Autonomic Provisioning of Backend Databases in Dynamic Content Web Servers

Gokul Soundararajan, Ashvin Goel and Cristiana Amza

Abstract

This paper introduces a self-configuring architecture for scaling the database tier of dynamic content web servers. We use a unified approach to load and fault management based on dynamic data replication and feedback-based scheduling. While replication provides scaling and high availability, feedback scheduling dynamically allocates tasks to commodity databases across workloads in response to peak loads or failure conditions thus providing quality of service. By augmenting the feedback loop with state awareness, we avoid oscillations in resource allocation.

We investigate our transparent provisioning mechanisms in the database tier using 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.

1 Introduction

This paper introduces a novel scheduling technique for on-demand resource allocation across multiple dynamic-content workloads that use a cluster-based database back-end. Dynamic content servers commonly use a three-tier architecture (see Figure 1) that consists of a front-end web server tier, an application server tier that implements the business logic, and a back-end database tier that stores the dynamic content of the site. Gross hardware over-provisioning for each workload's estimated peak load, in each server tier can become infeasible in the short to medium term, even for large sites. Hence, it is important to efficiently utilize available resources through dynamic resource allocation across all active applications. One such approach, the Tivoli on-demand business solutions [13], consists of dynamic provisioning of resources within the stateless web server and the application server. However, dynamic resource allocation among applications in the state-full database tier, which commonly becomes the bottleneck [1, 25], has received comparatively less attention.

Our approach interposes a scheduler between the application server(s) and the database cluster. This scheduler virtualizes the database cluster and the workload allocations within the cluster so that the application server sees a single database. In addition, a controller arbitrates resource allocations between the different workloads running on the web site.

We define quality of service as maintaining the average query latency for a particular workload under a predefined Service Level Agreement (SLA). Our dynamic database provisioning algorithm, called feedback-based scheduling (FBS), triggers adaptations in response to impending SLA violations. Furthermore, our algorithm removes resources from a workload's allocation when in underload. Due to the state-full nature of databases, the allocation of a new database to a workload requires the transfer of data to bring that replica up to date. We use an adaptation scheme called *warm migration* where: i) All databases in the workload's main partition are kept up-to-date and ii) We maintain a set of additional replicas within a staleness bound. These replicas constitute an overflow pool used for rapid adaptation to temporary load spikes.

Our dynamic resource allocation mechanism uses per-workload performance tracking to trigger control actions such as changes in per-workload allocations within the database tier. More importantly, our controller uses a state machine approach to track the system state during and in-between adaptations in order to trigger any subsequent adaptations only after the changes of previous adaptations have become visible. Latency sampling is thus suppressed while an adaptation is in progress (e.g., during data migration to bring a new replica up to date) because effects on latency cannot be reliably observed. This closes the feedback loop and avoids unnecessary overreaction and oscillation in the system.

Our experimental evaluation uses the TPC-W industry-standard e-commerce benchmark [27] modeling an on-line book-store and Rubis, an on-line auction site modeled after e-Bay [1]. We have implemented each respective web site meeting each benchmark specification, using three popular open source software packages: the Apache web server [4], the PHP web-scripting/application development language [20] and the MySQL database server [18]. Our exper-

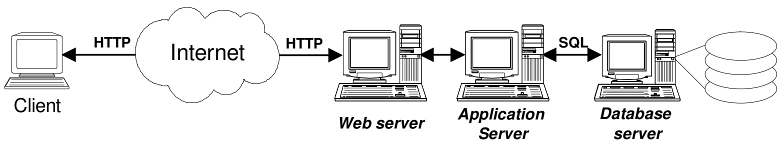


Figure 1: Common Architecture for Dynamic Content Sites

imental platform consists of a cluster of dual AMD Athlon PCs connected by 100Mbps Ethernet LAN and running RedHat Fedora Linux. Our largest experimental setup includes 8 database server machines, 2 schedulers, 1 controller and 12 web server machines.

Our evaluation shows that our feedback-based scheduling approach can handle rapid variations in an application's resource requirements while maintaining quality of service for each application. Our approach is significantly better than both a static read-any-write-all approach and a static partitioning approach under a variety of load scenarios. In addition, by monitoring system state, we avoid oscillations in resource allocation. Finally, we show that the same approach can be used to handle failure scenarios.

Section 2 describes the environment used for dynamic resource allocation. Section 3 describes our feedback-based dynamic resource allocation solution. Section 4 presents our benchmarks and experimental platform. We investigate dynamic allocation of database resources to workloads experimentally in Section 5. Section 6 discusses related work and Section 7 concludes the paper.

2 Environment

This section describes the environment that forms the basis for the implementation of the feedback-based scheduling algorithm discussed in section 3. In particular, we describe the programming model, the desired consistency, the cluster architecture and the data migration algorithm we employ.

2.1 Consistency and Programming Model

The consistency model we use for all our protocols is strong consistency or 1-copy-serializability [5], which makes the system look like one copy to the user. With 1-copy-serializability, conflicting operations of different transactions execute in the same order on all replicas (i.e., the execution of all transactions is equivalent to a serial execution, and that particular serial execution is the same on all replicas).

The user inserts transaction delimiters wherever atomicity is required in the application code. In the absence of transaction delimiters, each single query is considered a transaction and is automatically committed (so called "auto-commit" mode).

Our method requires that all tables accessed in a transaction and their access types (read or write) be known at the beginning of each transaction. During a pre-processing phase, we parse the application scripts to obtain a conservative approximation of this information. The pre-processor inserts a "table declaration" database query at the beginning of each script for all tables accessed and their access type. Although these queries are not actually executed by the databases, they form the basis of a conservative concurrency control protocol [5] based on conflict classes [19] that avoids deadlocks. We choose a protocol that avoids deadlocks, because the deadlock probability for a replicated database cluster becomes prohibitive in large clusters, due to the extra updates that each node performs on behalf of the other nodes [11].

