# Holistic Query Transformations for Dynamic Web Applications

**Amit Manjhi Charles Garrod Bruce M. Maggs**
**Todd C. Mowry Anthony Tomasic**

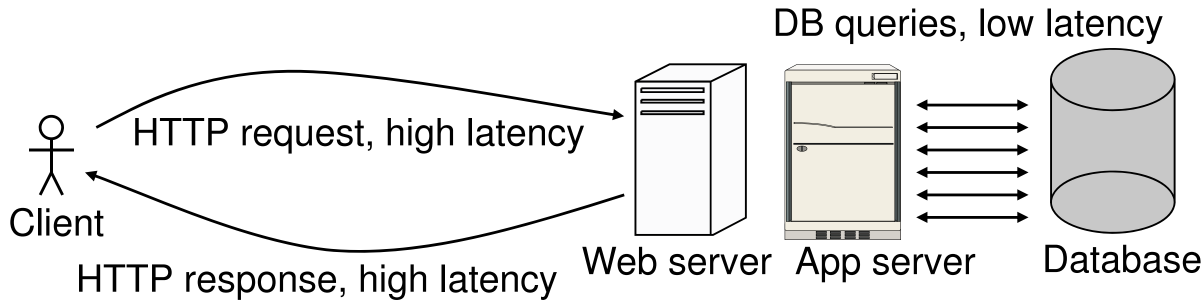
October 2008
CMU-CS-08-160

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

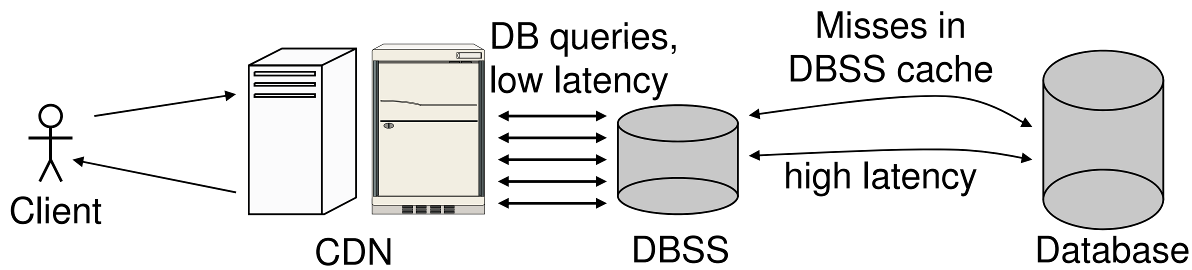
**Abstract**

A promising approach to scaling Web applications is to distribute the server infrastructure on which they run. This approach, unfortunately, can introduce latency between the application and database servers, which in turn increases the network latency of Web interactions for the clients (end users). In this paper we introduce the concept of source-to-source holistic transformations—transformations that seek to optimize both the application code and the database requests made by it, to reduce client latency. As examples of our concept, we propose and evaluate two source-to-source holistic transformations that focus on hiding the latencies of database queries. We argue that opportunities for applying these transformations will continue to exist in Web applications. We then present algorithms for automating these transformations in a source-to-source compiler. Finally, we evaluate the effect of these two transformations on three realistic Web benchmark applications, both in the traditional centralized setting and a distributed setting.

**Keywords:** Distributed Architecture, Web Applications Scalability, Holistic Optimization




(a) Latency in a traditional centralized architecture.




(b) Latency in a distributed architecture.

**Figure 1: Latency in traditional vs. distributed architectures.**

# 1 Introduction

Anyone on the Internet can access a Web application. As a result, Web applications suffer from unpredictable load, particularly due to breaking news (e.g., Hurricane Katrina) or popularity spikes (e.g., the Slashdot effect). To address the scalability challenge, Web applications increasingly use a distributed infrastructure. The distributed infrastructure inevitably introduces latency between the different tiers of the application, which in turn increases the latency experienced by users. User studies [1, 2] have shown that high user latencies drive customers away. Therefore user latencies must be kept low even when using a distributed architecture.

To ensure low user latencies, it is important to understand how this latency arises. A Web application is a collection of programs. On an HTTP request, an application server runs one or more of these programs to generate the response. These programs, in turn, issue database queries to obtain the data needed for generating the response. Frequently, the programs issue multiple database queries for each HTTP interaction: e.g., for the benchmark applications we study, the average number of queries per dynamic HTTP response varies between 1.8 and 9.1 (Table 3).

In a traditional centralized setting, these database queries are answered by a database server that is in the same administrative domain and connected to the application server(s) by a high bandwidth, low latency link. As a result, these multiple round-trips have little impact on the overall latency a user experiences. The user latency is dominated by the high latency of reaching the web server of the application. Figure 1(a) shows the different latency components in a traditional centralized setting.

In a distributed setting, an application may use geographically distributed Content Delivery Network (CDN) nodes to scale its web and application servers and geographically distributed Database Scalability Service (DBSS) nodes to scale its database [18, 20]. The database queries issued by an application server are handled by a DBSS node, which attempts to answer these queries from its query-result cache. If the request hits in the DBSS cache, the delay in obtaining the query result is minimal. However, if the request misses in the cache, the user must endure the delay in getting the response back from the home server database. This delay is typically long because the scalability service nodes are geographically distributed. Figure 1(b) shows the different latency components in a scalability service setting. Even after methods to boost the cache hit rate are employed by scalability service nodes, users are likely to experience high latency if multiple database requests miss the cache on an HTTP request.

To reduce the client latency, it is desirable to either eliminate database requests or hide their latencies. There are several reasons why opportunities to do so appear in current Web applications. First, these applications are typically written for a traditional centralized setting, in which there is minimal overhead in issuing multiple database requests. Not expecting a distributed environment, application developers frequently do not optimize for the number of database requests the application issues. Second, application developers often abstract database values as objects in the program, a paradigm that is also adopted by Object Relational Mapping tools [13, 32]. If they need multiple values, they just issue multiple queries. Third, there are instances where it is easier for developers to express their main logic in the procedural language because it is closer to how the data is actually presented to the user. Consequently, they issue multiple, short queries, as in the example in Figure 5.

In this work we propose two transformations that rewrite the application code to either eliminate database requests or hide their latencies. Our first transformation, the MERGING transformation, eliminates queries by clustering related queries. Our second transformation, the NONBLOCKING transformation, hides the long latency in fetching query results, by overlapping the execution of queries.

