al. 2003a, 2003b] and multiple application [Soundararajan et al. 2006] configurations through an in-depth presentation of dynamic provisioning of replicas in the database backend of dynamic content servers, including a discussion of algorithmic trade-offs, comprehensive performance evaluation, and sensitivity analysis.

A number of recent articles in the area of autonomic resource provisioning use resource allocation models based on analytical queuing models [Bennani and Menascé 2005; Woodside et al. 2006] or queuing models coupled with hill-climbing for searching through a multidimensional configuration space [Menasce et al. 2001], interpolation based on offline experiments [Walsh et al. 2004], utility models [Tesauro et al. 2005], machine learning algorithms [Tesauro et al. 2006], or dynamic programing [Karve et al. 2006] to adjust the mapping between a desired service level agreement and the amount of resources to be provisioned.

For instance, Karve et al. [2006] study resource allocation as an instance of workload placement in Web applications running on cluster servers. They formulate a dynamic programming problem where they assign capacities and demands to applications, classified as load independent (e.g., memory) or load dependent (e.g., CPU). They evaluate the effectiveness of their technique through simulations.

Most of these analytic performance models of Web servers have demonstrated good accuracy in simulations or experimentally. On the other hand, to the best of our knowledge, current performance prediction techniques for large data centers assume a single database server as backend [Walsh et al. 2004; Bennani and Menascé 2005; Woodside et al. 2006] or do not consider the database configuration parameters as part of their search space [Menasce et al. 2001; Karve et al. 2006].

Most of these solutions treat the system as a set of black boxes, hence, in theory, they could be applied to provisioning database replicas as well by simply adding boxes to an application's allocation based on model-based bottleneck detection. On the other hand, our article shows that, for a stateful system, awareness of the costs of provisioning and adaptation states in the particular system is a key feature for successful adaptation. Specifically, for a replicated database tier, the allocation of a new database replica to an application requires data migration, that is, the transfer of data to bring that replica up-to-date. Only if the resource allocation manager is conscious of the progress of data migration can it evaluate with reasonable accuracy an adaptation's impact on performance and, in particular, whether or not further provisioning is needed for that application. Moreover, accurately modeling a complex stateful system is a daunting task. Therefore, the applicability of generic queuing or utility models to database applications needs further investigation.

Apart from our previous work in conflict-aware database replication [Amza et al. 2003a, 2003b, 2005] a number of independent solutions exist that provide both scaling and strong consistency in replicated database clusters. They range from industry established solutions such as the Oracle RAC [ORACLE] to research and open-source prototypes [Kemme et al. 2001; Cecchet et al. 2004; Milan-Franco et al. 2004; Kemme and Alonso 2000b; Plattner and Alonso 2004; Daudjee and Salem 2004]. The industry solutions provide strong consistency, high availability, and good scalability, but they are costly and require specialized hardware such as Shared Network Disk [ORACLE]. The research prototypes use commodity software and hardware, but they typically rely on support for snapshot isolation inside the database [Plattner and Alonso 2004; Wu and Kemme 2005] and explore providing snapshot isolation semantics on a replicated database cluster [Wu and Kemme 2005; Lin et al. 2005; Plattner and Alonso 2004] instead of 1-copy serializability.

With a few notable exceptions [Kemme et al. 2001; Milan-Franco et al. 2004], these systems do not investigate database replication in the context of dynamic adaptation. Kemme et al. [2001] propose algorithms for database cluster reconfiguration through a staged approach similar to our database migration algorithm. The algorithms are, however, not evaluated in practice. Our article studies efficient methods for dynamically integrating a new database replica into a running system and provides a thorough system evaluation using realistic benchmarks. Milan-Franco et al. [2004] present both global and local adaptation to changing demand in a database cluster by adapting the placement of primary replicas and the degree of multiprogramming at each replica. Their work is, however, orthogonal to ours because they do not address replica provisioning.

Related, but orthogonal, efforts toward providing differentiated Quality of Service in dynamic content cluster servers include service differentiation based on tracking customer actions [Totok and Karamcheti 2006] and automatically determining correlations between system metrics and potential bottleneck states [Cohen et al. 2004; Parekh et al. 2006]. Instead of the conventional approach of labeling customers with service class types (e.g., the traditional gold, silver, bronze), Totok and Karamcheti [2006] determine the value of a customer dynamically from his/her actions such as putting items into the shopping cart. Based on a cumulative reward, a request scheduler assigns the client to its appropriate service class.

