DotSlash: Providing Dynamic Scalability to Web Applications with On-demand Distributed Query Result Caching[*]

Weibin Zhao, Henning Schulzrinne
Department of Computer Science
Columbia University
New York, NY 10027
{zwb,hgs}@cs.columbia.edu

Abstract

Scalability poses a significant challenge for today's web applications, mainly due to the large population of potential users. To effectively address the problem of short-term dramatic load spikes caused by web hotspots, we developed a self-configuring and scalable rescue system called DotSlash. The primary goal of our system is to provide dynamic scalability to web applications by enabling a web site to obtain resources dynamically, and use them autonomically without any administrative intervention. To address the database server bottleneck, DotSlash allows a web site to set up on-demand distributed query result caching, which greatly reduces the database workload for read mostly databases, and thus increases the request rate supported at a DotSlash-enabled web site. The novelty of our work is that our query result caching is on demand, and operated based on load conditions. The caching remains inactive as long as the load is normal, but is activated once the load is heavy. This approach offers good data consistency during normal load situations, and good scalability with relaxed data consistency for heavy load periods. We have built a prototype system for the widely used LAMP configuration, and evaluated our system using the RUBBoS bulletin board benchmark. Experiments show that a DotSlash-enhanced web site can improve the maximum request rate supported by a factor of 5 using 8 rescue servers for the RUBBoS submission mix, and by a factor of 10 using 15 rescue servers for the RUBBoS read-only mix.

1 Introduction

Scalability poses a significant challenge for today's web applications, mainly due to the large population of potential users. The phenomenon of web hotspots, also known as flash crowds or the Slashdot effect [1], is a well identified example. Traditionally, a web site has a fixed set of resources, leaving it unable to handle a large load increase without significant overprovisioning. It is unacceptable to sacrifice availability by losing valuable users in the critical period – "15 minutes of fame", but it is uneconomical to invest on more powerful infrastructure which is idle most of time.

To effectively address the problem of short-term dramatic load spikes caused by web hotspots, we developed a self-configuring and scalable rescue system called DotSlash [24, 25], which allows a web site to set up a distributed web server system in wide area networks on the fly. The primary goal of DotSlash is to provide dynamic scalability to web applications, i.e., to enable a web site to obtain resources dynamically, and use them autonomically without any administrative intervention in handling load spikes. There are four aspects of dynamic scalability for web applications, namely access network bandwidth, web server, application server, and database server. Our previous work on DotSlash [24, 25] can distribute static content and application programs dynamically from the origin server to rescue servers. As a result, the bottlenecks at the access network, web server, and application server can be effectively removed.

In this paper, we address the database server bottleneck within the DotSlash framework. Database scalability is an important issue for web applications. First, the database can be the most constrained resource in certain web applications such as on-line bookstores [4]. Secondly, after other bottlenecks have been removed, the database server will become a bottleneck at some point if the load continues to increase. There has been a large body of research work on database replication, partition, caching, and clustering for improving database scalability [16, 19, 3, 6, 10, 8]. However, existing systems often involve manual configurations, making them difficult to be deployed dynamically to new servers. To provide dynamic scalability, we need to reduce administrative intervention as much as possible. This paper presents a technique that allows a web site to set up on-demand distributed query result caching on the fly, which can greatly reduce the workload at the back-end database server, and thus increase the request rate supported at a DotSlash-enabled web site. The novelty of

[*]This work was supported in part by the National Science Foundation (ANI-0117738).

our work is that our query result caching is on demand, and operated based on load conditions. The caching remains inactive as long as the load is normal, but is activated once the load is heavy so as to improve database scalability. This approach offers good data consistency during normal load situations, and good scalability with relaxed data consistency for heavy load periods. On-demand caching also helps to keep our system simple. Since we don't need to replicate a lot of DBMS functions, the overhead of our caching is very low. Other important features of our query result caching include:

- Distributed: Our query result cache is fully distributed, and thus can scale well. Normally, each web server has its own query result cache. However, different web servers may share a cache so as to increase cache hit ratio, and further reduce the load at the back-end database.
- Self-configuring: The distributed cache is self-configuring, and fully controlled by the origin server. There are two major control parameters, namely expiration time for cached objects, and whether or not to perform cache invalidation. In general, a short expiration time in the order of 1 minute is used to bound the staleness of cached objects. A simple invalidation may be performed for cached objects based on local updates, and in the granularity of table column. Our cache invalidation scheme is designed to be simple, low overhead, and applicable to all applications.
- Transparent: Our caching system is transparent to web users and web applications. No change is needed at client-side web browsers and at server-side application programs. Furthermore, our caching system handles the HTTP *Cache-Control* header. If there is *no-cache* or *max-age=0* in the HTTP header of a client request, the query result cache will be bypassed.

The remainder of this paper is organized as follows. We first discuss related work in Section 2, and then describe our system design in details in Section 3. We give extensive performance evaluation for our prototype system in Section 4, and conclude in Section 5.

2 Related Work

Various systems have been developed to improve the scalability of web servers and application servers dynamically, such as Akamai [2], ACDN [17], and our previous work on DotSlash [24, 25]. However, dynamic scalability for databases is largely an open issue. Recently, Olston et al. [14] proposed a scalability service using multicast-based consistency management, but their service is not transparent since clients need to connect to proxy servers in order to use their service, and the fixed number of proxy servers may become a scalability bottleneck as the number of clients and home servers increases. Amza et al. [5] evaluated transparent scaling techniques for dynamic content web sites. Their results show that query result caching can significantly increase performance for read-mostly databases, whereas content-aware scheduling is effective for write-intensive databases.

Replication is a widely used mechanism to provide better scalability for databases. The Ganymed [16] system separates update and read-only transactions, and routes updates to a main database server and queries to read-only database copies. GlobeDB [19] uses partially replicated databases based on data partition to reduce update traffic. These systems can deliver better performance for a pre-configured number of database replicas, but they do not address deploying replicas dynamically to new servers.

Database caching [20, 3, 6, 10] is very effective in reducing the workload at the back-end database. However, existing systems need administrative intervention for cache deployment and setup, and the caching functionality is active in all cases. In contrast, our query result caching is activated only during heavy load periods, and can be deployed dynamically to new servers.

Database clustering [8, 15, 13] is a mechanism to pool database servers together so as to provide high availability and performance. While Oracle Real Application Clusters [15] uses a shared cache architecture, MySQL Cluster [13] is built on a shared-nothing architecture. Clustered JDBC [8] implements the Redundant Array of Inexpensive Databases concept, and provides a single virtual database to the application through the JDBC interface. Generally speaking, database clustering is a solution at the database server tier for availability, reliability, and performance. In contrast, DotSlash is a solution at the web/application server tier for dynamic scalability. Thus, our system is orthogonal to database clustering, and can be used together with database clustering at dynamic content web sites.

