Enabling On-demand Query Result Caching in DotSlash for Handling Web Hotspots Effectively[†]

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

Abstract— DotSlash is an automated web hotspot rescue system, which allows different web sites to form a mutual-aid community, and use spare capacity in the community to relieve web hotspots experienced by any individual site. DotSlash rescue services enable a web site to build an adaptive distributed web server system on the fly and replicate application programs dynamically, which relieve a spectrum of bottlenecks ranging from access network bandwidth to web servers and application servers. This paper presents DotSlash *Qcache* services that allow a web site to use on-demand distributed query result caching to greatly reduce the workload at read-mostly databases. The novelty of this work is that our query result caching is on demand and operated based on load conditions, which offers good data consistency for normal load and good scalability with relaxed data consistency under heavy load. DotSlash Qcache services complement DotSlash rescue services; together they provide a comprehensive solution to address different bottlenecks at multi-tier web sites. Experiments show that using DotSlash a web site can increase its maximum request rate supported by a factor of 10 for the RUBBoS read-only mix.

I. INTRODUCTION

Web hotspots are short-term dramatic load spikes, which can seriously degrade the service quality of affected web sites. Traditionally, a web site has a fixed set of resources, leaving it unable to handle a large load increase without significant overprovisioning. Since web hotspots are rare events, it is uneconomical to invest in more powerful infrastructure that is idle most of time. Small web sites often cannot afford a solution that employs server clusters, mirrored servers, or commercial content delivery networks (CDNs) [2]. However, a few such sites will invariably experience their "fifteen minutes of fame", typically by being mentioned on a high-volume news site such as Slashdot or CNN. Such flash crowds or "Slashdot effect" [1] will routinely cause small web sites to collapse.

To handle web hotspots cost-effectively, we have developed DotSlash, a self-configuring and scalable rescue system [43], [44], which allows different web sites to form a mutual-aid community, and use spare capacity in the community to relieve web hotspots experienced by any individual site [12]. DotSlash rescue services enable a web site to build an adaptive distributed web server system on the fly and replicate application programs dynamically, which relieve a spectrum of bottlenecks ranging from access network bandwidth to web servers and application servers.

To address the database server bottleneck, this paper presents DotSlash *Qcache* services, which allow a web site to use on-demand distributed query result caching to greatly reduce the workload at read-mostly databases. DotSlash Qcache services complement DotSlash rescue services; together they provide a comprehensive solution to address different bottlenecks at multi-tier web sites. Although our query result caching is not a new mechanism, its on-demand usage scenario in DotSlash is novel. Traditionally, when query result caching is enabled in a system, it stays on permanently. In contrast, our query result caching in DotSlash is on demand and operated based on load conditions: caching remains inactive as long as the load is normal, but is activated once the load is heavy. This approach offers good data consistency for normal load and good scalability with relaxed data consistency under heavy load.

The remainder of this paper is organized as follows. We discuss related work in Section II and introduce the DotSlash framework in Section III. We describe the design of DotSlash Qcache services in Section IV, evaluate our prototype system in Section V, and conclude in Section VI.

II. RELATED WORK

There have been a large body of work on improving the web site scalability under web hotspots. Most of them ([17], [39], [36], [26], [37], [15]) have focused on static content. To improve the application server scalability, application programs or components [28], [2], [44] can be offloaded from the origin server. Recently, Olston et al. [24] proposed a scalability service for databases using multicast-based consistency management. Although their system aims at broader web applications, its scalability gain under heavy load is unclear. Moreover, their service is not transparent since clients need to connect to their proxy servers in order to use the service. In contrast, our system targets read-mostly databases and scales well under dramatic load spikes. Also, our system is transparent to web users. 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. The performance evaluation of our prototype system confirmed that query result caching works well for read-mostly databases.

Caching is very effective for web content distributions. Web caching can cache HTML pages or page fragments at proxies [13], web servers [11], application servers [19], [6], and edge servers [2]. Database caching [3], [8], [20], [33] can cache data from back-end databases at caches closer to application servers. While caching in existing systems is active in all cases, our query result caching is activated only under heavy load, which minimizes the effect of caching on consistency while improving the system scalability.

Replication is a widely used mechanism for database scalability. Ganymed [27] separates update transactions from read-only transactions, and routes updates to a main database server and read queries to read-only database copies. GlobeDB [32] uses partially replicated databases based on data partition to reduce update traffic. Our current prototype uses a single back-end database server, which can be extended to support distributed database servers by incorporating database replication into our system.

Database clustering is a mechanism to pool database servers together so as to provide high availability and performance. While Oracle RAC (Real Application Clusters) [25] uses a shared cache architecture, MySQL Cluster [23] is built on a shared-nothing architecture. Clustered JDBC [10] 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 high availability and performance. In contrast, DotSlash is a solution at the web/application server tier for dynamic scalability. Thus, our system and database clustering are orthogonal, and they can be used together at dynamic content web sites.

III. THE DOTSLASH FRAMEWORK

We provide a brief overview of the DotSlash framework here; a more complete description is given in [43], [44].

A. DotSlash Rescue Model

In DotSlash, a web server is in one of the following three states at any time: in *SOS state* if it gets rescue services from others, in *rescue state* if it provides rescue services to others, and in *normal state* if it does neither. These three states are mutually exclusive: a server is not allowed to get rescue services and provide rescue services at the same time. This rule avoids complex rescue scenarios (e.g., a rescue loop), and keeps DotSlash simple and robust without compromising scalability.

B. DotSlash Usage Models

DotSlash allows different web sites to form a mutual-aid community, and use spare capacity in the community to relieve web hotspots experienced by any individual site. A web server joins a mutual-aid community by registering itself with a DotSlash service registry in the community, and contributing its spare capacity to the community. Under heavy load, a participating web server discovers and uses spare capacity at other web servers in the community via

DotSlash rescue services. A mutual-aid community has a set of DotSlash service registries, which use mSLP [45] to replicate service registration information from each other automatically. It is beneficial for a mutual-aid community to set up a DNS domain, which allows DotSlash service registries to be discovered via DNS SRV [46], eliminating manual configuration for registry discovery.

We consider three types of mutual-aid communities, namely open communities, closed communities, and flood-insurance closed communities.