Cohen et al. [2004] 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, in contrast to our work, it does not provide any information on how to adapt in order to avoid SLO violations. The Elba project [Parekh et al. 2006] is based on the similar idea of evaluating the performance of the system using several machine learning classifiers. Its novelty lies in advocating observing service level objectives even before deployment, with the goal of detecting errors and potential bottlenecks during the design and testing phases of the system.

Various scheduling policies for proportional share resource allocation can be found in the literature, such as STFQ [Goyal et al. 1996]. Steere et al. [1999] 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 paper discussing resource controllers [Diao et al. 2002; Li and Nahrstedt 1999], 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 [Diao et al. 2002]. 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.

Further challenges in adaptive database replication, which we do not address in this work, include adaptive replication in the wide area. While, as we have shown, strongly consistent replication can be achieved transparently in a local cluster, different methods may be needed in a geographically distributed system because of problems of latency, bandwidth, and possible disconnected operation. Among the recent efforts in this direction, GlobeDB [Sivasubramanian et al. 2005] addresses adaptive data replication at edge servers in a Content Delivery Network (CDN) and studies the trade-offs between data replication and data placement in this context. In GlobeDB, a centralized server observes the access patterns from different edge servers and places the database items (rows) onto edge servers based on the observed clustered accesses. The degree of replication is controlled by a cost function which reduces the response time or the number of updates. The authors use the TPC-W benchmark in the evaluation of their solution. While currently our solution is limited to a cluster of database replicas in a LAN environment, we are planning to explore wide-area extensions of our dynamic content server solution in future work.

8. CONCLUSIONS

In this article, we introduce a novel solution to resource provisioning in the database tier of a dynamic content site. Our self-configuration techniques allow the server the flexibility to dynamically reallocate database commodity nodes across multiple applications. Our approach can react to resource bottlenecks or failures in the database tier in a unified way.

We avoid modifications to the Web server, the application scripts, and the database engine. We also assume software platforms in common use: the Apache Web server, the MySQL database engine, and the PHP scripting language. As a result, our techniques are applicable without burdensome development and replace human management of the Web site. We use the shopping workload mix of the TPC-W benchmark and the RUBIS online auction benchmark to evaluate the reactivity and stability of our dynamic resource allocation protocol.

Our evaluation shows that our dynamic replica allocation approach can handle rapid variations in an application's backend resource requirements while maintaining the per-application query latency under a predefined SLA for our

two benchmarks. We show that an approach with partial-overlap in application allocations works significantly better than a full-overlap and a static partitioning approach, which suffer from workload interference and rigid allocations, respectively. We also show that introducing adaptation delay awareness in the feedback loop of the resource manager is key for avoiding oscillations in resource allocation. We finally show through our sensitivity analysis, that our delay aware provisioning technique obviates the need for careful handtuning of any system parameters.

REFERENCES

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

AMZA, C., COX, A., AND ZWAENEPOEL, W. 2003a. Conflict-aware scheduling for dynamic content applications. In *Proceedings of the 5th USENIX Symposium on Internet Technologies and Systems*. 71–84.

AMZA, C., COX, A., AND ZWAENEPOEL, W. 2005. A comparative evaluation of transparent scaling techniques for dynamic content servers. In *Proceedings of the 21st International Conference Data Engineering* (*ICDE'05*). 230–241.

AMZA, C., COX, A. L., AND ZWAENEPOEL, W. 2003b. Distributed versioning: Consistent replication for scaling backend databases of dynamic content web sites. In *Lecture Notes in Computer Science, vol. 2672*. M. Endler and D. C. Schmidt, Eds., Springer, 282–304.

APACHE. The Apache Software Foundation. `http://www.apache.org/`.

BENNANI, M. N. AND MENASCÉ, D. A. 2005. Resource allocation for autonomic data centers using analytic performance models. In *IEEE International Conference on Autonomic Computing* (*ICAC'05*). 62–69.

BERNSTEIN, P., HADZILACOS, V., AND GOODMAN, N. 1987. *Concurrency control and recovery in Database Systems*. Addison-Wesley, Reading, MA.

