GlobeCBC: Content-blind Result Caching for Dynamic Web Applications

Swaminathan Sivasubramanian[1], Guillaume Pierre, Maarten van Steen, and Gustavo Alonso[2]

[1] Vrije Universiteit, Amsterdam, The Netherlands
[2] ETH Zurich, Switzerland

{swami,gpierre,steen}@cs.vu.nl,alonso@inf.ethz.ch

Technical Report IR-CS-022, June, 2006.

Abstract. In this paper, we present GlobeCBC, a content-blind query caching middleware for hosting Web applications in an edge computing infrastructure. Unlike existing data caching middleware systems, GlobeCBC stores the query results independently and does not merge different query results. We study the potential performance of this approach using extensive experimentations on our prototype implementation and compare it with other systems over an emulated wide-area network. Our evaluations show that content-blind caching performs well in terms of client latency for applications that exhibit high locality. It allows the system to sustain higher throughput by offloading the origin server database. We also present the design and evaluation of different online cache replacement algorithms for edge servers that have limited resource capabilities. In our evaluations, we find that the best heuristic must exploit temporal locality and take into account the query execution cost simultaneously.

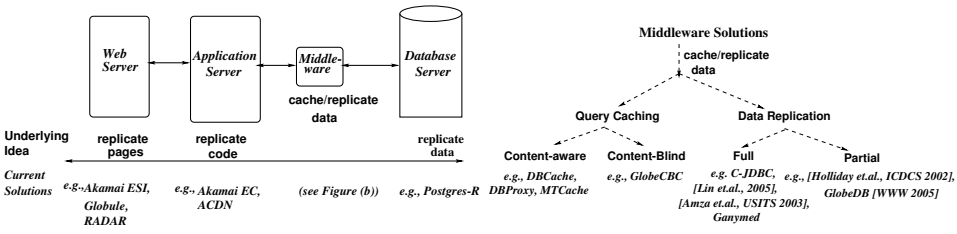
1 Introduction

Edge service architectures have become the most widespread platform for distributing Web content over the Internet. Commercial Content Delivery Networks (CDNs) like Akamai [4] and Speedera [34] deploy edge servers around the Internet that locally cache Web pages and deliver them to the clients. By serving Web content from edge servers located close to the clients, the response time for serving clients is reduced as each request need not travel across a wide-area network. Typically, edge servers store static Web pages. A plethora of techniques and commercial systems exist for replicating such pages [32].

In practice, however, a large amount of Web content is generated dynamically. These pages are generated upon request using Web applications that take, e.g., individual user profiles and request parameters into account when producing the content. These applications tend to run on top of databases. When a request arrives, the application examines the requests, issues the necessary read or update queries to the database, retrieves the data, and composes the page that is sent back to the client. Traditional CDNs use techniques such as fragment caching whereby the static fragments (and sometimes also certain dynamic parts) of a page are cached at the edge servers [30, **?**,**?**,**?**]. However, the increasing need for content personalization reduces the temporal locality among the fragment responses generated for different Web clients. Such applications require different solutions than fragment caching.

To handle such applications, CDNs employ edge computing infrastructures (e.g., Akamai ECI [4], ACDN [29]) where the application code is replicated at all edge servers and the data are kept in a centralized server. Mere replication of code has two drawbacks: (i) each data access incurs a wide-area network latency; and (ii) the central database server (which we call the origin server) becomes a potential bottleneck. This problem has gained significant interest among the research community, resulting in a variety of middleware solutions that cache or replicate data.

As seen in figure 1, data access middleware systems can be broadly classified into two types: (i) Query caching - systems that cache the results of database queries at the edge servers, and (ii) Data replication systems, which (fully or partially) replicate the underlying *database tuples*. These two approaches are suited for different kinds of Web applications. Query caching is suited for Web applications whose query workload



(a) Solutions across 3-tiers (b) Middleware solutions

Fig. 1. Variety of solutions that address problem of scalable Web hosting

exhibits high temporal locality and contain a small number of updates. If the workload exhibits poor temporal locality then data replication often proves beneficial [8, 14, 20, 31].

In this paper, we target the first kind of Web applications and explore the potential performance of a caching technique (that we call *content-blind* query caching) for hosting such Web applications. We present the design and implementation of *GlobeCBC*, a system that accelerates performance of Web applications by caching query results at the edge server. Unlike existing data caching middleware systems (e.g., [10, 22, 5]), GlobeCBC does not merge different query results at edge server databases but stores each query result independently. At the outset, this simple approach may look very limiting. However, as we show later in the paper, for many Web applications this approach avoids the overhead of query containment [6], query planning, query execution and cache management (thereby reducing the server overhead) while maintaining high cache hit ratio (thereby avoiding wide-area network latencies). We substantiate these claims with extensive experimentations on an emulated wide-area network test-bed for different kinds of Web applications.

An important issue in GlobeCBC is to determine which query results to cache (or rather which item to evict from the cache), when the maximum storage capacity has been reached. This decision is governed by several factors such as the cost of each query, temporal locality, and update workload. Instead of making explicit decisions periodically, we propose the use of online cache replacement algorithms. While online cache replacement and placement algorithms are well researched in the context of Web pages, those algorithms are not always best suited to query-caching systems. We present and evaluate the performance of different query-cache replacement algorithms for GlobeCBC. We show that the best strategy is the one that takes into account both temporal locality and the execution cost of each query.

The contributions of this paper are twofold. First, we explore and demonstrate the potential performance benefits of content-blind query caching system for different Web applications. Second, we propose and evaluate the performance of different cache replacement algorithms and show that an algorithm that takes both the query cost and the temporal locality into account performs the best.

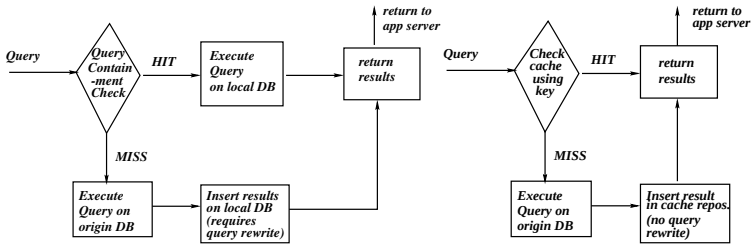
The rest of the paper is organized as follows: Section 2 presents the design issues in building a query caching system and motivates our design choices. Section 3 presents the GlobeCBC's architecture and Section 4 presents the experimentation results that compare the performance of GlobeCBC with other data caching and replication solutions. Section 5 presents the design and evaluation of different query replacement algorithms. Finally, Section 6 presents the related work and Section 7 concludes the paper.