3 System Design

This section describes our system design. We first outline our major design goals, chosen scalability mechanism, the application model, and our system architecture. We then give details about our cache-enhanced data driver, flexible query result caching engine, on-demand and self-configuring cache control, and unique data consistency approach.

3.1 Design Goals

Our design goals are dynamic scalability, self-configuration, and transparency. First, we aim to provide a mechanism that can be deployed to new servers

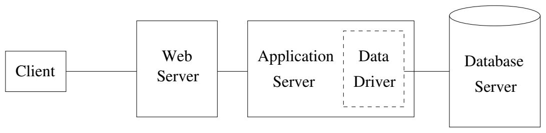


Figure 1: DotSlash Application Model

on demand so as to improve database scalability dynamically for web applications. Since deploying a scalability mechanism dynamically incurs an overhead at the origin server, we need to reduce this overhead as much as possible. Secondly, our system is designed to be self-configuring, handling dramatic load spikes autonomically without any administrative intervention. Finally, our system aims to be transparent to both web users and web applications. Without the need to change existing applications and user browsers, our system is easy to deploy.

3.2 Scalability Mechanisms

A spectrum of mechanisms can be used to improve database scalability. In general, caching and replication are good for read mostly databases, whereas partitioning may be useful when updates are frequent. For the purpose of handling web hotspots, we focus on read mostly databases, which are common for web applications such as news sites and web forums. Compared to replication, caching is easier to deploy dynamically, and incurs lower overhead at the origin server because data in caching is distributed from the origin server to caches on-demand, avoiding unnecessary data transfers. Thus, we narrow down our option to database caching.

In terms of database caching, we have two main design choices, namely table level caching and query result caching. Although table level caching [3, 6, 10] is more efficient in that it can answer arbitrary queries on cached tables, query result caching [20] is much simpler and can save expensive computations on cache hits. Thus, we chose to use query result caching in DotSlash.

3.3 Application Model

We consider the standard three-tier web architecture, shown in Figure 1. Application programs running at the application server access application data stored in the database server through a data driver, which is normally a system component of the application server. The data driver provides a standard API for web applications to store and retrieve data in the back-end database.

3.4 System Architecture

To provide application and user transparency, we add the query result caching functionality to the data driver shown in Figure 1. In our prototype system, we use the widely used LAMP configuration (i.e., Linux, Apache, MySQL, and PHP), where the PHP module resides in the Apache web server. Thus, for our prototype system, we provide query result caching at the PHP data driver for MySQL databases.

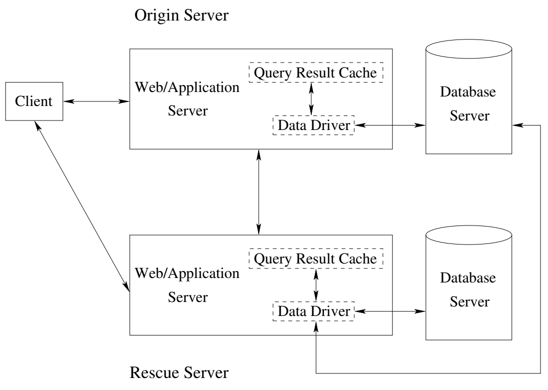


Figure 2: DotSlash System Architecture

The architecture of our system is illustrated in Figure 2. When the load is heavy, the origin server uses service discovery to discover spare capacity in a mutual-aid community, and sets up a number of rescue servers on the fly. Then, the origin server redirects a portion of client requests to its rescue servers using DNS round robin and HTTP redirect. A rescue server works as a reverse caching proxy for the origin server. In addition to caching static content from the origin server, a rescue server replicates PHP scripts dynamically from the origin server, and accesses the database at the origin server. To reduce the database workload at the origin server, query result caching is enabled at the origin server and all its rescue servers.

3.5 Data Driver

We modify the original PHP data driver for MySQL databases to provide query result caching. Since the data driver intercepts all database queries, it is straight forward to add additional processing functions without changing the application API and database interface.

Our caching-enhanced data driver handles each database query as follows. For a read-only query (i.e., the SQL *select* statement), the data driver first checks whether the query result is cached in the query result cache. If it is a cache hit, the data driver gets the result from the cache, and returns it to the application immediately. In case it is a cache miss, the data driver submits the query to the corresponding database, which can be its local database or the database at the origin server. Then, the data driver gets the query result from the database, saves it to the query result cache, and returns it to the application. For a write query (i.e., the SQL *insert*, *up-*

date, or *delete* statement), the data driver forwards the query directly to the database.

3.6 Query Result Cache

We keep the query result cache as a separate component from the data driver. The advantage for doing this is that we can experiment and use different engines as our caching storage.

The interface between the data driver and query result cache has two simple functions: *check_in* and *check_out*. The *check_in* function takes three parameters as input, namely query string, query result, and caching TTL. Essentially, a cached object is a key-value pair, using the query string as the key, and the query result as the value. The caching TTL is a control data, indicating the duration that a cached object to be kept as valid. The *check_in* function serializes the original query result structure into a byte stream, and saves it at the underlying caching storage engine. In contrast, the *check_out* function takes only one parameter as input, namely query string, and retrieves the query result from the underlying caching storage engine. For a cache hit, it de-serializes the query result byte stream into the original query result structure, and returns a pointer to the result structure. In case of a cache miss, it returns a NULL pointer.

We consider both disk and memory as our caching storage engine. For the disk storage engine, we use the GDBM library. In the *check_in* function, we first use the ELF hash algorithm to map the query string into a file name, and then store the query string, the serialized query result byte stream, the check_in timestamp, and the caching TTL into the file. Note that different query strings may be mapped into the same file name. Since the probability for this type of hash conflicts is very small, less than 1% in all of our test cases, we adopt a simple strategy for handling the conflicts: just letting the new query and its result overwrite the old one. This strategy keeps our system simple without losing much performance. In the *check_out* function, we use the same ELF hash algorithm to map the query string into a file name. If the file exists, we check whether the stored query string matches the input query string, and the cached result is not expired (i.e., check_out timestamp $-$ check_in timestamp $<$ caching TTL). If so, it is a cache hit; otherwise, it is a cache miss. We will address cache invalidation later separately in Section 3.8. Although the disk storage engine is simple and easy to use, its performance is not satisfactory since the disk becomes a bottleneck when the load is heavy. Thus, we turn our attention to the memory storage engine.