BORAL, H., ALEXANDER, W., CLAY, L., COPELAND, G., DANFORTH, S., FRANKLIN, M., HART, B., SMITH, M., AND VALDURIEZ, P. 1990. Prototyping Bubba, a highly parallel database system. In *IEEE Trans. Knowl. Data Engin. 2*, 4–24.

CECCHET, E., MARGUERITE, J., AND ZWAENEPOEL, W. 2004. C-jdbc: Flexible database clustering middleware. In *Proceedings of the USENIX Annual Technical Conference*.

COHEN, I., CHASE, J. S., GOLDSZMIDT, M., KELLY, T., AND SYMONS, J. 2004. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *Proceedings of the 6th Symposium on Operating System Design and Implementation* (*OSDI'04*). 231–244.

DAUDJEE, K. AND SALEM, K. 2004. Lazy database replication with ordering guarantees. In *20th International Conference on Data Engineering*, Boston, MA.

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

GOYAL, P., GUO, X., AND VIN, H. M. 1996. A Hierarchical CPU scheduler for multimedia operating system. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*. Seattle, WA.

GRAY, J., HELLAND, P., O'NEIL, P., AND SHASHA, D. 1996. The dangers of replication and a solution. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. Montreal, Quebec, Canada, H. V. Jagadish and I. S. Mumick, Eds., ACM Press, New York, NY, 173–182.

GUY, R., REIHER, P., RATHER, D., AND GUNTER, M. 1999. Rumor: Mobile data access through optimistic peer-to-peer replication. Lecture Notes in Computer Science, vol. 1552, 254–265.

HUTCHINSON, N., MANLEY, S., FEDERWISCH, M., HARRIS, G., HITZ, D., KLEIMAN, S., AND O'MALLEY, S. 1999. Logical vs. physical file system backup. In *Proceedings of the 3rd USENIX Symposium on Operating System Design and Implementation*. 239–249.

IBM CORPORATION. 2003a. Autonomic computing manifesto. http://www.research.ibm.com/autonomic/manifesto.

IBM CORPORATION. 2003b. Automated provisioning of resources for data center environments. http://www-306.ibm.com/software/tivoli/solutions/provisioning/.

KARVE, A., KIMBREL, T., PACIFICI, G., SPREITZER, M., STEINDER, M., SVIRIDENKO, M., AND TANTAWI, A. 2006. Dynamic placement for clustered web applications. In *Proceedings of the 15th International Conference on World Wide Web* (*WWW'06*) ACM Press, New York, NY, 595–604.

KEMME, B. AND ALONSO, G. 2000a. A new approach to developing and implementing eager database replication protocols. In *ACM Trans. DataB. Syst. 25*, 333–379.

KEMME, B. AND ALONSO, G. 2000b. Don't be lazy, be consistent: Postgres-R, a new way to implement Database Replication. In *Proceedings of the 26th International Conference on Very Large Databases*.

KEMME, B., BARTOLI, A., AND BABAOGLU, Ö. 2001. Online reconfiguration in replicated databases based on group communication. In *Proceedings of the International Conference on Dependable Systems and Networks* (*DSN'01*). 117–130.

LI, B. AND NAHRSTEDT, K. 1999. A control-based middleware framework for quality of service adaptations. *IEEE J. Select. Areas Comm.*.

LIN, Y., KEMME, B., PATIÑO-MARTÍNEZ, M., AND JIMÉNEZ-PERIS, R. 2005. Middleware-based data replication providing snapshot isolation. In *Proceedings of the ACM SIGMOD Conference (SiGMOD'05)*, F. Ozcan, Ed. ACM, Press, New York, NY. 419–430.

MENASCE;, D. A., BARBARA;, D., AND DODGE, R. 2001. Preserving QoS of e-commerce sites through self-tuning: A performance model approach. In *Proceedings of the 3rd ACM Conference on Electronic Commerce*. ACM Press, New York, NY. 224–234.

MILAN-FRANCO, J. M., JIMENEZ-PERIS, R., PATIO-MARTNEZ, M., AND KEMME, B. 2004. Adaptive middleware for data replication. In *Proceedings of the 5th ACM/IFIP/USENIX International Middleware Conference*.