2 Design Issues

2.1 Data granularity

The first and foremost design issue is to determine *what* data to cache and *how* to store them. The two possible design alternatives are *content-aware caching* and *content-blind caching*. The working of these two caching systems is pictorially described in Figure 2.

Content-aware caching systems run a DBMS at each edge server. As shown in Figure 2(a), each query received by the edge server is first checked to determine if it can be answered with the locally cached tuples (using a *query containment* procedure). If so, the query is executed locally. Otherwise, the query is executed on the origin server



(a) Content-aware Cache (b) Content-blind Cache

Fig. 2. Cache hit and miss processing in Query Caching systems

and the returned result tuples are inserted into the local database. This approach is storage efficient for queries that span a single database table as it does not store redundant tuples. Results of queries spanning multiple tables are usually stored separately.

In this paper, we pursue the relatively less explored content-blind caching approach (notably, a similar approach was explored [25]; we discuss it in detail in Section 6). The key difference between content-aware and content-blind caching is that the latter does not merge different query results and stores each result separately. Each query can be answered from the cache only if the result of the same query has been cached. There are several advantages to this approach. First, the process of checking if a query is cached or not becomes trivial and scales very well even for high loads (whereas query containment approaches are relatively expensive). Second, by caching results directly, the edge servers avoid the overhead of database query planning and query execution. In fact, edge servers do not even need to run a DBMS at all. This is especially beneficial under high load. Finally, cache replacement is quite simple as each result is stored independently.

The content-blind caching approach also has some shortcomings. First, by storing query results independently it possibly stores redundant data. This drawback also exists to a lesser extent in content-aware caching systems, as they also need to store the results of queries spanning multiple tables independently. Second, by skipping the query containment procedures, certain queries may not not be answered locally even if all the required tuples are available locally. However, as we show later in our experiments, this approach performs better than its caching counterparts for different Web applications.

2.2 Cache control and placement

Another design issue that needs to be addressed is to decide who selects which queries to cache (*cache control*) and how the selection is made (*cache placement*). Cache control can be *centralized*, where a central server collects query access patterns and decides which queries must be cached in which edge server or *distributed*, where each edge server independently decides which items to cache (or to evict from the cache). We choose the second option because it scales naturally with the addition of new edge servers and reduces the control overhead in the origin server.

Cache placement decides which set of query results to cache. This is important if the edge server does not have enough resources to cache all query results. Moreover, the problem of determining which results to keep in the cache gains significance, if the cached results are always kept in main memory instead of disk. Cache placement has

direct impact on the cache hit ratio and thus also on client latency and throughput, as well as the network traffic between the edge server(s) and the origin server. We choose an online caching algorithm by which an edge server caches the results of all queries it receives (unless explicitly specified). When the maximum storage capacity is reached, each edge server will run a cache replacement algorithm to determine the least beneficial cached query to evict from the cache. Although cache replacement algorithms are well researched in the context of static Web pages, only few efforts have been conducted to explore them in the context of database query caching [16, 7].

2.3 Consistency

Cached query results need to be updated or invalidated when the underlying database is updated. The problem of maintaining cache consistency has been extensively studied in the context of caching static Web pages [32]. However, consistency maintenance in query result caching is different from Web page caching for two reasons. First, an update to a single database record can affect multiple query results while at the same time it is more difficult to determine which cached result must be invalidated. In contrast, an update to a Web page usually affects only a single object. Second, the ratio of number of updates to number of reads is much higher when dealing with database-driven Web applications.

These two issues make the problem of maintaining consistency in a query-caching system challenging. To address them, we assume that the query workload consists of a fixed set of read and write query templates. A query template is a parameterized SQL query whose parameter values are passed to the system at runtime. This scheme is deployed, for example, using Java's prepared statement. In our system, we expect the developer to specify a priori which query template conflicts with which update template. Based on this, whenever an origin server receives an update query, it invalidates all results belonging to the conflicting templates. Although this assumption of data access through pre-defined query templates curtails the flexibility of the system in being able to handle new query types, it suits Web applications well.

Template-based invalidation prevents from invalidating all cached results upon each database update. However, there can still be a problem if the number of updates to the database is high or if there exists a significant number of conflicts between the query templates and update templates. In such cases, massive invalidations will lead to a low cache hit ratio, increased network traffic and increased server load, thereby leading to high client latencies. Typically, this problem is addressed by adoption of *weak consistency*. In this paper, we explore how different forms of weak consistency can be easily integrated into our system and show the potential scalability benefits obtained by them. We explain the consistency properties of our system in detail in the next section.

3 System Architecture

The architecture of our system is presented in Figure 3. An application is hosted by edge servers located across different regions in the Internet. Communication between edge servers usually traverses the wide-area network incurring wide-area latency. Each client is assumed to be redirected to its closest edge server using enhanced DNS-based redirection [18, **?**]. For each session, a client is assumed to be served by only one edge server. We assume that the application code is replicated at all edge servers. Furthermore, for each application, one of the edge servers acts as the *origin server*. The origin server hosts the complete database of the application.

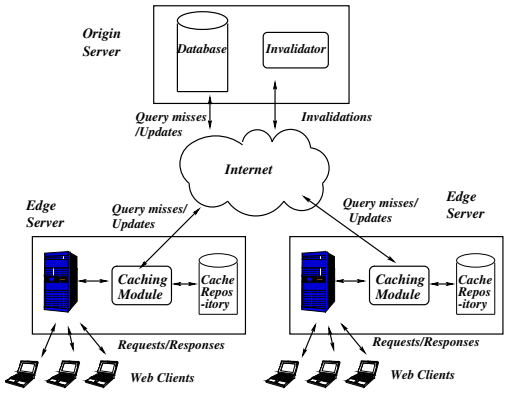


Fig. 3. System Architecture - Edge servers serving clients close to them and interactions between edge server and origin server goes through Wide-area network.

The two key components of our system are the *caching module* and the *invalidator*. The caching module is the middleware that runs in each edge server and intercepts all query execution requests between the application code and the origin server database. It is responsible for (i) determining if the requested read query's result is available in the local cache repository, (ii) returning query results (either from the local cache repository or by executing the query at the origin server) and (iii) adding/replacing new query results into the cache.

