The MACE Approach for Caching Mashups

Osama Al-Haj Hassan, Lakshmish Ramaswamy and John A. Miller
Computer Science Department, University of Georgia, Athens, GA 30602, USA
{hasan, laks, jam}@cs.uga.edu

ABSTRACT:

In recent years, Web 2.0 applications have experienced tremendous growth in popularity. Mashups are a key category of Web 2.0 applications which empower end-users with a highly personalized mechanism to aggregate and manipulate data from multiple sources distributed across the Web. Surprisingly, there are few studies on the performance and scalability aspects of mashups. In this paper, we study caching-based approaches to improve efficiency and scalability of mashups platforms. This paper presents MACE – a caching framework specifically designed for mashups. MACE embodies three major technical contributions. First, we propose a mashup structure-aware indexing scheme that is used for locating cached data efficiently. Second, we build taxonomy awareness into our system and provide support for range queries to further improve caching effectiveness. Third, we design a dynamic cache placement technique that takes into consideration the benefits and costs of caching at various points within mashups workflows. We report a set of experiments studying the effectiveness of the proposed mechanisms.

KEY WORDS:
Web 2.0, Mashups, Caching, B+ trees

INTRODUCTION

Web 2.0 is drastically changing the landscape of the World Wide Web by empowering end-users with new tools for enhanced interaction and participation. Among Web 2.0 applications, mashups (Programmable Web, 2009) are becoming increasingly popular as they provide end-users with high degrees of personalization. Conceptually, mashups are Web services that are created by end-users who also consume their results. They offer high level of personalization because they are developed by end-users themselves as opposed to regular Web services which are designed by professional developers. (Throughout this paper, the term Web services refers to traditional Web service model in which a service provider creates and deploys Web services).

Mashups basically collect data from several data sources distributed across the Web, which would then be aggregated, processed, and filtered to generate output which would be sent to end-user. Several mashup platforms exist on the Web including Yahoo Pipes (Yahoo Inc., 2007) and Intel MashMaker (Intel Corp., 2007). The unique features of mashups, represented in high personalization and end-user participation, pose new scalability challenges. First, giving end-users the privilege of designing their own mashups causes a mashup platform to host a large volume of mashups which implies that the scalability requirement for mashup platforms is much higher when compared with Web services portals. Second, large volumes of mashups also imply that the opportunities for data reuse are minimal unless specialized mechanisms to boost data reuse are adopted. Third, mashups fetch data from several data sources across the Web; these data sources differ in their characteristics and their geographical distribution. Finally, mashups may be designed by non-technical-savvy end-users, and hence they are not necessarily optimized for performance. Unfortunately, scalability and performance challenges of mashups received little attention from the research community. Although there have been some studies on the
performance of traditional orchestrated Web service processes (Chandrasekaran, 2003), to our best knowledge, no studies have investigated efficiency and scalability aspects of mashups or proposed techniques to tackle them. This paper explores caching as a mechanism to alleviate the scalability challenges of mashups. Caching is a proven strategy to boost performance and scalability of Web applications. For example, Web content delivery and Web services have long adopted caching (Wang, 1999). Several caching techniques have been specifically developed for Web services (Tatemura, 2005; Terry, 2003). However, most of these techniques cannot be directly used for mashups because of some significant differences between Web services and mashups (listed in Section 2). We need a caching framework that not only takes into account the structural characteristics of mashups but is also adaptive to the various dynamics of the mashup platform.

Contributions

This paper describes the design and evaluation of MACE (mashup cache) - a server-side cache framework for the mashup domain. MACE is sensitive to the structural composition of the mashups, and it can store results at intermediate stages of mashup workflows. The design of the MACE framework embodies three original contributions.

- We design a mashup structure-aware scheme for indexing cached data which enables MACE to efficiently discover whether any of the currently cached data can be reused in the execution of a newly created mashup.
- We incorporate taxonomy-awareness and provide support for range queries to further increase reuse of cached data.
- We present a dynamic cache point selection scheme that estimates the benefits and costs of caching data at different stages of mashup trees. Our approach selects a set of points that collectively maximize the benefit-to-cost ratio of caching data at those points.

We have evaluated the MACE framework through several sets of experiments. Our experiments demonstrate that MACE significantly improves the scalability and efficiency of mashup platforms. In the following sections, we discuss our motivations for doing this research and describe our mashup model and architecture. We follow that by proposing our index structure that eases accessing mashups. We then introduce two techniques to increase the utilization of our index, namely support of range queries and taxonomy awareness. Further, we describe our caching scheme for mashups. Finally, we investigate our model by discussing a detailed set of experiments.

MACE: BACKGROUND AND DESIGN OVERVIEW

In this section, we outline the challenges that need to be addressed in developing an effective cache framework for the mashup domain. Further, we develop a formal model for mashups which serves as an analytical framework for studying their performance.