Web applications are commonly written in a procedural language like Java or PHP whereas they issue database queries in a declarative language, typically SQL. Both transformations that we propose change the database queries as well as the application code surrounding them. They affect the program as a whole. Therefore we call them *holistic* transformations (Figure 2). To evaluate their effectiveness, we have applied it to three benchmark applications described in Section 4.1. While we currently applied them manually, we believe that the algorithms (described in Section 2.3 and Section 3.1) should be straightforward to automate in a source-to-source compiler [16, 25]. We also defer the detailed discussion of these two transformations and the related work to the technical report [19].

This paper makes the following contributions:

- We propose two holistic transformations to reduce the client latency in accessing a Web application, running on a distributed infrastructure.

- We discuss why opportunities for applying these transformations will continue to exist in Web applications and present algorithms for automating these transformations in a source-to-source compiler.

- Finally, we present extensive results of applying these two transformations on three Web benchmark applications—AUCTION, BBOARD, and BOOKSTORE, described in Section 4.1—both in the traditional centralized setting and a distributed setting. At least one transformation applies in 18.7%, 75.7%, and 59.4% of all dynamic interactions at runtime for the AUCTION, BBOARD and the BOOKSTORE, respectively. In a distributed setting, these transformations increase overall scalability by over 10% and latency for many interactions by over 50%.

**Roadmap.** Section 2 and 3 discuss the two transformations. Section 4 evaluates the effect of the two transformations on both latency and scalability. Section 5 summarizes the related work. Finally, Section 6 presents the summary.

# 2 The MERGING Transformation: Clustering Related Queries

We explain the MERGING transformation using an illustrative example. Consider the code fragment in Figure 3(a), which is taken from the AUCTION benchmark. The program issues several short related queries and then combines their results. In a DBSS setting, for each query that results in a cache miss at the DBSS node, the user must endure the long delay of accessing the home server database. Assuming a constant

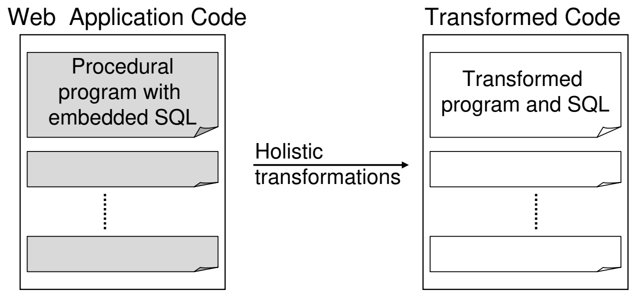



**Figure 2: The holistic transformations reduce the number of database queries that the Web application issues per HTTP request at runtime.**

```
$template:=SELECT from_user_id
           FROM comments
           WHERE to_user_id = ?;
$query:=set_params ($template, $to_id);
$result:=execute ($query);
foreach ($row in $result) {
   $from_id:=get_user_id ($row);
   $template:=SELECT user_name
              FROM users
              WHERE user_id = ?;
   $query:=set_params ($template, $from_id);
   $result2:=execute ($query);
}
```

(a) Original code

```
$template:=SELECT from_user_id, user_name
           FROM comments, users
           WHERE from_user_id = user_id
             AND to_user_id = ?;
$query:=set_params ($template, $to_id);
$result:=execute ($query)
```

(b) After the MERGING transformation

**Figure 3: A code fragment from the AUCTION application, showing the original code on the left, and the code after applying the MERGING transformation on the right. The code, an example of the Loop-to-join pattern, finds the names of users who have posted comments about a particular user. We focus on two base relations: `users` with attributes `user_id` and `user_name`, and `comments` with attributes `from_user_id` and `to_user_id`.**

hit rate at the DBSS cache, the client latency is proportional to the number of queries issued in an HTTP interaction. The MERGING transformation transforms the code to the equivalent code in Figure 3(b), merging all of the short inter-related queries into one join query. The program then needs to issue just one query instead of the previous $N + 1$ queries, assuming the loop is repeated $N$ times.

The remainder of this section discusses the effect we expect the MERGING transformation to have on the total work done by the system (Section 2.1), describes in more detail the code patterns for which we apply the transformation (Section 2.2), and describes our algorithms to implement it (Section 2.3).

## 2.1 Impact on the Total Work in the System

While it is possible for the MERGING transformation to either decrease or increase the total amount of work done in the system, we do not expect it to affect the total amount of work in the system. We use the term *work* to mean the use of any resources like disk I/O or CPU in the system. In most cases, we expect it to only change the division of work between the application and the database server(s), and not impact the total amount of work. This behavior is true for the example in Figure 3, which involves a simple one-to-one join operation. For a complete understanding of the consequences of this optimization, we next discuss instances in which applying this transformation might decrease or increase the total amount of work in the system.

```
$template1:=SELECT MAX(bid) FROM bids
            WHERE item_id = ?;
$query1:=set_params ($template1, $item_id);
$result1:=execute ($query1);
$template2:=SELECT COUNT(*) FROM bids
            WHERE item_id = ?;
$query2:=set_params ($template2, $item_id);
$result2:=execute ($query2);
```

(a) Original code

```
$template:=SELECT MAX(bid), COUNT(*)
           FROM bids
           WHERE item_id = ?;
$query:=set_params ($template, $item_id);
$result:=execute ($query);
```

(b) After the MERGING transformation

**Figure 4: An example of the merge-projection-predicates pattern, showing the original code on the left, and the code after applying the MERGING transformation on the right. The code fragment is a simplified version of the code from the AUCTION application, and finds the current maximum bid and the total number of bids for an item. We focus on the `bids` relation with the `bid` and the `item_id` attributes.**

Applying the MERGING transformation can decrease the total amount of work if the database is able to execute the queries more efficiently after applying the transformation, e.g., if the application code in Figure 3 involved a many-to-many join instead of an one-to-one join. The left hand side code will still implement a nested loop join in the application with a pre-determined outer table whereas on the right hand side, the database optimizer will be able to optimize the join based on the cardinality and selectivity estimations of the tables involved and the available indices. This reduction in work would be reflected in improved scalability in both the centralized as well as a distributed setting. Since deployed application code is unlikely to have such inefficiencies, we do not expect such opportunities to exist there. As expected, no opportunities for significantly reducing work were present in the three benchmark applications we used. Consequently, we did not see any measurable throughput improvements, i.e., improvements in number of interactions per second, due to the MERGING transformation in any of the benchmark applications.

Applying the MERGING transformation can increase the total amount of work in the system when one or more of the queries being merged is issued conditionally. For example, consider a slightly modified version of Figure 3. Imagine that the loop is executed only if the returned result contained at least ten rows. Then applying the transformation can increase the total amount of work if most of the returned results had fewer than ten rows. To decide whether to apply this transformation in such "speculative" situations or not, we use estimates of the relative costs of evaluating the query result and the frequencies with which the different queries are issued. In practice, we do not apply this transformation when doing so increases the total amount of work. Only once in our benchmark applications, we had to decide whether to apply this transformation speculatively or not: it occurred in the BBOARD benchmark and we decided to speculatively apply the transformation.