The invalidator is a stand-alone service that runs at the origin server. It is responsible for monitoring all update queries addressed to the database server, determining the list of query templates to invalidate and issuing invalidations. We will discuss the design and implementation of these components below.

3.1 Caching Module

The caching module is implemented as a PHP driver and can be added as a module to the Apache Web Server [9]. As said earlier, we assume that the application's query workload consists of a fixed set of read and write templates. Each query is identified by the module using the structure: $< template\text{-}id, query\ parameters >$ as key.

When a read query is issued, the caching module checks if the query result is cached in the local repository (using its key as the unique identifier). If found, then the result is returned immediately. Note that this is different from traditional edge database caching systems as the cached units are not the result tuples but the result structures. This allows the caching module to return the query result immediately without query planning and execution overheads.

If the incoming query is not found in the cache, then the query is executed at the origin server. Upon successful execution, the result is stored in the local repository. Note that the local cache repository does not use a DBMS to store query results. Instead, in our system, we store the result of each query as a separate file in the local file system. Of course, to improve performance it can also be kept in main memory. The caching module subscribes at the invalidator to the invalidation channels corresponding to the cached template-id (if it had not subscribed already). This allows the edge server to be informed of the updates to the underlying database that can *possibly* affect the cached results. This process of invalidation is described in detail in the next section.

When the caching module receives a write query (i.e., update/insert/delete), it is executed on the origin server's database. Subsequently, it sends a message (asynchronously) to the invalidator with the template identifier and the parameter (if applicable) of the write query. Note that since invalidation messages are sent asynchronously to the edge servers, the system does not provide strong consistency. We assume that this is acceptable for the hosted Web application. However, if a particular query template can not tolerate any staleness in its result, then the application developer can disable caching for the concerned query template.

3.2 Invalidator

The invalidator is a stand-alone service that runs in the origin server and is responsible for invalidating the cached results at edge servers when they become stale due to updates. As noted earlier, we assume that the Web application's workload consists of a fixed set of read and write query templates. Furthermore, we assume that the conflicts between read and write templates are marked offhand either manually by the developer or using automated code parsers. For example, a query template whose instances add a new book to the book table (e.g., "INSERT into books VALUES(..)") conflicts with query instances that find the newly added books.

The invalidator maintains such a conflict map for all the write query templates. For each write template, the invalidator maintains a publish-subscribe channel. Edge servers that cache a query result of template instance id subscribe to the channels of write template(s) that id conflicts with.

When a write query is performed on the origin server's database, the invalidator receives a message from the edge server's caching module. Upon its receipt, the invalidator sends the write query key (and the query parameter, if applicable) to all the subscribers in the channel corresponding to the write identifier. Upon receipt of the invalidation message, the caching module in each edge server invalidates instances of the query templates that conflicts with the write template. We discuss the process of template-based invalidation in detail in the next section.

3.3 Fine-grained Invalidation

Template-based invalidation helps in reducing the number of invalidations performed, provided there is no conflict between a result and update query even though they operate on the same database table(s). For example, in a bookstore application, the cached result of a query to find the best selling books is not affected by an update query to the book table for changing the price of a book. However, the cached result should be invalidated if a new book order is placed.

Such a simple coarse-grained template-based invalidation is sometimes too conservative as it invalidates all instances of a conflicting query template. For example, consider the following template: QT1: "SELECT price, stock, details from book where id=?" and update query template UT1: "Update price=price+1 from books where id=?". Using a simple coarse-grained invalidation scheme, an update to even a single book (e.g., $< UT1, 100 >$) will invalidate all cached instances of QT1.

To avoid this conservative invalidation, a simple extension is to take into account the parameter of the updated item and invalidate only those cached queries that are affected by the updated item. However, the system can determine which cached queries are affected by an update, without examining the content, only for simple queries (i.e., SQL queries which access information based on primary key). So, in the above example, when the system receives an update query with key: <UT1,$100 >$, the invalidator

invalidates only the cached item whose key is <QT1, 100>, provided QT1 is a simple query. As shown later, this simple extension improves the performance of applications whose workload has lots of simple queries, such as the TPC-W benchmark.

3.4 Tunable Consistency

Template-based invalidations help in reducing the number of invalidations to an extent. However, for Web applications that have a large number of read-write conflicts, the system will generate large numbers of invalidations. This can cause a poor cache hit ratio, leading to increased wide-area network traffic, increased origin server load, and therefore increased client latency. Applications that have these characteristics can usually scale only by employing weak consistency.

Traditionally, weak consistency protocols are employed along one of the axes: *time*, *order of operations* or *value* [35]. Time-based weak consistency protocols relax consistency by permitting inconsistency between the cached results and the underlying database for a bounded period of time. For example, in an online bookstore application, the administrator can allow the system to deliver the cached result of a best-sellers query provided the cached copy is at most one hour old.

Order-based weak consistency models are generally exploited in replicated databases. These models perceive every read/write operation as a transaction and allow the replicas to operate in a different state if the out-of-order transactions adhere to policy-specific rules. For example, [21] introduces the concept of N-ignorant transactions, where a transaction can be carried out in a replica while it is ignorant of N prior transactions in other replicas. The rules constraining the execution order of transactions can also be defined based on dependencies among transactions.

The third axis of expressing consistency is based on the concept of *value*. Value-based consistency schemes ensure that the difference between the value of a cached copy and that of the original data is bound by a given Δ. Value-based schemes can be applied only to objects that have a precise definition of value (e.g., stock quote prices).

In our system, we provide interfaces to the application developers (or system administrators) to *tune* their consistency bounds based on *time* and *order*. We do not support value-based mechanisms as our caching module is content-blind.

For time-based mechanisms, the system expects the application developer (or the system administrator) to set TTI_i (time-to-invalidate) values for each update template, U_i. Subsequently, if an update query of type U_i is received by the invalidator, the invalidator starts a timer for TTI_i seconds (unless a timer has already been started). After TTI_i seconds, all queued invalidation messages for conflicting read templates are sent.