- An open mutual-aid community is intended for a cooperative environment, e.g., within an organization. Such a community is simple and easy to use, but it does not provide security measures against attacks or abuse.
- In a closed mutual-aid community, only authorized web sites can provide or get rescue services, which protects against abuse by origin servers or bogus rescue servers. Based on the SLP (Service Location Protocol) [18] security features and public key cryptography, authentication is performed in accepting service registrations, choosing rescue server candidates, and accepting rescue requests. A community authority maintains a list of public keys for all authorized participating servers. When a web site is authorized to join a closed community, it creates a pair of public/private keys, and registers its public key with the community authority. For service registrations, a web server signs its registration using its private key, and a DotSlash service registry checks the public key specified in each registration and uses the key to verify the corresponding registration. For rescue server discovery, upon receiving a list of URL entries from a DotSlash service registry, a web server checks the public key specified in each URL entry and uses the key to verify the corresponding URL entry. For rescue requests, an origin server signs its rescue request using its private key, and a rescue server accepts a rescue request only if it is from a verified member of the community.
- To increase the incentive for providing DotSlash rescue services and reduce abuse, a flood-insurance closed community can be used. In such a community, an authorized web site needs to pay the community authority an insurance premium to obtain tokens for using rescue services. For example, $3 might be needed for 10 tokens [14]. The community authority maintains a list of all valid tokens in the community, and associates each token with a participating web server. To secure a rescue relationship, the origin server needs to transfer one token to the rescue server. When an origin server has used up all its tokens, it needs to buy new tokens for using rescue services. On the other hand, a rescue server can accumulate tokens for its own rescue needs or sell its tokens.

C. DotSlash Rescue Services

In DotSlash, a web site not only has a fixed set of *origin servers*, but also has a changing set of *rescue servers* drafted from other sites. DotSlash rescue services allow an origin

server to draft and release rescue servers fully automatically based on its load conditions. An origin web server discovers suitable rescue servers via wide-area service location, either among peer servers or from a dedicated pool of rescue servers, allocates them for temporary use, and redirects client requests to them. The rescue process is completely self-managing and transparent to clients.

DotSlash uses DNS round robin as the first level crude load distribution, and uses HTTP redirect as the second level fine-grained load balancing. When a rescue relationship is set up between two web servers, the rescue server assigns a unique virtual host name to the origin server, which is used by the origin server in its HTTP redirects to the rescue server. Also, the origin server adds the rescue server's IP address to its local DNS for round robin.

In DotSlash, a rescue server can serve the content of its origin server on the fly. A rescue server works as a reverse caching proxy for its origin server. It maps client requests either to its own content or to the content of its origin server. In addition to caching static content from the origin server, a rescue server replicates application programs dynamically from the origin server, and accesses databases at the origin server.

D. Adaptive Load Control

DotSlash uses two configurable parameters, lower threshold ρ^l and upper threshold ρ^u, to define three load regions: light load region $[0, \rho^l)$, desired load region $[\rho^l, \rho^u]$, and heavy load region $(\rho^u, 100\%]$. DotSlash measures utilization of different resources, e.g., our current prototype system measures network and CPU utilization. A web server's load region is determined as follows: the server is in the heavy load region if *any* resource is heavily loaded, in the light load region if *all* resources are lightly loaded, and in the desired load region otherwise.

In DotSlash, a web server uses adaptive load control to tune its resource utilization. More specifically, to keep a resource utilization ρ in the desired load region, a web server triggers overload control actions if $\rho > \rho^u$, and triggers under-load control actions if $\rho < \rho^l$. For example, an origin server can reduce its load level by allocating new rescue servers and increasing its redirect probability. On the other hand, a rescue server can raise its load level by accepting new rescue requests and increasing the allowed redirect rate for its origin servers.

IV. The Design of DotSlash Qcache Services

A. Motivations

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 a certain 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 [27], [32], [3], [8], [20], [10]. However, existing systems often involve
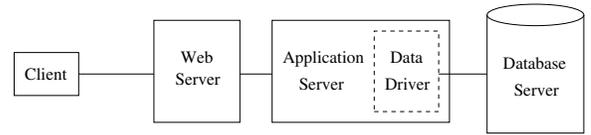

Fig. 1. DotSlash application model

manual configuration for different participating parties to work together, which makes these systems difficult to be deployed dynamically to new servers. Based on the existing research work and our DotSlash framework, we aim to develop a plug-in subsystem for DotSlash, which allows a dynamic content web site to improve its database scalability dynamically and in a fully automated way.

B. Design Goals

We have three major design goals: dynamic scalability, self-configuration, and transparency. First, we aim to provide a mechanism that can be deployed to new servers 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 strive 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 web users and applications. Without the need to change existing applications and user browsers, our system is easy to deploy.

C. 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 content management systems (CMS), blogs, and web forums. Compared to replication, caching is easier to deploy dynamically, and incurs lower overhead at the origin server because cached objects are 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], [8], [20] is more efficient in that it can answer arbitrary queries on cached tables, query result caching [33] is much simpler and can save expensive computations on cache hits. Thus, we chose to use query result caching in DotSlash.

D. 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
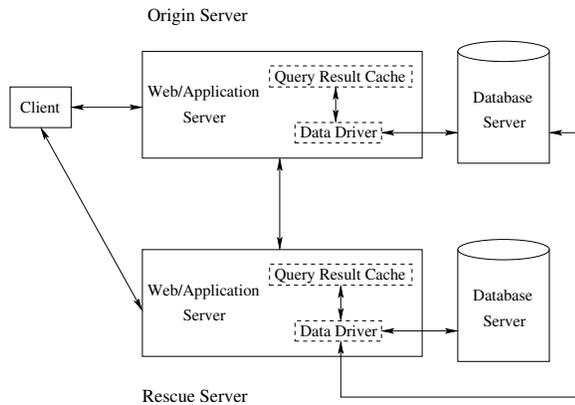
Fig. 2. Enabling query result caching in DotSlash

Configuration	Bottlenecks Addressed	Used By
Dots_Apache	Network and web server	All sites
Dots_Apache + Dots_PHP	Network, web server, and application server	Dynamic sites
Dots_Apache + Dots_PHP + Dots_MySQL	Network, web server, application server, and database server	Dynamic sites

of the application server. The data driver provides a standard API for web applications to store and retrieve data in back-end databases. In our prototype system, we use the common LAMP (Linux, Apache, MySQL, and PHP) configuration, where the PHP module resides in the Apache web server.