In the example in Figure 3, the MERGING transformation converted a loop in the application code to a database join. We call this pattern the "loop-to-join" pattern. In the next section we list all the different patterns we found in the three benchmark applications where the MERGING transformation applied.

## 2.2 Code Patterns Where the MERGING Transformation Applies

Based on our study of the three benchmark applications, we found three code patterns where the MERGING transformation applies.

**Loop-to-join.** The application first issues a query to get multiple values and then for each value (using a loop structure), issues another database query. This code pattern can be transformed to a single database join query, as in the example in Figure 3. Of the three patterns we found this pattern occurs the most frequently, occurring in all three benchmark applications that we study.

**Merge-projection-predicates.** The application issues multiple queries in succession that are identical except in the attributes they project. This code pattern can be transformed to a single database query

```
$template:=SELECT id, body FROM comments
          WHERE parent = 0 AND story = ?;
$query:=set_params ($template, $sid);
$result:=execute ($query);
push_comments (Stack, $result);
while (($comment:=pop (Stack)) != NULL) {
  print ($comment);
  $cid:=get_id ($comment);
  $template:=SELECT id, body FROM comments
            WHERE parent = ?  AND story = ?;
  $query:=set_params ($template, $cid, $sid);
  $result := execute ($query);
  push_comments (Stack, $result);
}
```

(a) Original code

```
$template:=SELECT id, body, parent
          FROM comments
          WHERE story = ?;
$query:=set_params ($template, $cid, $sid);
$result:=execute ($query);
```

(b) After the MERGING transformation

**Figure 5: An example of the merge-selection-predicates pattern, showing the original code on the left, and the code after applying the MERGING transformation on the right (We just show the database queries on the right). The simplified code fragment is from the BBOARD application, and shows all the comments on a story in a tree format. We focus on the `comments` relation with the `id`, `body`, `parent`, and `story` attributes.**

where the projection clause is a union of the projection clauses of the original queries. For instance, in the AUCTION benchmark example in Figure 4, a query to find the maximum bid on an item is followed by another query to find the number of bids for the same item.

For any merge-projection-predicates pattern, the MERGING optimization reduces the total work in the system. For the example in Figure 4, after the `MERGING` transformation, the database must lookup just one row instead of looking up the row twice, saving on both the disk I/O and CPU costs. While this pattern exists in the AUCTION and BBOARD benchmarks, we do not expect this pattern to occur frequently for a deployed Web application. This pattern might exist only when it is difficult to optimize away the pattern: e.g., (1) there is a significant time gap among the different component queries being issued, and (2) some component queries are issued speculatively.

**Merge-selection-predicates.** The application issues multiple queries in succession that are identical except in a selection clause. This code pattern can be transformed to a single database query where the differing selection clause is dropped and the attribute used in the dropped selection clause is added to the projection attributes. For example, in the BBOARD benchmark, when a user views a story, all the comments for the story are to be displayed in a tree format (The comments on a story can be viewed as a tree with the story being the root, each comment being a node of the tree, and comments which are replies to a particular comment, determining the children-parent relationships in the tree). In the original code, to achieve this task, a tree traversal is done, and at each tree node, a new query is issued to fetch the children comments. The issued query does a selection on the `parent` and the `story` attribute. Applying this transformation, all the comments on the story can be obtained using a single query: the issued query simply does a selection on the `story` attribute, as illustrated in Figure 5. We found this pattern only in the BBOARD application. It existed because the original code more closely reflects how the data is actually presented to the user.

## 2.3 Algorithm for Automating the MERGING Transformation

In this section we present an algorithm for automating the MERGING transformation, which should be straightforward to implement in a source-to-source compiler [16, 25]. As with any compiler transformation, the algorithm can bail out if it does not completely understand the program. Additionally, for ease of exposition, we assume that this transformation modifies the part of a program only up to the first update

```
SELECT user_name
FROM users
WHERE user_id = 5
```

(a) Invalidations in the original code

```
SELECT from_user_id, user_name
FROM comments, users
WHERE from_user_id = user_id AND to_user_id = ?
```

Invalidated if the query result contains the user_id 5

(b) Invalidations after applying the MERGING transformation

**Figure 6: Query results that are invalidated on an update with template as `UPDATE users SET user_name = ? WHERE user_id = ?` and `user_id` as 5, before and after applying the MERGING transformation. Since the MERGING transformation increases caching granularity, it leads to more invalidations, and consequently, less reuse of work.**

statement. The algorithm works by identifying the three code patterns: loop-to-join, merge-projection-predicates, and merge-selection-predicates, and then making appropriate changes in each case.

**Loop-to-join.** We build on the work done in optimizing nested queries over 25 years ago [14]. We first identify loops in the program. We then check if: (1) the loop iterates using the result of a previous query, (2) the loop issues a query in each iteration, and (3) the previous query is executed whenever the loop executes. Moreover, to avoid issuing speculative queries, we check if the loop is executed whenever the previous query is executed. Once the pattern is identified, we can use work done by [14] to replace the small queries by a merged query. Additionally, variables that used the result of any of the small queries must be reinitialized to use the result of the merged query.

**Merge-projection-predicates.** We check if (1) the second query is executed whenever the first query is executed (using control-flow-analysis [22]), and (2) the queries are identical except the projection predicates. If these pre-conditions are satisfied, the two queries can be merged as in Figure 4. Finally, variables that used the result of any of the queries being merged must be reinitialized to use the result of the merged query.

**Merge-selection-predicates.** We check if: (1) the outer query is executed whenever the loop is executed and vice versa (using control-flow-analysis), and (2) the queries are identical, except one selection clause. If these pre-conditions are met, the query is transformed as per the example in Figure 5. Finally, variables that used the result of any of the queries being merged must be reinitialized to use the result of the merged query.

## 2.4 Other Tradeoffs

There are other advantages and disadvantages of applying the MERGING transformation, beyond just reducing latencies.

**Interactions with query result caching.** For our DBSS setting, in which the query results are cached, the MERGING transformation, which merges short related queries into a long query, increases the caching granularity. Increasing the caching granularity implies that on an invalidation, a larger cache entry, which is more expensive to compute, is invalidated. For example, in Figure 6, rather than invalidating the result of a simple lookup query, in the transformed code, the update `UPDATE users SET user_name = ? WHERE user_id = 5` invalidates the result of the join query, a query result which is more expensive to compute.
Because of the possibility of increased invalidation overhead, the latency reduction due to this transformation must be weighed carefully against the increased invalidation, before applying this transformation in a setting that caches query results. For our benchmark applications, the increase in invalidations was minimal, and so we always decided to apply this transformation.