For order-based weak consistency mechanisms, we employ a mechanism similar to N-ignorant transactions. The system requires the application developer (or the administrator) to set the bound for each update template U_i, regarding the number of updates the invalidator can tolerate before invalidating the conflicting read templates (Max_Upds_i). To implement this mechanism, the invalidator maintains a counter that keeps track of the number of updates it has received for each update template (Num_Upds_i). When $Num\_Upds_i \geq Max\_Upds_i$, the invalidator sends out the invalidation messages and resets Num_Upds_i to 0.

The system can also support a combination of both time and order-based weak consistency mechanisms. Application developers or system administrators who want to get higher scalability using weak consistency, can simply tune one (or both) of these two parameters, TTI and Max_Upds. We call this means of employing weak consistency as *tunable consistency*.

4 Performance Evaluation

In this section, we compare the performance of content-blind caching to other solutions. We consider two different applications, a news Web site modelling `http://slashdot.org` and TPC-W benchmark, an industry standard e-commerce benchmark that models an online bookstore. We chose these two applications for their different data access characteristics. For example, in a typical news forum, most users are usually interested in the latest news and so the workload will usually exhibit high locality. On the other hand, in a bookstore application, the shopping interests of customers can be different thereby leading to much lower query locality. This allows us to study the behavior of content-blind caching for different data access patterns.

4.1 Performance results: Slashdot application

In this section, we present results of experiments performed using the RUBBoS benchmark, an open source benchmark that models an online news forum application similar to `slashdot.org` [3]. The benchmark is written in PHP. Its database consists of five tables, storing information regarding users, stories, comments, submissions and moderator activities. The database is filled with $500,000$ users, out of which 10% have moderator privileges, and $200,000$ comments. The size of the database is approximately $1.5\ GB$.

We deployed the GlobeCBC prototype across 3 identical edge servers each with dual-processor Pentium III 900 Mhz CPU, $1\ GB$ of memory and a $120\ GB$ IDE hard disk. Each edge server uses an Apache $2.0.49$ Web server with PHP $4.3.6$. We use PostgreSQL 7.3.4 for our database servers and PgPool for pooling database connections [27]. The origin server uses an identical configuration as the edge servers except that it acts just as a backend database and does not run a Web server. We emulate a wide-area network (WAN) among the servers by directing all the traffic to an intermediate router which uses the NISTNet network emulator [2]. This router delays packets sent between the different servers to simulate a realistic wide-area network. In the remaining discussion, we refer to links via NISTNet with a bandwidth of $50Mbps$ and a latency of 100ms as WAN links, and $100Mbps$ and 0 latency as LAN links. Note that these bandwidth and latency values are considerably optimistic, as the Internet bandwidth usually varies a lot and is constantly affected by network congestion. These values are chosen to model the best network conditions for a CDN built on an Internet backbone and are the *least favorable* conditions to show the best performance of any data caching or replication system. Any lower bandwidth or higher latency will only boost the performance of caching systems. For example, if the WAN delay in the route is set to a higher latency, say 500ms, then for edge computing infrastructures that do not replicate data, each database query will incur at least 500ms round trip latency (and more if each query is transmitted in multiple TCP packets). This value will therefore boost the performance of data caching solutions as they can answer queries locally.

We use three client machines to generate requests addressed to the three edge servers. The client workload for the system is generated by Emulated Browsers (EBs). The runtime behavior of an EB models a single active client session. Starting from the home page of the site, each EB uses a Customer Behavior Graph Model (a Markov chain with various interactions with the Web pages in a Web site as nodes and transition probabilities as edges) to navigate among Web pages, performing a sequence of Web interactions [33]. The behavior model also incorporates a think time parameter that controls the amount of time an EB waits between receiving a response and issuing the next

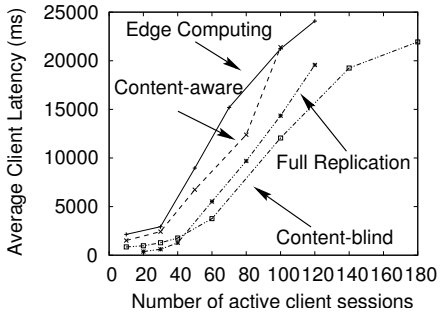


Fig. 4. Effect of Workload on Slashdot application

request, thereby modelling a human user more accurately. The user workload contains more than 15% interactions that lead to updates.

As shown in Figure **??**, we evaluated four system architectures using the aforementioned setup: (i) *Edge Computing:* the code is replicated in every edge server while the data remain centralized at the origin server. (ii) *Full replication:* the code and database are fully replicated in the edge server. All updates are performed at the origin server and are propagated asynchronously to the edge servers. (iii) *Content-aware caching:* the edge servers run a Web application server and a DBMS. The key difference between this system and full replication is that the edge servers run the query locally only if the *query containment* check results in a hit. We implemented the algorithm described in [5] for query containment. (iv) *Content-blind caching:* The edge servers run a Web application server and a cache repository. The origin server runs the database. Unless stated otherwise, in content-blind caching schemes, the invalidator is configured with the strongest possible consistency, $TTI = 0$ and $Max\_Upds = 0$. Note that full replication, content-aware and content-blind caching systems provide the same level of consistency while only the edge computing architecture provides the stronger level of consistency.

All experiments are started with a cold cache. The system is warmed up for 20 minutes, after which the measurements are taken for a period of 90 minutes. We assume the edge server to have infinite cache capacity, i.e., the size of cache repository was not restricted because we did not want the effect of cache replacement algorithms to affect the performance of content-aware and content-blind caching systems. We study the performance of different cache replacement algorithms in the next section. In all experiments, we measure the end-to-end *client latency*, which is the sum of the *network latency* (the time spent by the request traversing the WAN) and *internal latency* (the time spent by the request in generating the query responses and composing the subsequent HTML pages).

Results for WAN experiments We first evaluated the client latency for different architectures for only 1 edge server and the origin server. We studied the client latency for different client workloads. The results of our experiment are shown in Figure 4.

As seen in the figure, content-blind caching performs the best in terms of client latency (except for low loads) while edge computing performs the worst in all cases. It can also be seen that content-blind caching sustains higher load than full replication and the content-aware caching. This is quite remarkable considering that content-blind caching provides the same level of consistency. The edge computing infrastructure per-

forms worse than the other architectures. This is due to the fact that all data accesses incur a WAN latency in addition to the origin server becoming the scalability bottleneck thereby increasing the internal latency in generating query responses.