2.2 Cluster Architecture

Our dynamic-content cluster architecture consists of a set of schedulers, one per workload, that distribute incoming requests to a cluster of database replicas and deliver the responses back to the application server, as shown in Figure 2. Each per-workload scheduler may itself be replicated for availability. The application server interacts directly only with the scheduler in charge of

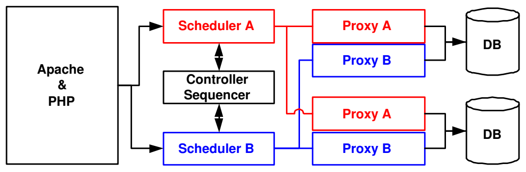


Figure 2: Cluster Architecture Design

the corresponding workload run by the application server. These interactions are synchronous, so that for each query, the application server blocks until it receives a response from the scheduler.

The schedulers use a set of database proxies, one at each database engine, to communicate with the databases. In addition, they use a sequencer that assigns a unique sequence number to each transaction, and a controller that arbitrates resource allocations between the different workloads. These components play crucial roles towards reaching our goals of strong consistency and high utilization, respectively. The sequence numbers are used to enforce a total ordering on conflicting update transactions at each database replica. Figure 2 shows the sequencer and controller together in one component since in practice they can be placed on the same machine or even inside the same process.

2.2.1 Operation

The web/application server sends the queries embedded in a script, one at a time, to the scheduler assigned to the corresponding workload for that script. For each workload, the corresponding scheduler maintains two data structures denoting the database *read-set* and the *write-set* for that workload. The write-set is the set of replicas where write queries (`INSERT`, `UPDATE`, and `DELETE`) are propagated to keep them up to date. In addition, the transaction delimiters are sent to all databases in a workload's write-set. Similarly, the read-set is the set of replicas where read queries (`SELECT`) may be sent for that workload. To maintain consistency, the read-set should be a subset of the write-set.

2.2.2 Consistent Asynchronous Replication within a Partition

Each per-workload scheduler maintains consistency within its workload allocation based on Distributed versioning [3], a replication algorithm that achieves 1-copy serializability and absence of deadlock through an asynchronous replication scheme augmented with version numbers, as described next.

The sequencer maintains a separate version number for each table in the database. Upon receiving a table pre-declaration from the scheduler, the sequencer assigns a version number for each table to be accessed by the transaction. Version number assignment is done atomically and in such a way that, if there is a conflict between the current transaction and an earlier one, the version numbers given to the current transaction for the tables involved in the conflicts are all higher than the version numbers received by the earlier conflicting transaction. Please see our previous paper [3] for more details on version assignment.

All operations on a particular table are executed at all replicas in version number order. In particular, an operation waits until its pre-assigned version is available at the database replica where it has been sent. New versions become available as a result of a previous transaction committing. The corresponding database proxy is in charge of keeping track of its database versions and of withholding queries until their pre-assigned version number for all accessed tables matches those of the database. Queries from the same script are issued in-order.

The scheduler sends write queries to all replicas in the workload's database write-set tagged with the version numbers obtained at the beginning of their enclosing transaction and relies on their asynchronous execution in order of version numbers. At a given time, a write query may have been sent to all write-set replicas, but it may have completed only at a subset of them. The scheduler [2, 3] maintains the completion status of outstanding write operations, and the current version for each table at all database replicas. Using this information, the scheduler sends a read that immediately follows a particular write in version number order to a replica where it knows the write has already finished (i.e., the corresponding required version has been produced). This scheduling algorithm allows asynchronous execution of write queries and avoids waiting due to read-write conflicts.

2.2.3 Fault Tolerance of Writes

To enable flexible extensions of a workload's database write-set for accommodating bottlenecks induced by faults or overload, the schedulers maintain persistent logs for all write queries of past transactions in their serviced workload. 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 in order of the version numbers for the corresponding write queries.

Each scheduler maintains a *global version vector* ($\vec{G}$) with one entry for each database table accessed by its workload. 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 newly incorporated database replica up-to-date upon database write-set extension through a process we call "data migration", described in the next section.

2.3 Data Migration

Due to the state-full nature of databases, the allocation of a database to a workload requires the transfer of data to bring a replica up to date. Our algorithm is designed to transfer the current state to the joining database with minimal disruption of transaction processing. As previously described, the scheduler maintains two version vectors: (1) global version vector ($\vec{G}$) and (2) replica version vector ($\vec{R}$). In this section, we describe how a new replica is brought "up-to-date" and incorporated into the *write-set* of the workload.

We describe the basic algorithm, then we describe an optimization we made to overcome the disadvantages in the basic scheme. Assume that there are replicas $D_0 \ldots D_n$ which can be allocated to a workload W whose write set is $\vec{W} = \{D_0 \ldots D_i\}$. We want to add replica D_j to the write set of workload W.

Intuitively, to bring a database up-to-date, the scheduler has to send to it all updates committed *before* adding database replica D_j to the write-set, for update log replay on D_j. The problem is that new transactions continue to update the databases in the write-set while data migration is taking place. Therefore, any updates made *after* the start of migration should be queued at replica D_j and applied only after migration is complete.

Although the previous algorithm is simple, it has a severe drawback. If the transfer of updates to D_j takes a long time, the queue of updates at D_j grows without bound. To correct this, the scheduler executes data migration in *stages*. 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*$(\vec{M})$ corresponding to all comitted 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. This reduces the number of logged updates to be sent after each stage until during the last stage, the scheduler is able to concurrently send new updates to the replica 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 $(\vec{T})$. The scheduler then sends old queries from the log up to the minimum of $\vec{G}[t]$ and $\vec{T}[t] - 1$ for each table t.

3 Dynamic Allocation of Resources

We define quality of service as maintaining the average query latency for a particular workload under a predefined Service Level Agreement (SLA). 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 controller. All schedulers use the same sampling interval for the purposes of maintaining these performance averages and communicating them to the scheduler. The controller, based on its global knowledge of each workload's service-level-agreement (SLA) requirements and their perceived performance makes database allocation decisions for all workloads. The decisions are communicated to the respective workload schedulers, which act accordingly by including or excluding databases from their database read and/or write sets for their corresponding workload. The controller increases a workload's allocation if the respective workload 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 workload.

Our dynamic provisioning algorithm called feedback-based scheduling (FBS) has two key components: (1) per-workload performance monitoring, and (2) system-state awareness through a state machine approach. Performance monitoring is used to trigger adaptations in response to impending Quality of Service (QoS) violations for any particular application. At the same time, the controller uses a state machine approach to track the system state during and in-between 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.