# 3 The NONBLOCKING Transformation: Prefetching Query Results

After issuing a database query, a Web application waits for the query result. In some cases this wait is unnecessary because the next database query does not depend on the answer to the current query. In such

```
$template1:=SELECT item_name
            FROM items i1, items i2
            WHERE i1.id = i2.related
              AND i2.id = ?;
$query1:=set_params ($template1, $id);
$result1:=execute ($query1);
$template2:=SELECT user_name FROM users
            WHERE user_id = ?;
$query2:=set_params ($template1, $user_id);
$result2 := execute ($query2);
```

(a) Original code

```
$template2:=SELECT user_name FROM users
            WHERE user_id = ?;
$query2:=set_params ($template2, $user_id);
execute_non_blocking ($query2);
$template1:=SELECT item_name
            FROM items i1, items i2
            WHERE i1.id = i2.related
              AND i2.id = ?;
$query1:=set_params ($template1, $id);
$result1:=execute ($query1);
$result2:=execute ($query2);
```

(b) After the NONBLOCKING transformation

**Figure 7: A simplified code fragment from the BOOKSTORE application, which finds the name of an item related to the item the user is viewing and the name of the user, given her id. We focus on two base relations: `users` with attributes `user_id` and `user_name`, and `items` with attributes `item_id`, `item_name`, and `related`. The left hand side shows the original code, while the right hand side shows the code after applying the NONBLOCKING transformation.**

cases, the client latency can be greatly reduced by overlapping the query executions. In this section we present the NONBLOCKING transformation, which can overlap executions of multiple queries that do not depend on each other by "prefetching" query results.

To illustrate how this transformation can be applied to a code fragment, consider Figure 7, which shows two functionally equivalent code fragments from the BOOKSTORE application. Figure 7b shows the code after applying the NONBLOCKING transformation. The method *execute_non_blocking* does not block and only serves to populate the cache with the query result. If the latency of the first database request is $t_a$ and the latency of the second request is $t_b$, this transformation reduces the overall latency from $t_a + t_b$ to $\max\{t_a, t_b\}$.

Ideally, whenever the program that dynamically generates the HTTP response starts running, we would like to issue prefetch requests for all queries that the program will issue during its execution. However, issuing a prefetch request for each query, at the start of the program's execution, is not always possible because: (1) one of the parameters of the query may be the result of a previous query, (2) the query may be conditionally issued and the condition uses the result of a previous query, and (3) there may be an update statement before the query that may affect the query result.

Formally, each program of a Web application can be represented as a directed acyclic graph, where the nodes are database accesses, and there is an edge between two nodes if one node has to be executed after the other node for correctness. Given this directed acyclic graph, a database access can be issued as soon as all database accesses that are its ancestors in the directed acyclic graph have completed. With this formulation, the client latency can be reduced significantly.

This transformation normally does not change the amount of work that must be done, it just improves the scheduling of the work. However, if a prefetch is issued for a query that is conditionally executed, the result of the prefetch will not always be used. While issuing such "speculative" prefetches increases the total work in the system, it allows trading off reduced latency for extra work. For our evaluation, we issued speculative prefetches whenever possible because more often than not, the result of the prefetches would be useful. The prefetches were wasted only when an error occurred in the execution of a query – an infrequent occurrence for the applications we studied.

Application of this transformation can be automated – we outline an algorithm for automatically applying this transformation in Section 3.1. Finally, in Section 3.2 we discuss implementation issues relating to this transformation.

### 3.1 Algorithm for Automating the NONBLOCKING Transformation

In this section we present an algorithm for automating the NONBLOCKING transformation, which should be straight-forward to implement in a source-to-source compiler [16, 25]. As with any compiler transformation, the algorithm can bail out if it does not completely understand the program. For ease of exposition, we make two assumptions. First, similar to the assumption in Section 2.3, we assume that the algorithm modifies a program of the application code only up to the first update statement. The work on query-update-independence [17] can be used to remove this restriction. Second, we assume that there are no edges in the database dependence graph of the program, as defined before. The dependence graph for a program can be computed using data-flow techniques [22]. After allowing for the assumptions, the algorithm is:

1. Let $\mathcal{Q}$ be the list of all the queries in the program that appear before any database update statements.
2. For every query $q \in \mathcal{Q}$, place a copy of all variable initializations that query $q$ uses directly or indirectly (through some other variable) at the beginning of the program. Next, put a *non-blocking-execute* function call for the query $q$ after all these variable initializations.

### 3.2 Implementation Issues

We now describe two issues regarding the implementation of the prefetch mechanism. We evaluate these two issues later in Section 4.7.

**Prefetching support in the runtime layer.** For prefetches to work, the runtime layer must support the execution of non-blocking queries. Furthermore, the DBSS node needed to maintain database connections in order to fulfill prefetch requests. The number of connections to use for this purpose is a configurable parameter. In our implementation, the DBSS node maintains a fixed number of total connections to the home server database as an admission control mechanism to avoid overloading the database. Since we did not want either the regular database requests or the prefetch requests to be unnecessarily delayed, we dynamically allocated the connections used for fulfilling prefetch requests, depending on how many were available after fulfilling the regular database requests. Of course, if the query result that a prefetch was requesting was already present in the cache, we just filtered the prefetch.

**Timing of prefetches.** To hide the latency due to a miss, it is critical that the prefetches be issued at the right time. On the one hand, if the prefetch is issued late, the query result will not have arrived in the cache by the time the query is issued. On the other hand, if the prefetch is issued well in advance, the query result might be invalidated by a later update. In both cases, the latency in obtaining the subsequent query result remains. In our experiments, we issued the prefetches at the earliest possible time: when the Web application started its execution. Even using this policy, none of our prefetches were invalidated before the query result was used.

## 4 Evaluation

We evaluate the two transformations–MERGING and NONBLOCKING–by applying them to three benchmark applications. We describe these benchmark applications in Section 4.1, and we describe our experimental methodology in Section 4.2. In Section 4.3 and Section 4.4 we evaluate the effects of these transformations on scalability and latency, both in the traditional centralized setting as well as the DBSS setting. Next, we list the frequencies with which the two transformations apply to our benchmark applications in Section 4.5. We finally present the detailed "coverage" results of the two transformations in Section 4.6 and Section 4.7.