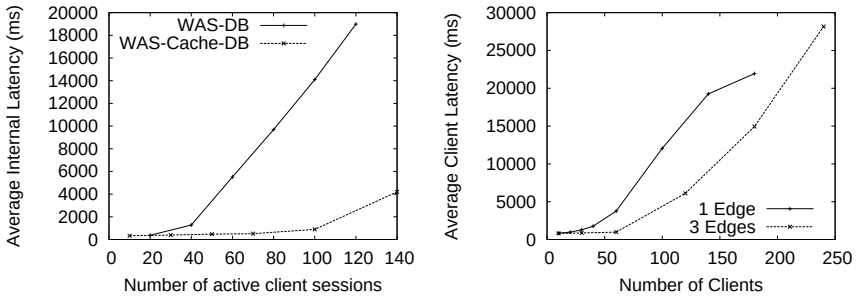
Full replication system performs marginally better than content-blind under very low client loads (up until 30 client sessions). This is because in a full replication system each query is answered locally thereby avoiding any wide-area network latency. During low workloads, the internal latency incurred in generating a query response is lower than the network latency incurred in answering a query. Therefore, full replication system performs marginally better in such workloads. On the other hand, content-blind caching outperforms full replication during high workloads for two reasons. First, the application's workload exhibits high temporal locality (yielding a cache hit ratio of up to almost 90%) thereby avoiding WAN latency. Second, the internal latency in generating a query response, for a content-blind caching system, is much lower than that of full replication as the caching system avoids query planning and execution latency.

Content-blind caching performs better than content-aware caching for all client workloads. Even though both systems are equally capable of capturing temporal locality, the former incurs more latency in generating query responses as it incurs the overhead of query containment, cache management (inserting and invalidating caches), query planning, and execution. On the other hand, content-blind caching avoids these overheads by storing the results of query instances separately, thereby resulting in lower client latency. Furthermore, the internal latency overhead increases with increase in client load, as we discuss next.

Understanding Internal latencies To better understand the effect of client workloads on query execution overheads and subsequently on the internal page generation latency, we isolated a single edge server and studied the latency in generating a page for two different systems: (i) a Pentium III dual processor machine that runs a vanilla Apache server and Postgresql $7.3.4$ (WAS-DB) and (ii) the same setup that runs the content-blind caching module, in addition to the Web server and database server (WAS-Cache-DB). For both setups, we evaluated the client latency for read-only client sessions for different loads. In both setups, the network latency incurred for generating a page is 0 as both the Web server and database reside on the same machine. The objective of this study is to measure the potential gain in internal latency by the use of content-blind caching which avoids query planning and execution latencies.

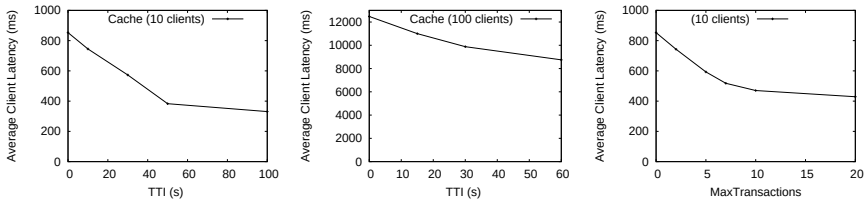
The results of our study are given in Figure 5(a). As seen in the figure, the internal latency of a system that employs content-blind caching (WAS-Cache-DB) is much lower than that of the traditional WAS-DB. In particular, the difference grows up to an order of magnitude for high loads. This study thus gives a clear insight into the potential gains of content-blind caching. It also explains the better performance of content-blind caching in comparison to full replication and content-aware caching system, which always incur the query planning and execution overheads.

Study with multiple edge servers As a next experiment, we compared the average client latency of a content-blind caching system with a single edge server to one with three edge servers. For the latter case, the client load was equally divided among the 3 edge servers. As seen in figure 5(b), the system with 3 edge servers performs significantly better in terms of client latency (by a factor of 2) compared to the single edge server system. This can of course be easily understood as the edge servers share their client load. The difference in client latency is negligible during low loads as a single edge server is able to sustain the client load under these conditions. Under high loads,



(a) Internal latency (b) Client latency

Fig. 5. Comparison of internal and client latency for plain 3-tier architecture with and without content-blind caching system



(a) Effect of TTI for 10 clients (b) Effect of TTI for 100 clients (c) Effect of Max_Upds for 10 clients

Fig. 6. Effect of TTI and Max_Upds

the 3-edge server system is able to sustain more load. However, as can be seen from the figure, the system does not offer a linear improvement and can only sustain 50% more load than the single edge server. This is because in both cases the database in the origin server remains the central scalability bottleneck.

Tunable Consistency For high workloads when the number of updates increases (recall that the fraction of update interactions is around 15%), immediate invalidations increase the system load leading to a higher client latency. If more scalability is desired, such situations demand the use of relaxed consistency models. As noted earlier, we adopt a tunable consistency model in which the application developer or the system administrator can specify the limits of weak consistency along the axes of time (TTI) and order (Max_Upds).

In this experiment, we study the potential gains in client latency of relaxing consistency with different values of TTI and Max_Upds in a system with a single edge server connected to the origin server through a WAN link. In the first set of experiments, we fixed Max_Upds at 0 and study the impact of TTI on client latency for a load of 10 and 100 client sessions. The results are given in Figure 6(a) and 6(b). As seen in these figures, for a TTI of 1 minute, we gain a factor of 3 in client latency under low load

and 30% improvement under high load. The difference in gains of client latency is due to the increase in internal latency.

In the second set of experiments, we studied the effect of Max_Upds on client latency. The results are given in Figure 6(c). As seen in the figure, with a Max_Upds value of 10, we can gain a factor of 2 in client latency.

The objective of this study is not to recommend the best value of TTI or Max_Upds, as the best values depend on the application needs and system costs. This experiment is rather meant to demonstrate that, provided the application can support it, controlled relaxation of consistency can produce significant gains in performance.

4.2 Performance results: TPC-W benchmark

We evaluated the performance of different systems for the TPC-W benchmark, an industry standard e-commerce benchmark that models an online bookstore like Amazon.com [33]. We used the open source PHP implementation of TPC-W [26]. We modified the client workload behavior such that the book popularity follows a Zipf distribution (with $alpha = 1$), which was found in a study that observed data characteristics of Amazon online bookstore [11].