3.1 Per-Workload Performance Monitoring

To avoid short and sudden spikes from triggering adaptation, the controller uses a smoothened response time. Smoothing is achieved through a com-

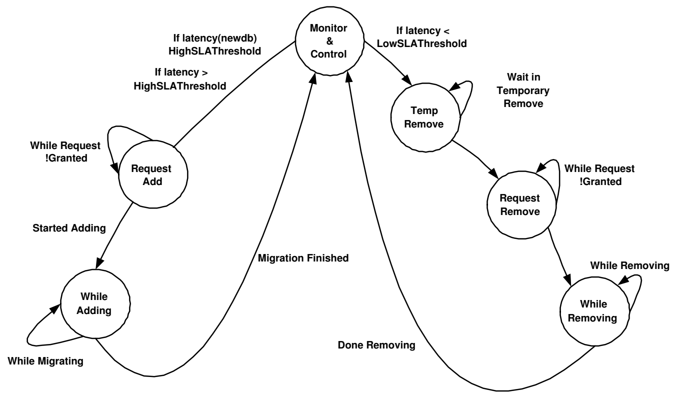


Figure 3: Controller State Machine

monly used [29] exponentially weighted running latency average of the form $WL = \alpha \times L + (1 - \alpha) \times WL$. A larger value of the α parameter makes the average more responsive to current changes in the latency. The controller uses two different values for α, 0.25 and 0.125 and two corresponding running averages for the latency, *HLatency (HL)* and *LLatency (LL)*. We choose two different parameters because we want the increase in allocation to be responsive to changes in latency, while we want to react to reduced allocation requirements slowly since removing a database is a potentially costly operation. Our results are, however, relatively independent of the precise value of the α parameter due to our use of state-awareness, described in the next section. In particular, we did not need to choose a different set of parameters for each benchmark.

3.2 Feedback State Machine

Figure 3 presents the main states used in the control loop of FBS. We explain the state transitions for adding and removing databases, respectively, in more detail in the following two sections.

3.2.1 Adding Databases to a Workload's Allocation

The controller starts in the initial `MonitorAndControl` state where the controller monitors the average latency received from each workload scheduler during each sampling period. If the average latency over the past sampling interval for a particular workload exceeds the `HighSLAThreshold`, hence an

SLA violation is imminent, the controller places a request to add a database to that workload's allocation. This request may involve data migration to bring up a new database for that workload. Since the request for adding a new database may not be fulfilled immediately, latency sampling is suspended at the controller until the request has been fulfilled and the result of the change can be observed. This implies waiting in potentially three states: (i) the `RequestAdd` state while a free machine can be found and allocated to the overloaded workload, (ii) the `DataMigration` state while the newly added database is brought up to date, if necessary and (iii) the `SystemStabilize` state, where we wait for the queries in the queues of the overloaded machines to be flushed out of the system before we restart system-wide monitoring. The `SystemStabilize` is done inside the `MonitorAndControl` state in Figure 3.

Our finite state machine approach avoids system instability due to oscillating between eagerly adding and removing databases for a particular workload. Adding a new database is triggered by system-wide overload. Hence, even after the load balancing algorithm starts to send new queries to the newly added database, the long queues that may be already present at the old databases continue to generate high latencies. This is independent of whether adding the new machine is enough to normalize the average latency in the system or not. We need to wait until all pre-existing queries are flushed out of the system before resuming overall latency sampling and potentially adding another database to the workload if the SLA is still not met at that time.

Since this wait may be long and will impact system reactivity to steep load bursts, we optimize waiting time by using the individual average latency generated at the newly added database as a heuristic. Since this database has no load when added, we use its latency exceeding the SLA as an early indication of a need for even more databases for that workload and we transition directly into the `RequestAdd` state in this case.

3.2.2 Removing a Database from a Workload's Allocation

The controller removes a database from a workload allocation in either of the following two scenarios: (i) the workload is in underload for a sufficient period of time and does not need the database (voluntary remove) or (ii) the system as a whole is in overload and fairness between allocations needs to be enforced (forced remove).

In the former case, as we can see from the finite state machine diagram in Figure 3, the removal path is conservative with an extra temporary remove state on the way to final removal of a database from a workload's allocation. This is another measure to avoid system instability by making sure that a workload 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 a workload's read-set (but not from its write set) if its average latency has been under the `LowerSLAThreshold` for the last sampling interval. We then proceed to loop inside the temporary remove state for a configurable number of sampling intervals `RemoveConfidenceNumber` before transitioning into the `RequestRemove` state. Subsequently, the control loop initiates the final removing procedure of the database from the workload's write-set and tracks this process until complete.

3.3 Main Scheduling Algorithm - Warm Migration

All databases in the workload's main partition are kept up-to-date by the scheduler. In addition, we maintain a set of additional replicas within a staleness bound. The overlap replicas are an overflow pool used for rapid adaptation to temporary load spikes since data migration onto them is expected to be relatively fast. Write-type queries are batched and sent periodically to update overlap replicas whenever they violate the staleness bound.

While the overlap region between workloads is configurable in our system, to simplify algorithm analysis, for the purposes of this paper, we will henceforth assume that the overlap region consists of all other replicas available in the system outside the workload's read-set partition.

3.4 Alternative Scheduling Algorithms

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

Specifically, the database write-set allocated by the controller to a particular workload could theoretically be completely decoupled from the read-set allocated to the same workload. The same applies for state-awareness which could be decoupled from dynamic resource allocation. Hence, we distinguish the following alternative scheduling algorithms using different database read and write sets and either state-aware or stateless scheduling.

3.4.1 Dynamic Allocation with Cold Migration

In this scheme, each workload is assigned a current partition of the database cluster. We keep up-to-date only the databases in the particular workload's allocation and any database within the partition can be selected to service a read of the particular workload. 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 a workload'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 workloads during periods of stable load increasing the probability that each individual working set fits in each database buffer cache.