We use *memcached* [12] as our memory storage engine, which is a generic high-performance distributed memory object caching system. Memcached uses a client-server model. At the server side, a daemon maintains cached objects in dynamically allocated memory. Each cached object is a key-value pair with an expiration time. At the client side, applications use the memcached API to access the cache. In our prototype system, we use an open source C library *libmemcache* [11] to access the cache. A query string cannot be used directly as the cache key in memcached because a valid memcached cache key cannot have any space characters. Thus, similar to the disk storage engine, we use the ELF hash algorithm to map a query string into a key for the cache object. Then, we store both the query string and the serialized query result byte stream as the value of the cache object. As the memcached server already handles the expiration time for cached objects, in the *check_out* function we only need to make sure that the stored query string matches the input query string for a cache hit.

In addition to high performance, memcached is very flexible. A memcached server can be co-located with the web/application server, or be on a separate machine. Also, a memcached server can be shared among an origin server and its rescue servers, or among a subset of rescue servers. Using a shared cache can help to reduce the workload at the origin database server. However, a shared cache may become a potential performance bottleneck, and accessing a remote cache from the application server incurs longer delays.

3.7 Cache Control

Our caching system has three control parameters, namely cache server, caching TTL, and cache invalidation. These parameters can be configured in the Apache web server configuration file *httpd.conf*, but the default setup should work well in most cases. The default value for cache server is NULL, meaning that our query result caching is off by default. To enable query result caching, a cache server must be specified in the form of *host_name:port_number*. Caching TTL is largely an application dependent parameter. We use a default value of 60 seconds for caching TTL in our prototype system based on our experimental results in Section 4.3. By default, the cache invalidation functionality is off in our system since there are trade-offs between scalability and data consistency, and we give more weight to scalability during heavy load periods so as to handle the workload well.

On-demand caching is the most unique feature of our system since our query result caching is activated only when the load is heavy. In our system, we use two configurable parameters, lower threshold ρ^l and upper threshold ρ^u, to define three load regions: lightly loaded region $[0, \rho^l)$, desired load region $[\rho^l, \rho^u]$, and heavily loaded region $(\rho^u, 100\%]$. We measure both network and CPU utilization to indicate the load level. A DotSlash-enabled web site uses rescue services to handle load spikes [24]. As a result, a web server has three DotSlash states: *SOS state* if it gets rescue services from others, *rescue state* if it provides rescue services to others, and *normal state* otherwise. We illustrate the operations of our on-demand caching in Figure 3, and give details next. Based on a web server's DotSlash states and load regions, our query result caching is activated if one of the following cases happens.

- When a web server is in the SOS state. In this case, the web server is an origin server with a number of rescue servers.
- When a web server is in the rescue state. In this case, the web server works as a rescue server for an origin server. For this case, the query result caching is used only for queries applied to the remote database at the origin server, but not for queries applied to the local database. In other words, the query result caching is used only for the rescue purpose at a rescue server.
- When a web server is in the normal state but its load level has exceeded the upper threshold. In this case, the web server is heavily loaded, but it has not set up any rescue servers yet.

Similarly, based on a web server's DotSlash states and load regions, our query result caching is de-activated if one of the following cases happens.

- When an origin server switches from the SOS state to the normal state
- When a rescue server switches from the rescue state to the normal state
- When a web server's load level falls below the lower threshold and it is in the normal state

Self configuration is another important feature of our caching system. When an origin server sets up its rescue servers, it passes the query result caching control parameters to its rescue servers, which include caching TTL and whether to perform cache invalidation. By doing so, a rescue server can manage cached objects based on the instructions from the origin server. In other words, an origin server can set up a distributed query result caching system on the fly using one set of control parameters.

Distributed caching is a natural feature of our system. In the default setup, each web/application server has its own, co-located query result cache, which provides the following advantages. First, we do not need any additional resources to run cache servers. Secondly, it is easier for administration since we can start or shutdown the cache server and web/application server at same time in one machine. Third, it is more efficient in terms of resource utilization because our query result caching is on demand, and the cache server is idle most of time. Finally, a naturally distributed caching system can avoid becoming a bottleneck since an origin server can bring

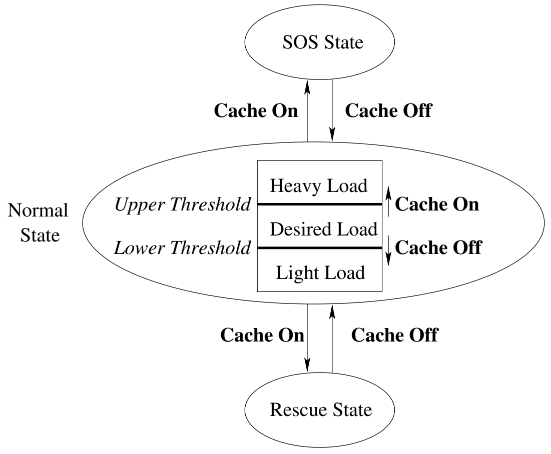


Figure 3: DotSlash on-demand caching, where caching is activated (cache on) and de-activated (cache off) based on the web server's DotSlash states (normal, SOS, and rescue) and load regions (desired load, heavy load, and light load)

in more query result caches as it obtains more rescue servers.

3.8 Data Consistency

Data consistency is an important issue for any caching systems. For our query result caching, we have four mechanisms for providing a better consistency with a low overhead. We use caching TTL to bound the staleness of cached objects, use a simple cache invalidation mechanism to discard stale results, do not cache any empty query result sets when cache invalidation is off so as to enable primary key based selections to observe the effect of new insertions immediately, and allow web users to use the HTTP *Cache-Control* header to bypass our query result caching.

Caching TTL is the most basic and simplest mechanism. In general, using a smaller caching TTL can improve the consistency between the cache and the backend database. Also, it can reduce the space requirement for caching storage. However, using a caching TTL that is too small than necessary will reduce the caching benefit. Based on our experimental results in Section 4.3, we use a default value of 60 seconds for caching TTL in our prototype system.

Our cache invalidation is designed to be simple, low overhead, and applicable to all applications. The basic invalidation scheme is as follows. Each web/application server maintains a list of the latest write timestamps (LWT) for all database tables and table columns that it has submitted write queries for. In general, an *insert* or *delete* statement will cause the LWT for a database table to be updated, and an *update* statement will cause the

LWTs for one or multiple table columns to be updated. When a cached result is checked out, we compare its check_in timestamp with all LWTs of its related tables and table columns. If its check_in timestamp is smaller in any of the above comparisons, the result is regarded as stale, and is discarded. It is easy to see that our cache invalidation is generic and application independent. Although application specific invalidation schemes such as template-based invalidation [9] may deliver better performance, they are difficult to be applied to our rescue system since a rescue server needs to be able to rescue any applications.