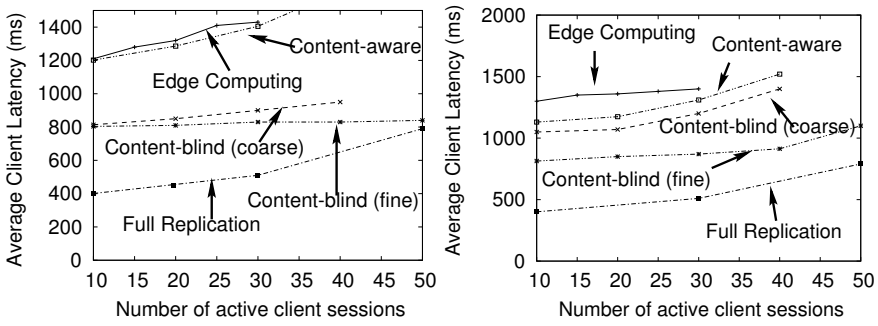
For this application, we studied the performance of four systems: edge computing, content-aware caching, content-blind caching and full replication. For content-blind caching, we studied the client latency for two kinds of invalidations: coarse-grained and fine-grained invalidation. The objective is to examine the potential benefits in employing the finer invalidation. Recall that fine-grained invalidation takes into the account the parameter of the update template to invalidate conflicting simple queries.

We studied the client latencies of these systems for two kinds of workloads: browsing (which consists of 95% browsing and 5% ordering interactions) and ordering workload (50% browsing and 50% shopping interactions). As seen in Figure 7, edge computing performs the worst in client latency while full replication performs the best. In particular, full replication performs better than content-blind caching because the TPC-W benchmark workload exhibits poor temporal locality and yields a hit ratio of at most 35% in our experiments. In such cases, data replication helps as each query can be answered locally. However, during high loads, the gain in latency decreases as the internal latency of full replication increases. These results are in line with our earlier results, where we demonstrated that applications whose workload exhibit poor temporal locality or has many updates are best hosted using data replication schemes [31].

Content-blind caching performs better than content-aware caching as both are able to capture temporal locality and the former benefits from reduced internal latency. Note that the gain in client latency in content-blind caching compared to edge computing in TPC-W benchmark is only about 50%, while it was a factor of 3 for the Slashdot application. This is again due to the poor temporal locality exhibited by the workload.

Between the two invalidation mechanisms of content-blind caching system, the fine-grained one consistently performs better than its coarse-grained counterpart. In particular, the difference in latency is especially high for the ordering workload. This is because the fraction of simple queries in this workload is higher than that of the browsing workload (recall that simple queries are queries that access data from a single table using primary keys). Since fine-grained invalidation performs fewer invalidation for instances of simple query templates, it improves overall client latency[3].

[3] The performance difference between these invalidation schemes in Slashdot application was negligible. This is because fine-grained invalidation improves performance only in the pres-



(a) Browsing (b) Ordering

Fig. 7. Performance of our architecture compared to edge computing for TPC-W benchmark

4.3 Discussion

As can be seen with these experiments, content-blind caching performs better than the other architectures while providing the same level of consistency guarantees, provided the application's data workload exhibits high temporal locality. Furthermore, for these applications, we also showed that our system allows administrators to achieve higher scalability employing relaxed consistency by simply tuning two variables, TTI and Max_Upds.

On the other hand, for applications that exhibit poor locality (such as the TPC-W benchmark), data replication schemes perform better than content-blind caching. Our hypothesis is that there exists no single solution that can perform the best for all Web applications and workloads. Different techniques are optimal for different applications. For example, in our earlier research, we demonstrated using a prototype system that applications with poor temporal locality can be hosted scalably using autonomic data replication [31]. As shown in this section, applications that exhibit high locality can benefit to a large extent from content-blind caching and outperform its replication counterparts. We are working on integrating our data replication and caching solutions, so that the system can dynamically choose the optimal technique for different database based on the observed workload. The design of such an integrated system is beyond the scope of this paper.

5 Cache Replacement

An important issue in any caching system is to determine which query results to cache and which ones to evict from the cache. This is usually a non-issue if the edge server has unlimited memory and storage resources. However, in CDNs, the edge servers are usually simple desktop servers with limited memory resources and simple disk access resources (for example, most of the servers use IDE disks). In such an environment, it is desirable to keep as many query results in main memory as possible and hence the problem of which results to keep in the cache gains significance. This issue is especially relevant in a collaborative CDN environment such as Globule [1] where the edge servers are usually end-user machines.

ence of simple queries. In the Slashdot application, the majority of the query workload consists of complex queries.

5.1 Cache Replacement

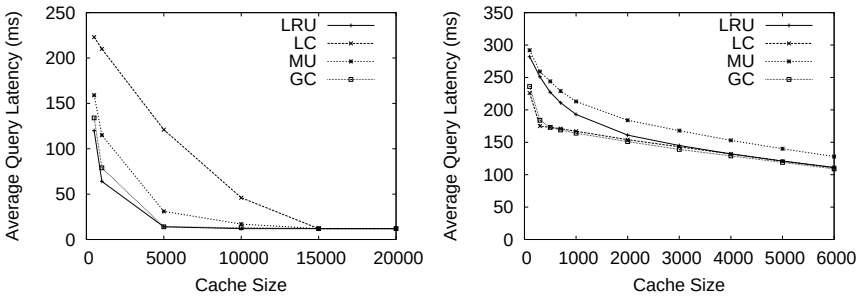
Cache replacement is simple in content-blind caching compared to content-aware caching. For example, in the latter, evicting a query result from the cache requires appropriate checks to ensure that query containment conditions of other query templates are not violated as results are merged. However, since content-blind caching stores result independently, each result can be replaced independently. GlobeCBC uses an online cache replacement mechanism to determine which results are more beneficial to retain in the cache when the cache is full. The query result replacement mechanism works as follows: When the cache is full, it must select one or more cached results to be removed so that resource constraints are met again.

The performance of the caching system depends to a large extent on the effectiveness of the cache replacement algorithm. An ideal cache replacement algorithm must take into account several metrics such as temporal locality, cost of the query, update characteristics of the database, etc. In this paper, we evaluate the average query latency of different cache replacement algorithms that operate on one or more of these metrics to find the one most suited for our system.