3.4.2 Dynamic Allocation with Hot Migration

Writes of all workloads are sent to all databases, while the read set of each workload 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 a workload's read-set allocation. On the other hand, this protocol does not extend to the general case of large clusters running many concurrent workloads where writes can become a bottleneck. In addition, since 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 replica.

3.4.3 Dynamic Allocation with Stateless Scheduling

Any of the dynamic allocation protocols with *warm*, *hot* or *cold* migration can be combined with a stateless controller. Instead of following our state machine approach, the controller simply reacts to any reported above-High or below-Low threshold average query latency during a particular sampling interval by increasing or decreasing the workload allocation, respectively. This technique is similar to overload adaption in stateless services [29] where simple smoothing of the average latency has been reported to give acceptable stability to short load spikes.

3.4.4 Read-Any Write-All

Writes of all workloads are sent to all databases. Each read query of any workload 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 workloads 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 workload under small load fluctuations. On the down-side, both reads and writes of all workloads share the buffer-cache on each node, hence poor performance can occur if the buffer-cache capacity is exceeded.

3.4.5 Static Partitioning

This is a standard static partitioning protocol, where each workload is assigned a fixed, pre-defined partition of the database cluster. The read-set of each workload is the same as the write-set. Both contain the machines within the fixed workload partition and never change.

4 Experimental Setup

4.1 Platform

We use the same hardware for all machines running the client emulator, the web servers, the schedulers and the database engines. Each is a dual AMD Athlon MP 2600+ computer with 512MB of RAM and 2.1GHz CPU. All the machines use the RedHat Fedora Linux operating system. All nodes are connected through 100Mbps Ethernet LAN. We used 8 machines to run our databases and 12 machines to operate the Apache 1.3.31 web-server, running the PHP implementation of the business logic of the TPC-W benchmark and Rubis benchmark.

4.2 Benchmarks

4.2.1 TPC-W E-Commerce Benchmark

The TPC-W benchmark from the Transaction Processing Council [27] 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 4 GB.

The inventory images, totaling 1.8 GB, are resident on the web server. We implemented the 14 different interactions specified in the TPC-W benchmark specification. Of the 14 scripts, 6 are read-only, while 8 cause the database to be updated. Read-write interactions include user registration, updates of the shopping cart, two order-placement interactions, two involving order inquiry and display, and two involving administrative tasks. We use the same distribution of script execution as specified in TPC-W. The complexity of the interactions varies widely, with interactions taking between 20 ms and 1 second on an unloaded machine. Read-only interactions consist mostly of complex read queries in auto-commit mode, 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.

4.2.2 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 [24].

4.3 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 to go from one interaction to another. The session time and think time are generated from a random distribution with a specified mean.

	1 db	**2 db**	**4 db**	**8 db**
Latency (msec)	946	1200	1179	1255
Throughput (WIPS)	14	26	54	105

Table 1: Scaling for TPC-W benchmark.

	1 db	**2 db**	**4 db**	**8 db**
Latency (msec)	1082	1079	1474	1679
Throughput (WIPS)	82	144	267	393

Table 2: Scaling for Rubis benchmark.

5 Experimental Results

5.1 Baseline Experiments

We run the TPC-W and Rubis benchmarks separately with increasing number of databases to plot scaling. We measured the scaling by first measuring the number of clients needed to generate latency equal to the SLA. Then, for different numbers of databases we multiplied the number of clients. For example, it takes 25 clients to saturate a single TPC-W database. For 2 databases, we ran 50 clients to see if we get double the throughput but with the same latency. Similarly, it takes 150 clients to saturate 1 Rubis database. Tables 1 and 2 show the scaling in terms of average query latency and overall site throughput for the TPC-W and Rubis benchmarks, respectively. We see that both workloads scale with more database replicas. In particular scaling is linear for the TPC-W workload up to 8 database replicas and slightly less for Rubis.

In the following, we study the adaptations of a single workload (section 5.2) and both workloads (section 5.3) under a variety of load patterns or faults (section 5.4).

All further experimental numbers are obtained running an implementation of our dynamic content server on a cluster of 8 database server machines. We use a number of web server machines sufficient for the web server stage not to be the bottleneck for either workload. The largest number of web server machines used for any experiment is 12. We use one scheduler per workload and one controller.

The input load function for each workload is in terms of number of clients used by the client emulator for that workload normalized to the number of clients necessary to saturate one database for that particular workload (which is considered a level 1 load).

The thresholds we use in the experiments are a `HighSLAThreshold` of 600 ms and a `LowSLAThreshold` of 200 ms. The `HighSLAThreshold` was chosen conservatively to guarantee a 1 second end-to-end latency at the client for each of the two workloads. To select the low load threshold, we use a threshold that is below 50% of the high threshold for stability in small configurations (i.e., adapting from 1 to 2 databases). We use a latency sampling interval of 10 seconds for the schedulers. Our state-aware protocol allows us to avoid careful tuning of the sampling interval. Since sampling is suppressed during adaptation periods when effects on latency cannot be reliably observed, we can choose a relatively short sampling interval with its implied high potential reactivity to load changes.

To quickly add new databases, the amount of data to be transferred should be kept small. The *batching size* was selected such that we are not continually sending batch updates but still minimizing the state to be migrated. The best balance was achieved by selecting the *batching size* to be 1000 updates which allows us to add at least a new database every 60 seconds and keep the batching overhead to a minimum. The *RemoveConfidenceNumber* was selected to be 5 times the time to migrate so we chose it to be 5 minutes.

5.2 Single Workload Adaptations

In this section we discuss adaptations to load and their impact on performance for our main protocol with warm migration in comparison with two static algorithms: a static partitioning algorithm and a read-any-write-all algorithm.