Note that our cache invalidation is based on local write queries only, which may sound limited at the first look. We choose to do so for the following reasons. First, it helps to keep our system simple. In our rescue system, rescue servers do not know each other, and the set of rescue servers for an origin server is changing from time to time. Thus, to reliably distribute all write queries to all caches, we have to forward all write queries to the origin server, and rely on the origin server to distribute the write queries to all its rescue servers. This will incur an overhead at the origin server which is already heavily loaded. Secondly, local update based invalidation is useful for some applications since it enables a web user to observe the effect of its write queries immediately even when our caching is on. Third, the data inconsistency caused by the local update based invalidation does not exist when the load is normal since our query result caching is on demand, and it is activated only when the load is heavy. Finally, an origin server may need to give up cache invalidation at all in order to handle extremely heavy load.

In case that our query result caching is on but cache invalidation is off, we do not cache any empty query result sets so as to enable primary key based selections to observe the effect of new insertions immediately. This feature is useful for certain applications such as on-line user registrations. The often used database query sequence for a on-line user registration is as follow: (1) using a *select* to check whether the key, user ID, to be inserted has already existed in the database, (2) using an *insert* to insert the registration information into the database, and (3) using a *select* to retrieve the information based on the same key from the database to provide a confirmation for the registration. Clearly, for a successful user registration, the *select* in step (1) returns an empty result set, whereas the *select* in step (3) returns a result set with just one row.

For a read-only database query, by default our data driver first checks it against the query result cache, and then submit the query to the back-end database only if it is a cache miss. However, we provide a way for web users to bypass our query result caching, and get query results directly from the back-end database. This function is achieved by handling the HTTP *Cache-Control* header at our data driver. We have a DotSlash module for the Apache web server, which implements our rescue services. For each client request, our DotSlash module passes the HTTP *Cache-Control* header to our data driver. If there is *no-cache* or *max-age=0* in the header, then our data driver will submit a read query directly to the database without checking the query result cache first, but the query result obtained from the database will be still stored in our query result cache.

4 Evaluation

This section provides the performance evaluation of our prototype system using the RUBBoS bulletin board benchmark [18]. After a brief description of the RUBBoS benchmark and our experimental setup, we discuss how to choose a proper caching TTL, we then give experimental results for the RUBBoS read-only and submission mixes.

4.1 Benchmark Description

Our caching system targets read mostly databases, which are common for web applications such as news sites, blogs, and web forums. To evaluate our prototype system, we use the RUBBoS benchmark, which is a bulletin board benchmark modeled after an online news forum like Slashdot [21], and has been used in a number of systems [20, 14, 25].

RUBBoS supports discussion threads. Each thread has a story at its root, and a number of comments for that story, which may be nested. There are two types of users in RUBBoS: regular users who browse and submit stories and comments, and moderators who in addition review stories and rate comments. The PHP version of RUBBoS has 19 PHP scripts, and the size of script files varies between 1 and 7 KB. The database has a size of 439 MB, and contains $500,000$ users and 2 years of stories and comments. There are 15 to 25 stories per day, and 20 to 50 comments per story. The length of story and comment bodies is between 1 and 8 KB.

We use RUBBoS clients written in Java to generate workloads. Each RUBBoS client can emulate a few hundred HTTP clients. An HTTP client issues a sequence of requests using a think time that follows a negative exponential distribution, with an average of 7 seconds [22]. If the request rate to be generated is high, multiple RUBBoS clients are used, each running on a separate machine. We use 7 seconds [7] as the timeout value for getting the response for a request. If more than 10% [7] of issued requests time out, the web server is considered overloaded.

RUBBoS has two major workload mixes, read-only and submission. The read-only mix calls browse scripts, story/comment view scripts, and search scripts with a probability of $2/3$, $1/6$, and $1/6$, respectively. The submission mix calls update scripts with a probability of $1/10$. The update scripts have both read and write database queries. As a result, 2% of the total database queries in the submission mix are write queries. A special property of the RUBBoS workload mixes is that for the same request rate, its read-only mix triggers a higher workload at the database than its submission mix. The reason is that each pre-generated story in the database has 20 to 50 comments. In contrast, a newly submitted story has no comments at all, or only a few comments. Each emulated RUBBoS client always starts with, and often returns to the *Stories Of The Day* page, which has the most recent 10 stories.

4.2 Experimental Setup

The goal of our query result caching is to relieve the database server bottleneck, and thus to increase the maximum request rate supported by a web site. Since our focus is on scalability, and the bottleneck we addressed is CPU, all our experiments described in this paper are carried out in our local area network, which bears similarity to the setting of a hosting company. We use a cluster of Linux machines connected via 100 Mb/s fast Ethernet. These machines have three different configurations. The configuration for web/application servers has a 3 GHz Intel Pentium 4 CPU, and 2 GB of memory, running Red Hat Enterprise Linux AS v.3, with Linux kernel 2.4.21-32.0.1.EL. The configuration for the database server has a 2 GHz AMD Athlon XP CPU, and 1 GB of memory, running Red Hat 9.0, with Linux kernel 2.4.20-20.9. The configuration for client emulators has a 1 GHz Intel Pentium III CPU, and 512 MB of memory, running Red Hat 9.0, with Linux kernel 2.4.20-20.9.

We run a varying number of web/application servers in different experiments. All web/application servers run Apache 2.0.49, configured with PHP 4.3.6, *worker* multi-processing module, proxy modules, cache modules, and our DotSlash module [24]. The PHP module includes our DotSlash extension for dynamic script replication [25], and the cache-enhanced data driver for MySQL database to support our query result caching. The cache server is started and shutdown along with the Apache web server. A co-located cache server has a storage space limit of 200 MB, whereas a shared cache server has a storage space limit of 1 GB.

The database server runs MySQL 4.0.18, which provides a number of different storage engines. Based on our evaluation, the default MySQL storage engine MyISAM delivers a better performance than the InnoDB storage engine for our chosen benchmark RUBBoS. Thus, we use MyISAM tables in all our experiments. To enhance the performance of MyISAM tables during heavy update periods, we configure MySQL with *delay_key_write=all* to delay the writing of index data to disk [23]. To support a large number of concurrent connections, we configure MySQL with *open_files_limit=65535*, and *max_connections=8192*. MySQL has a warm-up stage to load the table index information into memory. To obtain consistent results in the steady state of MySQL, we restart the MySQL server after each run of our experiments, and warm up it before each experiment using the read-only mix with 1400 emulated clients. This workload causes the database server CPU to be loaded around 70%. After each run of the submission mix, the RUBBoS database is restored to its original content so that all experiments start with the same database content.