E. DotSlash Qcache Services

DotSlash Qcache services allow an origin server and its rescue servers to use on-demand query result caching to reduce the database workload at the origin server. Since the data driver (as shown in Figure 1) intercepts all database queries, we enhance it with query result caching without changing the application API and database interface. In our prototype system, we extend the original PHP data driver for MySQL databases with a query result cache. Figure 2 illustrates how to enable query result caching in DotSlash. Note that a client request can be redirected from the origin server to the rescue server via either DNS round robin or HTTP redirect. Also note that a rescue server may need to access remote databases at the origin server in addition to its local databases. We will discuss DotSlash data driver control in details in Section IV-I.

F. DotSlash Configurations

Our open-source prototype implementation of DotSlash [41] has three major components: Dots_Apache, Dots_PHP, and Dots_MySQL. *Dots_Apache* is an Apache module that supports basic DotSlash functions including workload monitoring, rescue server discovery, rescue relationship management, request redirection, dynamic virtual hosting, and dynamic DNS update. *Dots_PHP* is an extension for the PHP module of Apache that supports replicating PHP scripts dynamically. *Dots_MySQL* is a caching-enhanced PHP data driver for MySQL databases that supports caching database query results on demand. DotSlash can be used in three different configurations as shown in Table I, where Dots_Apache and Dots_PHP provide DotSlash rescue services, and Dots_MySQL provides DotSlash Qcache services.

G. On-demand Query Result Caching

We employ on-demand query result caching in DotSlash: caching remains inactive as long as the load is normal, but is activated once the load is heavy. This novel approach offers good data consistency for normal load and good scalability with relaxed data consistency under heavy load.

The on-demand query result caching control is based on two factors, namely the web server's DotSlash state (normal, SOS, or rescue) and load region (desired load, heavy load, or light load). We show the control of our on-demand query result caching in Figure 3. Caching is activated if a web server is in the SOS state (i.e., an origin server), or if a web server is in the rescue state (i.e., a rescue server), or if a web server is in the normal state and its load is above the upper threshold. On the other hand, caching is de-activated when an origin server switches from the SOS state to the normal state, or when a rescue server switches from the rescue state to the normal state, or when a web server is in the normal state and its load is below the lower threshold.

H. Other Query Result Caching Features

Self-configuration is an important feature of our query result caching. When an origin server sets up its rescue servers, it passes the query result caching control parameters to its rescue servers. Although our current prototype system only uses caching TTL, other caching control parameters (such as validation flag) can be added to our system as needed. By doing so, a rescue server can manage cached objects based on the instructions from the origin server. In this way, an origin server can set up a distributed query result caching system on the fly using one set of control parameters.

Distributed query result caching is a natural feature of our system. By default, each web/application server has its own, co-located query result cache. An origin server can obtain more query result caches as it drafts more rescue servers. Using co-located query result caches eliminates dedicated cache servers, which is well-suited for DotSlash in terms of resource utilization efficiency because our query result caching is on demand, and it is off most of time. Note that our system can use a dedicated query result cache server which is shared among an origin server and its rescue servers, or among a subset of rescue servers. Doing so can reduce the workload at the origin database server. However, a shared cache may become a potential performance bottleneck, and accessing a remote cache incurs longer delays (see Section V-D for experimental results).

Our query result caching is transparent to web users and applications. Without the need to change client-side web
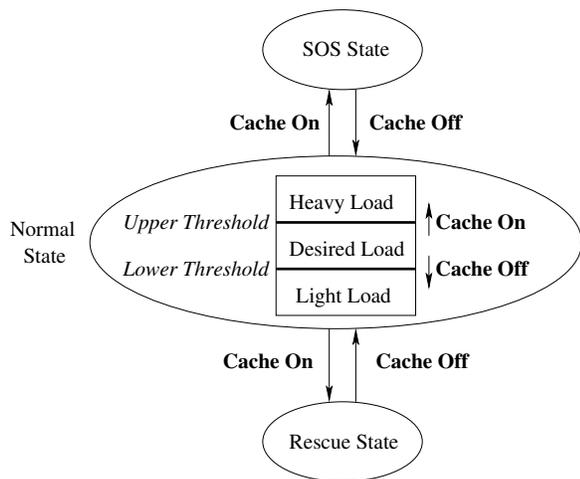
Fig. 3. DotSlash on-demand query result caching, where caching is activated (cache on) or de-activated (cache off) based on the web server's DotSlash state (normal, SOS, or rescue) and load region (desired load, heavy load, or light load).

browsers and server-side application programs, our system is easy to deploy. Furthermore, we provide a way for web users to bypass our query result caching. Our current prototype system uses the HTTP *Cache-Control* header for this purpose as follows. If there is *no-cache* or *max-age=0* in the HTTP Cache-Control header of a client request, DotSlash will handle that request without using query result caching.

I. Data Driver Design

To support DotSlash Qcache services, we enhance the data driver for databases with a query result cache, and add a control scheme to the data driver to handle database queries based on our needs.

We made two design choices for our caching-enhanced data driver. First, rescue servers only handle read-only database queries. As a result, all write database queries are handled by the origin web server. This design choice is mainly for security reasons because an origin server is unlikely to allow rescue servers to update its databases. Secondly, under heavy load we turn off write queries temporarily for regular users, but still allow site administrators (or a small group of premium users) to perform necessary updates. This design choice is mainly for scalability considerations because database systems often use locking (e.g., table locking in MySQL) to control concurrent read/write accesses to the same database table, and a large number of read/write contentions can seriously degrade the database performance (see Section V-E for experimental results).

Our caching-enhanced data driver works as a regular data driver for normal load, but it trades data consistency for scalability under heavy load. More specifically, our caching-enhanced data driver provides two levels of data consistency guarantees once caching is activated. The first consistency level is intended for site administrators who can continue to perform both read and write queries, and get an up-to-date

view of database states without using query result caching. The second consistency level is intended for regular users who can only perform read-only queries, and get a delayed view of database states by using query result caching. We use an application-specific caching TTL to bound the staleness of cached objects (see Section V-C for details).