MySQL. MySQL. http://www.mysql.com.

ORACLE. Oracle Real Application Clusters 10g. http://www.oracle.com/technology/products/database/clustering/.

PAREKH, J., JUNG, G., PU, G. S. C., AND SAHAI, A. 2006. Improving performance of internet services through reward-driven request prioritization. In *Proceedings of the 14th IEEE International Workshop on Quality of Service* (*IWQoS'06*).

PLATTNER, C. AND ALONSO, G. 2004. Ganymed: Scalable replication for transactional Web applications. In *Proceedings of the 5th ACM/IFIP/Usenix International Middleware Conference*.

RANJAN, S., ROLIA, J., FU, H., AND KNIGHTLY, E. 2002. QoS-driven server migration for internet data centers. In *10th International Workshop on Quality of Service*.

SHEN, K., YANG, T., CHU, L., HOLLIDAY, J. L., KUSCHNER, D., AND ZHU, H. 2001. Neptune: Scalable replica management and programming support for cluster-based network services. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems*. 207–216.

SIVASUBRAMANIAN, S., ALONSO, G., PIERRE, G., AND VAN STEEN, M. 2005. Globedb: autonomic data replication for Web Applications. In *Proceedings of the 14th International World Wide Web Conference* (*WWW'05*), A. Ellis and T. Hagino, Eds. ACM, Press, New York, NY. 33–42.

SLASHDOTEFFECT. Slashdot: Handling the loads on 9/11. http://slashdot.org.

SOUNDARARAJAN, G., AMZA, C., AND GOEL, A. 2006. Database replication policies for dynamic content applications. In *EuroSys'06: Proceedings of the EuroSys Conference (EUROSYS'06)*. ACM, Press, New York, NY. 89–102.

STEERE, D. C., GOEL, A., GRUENBERG, J., MCNAMEE, D., PU, C., AND WALPOLE, J. 1999. A Feedback-driven proportion allocator for real-rate scheduling. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*.

TERRY, D. B., THEIMER, M. M., PETERSEN, K., DEMERS, A. J., SPREITZER, M. J., AND HAUSER, C. H. 1995. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings of the 15th Symposium on Operating Systems Principles*. Cooper Mountain, CO. 172–183. http://www.parc.xerox.com/bayou/.

TESAURO, G., DAS, R., JONG, N., , AND BENNANI, M. 2006. A hybrid reinforcement learning approach to autonomic resource allocation. In *Proceedings of the 3rd International Conference on Autonomic Computing* (*ICAC'06*).

TESAURO, G., DAS, R., WALSH, W. E., AND KEPHART, J. O. 2005. Utility-function-driven resource allocation in autonomic systems . In *Proceedings of the 2nd IEEE International Conference on Autonomic Computing* (*ICAC'05*). 70–77.

TOTOK, A. AND KARAMCHETI, V. 2006. Issues in bottleneck detection in multitier enterprise applications. In *IWQoS 2006: Proceedings of the 14th IEEE International Workshop on Quality of Service* (*IWQS'06*).

TPC. Transaction Processing Council. `http://www.tpc.org/`.

WALSH, W. E., TESAURO, G., KEPHART, J. O., AND DAS, R. 2004. Utility functions in autonomic systems. In *Proceedings of the 1st International Conference on Autonomic Computing* (*ICACS'04*).

WEIKUM, G. AND VOSSEN, G. 2002. *Transactional Information Systems. Theory, Algorithms and the Practice of Concurrency Control and Recovery*, 2nd Ed. Addison-Wesley, Reading, MA.

WELSH, M. AND CULLER, D. 2003. Adaptive overload control for busy internet servers. In *Proceedings of the 5th USENIX Symposium on Internet Technologies and Systems*.

WOODSIDE, M., ZHENG, T., AND LITOIU, M. 2006. Service system resource management based on a tracked layered performance model. In *Proceedings of the 3rd International Conference on Autonomic Computing* (*ICAC'06*). 123–133.

WU, S. AND KEMME, B. 2005. Postgres-r(si): Combining replica control with concurrency control based on snapshot isolation. In *Proceedings of the 21st International Conference on Data Engineering* (*ICDE'05*). 422–433.

Received July 2005; revised June 2006; accepted August 2006