We run the *dot-slash.net* DNS domain for dynamic DNS name registrations. Also, we run a DotSlash service registry to allow registration and discovery of DotSlash rescue services.

4.3 Caching TTL

Caching TTL for query results is application dependent in that different applications may need to use different caching TTLs based on their consistency requirements. But in general the caching TTL should be small, and in a range of a few seconds to a few minutes. One way to find out the proper caching TTL in an acceptable range is to look at the relationship between the caching TTL and hit ratio of our query result caching.

For the RUBBoS application, we perform a set of experiments using different caching TTLs, and examine the effect on the hit ratio of our query result caching. In the experiments, we use the read-only mix, and each experiment runs for 10 minutes. In the beginning of each run, the query result cache is empty. We experiment with the following caching TTLs: 0, 1, 2, 4, 8, 15, 30, 60, 120, 240, and 480 seconds, where 0 means that the caching TTL is infinity. We show our experimental results in Figure 4, where the result for TTL $= 0$ is given at TTL $= 960$ seconds since the run length of our experiments is 600 seconds, and any TTL bigger than 600 seconds is equivalent to infinity. We can observe that the cache hit ratio increases when the caching TTL increases. Even with a very small caching TTL, we can still have a good hit ratio, meaning that the query result caching is very effective for the RUBBoS application. For example, we have a hit ratio of 66.9% with a caching TTL of only 1 second, and we achieve a hit ratio of 89.4% when the caching TTL is 60 seconds. In all our following experiments, we use 60 seconds as the default caching TTL since it is good enough to bound the staleness of cached objects in RUBBoS, and it achieves a good hit ratio close to 90%.

Table 1: Performance summary for the RUBBoS read-only mix: the maximum request rate supported, and the number of rescue servers used and the origin web server cache hit ratio at the peak rate. Test cases: READ1 – no rescue, no cache; READ2 – no rescue, with a co-located cache; READ3 – with rescue, no cache; READ4 – with rescue, with a co-located cache; and READ5 – with rescue, with a shared cache.

Test case	Max rate supported (requests/second)	Number rescue servers used	Cache hit ratio	Performance compared with READ1	Performance compared with READ3
READ1	117	no rescue	no cache	100%	
READ2	125	no rescue	91%	107%	
READ3	249	4	no cache	213%	100%
READ4	1151	15	87%	984%	462%
READ5	828	13	93%	708%	333%

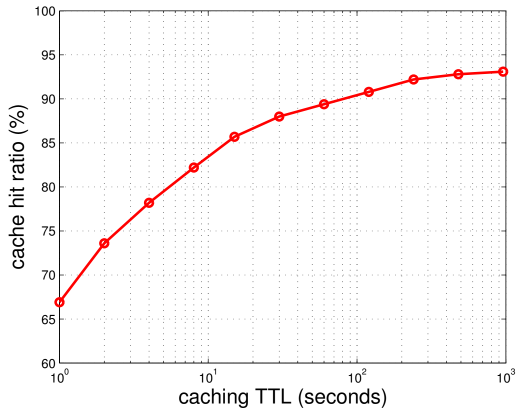


Figure 4: The relationship between the caching TTL and hit ratio of our query result caching in a set of 10-minute read-only experiments

4.4 Results for Read-Only Mix

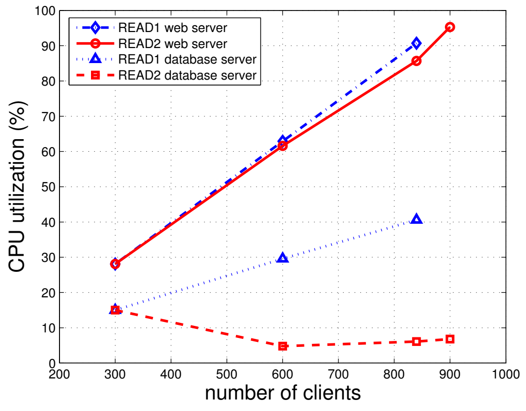
Query result caching aims at read mostly databases, and delivers the best performance for read-only accesses. Thus, we first test our caching system with the RUBBoS read-only mix. Depending on whether rescue servers are available, whether the query result caching is enabled, and whether each web/application server has a co-located cache or uses a shared cache server running on a separate machine, we have the following five test cases:

- *READ1* – no rescue, no cache
- *READ2* – no rescue, with a co-located cache
- *READ3* – with rescue, no cache
- *READ4* – with rescue, with a co-located cache
- *READ5* – with rescue, with a shared cache

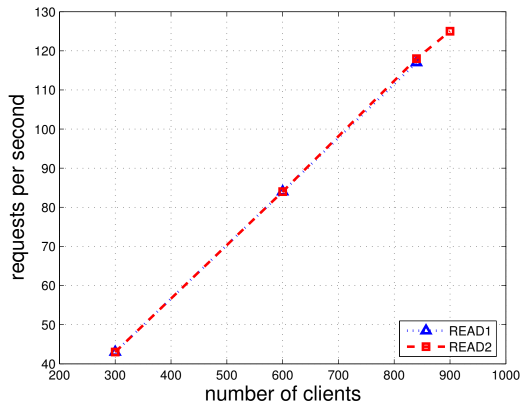
Table 1 summaries the performance of our prototype system for the RUBBoS read-only mix. Without using our rescue and caching services, a web server can only support a request rate of 117 requests/second. The request rate supported increases to 249 requests/second by using our rescue services only with 4 rescue servers, and increases to 1151 requests/second by using our rescue and caching services together with 15 rescue servers. Compared with READ1, the improvement achieved is 213% in READ3 and 984% in READ4. Compared with READ3, the improvement achieved is 462% in READ4.

Figure 5 shows the experimental results for test case READ1 and READ2, where rescue servers are not available. We give the CPU utilization for the web server and database server in Figure 5(a), and present the request rate supported in Figure 5(b). We observe that the web server CPU is the bottleneck. When the load is light with 300 clients, the caching is not activated. Thus, we have the same CPU utilization for READ1 and READ2. When the load is heavy with 840 clients, the caching is turned on, and we can observe a big difference in CPU utilization. The database server CPU utilization is 41% in READ1, but is only 6% in READ2, meaning that the caching is very effective in reducing the database workload. At the same time, the web server CPU utilization decreases from 91% to 86% by using the caching, indicating that getting query results from the cache incurs less cost than accessing the database directly. The maximum request rate supported is 117 and 125 requests/second in READ1 and READ2, respectively. The cache hit ratio is 91% in READ2. In summary, even without using rescue servers, the query result caching is useful in heavily loaded situations. However, the caching itself cannot remove the web server bottleneck.