The design of our caching-enhanced data driver targets hotspot rescue for read-mostly databases, which are common for content management systems (CMS), blogs, and web forums. This design is not intended to be applicable to all web applications, e.g., it should not be used by e-commerce sites (modeled by benchmarks such as RUBiS [30] and TPC-W [38]) that have frequent updates and strong consistency requirements.

Our caching-enhanced data driver handles database queries based on three factors, namely the web server's query result caching state, the client request HTTP Cache-Control header, and the client request type. Our query result caching is a per-server state, which is on or off as illustrated in Figure 3. A client request can bypass our query result caching using the HTTP Cache-Control header as described in Section IV-G. A rescue server distinguishes two types of client requests, *regular* and *rescue*, based on the request's HTTP *Host* header. If the HTTP Host header uses an origin server name such as *www.origin.com*, or an assigned virtual host name such as *vh1.www.rescue.com*, then the request is treated as a rescue request, otherwise as a regular request.

We show the control of our caching-enhanced data driver in Table II. There are four cases.

- For case 1, query result caching is off. Then the data driver handles all database queries normally by forwarding the queries directly to the database.
- For case 2, query result caching is on and caching is not bypassed. Then the data driver turns off all write queries (e.g., a SQL *insert*, *update*, or *delete* statement), and returns an error message, such as "Due to heavy load, write operations to databases at web site *http://www.origin.com* have been temporarily turned off", to the applications. At the same time, the data driver handles all read-only queries (e.g., a SQL *select* statement) as follows. It first checks a read-only query against the query result cache. If there is a cache hit, the data driver gets the query result

TABLE II

DOTSLASH CACHING-ENHANCED DATA DRIVER, WHICH HANDLES DATABASE QUERIES BASED ON THE WEB SERVER'S QUERY RESULT CACHING STATE (ON OR OFF), THE CLIENT REQUEST HTTP CACHE-CONTROL HEADER (BYPASS CACHING OR NOT), AND THE CLIENT REQUEST TYPE (RESCUE OR REGULAR).

Case	Cache On	Bypass Caching	Rescue Request	Database Write	Database Read
1	no	–	–	normal	normal
2	yes	no	–	turn off	cache+DB
3	yes	yes	no	normal	DB+cache
4	yes	yes	yes	redirect	redirect

from the cache and returns the result to the application immediately. In case of a cache miss, the data driver submits the query to the corresponding database, which can be a local database or a remote database at the origin server; then the data driver obtains the query result from the database, saves it to the query result cache, and returns it to the application.

- For case 3, query result caching is on, caching is bypassed, and the request is a regular request. Then the data driver forwards all database queries directly to the database. For a read-only query, the data driver saves the query result to the cache before returning it to the application.

- For case 4, query result caching is on, caching is bypassed, and the request is a rescue request. Then the request is redirected back to the origin web server via HTTP redirect, which ensures that a client request that needs to bypass caching can always be handled by the origin web server. For this purpose, an origin server does not apply HTTP redirect to client requests that need to bypass caching. However, client requests could be distributed to rescue servers due to the origin server's DNS round robin. This is why we need to use HTTP redirect in case 4. Note that a rescue server uses the origin server's IP address in its HTTP redirects to bypass the origin server's DNS round robin mechanism.

J. Query Result Cache Design

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

The data driver uses the query result cache via two interface functions: *check_in* and *check_out*. The *check_in* function takes the query string, query result, and caching TTL as input parameters, serializes the query result into a byte stream, and saves it to the caching storage engine. The *check_out* function takes the query string as the input parameter and retrieves the query result. For a cache hit, the *check_out* function deserializes the query result byte stream into the original query result data structure and returns a pointer to the result structure. In case of a cache miss, the *check_out* function returns a NULL pointer.

Both disk and memory can be used as our caching storage engine. Due to performance considerations, we choose to use a memory storage engine called *memcached* [22], which employs 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, we use an open-source C library *libmemcache* [21] to access the cache. In the *check_in* function, we first use the ELF hash algorithm [7] to map the query string into a cache key, and then store the query string and the query result as the cache value, using the caching TTL as the expiration time. Note that different query strings might be mapped into the same cache key with a small probability, which is less than 1% in our experiments. To handle this type

of hash conflicts, we let 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 cache key. If a cached object is found for the key, we check whether the stored query string matches the input query string. If so, it is a cache hit; otherwise, it is a cache miss.

V. EVALUATION

We use the maximum request rate supported by a web site as the major performance metric. We evaluate DotSlash rescue services and Qcache services individually as well as together through experiments, and examine the system performance improvement in different situations.

A. Benchmark Description

We evaluate our prototype system using the RUBBoS bulletin board benchmark [29], which is modeled after an online news forum like Slashdot [34].

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 consists of 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 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 [38]. If the request rate to be generated is high, multiple RUBBoS clients are used, each running on a separate machine. We use 7 seconds [9] as the timeout value for getting the response for a request. If more than 10% [9] of issued requests time out, the web server is considered as being overloaded.

RUBBoS has two major workload mixes, read-only and submission. The read-only mix invokes browse scripts, story/comment view scripts, and search scripts with a probability of 2/3, 1/6, and 1/6, respectively. The submission mix invokes 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 causes a higher workload at the database than its submission mix. This is due to two reasons. First, each pre-generated story has 20 to 50 comments, whereas a newly submitted story has only a few comments or no comments at all. Secondly, each emulated RUBBoS client always starts with, and often returns to the *Stories Of The Day* page, which has the most recent 10 stories.

B. Experimental Setup

In our previous work [43], [44], we have evaluated Dot-Slash rescue services across wide area networks and in our local area network. The goal of this paper is to address the database server CPU bottleneck by enabling on-demand query result caching in DotSlash. For the convenience of managing experiments and without loss of generality, all experiments described in this paper are performed in our local area network.

We use a cluster of Linux machines connected via 100 Mb/s fast Ethernet. These machines have three different configurations. Each web/application server 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 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. Each client emulator machine 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.