### 4.1 Benchmark Applications

We used three publicly available Web benchmark applications that extensively use a database and represent real-world applications: RUBiS [26], an auction system modeled after `ebay.com`, RUBBoS [27], a simple

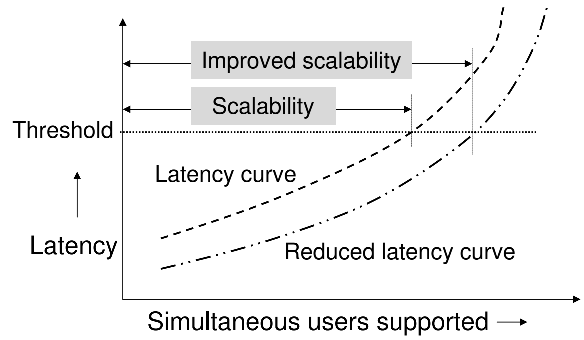



**Figure 8: Impact of latency on scalability.**

bulletin-board-like system inspired by `slashdot.org`, and TPC-W [36], a transactional e-Commerce application that captures the behavior of clients accessing an online book store. We used Java implementations of these applications. We refer to these applications as AUCTION, BBOARD, and BOOKSTORE, respectively.

## 4.2 Experimental Methodology

We used similar methodology as in [20]. We report results for a simple three-node configuration: a home server that runs MySQL4 [23] as its database management system, a DBSS node that provides answers to database queries using its store of the cached query results, and a CDN node providing the functionality of a web and application server, running on Emulab [37]. Cached query results were kept consistent with the home server's database using non-transactional invalidation of cached query results.

The home server machine had an Intel P-III 850 MHz processor with 512 MB of memory, while the DBSS node and the CDN node had an Intel 64-bit Xeon processor with 2GB of memory. In all experiments, the home server and DBSS node were connected by a high latency, low bandwidth duplex link (100 ms latency, 2 Mbps). The CDN node and the DBSS were connected by a low latency, high bandwidth link (5ms latency, 20 Mbps). Each client was connected to the DBSS node by a low latency, high bandwidth duplex link (5 ms latency, 20 Mbps).

Because the overhead for emulating clients is low, a single additional Emulab node was used to emulate all clients. As in the TPC-W [36] specification, clients simulate human usage patterns by issuing an HTTP request, waiting for the response, and pausing for a *think time* before requesting another Web page; the think time is drawn from a negative exponential distribution with a mean of seven seconds.

Each experiment ran for ten minutes, and the DBSS node started with a cold cache each time. *Scalability* was measured as the maximum number of users that could be supported while keeping the response time below two seconds for 90% of the HTTP requests.

## 4.3 Scalability Impact of the Transformations

So far, we have focused on how the MERGING and NONBLOCKING transformations reduce the latency of an HTTP request in a distributed setting. However, as Figure 8 shows, client latency varies depending on how many simultaneous users are supported. In our work, we use scalability as the single unifying metric. We measure scalability as the number of simultaneous users that can be supported with latency remaining under a threshold. Figure 8 shows how a reduction in latency improves the scalability metric. Because of the reduced latency, the scalability in the figure increases from "scalability" to "improved scalability."

Figure 9 plots the scalability of an application as a function of the code transformations used, for all three benchmark applications. The y-axis plots scalability, measured as specified in Section 4.2. On the x-axis, we consider five cases: one corresponding to not using the DBSS, one corresponding to using the DBSS but no transformations, and the other three corresponding to using either or both the transformations.

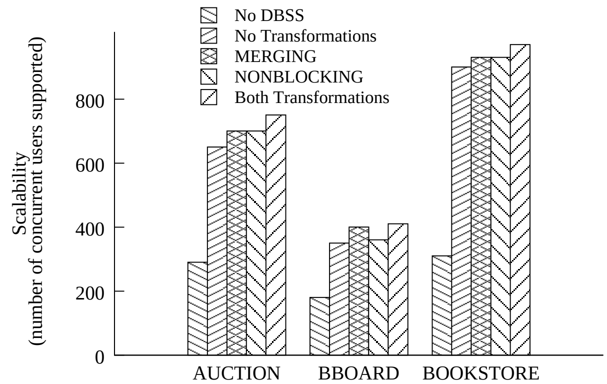



**Figure 9: Scalability impact of the transformations. For comparison, we include the scalability numbers without a DBSS, the leftmost bar for each application.**


**Figure 10: Impact of the** MERGING **and** NONBLOCKING **transformations on latency. We show the average latency for two dynamic interactions in the** BBOARD **benchmark. The graph shows that the** MERGING **transformation has a significant impact on the average latency.**

For clarity, we did not plot the bar for using the transformations in the centralized setting. The results were identical to not using the transformation in the centralized setting, showing that these transformations do not have any effect on the performance in a centralized setting. In all applications, using a DBSS significantly increased scalability. Turning on the transformations further improved scalability. The MERGING transformation has the most effect on the BBOARD application and the least effect on the BOOKSTORE application. On the other hand, the NONBLOCKING transformation has the most effect on the BOOKSTORE benchmark and the least effect on the BBOARD benchmark.

The two transformations: MERGING and NONBLOCKING, are complementary. While the MERGING transformation can be applied only when the queries are *related*, the NONBLOCKING transformation can be applied only when the queries are *not related*. Consequently, we expect that both transformations must be applied for the best scalability. Figure 9 shows that the scalability indeed increases the most when both transformations are applied simultaneously.

## 4.4 Latency Impact of the Transformations

Even though a single unifying metric like 'scalability' is helpful in comparisons, it is not always able to correctly portray the magnitude of a change. The scalability improvements due to these transformations, at around 10%, seem minor. To understand the results better, we plot the average latencies for two common interactions in the BBOARD application. (We chose BBOARD because the transformations have the greatest effect on latency in the BBOARD benchmark.)