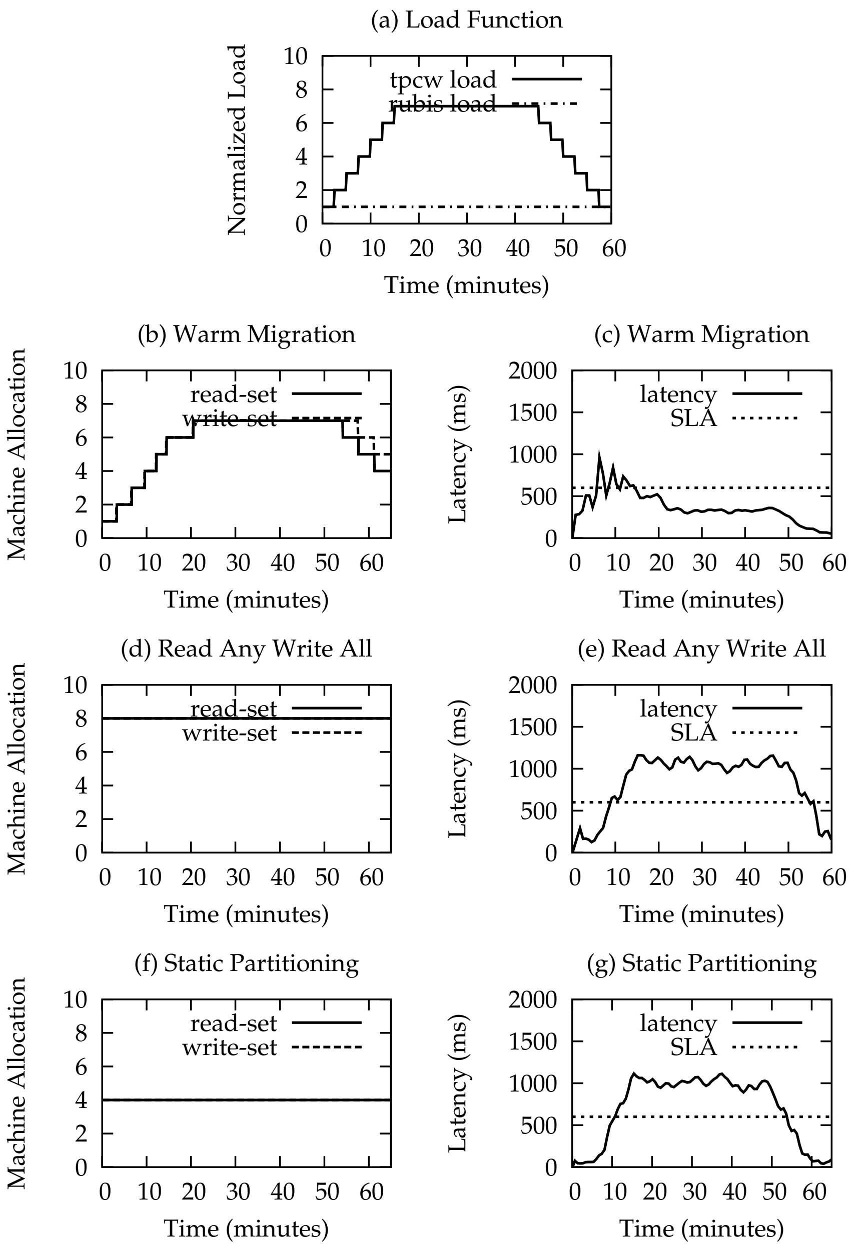


Figure 4: Comparison of Various Scheduling Schemes

Figure 4 shows in the x-axis time in seconds and, in the y-axis the input load function (top), machine allocation in terms of number of machines used in read and write sets over time (left) and query latency (right) respectively for each protocol.

We observe the behavior of only one workload, TPC-W, under a wide variation in load (between a level 1 load and a level 7 load and back) while the load of Rubis is kept constant at level 1 for the duration of the experiment. All performance numbers are derived through measurements using our experimental platform with up to 8 databases.

The overall result is that our system performs well as a result of its flexible machine allocation with very infrequent and brief SLA violations. Our system performs significantly better than either static algorithm which exhibit sustained and much higher latency SLA violations. In the static partitioning algorithm this is induced by insufficient resources, since this algorithm splits resources in half (4-4) among the two workloads independent of usage. The poor performance of the read-any-write-all algorithm is explained by the interference of the read sets of the two workloads in the buffer cache of the database since any request can be scheduled on any database at any point in time.

In more detail, we see that, while the TPC-W load is ramping up, the overall system is in underload and spare resources can be allocated to TPC-W. Note that load steps are about 2.5 minutes wide, giving a 7 fold load increase within 15 minutes to show that we have the agility to adapt to relatively steep load increases. We can see that the latency goes briefly over the TPC-W SLA (600 ms) as the system receives even more load while adapting to previous load increases. However, the system catches up quickly and the latency becomes under control immediately after the last load step.

Finally, in the last part of the experiment, we see that, as the load decreases, machines are gradually removed from the TPC-W workload allocation. From the warm migration allocation graph we see that write set removals lag behind the read set removals due to the more conservative nature of write-set adaptations.

5.3 Multiple Workload Adaptations

In this section we show the flexibility and robustness of our system under different combinations of load scenarios for the two workloads. In particular we show that our system switches rapidly from opportunistic use of spare databases to meet individual SLA's for either of the two workloads when spare resources exist to enforcing fairness between allocations when we are in overload for the system as a whole. Figure 5 shows a complex scenario where both loads fluctuate with periods of underload and overload for the system as a whole. Since our total capacity is 8 database machines, we see that, in the first part of the graph, while the TPC-W and Rubis loads are ramping up, the overall system is in underload and spare resources can still be allocated to workloads. During the stable load portion up until around 50 minutes, the system is loaded at estimated capacity. Then we induce further increases in the Rubis load until the respective loads are at levels 6 and 5 respectively, then hold these loads that exceed the total system capacity until the end of the experiment.

From the per-workload machine allocations of the adaptive partitioning with warm migration, we see that the TPC-W workload allocation increases, closely following the load increase just as before. Moreover, since each load step is longer than before (5 minutes), we can see that the system adaptation is even more graceful, keeping the latency under the SLA for almost the entire time when enough resources exist (up until almost 60 minutes). The same holds for Rubis where the system easily keeps the latency under the SLA. Note that even when we induce a Rubis load of 3, hence theoretically three databases might be needed to accommodate Rubis, the system increases its allocation to only as many resources as needed to meet the Rubis SLA. Because of the more lightweight and irregular nature of the Rubis workload, there are very brief oscillations between 3 and 1 databases (mostly in the Rubis read-set) while 2 databases appear to be sufficient for the most part.