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 Dots_Apache. Also, the PHP module includes our Dots_PHP and Dots_MySQL. By default, memcached works as a co-located cache server with a storage space limit of 200 MB. When it works as a shared cache server, memcached has a storage space limit of 1 GB. The database server runs MySQL 4.0.18 using the default MyISAM storage engine. To enhance MyISAM performance under heavy updates, we configure MySQL with *delay_key_write=all* to delay writing index data to disk [40]. To support a large number of concurrent connections, we configure MySQL with *open_files_limit=65535* and *max_connections=8192*. We use our *dot-slash.net* domain for dynamic DNS updates, and use the enhanced Service Location Protocol [42] for rescue server discovery.

C. Caching TTL

In DotSlash, each web server has a configurable parameter called caching TTL, which is used to control how long query results can be cached. This parameter is passed from an origin server to all its rescue servers; and a rescue server caches query results from an origin server based on the origin server's caching TTL parameter.

In general, the caching TTL for query results is an application-dependent parameter since different applications may need to use different caching TTLs based on their data consistency requirements. For RUBBoS, we use 60 seconds as the caching TTL because it is good enough to bound the staleness of cached objects in RUBBoS.

There is a trade-off in choosing the caching TTL parameter: decreasing this parameter will improve data consistency, whereas increasing this parameter will improve caching performance. Figure 4 shows the relationship between the caching TTL and query result cache hit ratio in a set of 10-minute experiments for the RUBBoS read-only mix. We observe that the cache hit ratio increases as the caching TTL increases.
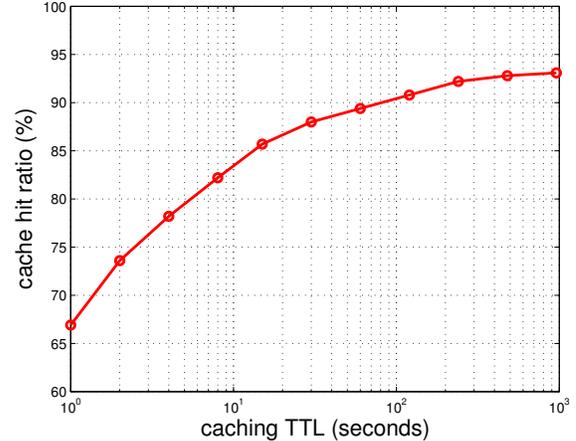

Fig. 4. The relationship between the caching TTL and query result cache hit ratio in a set of 10-minute experiments for the RUBBoS read-only mix

For our chosen caching TTL 60 seconds, the cache hit ratio is 89.4%.

D. Results for Read-only Mix

We first test our prototype system using the RUBBoS read-only mix. Depending on whether rescue servers are available, whether 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 five test cases for the read-only mix as follows.

- $READ$: no rescue, no cache.
- $READ_c$: no rescue, with a co-located cache.
- $READ_r$: with rescue, no cache.
- $READ_{r,c}$: with rescue, with a co-located cache.
- $READ_{r,sc}$: with rescue, with a shared cache.

Table III summarizes our experimental results for the RUBBoS read-only mix. Without using DotSlash rescue and Qcache services, a web server can only support a request rate of 117 requests/second. The request rate supported increases to 249 requests/second by only using DotSlash rescue services with 4 rescue servers, and increases to 1151 requests/second by using DotSlash rescue and Qcache services together with 15 rescue servers. Compared to $READ$, $READ_r$ and $READ_{r,c}$ achieve an improvement of 213% and 984%, respectively. Compared to $READ_r$, $READ_{r,c}$ achieves an improvement of 462%. Note that the reason for using 4 and 15 rescue servers in $READ_r$ and $READ_{r,c}$, respectively, is that the origin server and rescue servers have a CPU utilization in the desired load region $[\rho^l, \rho^u]$ (see Section IV-G), which is configured as $[45\%, 70\%]$ in our experiments. Next, we give details for each test case using the read-only mix.

Figure 5 shows the experimental results for $READ$ and $READ_c$, 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.

TABLE III
SUMMARY OF EXPERIMENTAL RESULTS FOR RUBBOS READ-ONLY MIX

Test Case	Max Rate (reqs/s)	Compared to $READ$	Compared to $READ_r$	Rescue Servers
$READ$	117	100%		
$READ_c$	125	107%		
$READ_r$	249	213%	100%	4
$READ_{r,c}$	1151	984%	462%	15
$READ_{r,sc}$	828	708%	333%	13

When the load is light with 300 clients, caching is not activated. Thus, we have the same CPU utilization for $READ$ and $READ_c$. When the load is heavy with 840 clients, caching is turned on, and we can observe a big difference in CPU utilization. The database server CPU utilization is 41% in $READ$, but is only 6% in $READ_c$, meaning that 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 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 $READ$ and $READ_c$, respectively. The cache hit ratio is 91% in $READ_c$. In summary, even without using rescue servers, query result caching is useful under heavy load. However, caching itself cannot remove the web server bottleneck.

Figure 6 shows the experimental results for $READ_r$, $READ_{r,c}$, and $READ_{r,sc}$, 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 $READ_{r,sc}$ in Figure 6(a), present the request rate supported in Figure 6(b), and display the average response time in Figure 6(c).
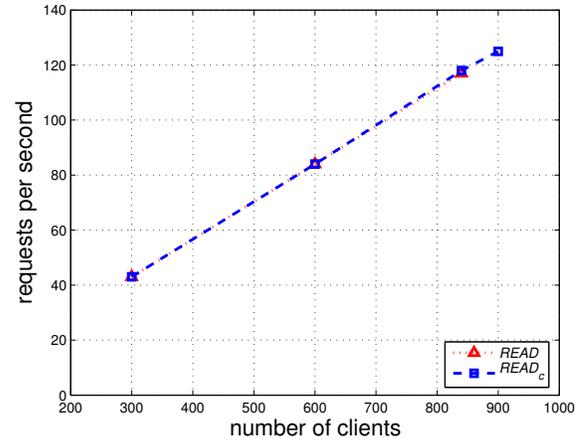
For $READ_r$, the origin database server gets overloaded quickly without using query result caching. The maximum request rate supported is 249 requests/second, obtained using 1800 clients and 4 rescue servers. Under this load, the origin database server CPU utilization is 97%.