Figure 10 shows the effect of the transformations on the average latency for the two interactions, executing

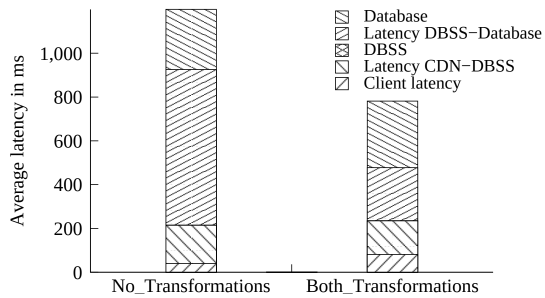



**Figure 11: Impact of the two transformations on the average latency of a dynamic interaction in the BBOARD application, executing in a DBSS setting.**

**Table 1: Runtime HTTP interactions in which the MERGING and NONBLOCKING transformation apply. The "either" column represents interactions in which at least one of the two transformations apply. The "static" column represents interactions in which a static HTML file is returned (clearly, neither transformation can apply to such interactions).**

| | % of runtime HTTP interactions | | | |
|---|---|---|---|---|
| Application | Static | MERGING applied | NONBLOCKING applied | Either |
| AUCTION | 15.9% | 15.2% | 3.8% | 15.7% |
| BBOARD | 7.4% | 69.8% | 28.5% | 70.1% |
| BOOKSTORE | 0.0% | 0.8% | 58.6% | 59.4% |

in a DBSS setting. Applying both transformations reduces latency by over 50%. Of the two transformations, the MERGING transformation causes a greater reduction in latency. The larger impact of the MERGING transformation agrees with the scalability results in Figure 9. Figure 10 also shows that both transformations are complementary: applying both reduces the latency more than applying either of them alone.

Figure 11 evaluates the impact of the two transformations on the average latency of a dynamic interaction in the BBOARD application, executing in a DBSS setting. The latency consists of five components: the client latency including the execution time at the CDN, the network latency from the CDN to the DBSS, the time spent at the DBSS, the latency from the DBSS to the database, and the time spent at the database. Almost all the latency decrease is due to a reduction in the network latency from the CDN to the DBSS.

A latency decrease often does not result in a commensurate scalability increase. For example, while these two transformations reduce the average latency by 38% for the BBOARD application (Figure 11), they increase the scalability by only about 10% (Figure 9). To understand this difference, we need to refer back to Figure 8. In the figure, the latency decrease and the scalability increase are related by the slope of the latency-users curve. This slope governs how much the scalability will increase due to a decrease in latency. Steeper the curve, more is the slope, and less is the impact on scalability due to any reduction in latency. There is another factor that contributes to why a latency decrease does not result in a commensurate scalability increase. These transformations sometimes tend to impact low-latency interactions more than high-latency interactions. For the ViewStory interaction of the BBOARD application, while the latency reduction for low-latency interactions[1] was 78%, the latency reduction for the high-latency interactions was only 58%.

**Table 2: Frequency of occurrence of different patterns in which the MERGING transformation applies.**

| Application | Total query templates | % of query templates where the patterns apply: *Loop-to-join* | *Merge-projection-predicates* | *Merge-selection-predicates* |
|---|---|---|---|---|
| AUCTION | 28 | 25.0% | 3.6% | 0.0% |
| BBOARD | 38 | 26.3% | 13.2% | 5.3% |
| BOOKSTORE | 28 | 7.1% | 0.0% | 0.0% |

**Table 3: Average number of database queries per dynamic HTTP response for the three benchmarks. For our benchmark applications, the MERGING transformation does not affect the cache hit ratio.**

| Application | *Cache hit ratio* | Average database queries per dynamic HTTP interaction: *original code* | *after* MERGING | *% decrease* |
|---|---|---|---|---|
| AUCTION | 57.4% | 2.6 | 2.1 | 19% |
| BBOARD | 75.5% | 9.1 | 1.9 | 79% |
| BOOKSTORE | 66.4% | 1.78 | 1.77 | 1% |

## 4.5 Applicability of the Transformations

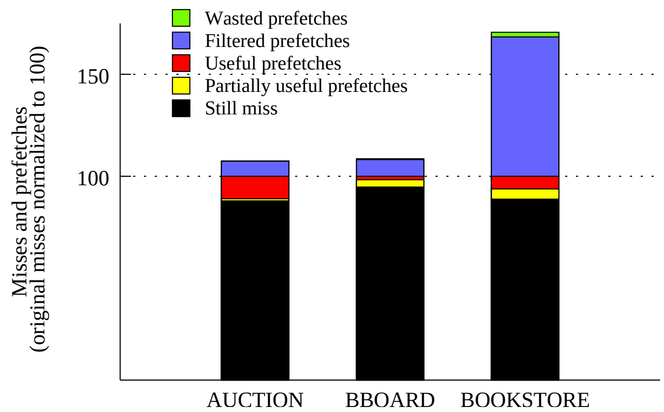
Table 1 lists the percentage of runtime HTTP interactions in which these transformations apply. The "either" column represents interactions in which at least one of the two transformations apply. The "static" column represents interactions in which a static HTML page is returned. Clearly, neither transformation can apply to such interactions. Even after including the static interactions (interactions which return an HTML file), one of these transformations applied to over 15%, 70%, and 59% of all runtime HTTP interactions for the AUCTION, BBOARD, and the BOOKSTORE benchmarks, respectively. For the BBOARD application, the MERGING transformation applies to over 69% of all HTTP interactions; this high percentage is one of the reasons why the MERGING transformation is particularly effective in increasing scalability (Figure 9) and reducing latency (Figure 10) of the BBOARD application. A similar argument can be made for the NONBLOCKING transformation and the BOOKSTORE application.

## 4.6 Coverage of the MERGING Transformation

Table 2 lists the total number of query templates per benchmark application and the number of times we could find the different patterns described in Section 2.2. We found the maximum number of patterns in the BBOARD benchmark, where the MERGING transformation could be applied to almost half of the query templates. In contrast, the BOOKSTORE benchmark had the fewest opportunities for applying this transformation. As for the patterns, the most frequently occurring pattern was loop-to-join, while the most uncommon pattern was merge-selection-predicates.

Table 3 lists the average number of database queries per dynamic HTTP response for all three benchmark applications both before and after applying the MERGING transformation, and computes the percentage decrease due to the transformation. The maximum decrease, 79%, occurs for the BBOARD benchmark and the minimum decrease, 1%, occurs for the BOOKSTORE benchmark. These results are in line with the results

[1] All interactions that had a latency below the threshold were categorized as low-latency interactions. The latencies were measured after the two transformations were applied.




| Application | Original misses normalized to 100% | | | Filtered Pfs | Wasted Pfs |
|---|---|---|---|---|---|
| | Still miss | Partial Pfs | Useful Pfs | | |
| AUCTION | 87.9 | 11.0 | 1.1 | 7.4 | 0.1 |
| BBOARD | 94.7 | 1.7 | 3.6 | 8.2 | 0.4 |
| BOOKSTORE | 88.8 | 6.2 | 5.0 | 68.4 | 2.2 |

**Figure 12: Impact of the NONBLOCKING transformation on the total number of misses, for the three benchmark applications. We use 'pfs' as a short-hand for prefetches.**

in Table 2 where the MERGING transformation applies most to the query templates of BBOARD benchmark, and least to the query templates of the BOOKSTORE benchmark. The table also provides the cache hit rates for the benchmark applications. From the cache hit rates, the average number of round trips from the DBSS node to the back-end database node that the MERGING transformation saves can be easily computed: 0.21, 1.76, and 0.003 round-trips are saved for the AUCTION, BBOARD, and BOOKSTORE benchmarks. These huge savings for the BBOARD application is reflected in Figure 11: the average time waiting to complete a database request decreases by almost 400 ms.