The remaining latency graphs show the impact on performance on the two static scheduling algorithms: static partitioning and read-any write-all. When compared with Dynamic Partitioning with Warm Migration, we see that read-any-write-all performs very poorly for TPC-W in both underload and overload reaching latencies almost 3 times the allowed latency. For Rubis it also reaches latencies of 1.5 seconds in overload (after 60 minutes) due to the massive interference between the two workloads in the buffer cache of the database.

During the last portion of the run (after 60 minutes) once our controller detects that the whole system is in overload, it enforces a fair 4-4 allocation scheme between the two workloads. We see that two consecutive forced removes of machines from the 6-database TPC-W allocation are initiated in this case. The write set still lags behind the read set in this case due to

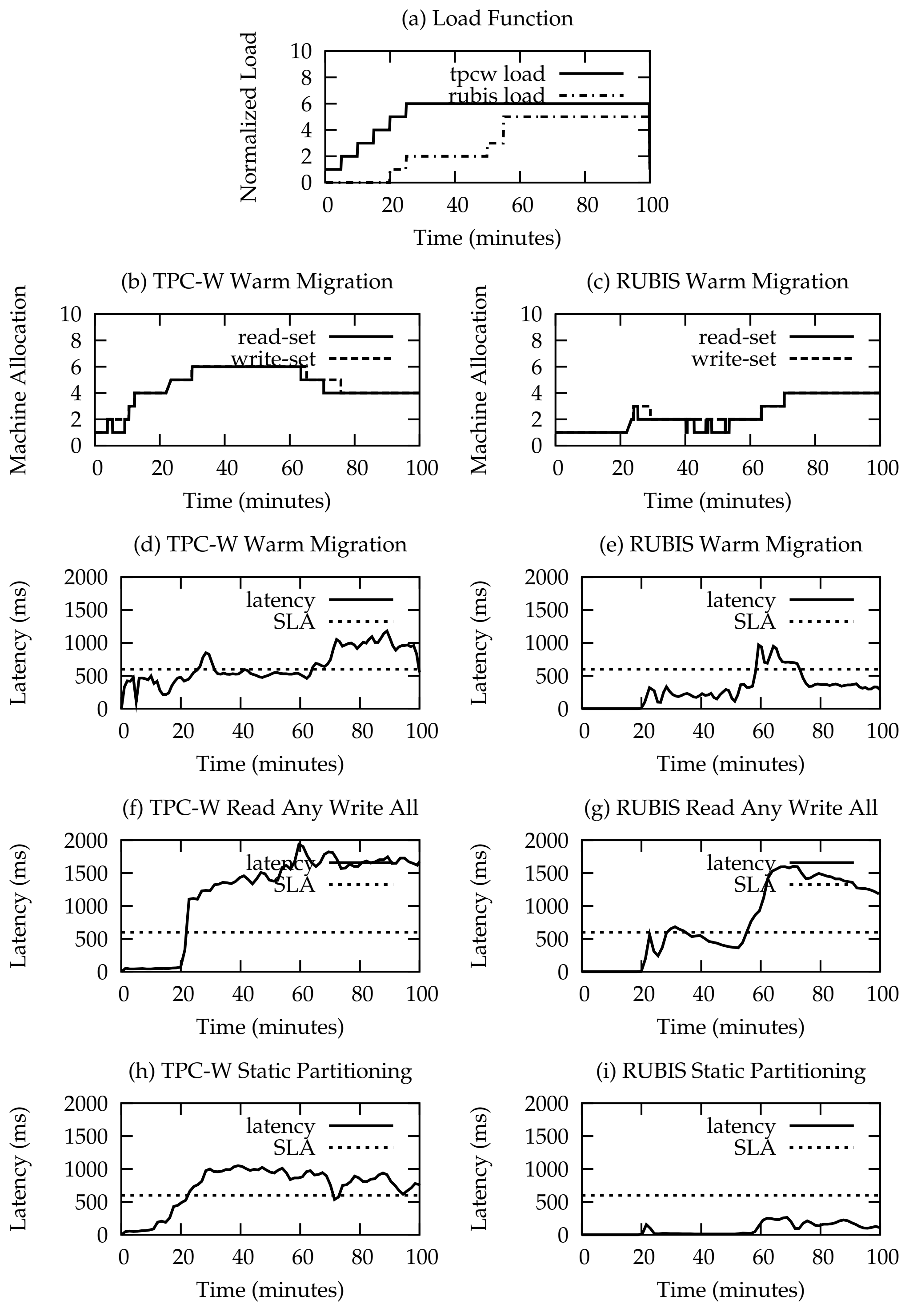
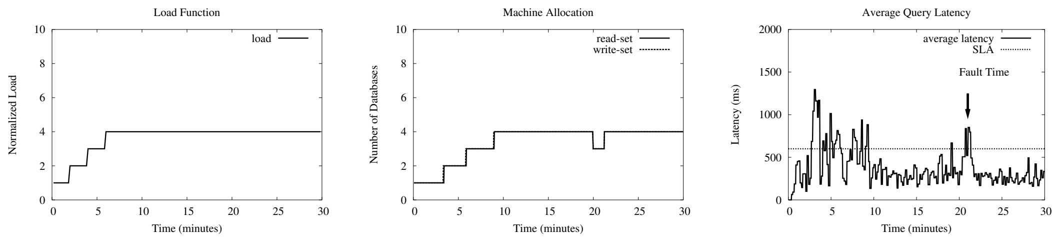


Figure 5: TPC-W Results of Multiple Scheduling



(a) Load Function (b) Machine Allocation (c) Query Latency

Figure 6: Adaptation to Faults for TPC-W.

the need to let on-going update transaction finish on the removed database. The two machines removed from the TPC-W workload are added to Rubis as they become available resulting in the half-half allocation enforced during overload. As a result, our dynamic partitioning behaves similarly to static partitioning for both workloads. The conspicuous exceptions, the two consecutive spikes in Rubis latency during this period, are due to the buffer-cache cold misses when adding the two machines previously running TPC-W, to the Rubis read set.