We designed and evaluated the performance of the following cache replacement algorithms: (i) *Least Recently Used (LRU)*. LRU always deletes the recently used cached result with the new result. The intuition is that the most recently accessed queries are most likely to be accessed again. Previous research on Web page caching suggests that strategy performs best if all cached items are equal. (ii) *Least Cost (LC)*. Each query takes different time to execute in the database server. This is usually modelled by a *query cost* parameter, which is used in the query planning. LC replaces the new result with the cache item that has the least query cost. The intuition is that a cache hit on a high cost query is more valuable than hit on a low cost query as it will offload the origin server to a great extent. (iii) *Most Updated (MU)*. Each edge server counts the number of invalidations received by each template. The system replaces the result whose parent template is most invalidated. The intuition is caching the least updated queries might improve the cache hit ratio. (iv) *Greedy-Cost (GC)*. LC optimizes on internal latency but does not exploit the locality of requests. On the other hand, LRU exploits the locality of requests but ignores the individual characteristics of the cached items. Greedy-cost aims to capture the best properties of these two algorithms. The algorithm associates a value, C_i, with each cached query result q_i. Initially, when a result is brought into the cache, C_i is set to be the time incurred in bringing it. When a replacement needs to be made, the result with the lowest C value, C_{min}, is replaced, and then all results reduce their C values by C_{min}. If a query result is accessed, its C value is restored to its initial value. Thus, the C values of recently accessed results retain a larger portion of the original cost than those of results that have not been accessed for a long time. This way, GC exploits temporal locality and takes query cost into account at the same time. This algorithm is similar to the Greedy-Dual algorithm used in Web caching [12].

5.2 Evaluation Results

We evaluated the performance of different cache replacement algorithms using trace-driven simulations. We collected query traces of the RUBBoS and TPC-W benchmark (shopping mix). Each trace represents two hours of execution with a workload of 10 active client sessions. The cost of each query template is assigned as the average query response time of all query instances of the given template. This was also used as the internal latency in calculating the overall query response time. We fixed the WAN latency to be 100ms.



(a) Slashdot (b) TPC Shopping

Fig. 8. Performance of different cache replacement algorithms for different cache sizes

We replayed these traces with different replacement algorithms, assuming that the latency of a cache hit is 0 and the latency of a cache miss is the sum of internal latency and WAN latency (if incurred). Note that the average query latency is different from the client-perceived latency as each client request may trigger multiple query requests (on average 4 for Slashdot) to generate a page. Our experiments were started with a cold cache and the measurements were taken for the entire run. We report the average query latency for various cache sizes.

The results of our experiment are given in Figure 8. For both applications, GC performs the best or close to the best. LRU performs the best for the Slashdot application due to the high temporal locality of the query workload and GC performs almost as good as LRU. However, LRU performs poorly for the TPC-W application as the query workload does not exhibit high locality. As a consequence, it constantly replaces high-cost queries (which form almost the 30% of query workload). This leads to an increase in internal latency of queries.

LC performs poorly for the Slashdot application as it is not capable of exploiting temporal locality. However, it does well for TPC-W as it retains the high cost queries and hence is able to optimize on high cost queries which exhibit high locality. MU performs poorly for both applications as it does not exploit locality and also does not take into account the cost of the queries. MU can be useful only if the workload generates a significant number of invalidations. However, in the TPC-W application, even though many updates are issued, the template-based query invalidation does a reasonable job in reducing the number of invalidations thereby reducing MU's performance gain.

In summary, among all strategies, GC performs (close to) the best by virtue of being able to exploit locality and keeping a good balance in managing high-cost queries.

6 Related Work

As shown in Figure 1, a number of systems have been developed to handle Web application hosting [4, 13, 29]. These systems replicate code at the replica servers, but do not replicate the application data. These systems figure in our evaluations as edge computing system. However, as we have shown, mere replication of code will lead to incurring WAN latency for each data access and direct all the load to the central database. Hence, these systems are not suitable only if the applications requires significant interactions with the backend database.

To address this limitation, commercial database caching systems such as DBCache [10], MTCache [22] and DBProxy [5] cache the results of selected queries and keep them consistent with the underlying database. These systems fall in the category of *content-aware* caching systems. They store the database tuples that form the results of queries in the DBMS running in the edge server (provided the query is on a single table) and merge different query results. Note that a similar approach known as semantic caching was proposed for client-server database systems in [16]. These systems are built to be very flexible and can support different types of applications (i.e., not just to template-based Web applications).

However, this flexibility comes at a cost. As shown in Figure 2(a), in these systems, each query needs to be subject to a local query containment check procedure to determine if the edge database server has all the data required to answer the query completely. Even if the containment test results in a cache hit, the system incurs query planning and execution overhead. Moreover, inserting and removing items from the local cache also is a non-trivial process. On the other hand, GlobeCBC is limited in flexibility and is suited only for Web applications whose workload usually consists of a small set of read and write templates. This allows the system to avoid query execution overheads and achieve better client performance even under heavy loads, provided the workload exhibits good locality as we have shown in our performance results. Furthermore, content-blind caching also allows the system to add or remove items from the cache easily as each item is treated and stored separately.

Many middleware systems have been proposed for scalable replication of a database in a cluster of servers [14, 8, 24, 28]. However, the focus of these works is to improve the throughput of the backend database within a cluster environment. On the other hand, the focus of our work is to improve the client-perceived performance in a CDN environment where edge servers are spread across a wide-area network. Furthermore, as we showed earlier, data replication is useful if the workload exhibits poor locality and low number of updates. If the number of updates increases, then autonomic replication solutions can be envisaged [31].

In [25], the authors investigate the use of a similar query caching system. Compared to their work, in this paper, we explore the potential performance gains of content-blind caching systems for different applications through extensive evaluations of different edge computing architectures in an emulated wide-area network. We also propose and evaluate different cache replacement algorithms to address the case of resource-constrained edge servers. These issues and experiments were not studied in [25].

7 Conclusions and Future Work

In this paper, we presented GlobeCBC, a content-blind query caching middleware for hosting Web applications in an edge computing infrastructure. Unlike existing data caching middleware systems, content-blind caching systems do not merge different query results and store the query results as result structures independently. We studied the potential performance of this approach using extensive experimentations on our prototype implementation and compared it with the other systems over an emulated wide-area network. Our evaluations show that content-blind caching performs well in terms of client latency for applications that exhibit high locality and is also able to sustain more load by offloading the origin server database. We also showed that when applications can support it, the system administrators can still improve performance by relaxing the data consistency in a simple fashion. We proposed and evaluated different

online query replacement algorithms which will be useful for resource-constrained edge servers. In our evaluations, we found that the best algorithm must take into account both the query execution cost and the temporal locality.