## 4.7 Coverage of the NONBLOCKING Transformation

Figure 12 plots how effective the NONBLOCKING transformation is in hiding the cache misses. The y-axis plots the misses and prefetches for each of the benchmarks (The original misses have been normalized to 100). After prefetches are issued, these misses either still remain a miss (*still miss*), meaning no prefetch was issued for them, or the prefetches were completely (*useful prefetches*) or partially useful (*partially useful prefetches*) in hiding the latency of a miss. Some prefetches were filtered by the caching layer because the query result was present in the cache (*filtered prefetches*). The final category was of those prefetches that were issued speculatively, and never used (*wasted prefetches*). This category of prefetches wastes bandwidth and CPU cycles. As the figure shows, the fraction of such prefetches is fairly small, which justifies our decision for issuing speculative prefetches in Section 3.

From Table 1 we expect many prefetches to be issued for the BOOKSTORE application. Figure 12 shows that while this is true, most of the prefetches turn out to be useless since they are filtered by the caching layer. Still, this transformation is able to hide latency for around 10% of the misses. For the AUCTION application, even though fewer prefetches are issued, it is still able to hide latency for around 10% of the misses. For the BBOARD application, fewest prefetches are issued, and the transformation is able to hide latency for only about 4% of the misses. The scalability improvement due to this transformation depends to some degree on the percentage of misses that the prefetches are able to hide: whereas we see a small impact of this transformation on the scalability of the AUCTION and the BOOKSTORE applications, the impact is almost zero on the scalability of the BBOARD application (Figure 9).

# 5 Related Work

Related research can be classified into two main areas: (1) prior work related to the MERGING transformation, and (2) prior work related to the NONBLOCKING transformation. We discuss each in turn.

## 5.1 Work Related to the MERGING Transformation

The work closest to our MERGING transformation is Cassyopia [30], a vision paper that proposes the use of compiler techniques for clustering system calls so that the overhead of crossing address spaces is reduced. Our technique is fundamentally similar to Cassyopia: we want to reduce the latency due to multiple database queries by clustering these database queries. However, there are significant differences. First, the domains are different: our work seeks to hide the overhead of network latency whereas their work seeks to hide the overhead of context switching between processes. Second, we identify important patterns where the MERGING transformation can be applied. Third, we argue why such patterns will continue to exist in future Web applications.

Most database vendors support stored procedures [35] which allow applications to invoke a block of procedural and declarative code at the database. Our approach of merging queries has several key advantages over using stored procedures. First, it is significantly harder for the database to optimize the execution of queries that use stored procedures than to optimize the execution of SQL queries [6, 7]. Second, it is significantly harder to maintain the consistency of a cache containing results of stored procedures. (If the results of stored procedures are not cached, no work is offloaded from the home server database.)

Work on optimizing the execution of nested queries [11] has mostly focused on decorrelation techniques [10, 14], which try to transform a given nested query into a form that does not use the nested sub-query construct. Guravannavar et al. [12] propose improved nested iteration methods as an alternative to decorrelation. Decorrelation techniques enable the query optimizer to use better plans such as hash join for evaluating the nested query. The MERGING transformation for the loop-to-join pattern essentially performs decorrelation. Compared to nested query optimization, the differences are: (1) the transformation is carried out by the compiler instead of the database optimizer, and (2) the primary motivation for the transformation is to reduce the number of round-trips an application needs to make to access its data instead of improving the performance.

Some database optimizers implement multi-query optimization [31, 33], where they identify common sub-expressions in a sequence of queries to speed up all the queries. However, to be applicable, these optimizers need to see a batch of queries at once—a model different from how the Web applications normally work, where they have at most one outstanding query. Moreover, in a multi-query optimization setting, the database does most of the work, while for the MERGING transformation, the compiler does most of the work: it needs to understand database queries as well as the procedural code surrounding them, identify patterns in the application code, and then transform the patterns accordingly.

Object relational mapping tools like Java's Hibernate [13] and Ruby-on-Rails's Active Records [32] result in Web application code that issue several simple, related queries, all of which can be merged into a single query. Both Hibernate and Active Records provide hooks to replace these related queries by a single query: Hibernate users can write queries in HQL, the Hibernate Query Language, whereas Active Record users can write explicit SQL queries. Our work seeks to automate this process of merging related queries.

The MERGING transformation can be viewed as a repartitioning of work between the application and database server. Yang et al. [38] present techniques to automatically partition a Web application into client and server parts, in order to optimize the application's response time. To be applicable, the Web application must be written in a custom language Hilda [39]. Similarly, the Abacus system [3] automates the placement of objects written in a custom language. In contrast, the MERGING transformation does not require applications to be rewritten in a custom language; it can be applied directly to legacy applications.

### 5.2 Work Related to the NONBLOCKING Transformation

The NONBLOCKING transformation aims to hide the latency of a miss in the DBSS cache by prefetching the query result. Much prior work has been done on prefetching. A commonly used technique is to issue prefetches by predicting, based on past accesses, what data is needed next. This technique has been used widely for hiding latency of page faults in virtual memory systems [8], reducing access times of static Web pages [9, 24, 28], and improving the overall performance of file systems [15]. There are two main drawbacks to this technique. First, a certain number of accesses are necessary to bootstrap this prediction mechanism. No prefetches can be issued while this bootstrapping is in progress. Second, this technique no longer remains useful when the access pattern changes.

Patterson et al. [29] propose an alternative approach to predicting based on past accesses: applications are manually modified to generate hints about their access patterns. Follow-up work by Chang et al. [5] automates this process of generating the access hints. Similarly, Mowry et al. [21] and Brown et al. [4], show how compiler analysis integrated with simple operating systems support and a runtime layer, to adjust to dynamic conditions, can be used to effectively manage physical memory for out-of-core applications. The compiler analysis was used to insert prefetch and release instructions in the application code. While the goal of using application-specific knowledge to hide latencies is the same in these efforts as in our system, we focus on a different domain than virtual memory references and file reads and writes. Plus, their work did not require analysis of SQL code embedded in a program.