Figure 6 shows the experimental results for test case READ3, READ4, and READ5, where a varying number of rescue servers are used. By using a sufficient number of rescue servers, the origin web server is no longer a bottleneck. We give the CPU utilization for the origin database server and the shared cache server used in READ5 in Figure 6(a), present the request rate sup-

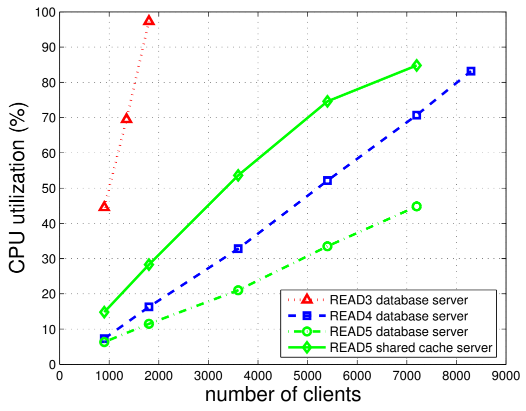


(a) The CPU utilization for the web server and database server

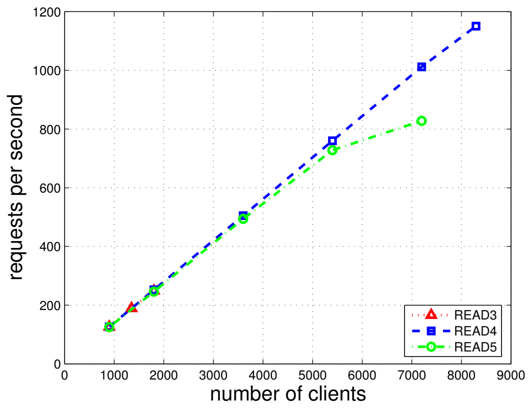


(b) The request rate supported

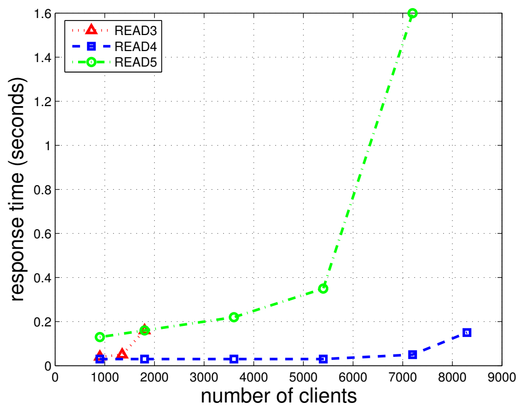
Figure 5: Experimental results for the RUBBoS read-only mix when rescue servers are not available. Test cases: READ1 – no rescue, no cache; and READ2 – no rescue, with a co-located cache.



(a) The CPU utilization for the origin database server and the shared cache server used in READ5



(b) The request rate supported



(c) The average response time

Figure 6: Experimental results for the RUBBoS read-only mix when rescue servers are available. Test cases: READ3 – with rescue, no cache; READ4 – with rescue, with a co-located cache; and READ5 – with rescue, with a shared cache.

ported in Figure 6(b), and display the average response time in Figure 6(c).

In test case READ3, the origin database server gets loaded quickly without using the query result caching. The maximum request rate supported is 249 requests/second, obtained with 1800 clients, and 4 rescue servers. Under this load, the origin database server CPU utilization is 97%.

In test case READ4, each web/application server uses a co-located query result cache, which greatly reduces the database workload. For 1800 clients, the origin web server uses 4 rescue servers, and the measured request rate is 252 requests/second. Under this load, the origin database server CPU utilization is only 16%, which is a huge reduction compared to the 97% CPU utilization in READ3. The maximum request rate supported is 1151 requests/second, obtained with 8295 clients, and 15 rescue servers. Under this load, the origin database server CPU utilization is 83%, and the origin web server cache hit ratio is 87%. For an experiment of this scale with 8295 clients, we use 38 machines: 21 for emulating clients, 15 as rescue servers, 1 as the origin web server, and 1 as the origin database server.

In test case READ5, all web/application servers use a shared query result cache server running on a separate machine, which further reduces the database workload. For example, for a load of 5400 clients, the origin database server CPU utilization is only 34%, compared to the 52% CPU utilization in READ4. However, the shared cache server itself becomes a bottleneck as the load increases because it gets loaded quickly than the origin database server does. The maximum request rate supported is 828 requests/second, obtained with 7200 clients, and 13 rescue servers. Under this load, the CPU utilization for the origin database server and the shared cache server is 45% and 85%, respectively, and the cache hit ratio at the shared cache server is 93%. From Figure 6(c), we observe that the average response time in READ5 is much longer than that in READ4. The reason is that using a shared cache incurs longer delays for handling client requests due to remote cache accesses. In general, a shared cache should be used with cautions since it is a single point of failure, it is a potential performance bottleneck, and it incurs longer delays. Note that it is possible to divide rescue servers into groups, and each rescue server group uses a separate shared cache, which has the potential to keep the shared cache in each group from overloaded, and reduce the database workload as much as possible. However, this method has administration overheads in forming groups and determining the right size of each group. As our goal is to build an autonomic system, we will not explore further along that direction in this paper.

4.5 Results for Submission Mix

Query result caching is used only by read queries; all write queries are submitted to the origin database server directly. In this section, we test our caching system with the RUBBoS submission mix, which has about 2% write queries. For simplicity, we use co-located query result caches in all our experiments for the submission mix. Depending on whether rescue servers are available, whether the query result caching is enabled, and whether cache invalidation is used, we have the following six test cases:

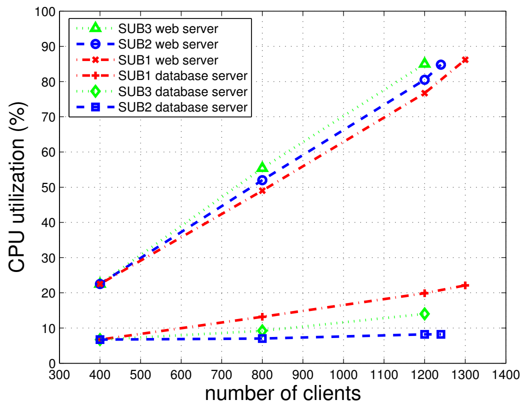
- *SUB1* – no rescue, no cache
- *SUB2* – no rescue, with cache, no invalidation
- *SUB3* – no rescue, with cache, with invalidation
- *SUB4* – with rescue, no cache
- *SUB5* – with rescue, with cache, no invalidation
- *SUB6* – with rescue, with cache, with invalidation

Table 2 summarizes the performance of our prototype system for the RUBBoS submission mix. Without using our rescue and caching services, a web server can only support a request rate of 180 requests/second. The request rate supported increases to 580 requests/second by using our rescue services only with 4 rescue servers, increases to 871 requests/second by using our rescue and caching without invalidation services together with 8 rescue servers, and increases to 701 requests/second by using our rescue and caching with invalidation services together with 6 rescue servers. Compared with SUB1, the improvement achieved is 322% in SUB4, 484% in SUB5, and 389% in SUB6. Compared with SUB4, the improvement achieved is 150% and 121% in SUB5 and SUB6, respectively.