5.4 Adaptation to Failures

We use the load function shown in Figure 6 and induced a fault 20 minutes into the experiment. To provide fault tolerance, in this experiment, we used warm migration as our underlying migration scheme. As the load for TPC-W increases, the number of machines allocated to handle the load also increases. After 20 minutes of running time, we induce a fault in the one of the databases used by TPC-W. At this moment, the latency of TPC-W is approximately 300 milliseconds, therefore the controller does not take any action. Due to the database fault, the latency of TPC-W rapidly increases from 300 ms until it violates the SLA at time 22 minutes as shown in Figure 6. When the SLA is violated, the controller adapts by adding another database. At this point, the latency drops to pre-fault levels.

5.5 Detailed Experimental Justification of Design Decisions

In this section, we present experimental justifications for our choice of warm migration versus cold migration and for our use of state awareness during adaptation.

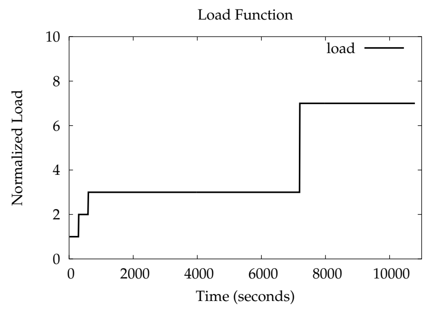
5.5.1 Comparison with Cold Migration

We use TPC-W as our benchmark in this experiment. As shown in Figure 7(a), we subject the system to a load that needs 3 databases to satisfy the SLA. After 2 hours (7200 seconds) elapsed time, we suddenly increase the load to level 7. Figure 7(b) shows the allocation of databases using cold migration. We see that the width of each adaptation step widens at each database addition in the cold migration case. This is because the amount of data to be transferred accumulates as suddenly a lot more transactions from the new clients need to be executed on these databases in addition to the application of the large 2-hours worth of log of missed updates. Hence, the system has a hard time catching up.

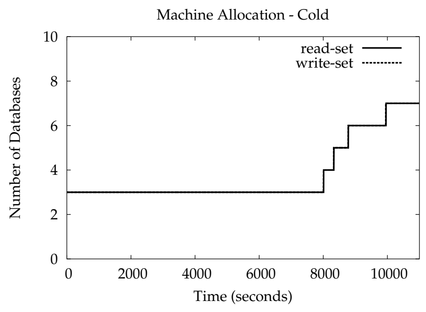
On the other hand, warm migration adapts faster to the large load spike. Both the intensity and duration of the resulting latency spike are reduced compared to cold migration, because databases are maintained relatively up-to-date through periodic batched updates.

5.5.2 Comparison with Stateless Scheduling

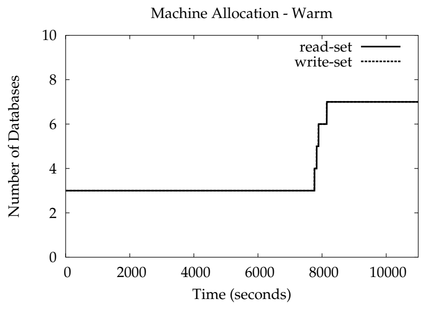
In Figure 8, we show the excessive oscillations that occur if the system state is not incorporated in the decision making process. In addition, we show that by taking into account the state of the system, our controller avoids these oscillations.



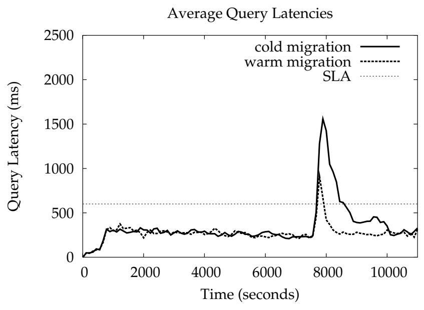
(a) Load Function



(b) Cold Migration Machine Allocation



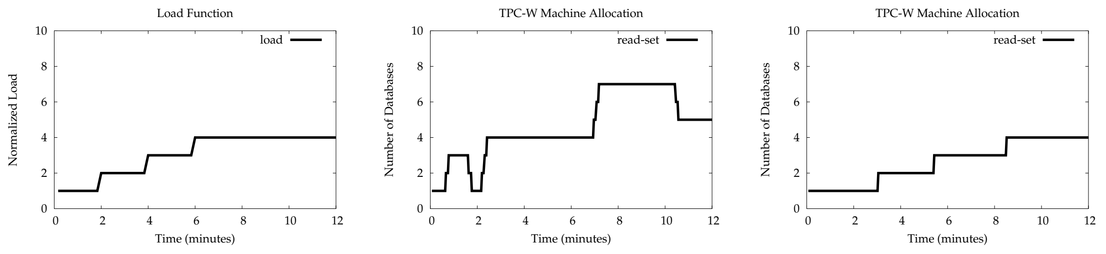
(c) Warm Migration Machine Allocation



(d) Average Query Latency

Figure 7: Comparison of Query Latencies Spikes with Cold/Warm Migration.

We disable the state awareness component of the controller and we use hot migration as our underlying migration scheme. From Figure 8, we can clearly see that without state awareness, the controller overreacts and allocates more than the number of databases needed to meet the SLA. This is because the latency normalizes only after the queries that caused congestion, still executing on the old database read-set are flushed out of the system. The overallocation causes further oscillation since it causes the latency to dip below the `LowSLAThreshold`. As a result, the controller removes the overallocated databases. The situation would be even worse with stateless warm or cold migration because the latency would continue to increase during migration. As a result, the controller would keep adding databases each



(a) Load Function (b) without State Awareness (c) with State Awareness

Figure 8: Comparison of Machine Allocation with versus without State Awareness (shown for TPC-W)

sampling interval throughout the data migration period.

On the other hand, by incorporating state awareness in the decision making process, we avoid oscillations. Figure 8 shows that the controller adds databases to meet the demand without overallocation. In addition, the controller maintains the latency within the SLA bounds. Although the controller makes some incorrect judgements, it only makes them in the read-set and avoids the costly mistakes in the write-set.