Motivation

Although mashups are in-essence Web services, there are significant differences between mashups and traditional Web services. As exemplified by Yahoo Pipes (Yahoo Inc., 2007), mashup platforms typically host several thousands of distinct mashups, whereas the number of distinct Web services in a typical Web services portal is relatively small. The frequency of
Table 1. Comparison between Web service portals and mashup platforms

<table>
<thead>
<tr>
<th>Comparison Criteria</th>
<th>Web Service Portals</th>
<th>Mashup Platforms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Created By</td>
<td>Developers</td>
<td>End-users</td>
</tr>
<tr>
<td>Number of distinct Web services/mashups</td>
<td>Several hundreds</td>
<td>Several thousands</td>
</tr>
<tr>
<td>Degree of personalization</td>
<td>Limited</td>
<td>High</td>
</tr>
<tr>
<td>Adherence to design guidelines</td>
<td>Strong</td>
<td>Weak</td>
</tr>
<tr>
<td>Ease of data reuse</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>Optimized at design time</td>
<td>Typical</td>
<td>Rare</td>
</tr>
<tr>
<td>Data sources</td>
<td>Mostly internal</td>
<td>Mostly external</td>
</tr>
</tbody>
</table>

execution of most individual mashups is expected to be modest (the request rate experienced by the mashup platforms may still be very high due to the large number of mashups they host). Thus, the data generated in a mashup platform is substantially greater than its Web services counterpart, whereas the opportunity for data reuse is much lower. As Web services are authored by professional developers, they are optimized for performance, and they usually adhere to certain broad guidelines with respect to their overall structures. Therefore, it is possible for a human to identify the stages of Web services at which the results should be cached. On the other hand, mashups can be extremely heterogeneous in terms of their structure, as they are created by large sets of individuals with varying degrees of technical expertise. Further, most mashups require data from external sources, which implies that the costs of executing them depend upon external conditions upon which the mashup platform has little control. A comparison between Web service portals and mashup platforms is summarized in Table 1. Because of these differences, traditional Web service caching schemes are inadequate for mashups. Mashups demand a more dynamic caching strategy, wherein: (a) the intermediate results of mashup computations can be stored for future use; (b) the cache enables the intermediate results of one mashup to be used in another; and (c) the caching decisions are based upon dynamic benefit-cost analysis which also take into account the external conditions. **Mashup Model**

In this section, we develop a formal model for mashups. A mashup platform can be thought of as a system that contains a set of mashups; a mashup fetches data from sources that are distributed across the Internet, processes the fetched data in ways specified by the end-users, and dispatches the processed data to the end-users who again are distributed over a wide-area network. $MpSet = \{Mp_0, Mp_1, ..., Mp_{N-1}\}$ represents the mashups existing in the mashup platform at a given point in time. The mashup platform includes a set of classes of basic processing operators such as filter, sort, join, truncate, count, location-extraction, reverse, subelement, tail, and unique. For ease of modeling, we introduce two special classes of operators. The fetch operator class corresponds to the function of retrieving data required for a mashup from an external or an internal source, and a dispatch operator class represents the function of dispatching the mashup results to the end-user. $OpSet = \{op_0, op_1, ..., op_{M-1}\}$ denotes the set of classes of operators available in the mashup platform. Without loss of generality, operators classes $op_{M-2}$ and $op_{M-1}$ correspond to the fetch (represented as $fo$) and dispatch ($do$) operators classes, respectively. The rest of the $OpSet$ elements are data processing operators classes. Each operator class may specify certain requirements on the number of inputs that are fed into it and the type and formats
of these inputs. Also, each operator class always produces the same type of output. For example, the sort operator class expects a single table with possibly multiple rows and columns as input, and produces a table with the same number of rows and columns as output.

A mashup comprises of a set of operator instances chosen from the \( \text{OpSet} \). Every mashup contains one or more instances of the fetch operator class and one instance of the dispatch operator class. Specifically, a mashup is modeled as a tree with each node corresponding to a mashup operator instance. In this tree, the output of an operator node forms (one of the) inputs of its parent node. Furthermore, the dispatch operator instance always forms the root of the tree, and each leaf node corresponds to a fetch operator instance. \( \{nd_i^0, nd_i^1, \ldots, nd_i^{q-1}\} \) represent the nodes in the tree of the mashup \( Mp_i \), where each node corresponds to an instance of operator class from \( \text{OpSet} \). We note that while an individual mashup is modeled as tree, multiple mashups might share data sources, thus forming directed acyclic graphs (DAGs). Although some mashups update the original data sources, this work focuses on those that process data from remote sources rather than the ones that update them.