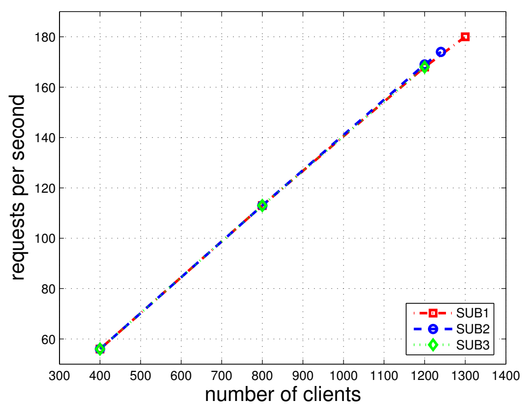
Figure 7 shows the experimental results for test case SUB1, SUB2, and SUB3, where rescue servers are not available. We give the CPU utilization for the web server and database server in Figure 7(a), and present the request rate supported in Figure 7(b). We observe that the web server CPU is the bottleneck. When the load is light with 400 clients, the caching is not activated. Thus, we have the same CPU utilization for SUB1, SUB2, and SUB3. When the load is heavy with 1200 clients, the caching is turned on. However, the performance is not improved by only using the caching because it reduces the database workload but increases the workload at the web server due to a low cache hit ratio, and the web server is the bottleneck. The maximum request rate supported is 180, 174, and 168 requests/second in SUB1, SUB2, and SUB3, respectively. Note that the number of clients supported is 1300 in SUB1, 1240 in SUB2, and 1200 in SUB3. The cache hit ratio is 76% in SUB2 and 39% in SUB3, which are much lower compared to the close to 90% cache hit ratio in the RUBBoS read-only mix.

Table 2: Performance summary for the RUBBoS submission mix: the maximum request rate supported, and the number of rescue servers used and the origin web server cache hit ratio at the peak rate. Test cases: SUB1 – no rescue, no cache; SUB2 – no rescue, with cache, no invalidation; SUB3 – no rescue, with cache, with invalidation; SUB4 – with rescue, no cache; SUB5 – with rescue, with cache, no invalidation; and SUB6 – with rescue, with cache, with invalidation.

Test case	Max rate supported (requests/second)	Number rescue servers used	Cache hit ratio	Performance compared with SUB1	Performance compared with SUB4
SUB1	180	no rescue	no cache	100%	
SUB2	174	no rescue	76%	97%	
SUB3	168	no rescue	39%	93%	
SUB4	580	4	no cache	322%	100%
SUB5	871	8	70%	484%	150%
SUB6	701	6	39%	389%	121%



(a) The CPU utilization for the web server and database server

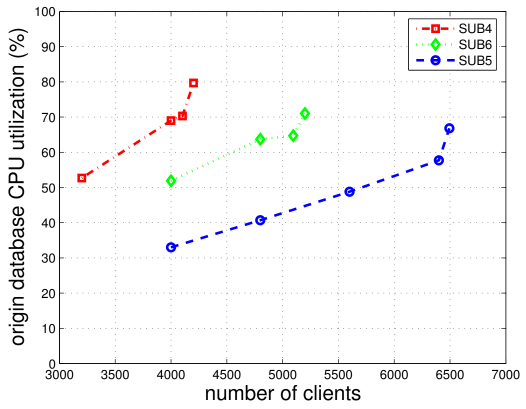


(b) The request rate supported

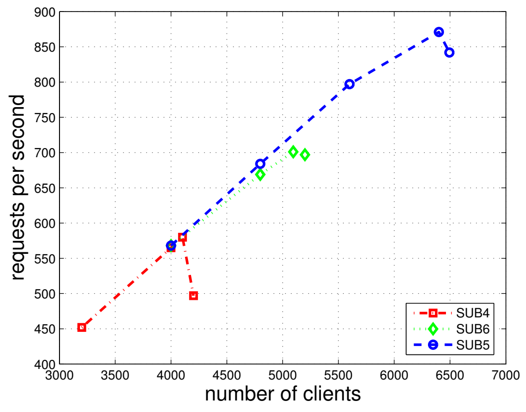
Figure 7: Experimental results for the RUBBoS submission mix when rescue servers are not available. Test cases: SUB1 – no rescue, no cache; SUB2 – no rescue, with cache, no invalidation; and SUB3 – no rescue, with cache, with invalidation.

Figure 8 shows the experimental results for test case SUB4, SUB5, and SUB6, where a varying number of rescue servers are used. By using a sufficient number of rescue servers, the origin web server is no longer a bottleneck. We give the origin database server CPU utilization in Figure 8(a), present the request rate supported in Figure 8(b), and display the rate of locks waited at the origin database server in Figure 8(c).

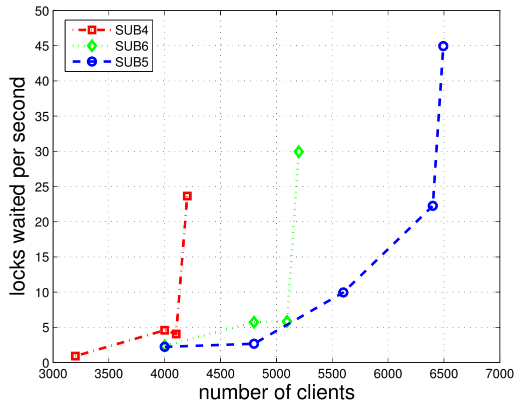
Based on Figure 8(a) and 8(b), we observe that the origin database server CPU utilization at the peak rate is only 58%, 65%, and 70% in SUB5, SUB6, and SUB4, respectively, which are much lower compared to the more than 80% CPU utilization in the RUBBoS read-only mix. This leads us to locate other bottlenecks in the system besides the database CPU utilization. In fact, for the RUBBoS submission mix, the rate of database locks waited becomes a performance bottleneck well before the database CPU gets overloaded. MySQL uses table locking in its default storage engine MyISAM to control concurrent read/write accesses to the same database table. Table locking allows many threads to read from a table at the same time; but a thread must get an exclusive write lock to write to a table. During an update to a database table, all other threads that need to access this particular table must wait until the update is done. In MySQL, the number of table access contentions caused by table locking is indicated by a status variable called *table_locks_waited*. As the number of clients increases, both the read access rate and the write access rate go up in the RUBBoS submission mix. As a result, the rate of locks waited increases. At certain point, the number of table access contentions increases abruptly, which causes the database performance degraded seriously. Using the query result caching reduces the read access rate to the origin database, which in turn reduces the table access contentions as well as the database workload.