6 Related Work

This paper addresses the hard problem of dynamic resource allocation within the database tier, advancing the research area of autonomic computing [12]. Autonomic computing is the application of technology to manage technology, materialized into the development of automated self-regulating system mechanisms. This is as a very promising approach to dealing with the management of large scale systems, hence reducing the need for costly human intervention.

Our study draws on recently proposed transparent scaling through content-aware scheduling in replicated database clusters [6, 17, 21, 23] and in particular on our own previous work [2, 3] on asynchronous replication with conflict-aware scheduling.

Various scheduling policies for proportional share resource allocation can be found in the literature, including Lottery scheduling [28] and STFQ [10]. Steere et al. [26] describe a feedback-based real-time scheduler that provides reservations to applications based on dynamic feedback, eliminating the need to reserve resources a priori. Our feedback-based resource scheduler differs from proportional schedulers in that it uses both performance feedback and state awareness to achieve performance targets.

In numerous other papers discussing controllers [7, 8, 14] 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 [7]. 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.

Finally, our work is related but orthogonal to ongoing projects in the areas of self-managing databases that can self-optimize based on query statistics [16], or detect and adapt to their workload [9, 15], and to work in automatically reconfigurable static content web servers [22] and application servers [14].

7 Conclusion

In this paper, we introduce a novel solution to resource provisioning in the database tier of a dynamic content site. Our self-configuration software allows the server the flexibility to dynamically reallocate database commodity nodes across multiple workloads. 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 on-line auction benchmark to evaluate the reactivity and stability of our dynamic resource allocation protocol.

Our evaluation shows that our feedback-based scheduling approach can handle rapid variations in an application's backend resource requirements while maintaining quality of service across applications. We show that our approach works significantly better than read-any-write all and static partitioning approaches which suffer from workload interference and rigid allocations, respectively.

Moreover since our scheduling reactivity is limited by the length and load induced by data migration, we provide a good compromise through our warm migration approach. We show that warm migration works well for scenarios both with and without faults. Finally we show that keeping track of system state is key for avoiding oscillations in resource allocation. Our state-based approach thus obviates the need for careful hand tuning of sampling intervals or any other system parameters.

References

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

[2] C. Amza, A. Cox, and W. Zwaenepoel. Conflict-aware scheduling for dynamic content applications. In *Proceedings of the Fifth USENIX Symposium on Internet Technologies and Systems*, pages 71–84, March 2003.

[3] C. Amza, A. Cox, and W. Zwaenepoel. Distributed versioning: Consistent replication for scaling back-end databases of dynamic content web sites. In *4th ACM/IFIP/Usenix International Middleware Conference*, June 2003.

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

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

[6] Emmanuel Cecchet, Julie Marguerite, and Willy Zwaenepoel. C-jdbc: Flexible database clustering middleware. In *Proceedings of the USENIX 2004 Annual Technical Conference*, Jun 2004.

[7] Yixin Diao, Joseph L. Hellerstein, and Sujay Parekh. Optimizing quality of service using fuzzy control. In *DSOM '02: Proceedings of the 13th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, pages 42–53. Springer-Verlag, 2002.

[8] Yixin Diao, Joseph L. Hellerstein, Adam J. Storm, Maheswaran Surendra, Sam Lightstone, Sujay S. Parekh, and Christian Garcia-Arellano. Incorporating cost of control into the design of a load balancing controller. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 376–387. IEEE Computer Society, 2004.

[9] S. Elnaffar, P. Martin, and R. Horman. Automatically Classifying Database Workloads. In *Proceedings of the ACM Conference on Information and Knowledge Management*, November 2002.

[10] Pawan Goyal, Xingang Guo, and Harrick M. Vin. A Hierarchical CPU Scheduler for Multimedia Operating System. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, Seattle, WA, October 1996.

[11] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173–182, 1996.

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

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

[14] Ed Lassettre, D. W. Coleman, Yixin Diao, Steve Froehlich, Joseph L. Hellerstein, L. Hsiung, T. Mummert, M. Raghavachari, G. Parker, L. Russell, Maheswaran Surendra, V. Tseng, N. Wadia, and P. Ye. Dynamic surge protection: An approach to handling unexpected workload surges with resource actions that have lead times. In Marcus Brunner and Alexander Keller, editors, *DSOM*, volume 2867 of *Lecture Notes in Computer Science*, pages 82–92. Springer, 2003.

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

[16] Microsoft Research. AutoAdmin: Self-Tuning and Self-Administering Databases. http://www.research.microsoft.com/research/dmx/AutoAdmin, 2003.

[17] J. M. Milan-Franco, Ricardo Jimenez-Peris, Marta Patio-Martnez, and Bettina Kemme. Adaptive middleware for data replication. In *Proceedings of the 5th ACM/IFIP/USENIX International Middleware Conference*, October 2004.

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

[19] M. Patiqo-Martinez, R. Jiminez-Peris, B. Kemme, and G. Alonso. Scalable replication in database clusters. In *International Symposium on Distributed Computing*, pages 315–329, October 2000.

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

[21] Christian Plattner and Gustavo Alonso. Ganymed: Scalable Replication for Transactional Web Applications. In *Proceedings of the 5th ACM/IFIP/Usenix International Middleware Conference*, October 2004.

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

[23] U. Rhom, K. Bhom, H.-J. Schek, and H. Schuldt. Fas - a freshness-sensitive coordination middleware for a cluster of olap components. In *Proceedings of the 28th International Conference on Very Large Databases*, pages 134–143, August 2002.

[24] Kai Shen, Tao Yang, Lingkun Chu, JoAnne L. Holliday, Doug Kuschner, and Huican Zhu. Neptune: Scalable replica management and programming support for cluster-based network services. In *Proceedings of the Third USENIX Symposium on Internet Technologies and Systems*, pages 207–216, March 2001.

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

[26] David C. Steere, Ashvin Goel, Joshua Gruenberg, Dylan McNamee, Calton Pu, and Jonathan Walpole. A Feedback-driven Proportion Allocator for Real-Rate Scheduling. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, February 1999.

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

[28] Carl A. Waldspurger and William E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*, Monterey, CA, November 1994.

[29] Matt Welsh and David Culler. Adaptive overload control for busy internet servers. In *Proceedings of the Fifth USENIX Symposium on Internet Technologies and Systems*, March 2003.