For $READ_{r,c}$, 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 97% CPU utilization in $READ_r$. The maximum request rate supported is 1151 requests/second, obtained using 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.

For $READ_{r,sc}$, all web/application servers use a shared query result cache server running on a separate machine, which can further reduce the database workload. For 5400
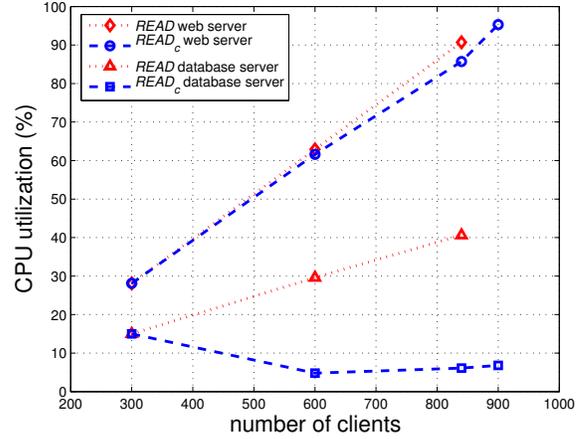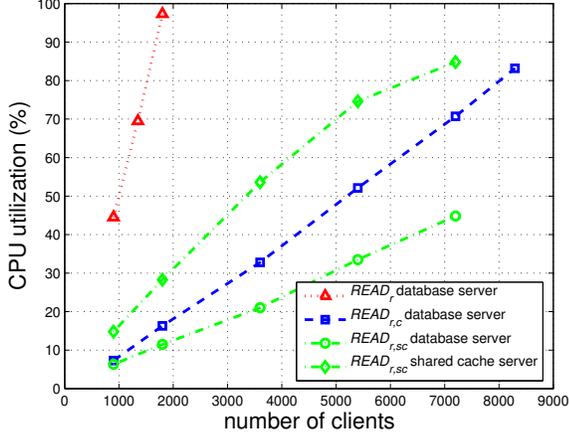

(a) The CPU utilization for the web server and database server

(b) The request rate supported

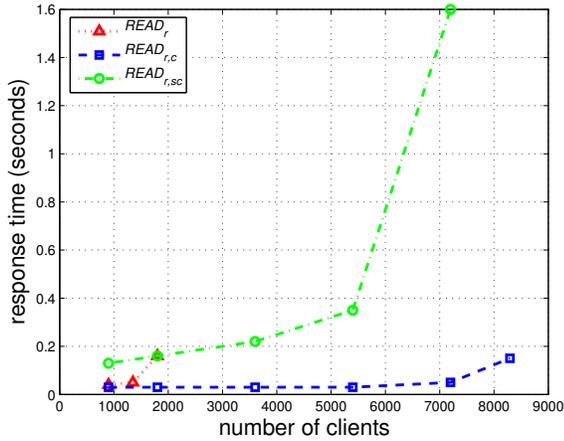Fig. 5. Experimental results for the RUBBoS read-only mix when rescue servers are not available

clients, the origin database server CPU utilization is only 34%, compared to 52% CPU utilization in $READ_{r,c}$. However, the shared cache server itself becomes a bottleneck since it gets loaded more quickly than the origin database server does. The maximum request rate supported is 828 requests/second, obtained using 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%. In Figure 6(c), the average response time in $READ_{r,sc}$ is much longer than that in $READ_{r,c}$ since using a shared cache incurs longer delays for remote cache accesses. In general, a shared cache is a single point of failure and a potential performance bottleneck, and it incurs longer delays. Note that it is possible to divide rescue servers into groups, and use a separate shared cache in each group, which has the potential to keep the shared cache in each group from being overloaded, and reduce

(a) The CPU utilization for the origin database server and the shared cache server used in $READ_{r,sc}$
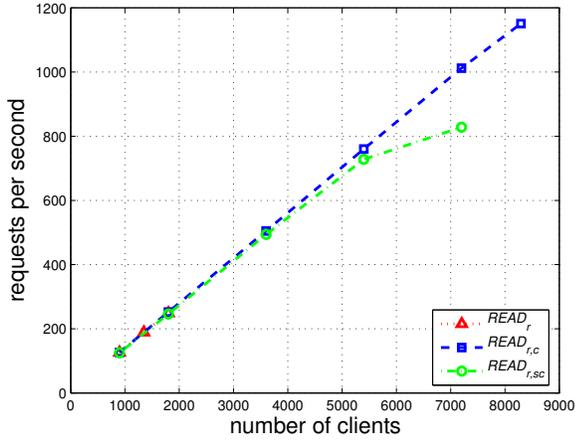
(b) The request rate supported

(c) The average response time

Fig. 6. Experimental results for the RUBBoS read-only mix when rescue servers are available

Test Case	Max Rate (reqs/s)	Compared to SUB	Compared to SUB_r	Rescue Servers
SUB	180	100%		
SUB_c	174	97%		
SUB_r	580	322%	100%	4
$SUB_{r,c}$	871	484%	150%	8

the database workload as much as possible. However, this method incurs administrative overhead in forming groups and determining the right size of each group. As our goal is to build an autonomic system, we will not explore this approach in more depth.

In summary, by using DotSlash rescue and Qcache services together, a web site can improve its maximum request rate supported by a factor of 10 for the RUBBoS read-only mix. Although the major performance gain comes from the Qcache services, the rescue services are the fundamental framework upon which the Qcache services are built. Moreover, the efficiency of the Qcache services depends on the cache hit ratio.
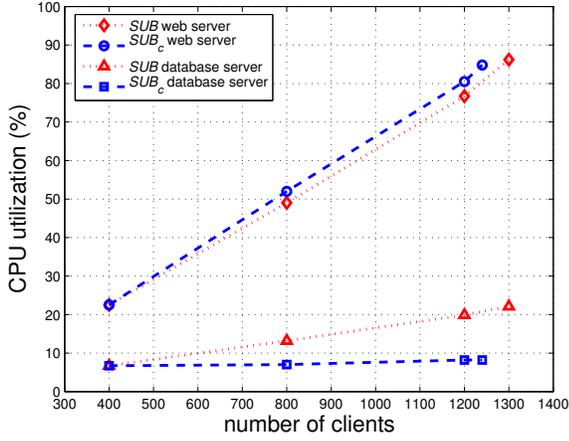
E. Results for Submission Mix