(a) The origin database server CPU utilization



(b) The request rate supported



(c) The rate of locks waited at the origin database server

Figure 8: Experimental results for the RUBBoS submission mix when rescue servers are available. Test cases: SUB4 – with rescue, no cache; SUB5 – with rescue, with cache, no invalidation; and SUB6 – with rescue, with cache, with invalidation.

In test case SUB4, the query result caching is not used. As the load increases, the read access rate to the origin database increases quickly along with the write access rate. The maximum request rate supported is 580 requests/second, obtained with 4103 clients, and 4 rescue servers. Under this load, the origin database server has a 70% CPU utilization, and an average of 4 locks waited per second.

In test case SUB5, each web/application server uses a co-located query result cache without invalidation. The caching greatly reduces the read access rate to the origin database. The maximum request rate supported is 871 requests/second, obtained with 6400 clients, and 8 rescue servers. Under this load, the origin database server has a 58% CPU utilization, and an average of 22 locks waited per second. The cache hit ratio is 70%.

In test cast SUB6, each web/application server uses a co-located query result cache with invalidation. Although using invalidation lowers the cache hit ratio, the caching still reduces the read access rate to the origin database by a large percentage. The maximum request rate supported is 701 requests/second, obtained with 5096 clients, and 6 rescue servers. Under this load, the origin database server has a 65% CPU utilization, and an average of 6 locks waited per second. The cache hit ratio is 39%.

5 Conclusions

In this paper, we have described the DotSlash query result caching, which is a self-configuring, transparent, and on demand distributed caching system. Through our preliminary experimental results, we have demonstrated that our caching system is very effective for read-mostly databases. By using DotSlash rescue and caching services together, a web site can improve the maximum request rate supported by a factor of 5 to 10.

References

[1] Stephen Adler. The slashdot effect: An analysis of three Internet publications. http://ssadler.phy.bnl.gov/adler/SDE/SlashDotEffect.html.

[2] Akamai homepage. http://www.akamai.com/.

[3] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. DBProxy: A dynamic data cache for web applications. In *International Conference on Data Engineering (ICDE)*, Bangalore, India, March 2003.

[4] 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 *International Workshop on Web Content Caching and Distribution (WCW)*, Boulder, Colorado, August 2002.

[5] C. Amza, A. L. Cox, and W. Zwaenepoel. A comparative evaluation of transparent scaling techniques for dynamic

content servers. In *International Conference on Data Engineering (ICDE)*, Tokyo, Japan, April 2005.

[6] C. Bornhovd, M. Altinel, C. Mohan, H. Pirahesh, and B. Reinwald. Adaptive database caching with DBCache. *IEEE Data Engineering Bulletin*, 27(2):11–18, June 2004.

[7] V. Cardellini, M. Colajanni, and P.S. Yu. Geographic load balancing for scalable distributed web systems. In *International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, San Francisco, California, August 2000.

[8] E. Cecchet. C-JDBC: a middleware framework for database clustering. *IEEE Data Engineering Bulletin*, 27(2):16–26, June 2004.

[9] C. Y. Choi and Q. Luo. Template-based runtime invalidation for database-generated web contents. In *Asia Pacific Web Conference (APWeb)*, Hangzhou, China, April 2004.

[10] P. Larson, J. Goldstein, H. Guo, and J. Zhou. MTCache: Mid-tier database caching for SQL server. *IEEE Data Engineering Bulletin*, 27(2):35–40, June 2004.

[11] Libmemcache homepage. http://people.freebsd.org/˜seanc/libmemcache/.

[12] Memcached homepage. http://www.danga.com/memcached/.

[13] MySQL cluster. http://www.mysql.com/products/database/cluster/.

[14] C. Olston, A. Manjhi, C. Garrod, A Ailamaki, B. M. Maggs, and T. C. Mowry. A scalability service for dynamic web application. In *The Conference on Innovative Data Systems Research (CIDR)*, Asilomar, CA, January 2005.

[15] Oracle real application clusters (RAC). http://www.oracle.com/technology/products/database/clustering/index.html.

[16] C. Plattner and G. Alonso. Ganymed: Scalable replication for transactional web application. In *ACM/IFIP/USENIX International Middleware Conference*, Toronto, Canada, October 2004.

[17] M. Rabinovich, Z. Xiao, and A. Aggarwal. Computing on the edge: A platform for replicating Internet applications. In *International Workshop on Web Caching and Content Distribution (WCW)*, Hawthorne, NY, September 2003.

[18] RUBBoS: Rice university bulletin board system. http://www.cs.rice.edu/CS/Systems/DynaServer/RUBBoS/.

[19] S. Sivasubramanian, G. Alonso, G. Pierre, and M. van Steen. GlobeDB: Autonomic data replication for web applications. In *International World Wide Web Conference*, Chiba, Japan, May 2005.

[20] S. Sivasubramanian, G. Pierre, M. van Steen, and G. Alonso. GlobeCBC: Content-blind result caching for dynamic web applications. Submitted for publication, Vrije Universiteit, June 2005.

[21] Slashdot homepage. http://slashdot.org/.

[22] Transaction processing performance council. http://www.tpc.org/tpcw/.

[23] J. D. Zawodny and D. J. Balling. *High Performance MySQL: Optimization, Backups, Replication, and Load Balancing*. O'Reilly, 2004.

[24] W. Zhao and H. Schulzrinne. DotSlash: A self-configuring and scalable rescue system for handling web hotspots effectively. In *International Workshop on Web Caching and Content Distribution (WCW)*, Beijing, China, October 2004.

[25] W. Zhao and H. Schulzrinne. DotSlash: Handling web hotspots at dynamic content web sites. In *IEEE Global Internet Symposium*, Miami, Florida, March 2005.