Even though GlobeCBC performs very well for applications with high query locality, it is outperformed by database tuple replication when the workload has poor locality. We are therefore working on integrating GlobeCBC with our data replication solution, GlobeDB [31]. We believe that such an integrated system will be able to perform well for all kinds of Web applications, by being able to select the best performing strategy for a given database based on the observed workloads.

References

1. Globule: An User-centric Content Delivery Network, `http://www.globule.org`.
2. NISTNet: A Network emulation tool, `http://snad.ncsl.nist.gov/itg/nistnet/`.
3. RUBBoS benchmark, `http://jmob.objectweb.org/rubbos.html`.
4. Akamai Inc. Edge Computing Infrastructure.
5. K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. DBProxy: A dynamic data cache for web applications. In *Proceedings of Internation Conference on Data Engineering*, pages 821–831, 2003.
6. K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. Scalable template-based query containment checking for web semantic caches. In *Proceedings of Internation Conference on Data Engineering*, pages 493–504, 2003.
7. K. Amiri, R. Tewari, S. Park, and S. Padmanabhan. On space management in a dynamic edge data cache. In *WebDB*, pages 37–42, 2002.
8. C. Amza, A. Cox, and W. Zwaenepoel. Conflict-aware scheduling for dynamic content applications. Proceedings of the Fifth USENIX Symposium on Internet Technologies and Systems, 2003.
9. Apache Web Server. `http://www.apache.org/`.
10. C. Bornhvd, M. Altinel, C. Mohan, H. Pirahesh, and B. Reinwald. Adaptive database caching with DBCache. *Data Engineering*, 27(2):11–18, June 2004.
11. E. Brynjolfsson, M. Smith, and Y. Hu. Consumer surplus in the digital economy: Estimating the value of increased product variety at online booksellers. MIT Sloan Working Paper No. 4305-03., 2003.
12. P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems (USITS-97)*, Monterey, CA, 1997.
13. P. Cao, J. Zhang, and K. Beach. Active cache: Caching dynamic contents on the Web. In *Proceedings of the Middleware Conference*, pages 373–388, Sept. 1998.
14. E. Cecchet. C-JDBC: a middleware framework for database clustering. *Data Engineering*, 27(2):19–26, June 2004.
15. J. Challenger, P. Dantzig, and K. Witting. A fragment-based approach for efficiently creating dynamic web content. *ACM Transactions on Internet Technology*, 4(4), Nov 2004.
16. S. Dar, M. J. Franklin, B. Jonsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *VLDB '96: Proceedings of the 22th International Conference on Very Large Data Bases*, pages 330–341, 1996.
17. A. Datta, K. Dutta, H. Thomas, D. VanderMeer, Suresha, and K. Ramamritham. Proxy-based acceleration of dynamically generated content on the world wide web: an approach and implementation. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 97–108. ACM Press, 2002.
18. J. Dilley, B. Maggs, J. Parikh, H. Prokop, R. Sitaraman, and B. Weihl. Globally distributed content delivery. *IEEE Internet Computing*, 6(5):50–58, 2002.

19. Z. Fei, S. Bhattacharjee, E. W. Zegura, and M. H. Ammar. A novel server selection technique for improving the response time of a replicated service. In *Proceedings of INFOCOM*, pages 783–791, March 1998.
20. J. Holliday, D. Agrawal, and A. Abbadi. Partial database replication using epidemic communication. In 22nd International Confernce on Distributed Computing Systems (ICDCS), pages 485–493, Vienna, Austria, July 2002., 2002.
21. N. Krishnakumar and A. J. Bernstein. Bounded Ignorance: A Technique for Increasing Concurrency in a Replicated System. *ACM Transactions on Database Systems*, 4(19):586–625, 1994.
22. P. Larson, J. Goldstein, H. Guo, and J. Zhou. MTCache: Mid-tier database caching for SQL server. *Data Engineering*, 27(2):27–33, June 2004.
23. W.-S. Li, O. Po, W.-P. Hsiung, K. S. Candan, and D. Agrawal. Engineering and hosting adaptive freshness-sensitive web applications on data centers. In *Proceedings of the Twelfth international conference on World Wide Web*, pages 587–598. ACM Press, 2003.
24. Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *SIGMOD Conference*, 2005.
25. C. Olston, A. Manjhi, C. Garrod, A. Ailamaki, B. M. Maggs, and T. C. Mowry. A scalability service for dynamic web applications. In *CIDR*, pages 56–69, 2005.
26. PGFoundry. PHP Implementation of TPC-W Benchmark, `http://pgfoundry.org/projects/tpc-w-php/`.
27. PGPool: Connection Pooler. `http://pgfoundry.org/projects/pgpool/`.
28. C. Plattner and G. Alonso. Ganymed: Scalable replication for transactional web applications. In *Proceedings of the International Middleware Conference*, Toronto, Canada, Oct. 2004.
29. M. Rabinovich, Z. Xiao, and A. Agarwal. Computing on the edge: A platform for replicating internet applications. In *Proceedings of the Eighth International Workshop on Web Content Caching and Distribution*, pages 57–77, Hawthorne, NY, USA, Sept. 2003.
30. M. Rabinovich, Z. Xiao, F. Douglis, and C. Kalmanek. Moving edge-side includes to the real edge- the clients. In *USENIX Symposium on Internet Technologies and Systems*, 1997.
31. S. Sivasubramanian, G. Alonso, G. Pierre, and M. van Steen. GlobeDB: autonomic data replication for web applications. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 33–42, New York, NY, USA, 2005. ACM Press.
32. S. Sivasubramanian, M. Szymaniak, G. Pierre, and M. van Steen. Replication for web hosting systems. *ACM Computing Surveys*, 36(3), Sept. 2004.
33. W. Smith. TPC-W: Benchmarking an e-commerce solution. http://www.tpc.org/tpcw/tpcw_ex.asp.
34. Speedera Inc. `http://www.speedera.com`.
35. H. Yu and A. Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Trans. Comput. Syst.*, 20(3):239–282, 2002.