## 6 Summary

To meet their scalability needs, Web applications increasingly use a distributed server infrastructure. Inevitably, the network latency between the application and database servers increases in such settings. Two examples of such settings are: (i) a DBSS setting [18, 20] where different third party services may manage the application server(s) and the database server(s), (ii) a shared Web-service hosting scenario where the application and the database server typically run on separate clusters of machines, and typical latencies are between 16ms and 20ms [34]. Since a single HTTP request of a dynamic Web application typically results in multiple database queries, even a slight increase in the latency between the application and database server(s) increases the client latency significantly. In this work we proposed two holistic transformations—MERGING and NONBLOCKING—which can be implemented in a source-to-source compiler [16, 25]. These transformations reduce the latency by either clustering related queries or overlapping query execution. By manually inspecting our application code, we found opportunities to apply these transformations in 18.7%, 75.7%, and 59.4% of all dynamic interactions at runtime for the AUCTION, BBOARD, and the BOOKSTORE, respectively. These transformations had almost no impact on the scalability in a centralized setting. However, in a distributed setting, these transformations increase scalability by over 10% and reduce latency by over 50% in many cases. These two transformations will continue to be useful as the two trends— using distributed infrastructures and issuing more database requests per HTTP requests—continue.

## References

[1] Akamai Technologies Inc. and Jupiter Research Inc. Akamai and Jupiter Research identify '4 seconds' as the new threshold of acceptability for retail web page response times. `http://www.akamai.com/html/about/press/releases/2006/press_110606.html`.

[2] Akamai Technologies Inc. and Quocirca. Akamai and Quocirca identify '4 second' performance threshold for European web-based enterprise applications. `http://www.edgejava.net/html/about/press/releases/2007/press_110707.html`.

[3] K. Amiri, D. Petrou, G. R. Ganger, and G. A. Gibson. Dynamic function placement for data-intensive cluster computing. In *USENIX Annual Technical Conference*, 2000.

[4] A. D. Brown and T. C. Mowry. Taming the memory hogs: using compiler-inserted releases to manage physical memory intelligently. In *Proc. OSDI*, 2000.

[5] F. Chang and G. A. Gibson. Automatic I/O hint generation through speculative execution. In *Proc. OSDI*, 1999.

[6] S. Chaudhuri. An overview of query optimization in relational systems. In *Proc. PODS*, 1998.

[7] S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. *ACM Transactions on Database Systems*, 24(2), 1999.

[8] K. M. Curewitz, P. Krishnan, and J. S. Vitter. Practical prefetching via data compression. In *Proc. SIGMOD*, 1993.

[9] L. Fan, P. Cao, W. Lin, and Q. Jacobson. Web prefetching between low-bandwidth clients and proxies: potential and performance. In *Proc. SIGMETRICS*, 1999.

[10] R. A. Ganski and H. K. T. Wong. Optimization of nested SQL queries revisited. *SIGMOD Record*, 16(3), 1987.

[11] G. Graefe. Executing nested queries. In *Conference on Database Systems for Business, Technology and the Web*, 2003.

[12] R. Guravannavar, H. S. Ramanujam, and S. Sudarshan. Optimizing nested queries with parameter sort orders. In *Proc. VLDB*, 2005.

[13] Hibernate. Relational persistence for Java and .NET. `http://www.hibernate.org`.

[14] W. Kim. On optimizing an SQL-like nested query. *ACM Transactions on Database Systems*, 7(3), 1982.

[15] T. M. Kroeger and D. D. E. Long. Predicting file system actions from prior events. In *Proc. USENIX Annual Technical Conference*, 1996.

[16] L. J. Hendren et al. Soot: a Java optimization framework. `http://www.sable.mcgill.ca/soot/`.

[17] A. Y. Levy and Y. Sagiv. Queries independent of updates. In *Proc. VLDB*, 1993.

[18] A. Manjhi, A. Ailamaki, B. M. Maggs, T. C. Mowry, C. Olston, and A. Tomasic. Simultaneous scalability and security for data-intensive web applications. In *Proc. SIGMOD*, 2006.

[19] A. Manjhi, C. Garrod, B. M. Maggs, T. C. Mowry, and A. Tomasic. Holistic query transformations for dynamic web applications. Technical Report CMU-CS-08-160, Carnegie Mellon University, 2008.

[20] A. Manjhi, P. B. Gibbons, A. Ailamaki, C. Garrod, B. M. Maggs, T. C. Mowry, C. Olston, and A. Tomasic. Invalidation clues for database scalability services. In *Proc. ICDE*, 2007.

[21] T. C. Mowry, A. K. Demke, and O. Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proc. OSDI*, 1996.

[22] S. S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[23] MySQL AB. MySQL database server.

[24] A. Nanopoulos, D. Katsaros, and Y. Manolopoulos. A data mining algorithm for generalized web prefetching. *IEEE Transactions on Knowledge and Data Engineering*, 15(5), 2003.

[25] ObjectWeb Consortium. ASM. `http://asm.objectweb.org`.

[26] ObjectWeb Consortium. Rice University bidding system. `http://rubis.objectweb.org/`.

[27] ObjectWeb Consortium. Rice University bulletin board system. `http://jmob.objectweb.org/rubbos.html`.

[28] V. N. Padmanabhan and J. C. Mogul. Using predictive prefetching to improve World Wide Web latency. *SIGCOMM Comput. Commun. Rev.*, 26(3), 1996.

[29] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proc. SOSP*, 1995.

[30] M. Rajagopalan, S. K. Debray, M. A. Hiltunen, and R. D. Schlichting. Cassyopia: compiler assisted system optimization. In *HOTOS'03: Proceedings of the 9th conference on Hot Topics in Operating Systems*, 2003.

[31] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. *SIGMOD Record*, 29(2), 2000.

[32] Ruby on Rails. Active Records. `http://www.rubyonrails.org`.

[33] T. K. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems*, 13(1), 1988.

[34] Simple measurements on the infrastructure of Dreamhost, a leading Web-hosting company. `http://www.dreamhost.com/`.

[35] M. Stonebraker, J. Anton, and E. Hanson. Extending a database system with procedures. *ACM Transactions on Database Systems*, 12(3), 1987.

[36] Transaction Processing Council. TPC-W specification, version 1.7.

[37] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. OSDI*, 2002.

[38] F. Yang, N. Gupta, N. Gerner, X. Qi, A. Demers, J. Gehrke, and J. Shanmugasundaram. A unified platform for data driven web applications with automatic client-server partitioning. In *Proc. WWW*, 2007.

[39] F. Yang, J. Shanmugasundaram, M. Riedewald, and J. Gehrke. Hilda: A high-level language for data-driven web applications. In *Proc. ICDE*, 2006.