Based on Section IV-I, DotSlash turns off database write queries temporarily for regular users under heavy load. We disable this feature in testing our prototype system against the RUBBoS submission mix, which has about 2% write queries. We choose to do so for two reasons. First, turning off all write queries will convert the submission mix into a read-only mix, which we have evaluated in the last section. Secondly, allowing site administrators to perform necessary updates in our system is roughly equivalent to having a small percentage of write queries in the submission mix. Depending on whether rescue servers are available and whether query result caching is enabled, we have four test cases for the submission mix as follows.
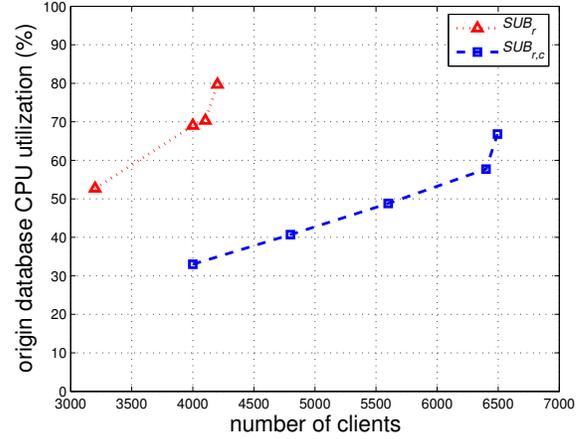
- SUB: no rescue, no cache.
- SUB_c: no rescue, with cache.
- SUB_r: with rescue, no cache.
- $SUB_{r,c}$: with rescue, with cache.

Table IV summarizes our experimental results for the RUBBoS submission mix. Without using DotSlash rescue and Qcache services, a web server can only support a request rate of 180 requests/second. The request rate supported increases to 580 requests/second by only using DotSlash rescue services with 4 rescue servers, and increases to 871 requests/second by using DotSlash rescue and Qcache services together with 8 rescue servers. Compared to SUB, SUB_r and $SUB_{r,c}$ achieve an improvement of 322% and 484%, respectively. Compared to SUB_r, $SUB_{r,c}$ achieves an improvement of 150%. Next, we give details for each test case using the submission mix.
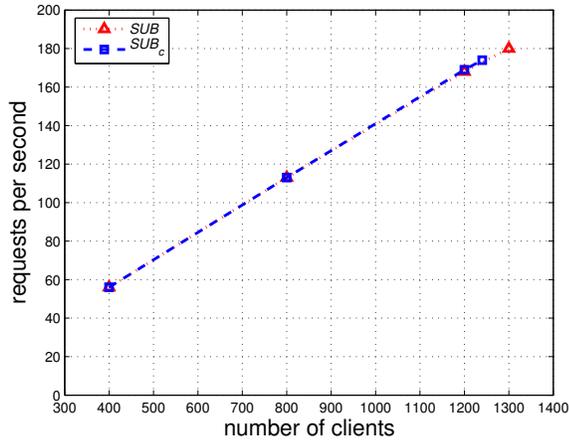
Figure 7 shows the experimental results for SUB and SUB_c, where rescue servers are not available. We give the
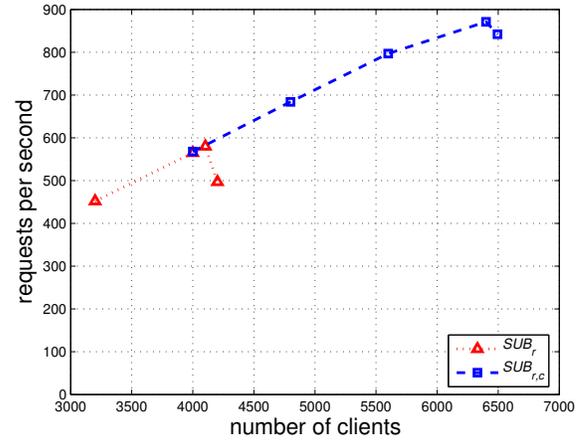
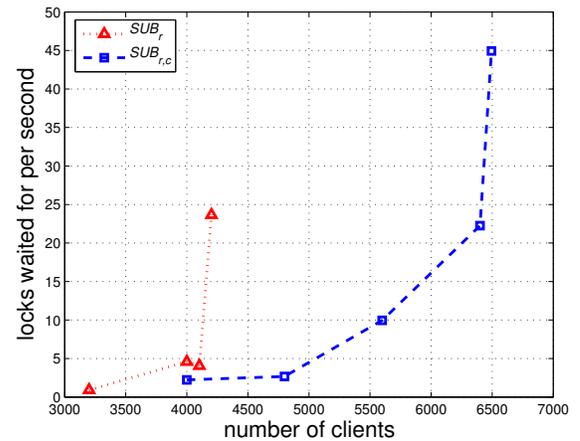(a) The CPU utilization for the web server and database server

(b) The request rate supported

Fig. 7. Experimental results for the RUBBoS submission mix when rescue servers are not available

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, caching is not activated. Thus, we have the same CPU utilization for SUB and SUB_c. When the load is heavy with 1200 clients, caching is turned on. However, the performance is not improved by only using query result caching because it reduces the database workload but increases the web server workload due to a low cache hit ratio, and the web server is the bottleneck. The maximum request rate supported is 180 and 174 requests/second in SUB and SUB_c, respectively. Note that the number of clients supported is 1300 in SUB and 1240 in SUB_c. The cache hit ratio is 76% in SUB_c, which is much lower compared to around 90% cache hit ratio in the RUBBoS read-only mix.

Figure 8 shows the experimental results for SUB_r and $SUB_{r,c}$, where a varying number of rescue servers are used.

(a) The origin database server CPU utilization

(b) The request rate supported

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

Fig. 8. Experimental results for the RUBBoS submission mix when rescue servers are available

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 for 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% and 70% in $SUB_{r,c}$ and SUB_r, respectively, which are much lower compared to over 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 for 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*. In the RUBBoS submission mix, both read and write access rates go up as the number of clients increases. As a result, the rate of locks waited for increases. At a certain point, the number of table access contentions increases abruptly, which causes the database performance to degrade seriously. Using query result caching reduces the read access rate to the origin database, which in turn reduces the number of table access contentions as well as the database workload.

For SUB_r, 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 using 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 for per second.

For $SUB_{r,c}$, each web/application server uses a co-located query result cache, which greatly reduces the read access rate to the origin database. The maximum request rate supported is 871 requests/second, obtained using 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 for per second. The origin web server cache hit ratio is 70%.

In summary, by using DotSlash rescue and Qcache services together, a web site can improve its maximum request rate supported by a factor of 5 for the RUBBoS submission mix, where the major performance gain comes from the rescue services. Comparing this performance improvement with that of the read-only mix, we observe a difference of a factor of 2. The main reason is that write queries not only reduce the cache hit ratio, but also increase database access contentions. To allow single-server databases to survive web hotspots, DotSlash turns off write queries temporarily for regular users under heavy load.

VI. CONCLUSIONS

In this paper, we have described how to enable on-demand distributed query result caching in DotSlash for handling web hotspots effectively. We have discussed the DotSlash system architecture and the design of DotSlash Qcache services, and evaluated DotSlash rescue and Qcache services individually as well as together. Through our experimental results, we have demonstrated that using DotSlash rescue and Qcache services together is very effective for hotspot rescue at dynamic content web sites with read-mostly databases. For example, using DotSlash a web site can increase its maximum request rate supported by a factor of 10 for the RUBBoS read-only mix.

For future work, we plan to investigate how the hotspot rescue techniques we developed in DotSlash can be applied to other systems beyond web servers. Handling short-term dramatic load spikes caused by hotspots is an issue common to many Internet servers such as game servers, SOAP servers [35], and SIP proxy servers [31]. Similar issues may also arise in other types of systems such as peer-to-peer systems and Grid computing systems [16]. Currently, our prototype system [41] only supports open mutual-aid communities. We plan to add security mechanisms to our system to against attacks, and enable insurance mechanisms in our system to promote the incentive for providing DotSlash rescue services and reduce abuse.

REFERENCES

[1] S. 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] BEA WebLogic. http://www.bea.com/products/weblogic/server/.

[7] Andrew Binstock. Hashing rehashed. Dr. Dobb's, April 1996.

[8] 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.

[9] 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.

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

[11] J. Challenger, P. Dantzig, A. Iyengar, M. Squillante, and L. Zhang. Efficiently serving dynamic data at highly accessed web sites. *IEEE/ACM Transactions on Networking*, 12(2):233–246, April 2004.

[12] E. Coffman, P. Jelenkovic, J. Nieh, and D. Rubenstein. The Columbia hotspot rescue service: A research plan. Technical Report EE2002-05-131, Department of Electrical Engineering, Columbia University, May 2002.

[13] A. Datta, K. Dutta, H. Thomas, D. Vandermeer, and K. Ramamritham. Proxy-based acceleration of dynamically generated content on the world wide web: An approach and implementation. *ACM Transactions on Database Systems*, 29(2):403–443, June 2004.

[14] Amazon EC2. Amazon elastic compute cloud. http://www.amazon.com/gp/browse.html?node=201590011.

[15] P. Felber, T. Kaldewey, and S. Weiss. Proactive hot spot avoidance for web server dependability. In *IEEE Symposium on Reliable Distributed Systems (SRDS)*, Florianopolis, Brazil, October 2004.

[16] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. *International Journal of High Performance Computing Applications*, 15(3):200–222, August 2001.

[17] M. J. Freedman, E. Freudenthal, and D. Mazires. Democratizing content publication with Coral. In *USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, California, March 2004.

[18] E. Guttman, C. E. Perkins, J. Veizades, and M. Day. Service location protocol, version 2. RFC 2608, Internet Engineering Task Force, June 1999.

[19] IBM WebSphere. http://www-306.ibm.com/software/websphere/.

[20] 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.

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

[22] memcached homepage. http://www.danga.com/memcached/.

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

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

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

[26] V. Padmanabhan and K. Sripanidkulchai. The case for cooperative networking. In *International Workshop on Peer-to-Peer Systems (IPTPS)*, Cambridge, Massachusetts, March 2002.

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

[28] 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, New York, September 2003.

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

[30] RUBiS: Rice university bidding system. http://www.cs.rice.edu/CS/Systems/DynaServer/RUBiS/.

[31] K. Singh and H. Schulzrinne. Failover and load sharing in SIP telephony. In *International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*, Philadelphia, Pennsylvania, July 2005.

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

[33] S. Sivasubramanian, G. Pierre, M. v. Steen, and G. Alonso. GlobeCBC: Content-blind result caching for dynamic web applications. Technical Report IR-CS-022, Vrije Universiteit, June 2006.

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

[35] Simple object access protocol (SOAP). http://www.w3.org/2002/ws/.

[36] T. Stading, P. Maniatis, and M. Baker. Peer-to-peer caching schemes to address flash crowds. In *International Workshop on Peer-to-Peer Systems (IPTPS)*, Cambridge, Massachusetts, March 2002.

[37] A. Stavrou, D. Rubenstein, and S. Sahu. A lightweight, robust P2P system to handle flash crowds. In *IEEE International Conference on Network Protocols (ICNP)*, Paris, France, November 2002.

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

[39] L. Wang, K. Park, R. Pang, V. Pai, and L. Peterson. Reliability and security in the CoDeeN content distribution network. In *Annual Usenix Technical Conference*, Boston, Massachusetts, June 2004.

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

[41] Weibin Zhao and Henning Schulzrinne. DotSlash—an automated web hotspot rescue system project. http://dotslash.sourceforge.net/.

[42] Weibin Zhao and Henning Schulzrinne. Service location protocol enhancements project. http://mslp.sourceforge.net/.

[43] Weibin Zhao and Henning 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.

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

[45] Weibin Zhao, Henning Schulzrinne, and E. Guttman. Mesh-enhanced service location protocol (mSLP). RFC 3528, Internet Engineering Task Force, April 2003.

[46] Weibin Zhao, Henning Schulzrinne, E. Guttman, C. Bisdikian, and W. Jerome. Remote service discovery in the service location protocol (SLP) via DNS SRV. RFC 3832, Internet Engineering Task Force, July 2004.