Each operator in \( \text{OpSet} \) is associated with two functions. The \textit{cost function}, represented as \( CF^{op_j}(s_0, s_1, \ldots, s_{q-1}) \) for operator \( op_j \) represents the cost of performing the operation. The parameters \( s_0, s_1, \ldots, s_{q-1} \) represent the sizes of the inputs to the operator \( op_j \). The concept of cost function is generic, and it can be measured in a variety of ways including latency involved in performing the operation and the computational/communication load imposed by the operation. In this paper, we quantify the cost of an operator instance through its latency. The output size estimation function, represented as \( OSF^{op_j}(s_0, s_1, \ldots, s_{q-1}) \) captures the size of the output of the operator instance \( op_j \), where \( s_0, s_1, \ldots, s_{q-1} \) are the sizes of the inputs. The \textit{cost value} of a node \( nd_i \) (denoted as \( CV^{nd_i} \)) in the mashup \( Mp_i \) is the value of the cost function of the corresponding operator instance on the specific inputs indicated in the mashup tree. Similarly, the output size value \( OSV^{nd_i} \) of the node \( nd_i \) is output size function of the corresponding operator instance evaluated on the inputs specified by the mashup tree.
The total cost of executing the mashup $M_{pi}$ is the sum of the cost values of all its operators instances. Similarly, the output size of mashup $M_{pi}$ is $OSV$ of its root node. Henceforth, for ease of representation, we refer to instances of operators classes as operators.

**High Level Architecture**

Figure 1 illustrates the architecture of the MACE system. The MACE system is co-located with the mashup platform. However, it is designed such that the mashup platform itself would require minimal modifications to work in conjunction with MACE.

In order for MACE to select stages at which data will be cached, it continuously observes the execution of mashups, and collects statistics such as request frequencies, update rates and costs and output size values at various nodes of the mashups. It then performs cost-benefit analysis of caching at different nodes of mashups, and chooses a set of nodes that are estimated to yield best benefit-cost ratios. An operator node in a mashup tree that is chosen for caching by MACE (i.e., the results until that stage of the mashup execution would be stored) is called a **cache point**.

Any node in the mashup tree except the root of the tree (corresponding to the dispatch operator) can potentially be chosen as a cache point. This set of nodes is called the **potential cache point set** ($PcpSet$), and each individual node in this set is referred to as **potential cache point**.

MACE also interacts with the mashup editor to obtain newly created mashups. For each new mashup, the MACE platform analyzes whether any of the cached results can be substituted for part of the mashup workflow. If so, the mashup is modified so that cached data is re-used, and only the additional operations required for completing the mashup are performed. The modified mashup is then provided to the mashup platform for execution. In addition to these two main features, the MACE platform also incorporates the basic cache functionalities such as replacement scheme and data consistency mechanism. This paper focuses on the design of a dynamic cache point determination technique and mechanism to re-use the cached data for substituting parts of incoming mashups. The next sections describe these two unique features of the MACE platform.

**CACHE INDEXING FOR EFFICIENT DATA REUSE**

Determining points of data reuse in newly created mashups is not a straightforward task. Notice that a cache point represents the results of computations occurring in a **subtree** of the mashup tree. This subtree itself might have one or more branches with fetch operators at the
leaves. The results at a particular cache point $Cp_h$ can be reused for a new mashup $Mp_i$ if and only if the subtree represented by $Cp_h$ exactly matches a subtree in $Mp_i$. By exact matching, we mean that a subtree of the incoming mashup has the same structure as that of the subtree represented by $Cp_h$, and parameters of operators in both subtrees are the same. Since mashup platforms support large numbers of mashups, we need a scalable mechanism to find out whether one or more subtrees of a new mashup match existing cache points. MACE includes a novel cache point indexing scheme to address this issue, which is explained later in this section. If one or more subtrees of a new mashup $Mp_i$ are found to match existing cache points in MACE system, $Mp_i$ is modified as follows. For each subtree that matches an existing cache point, the subtree is replaced with a fetch operator that references the cached data corresponding to the cache point. For example, if an arbitrary subtree $St_q$ of a $Mp_i$, matches an existing cache point $Cp_h$, $St_q$ is replaced with a fetch operator that refers to the cached data corresponding to $Cp_h$. The modified mashup is then sent to the mashup platform which executes it. Figure 2 illustrates the modification of a new mashup to reuse data available in the cache. We now explain our mashups representation and indexing that enables efficient discovery of cache points.

**B+ Tree Mashup Index**

We use the B+ tree structure to index cache points. Each operator in $OpSet$ is given a unique identification string. A mashup workflow is represented by concatenating its operators’ unique identification strings. Unlike other operators, join operators have two components to be joined and that makes its representation a little bit different than regular operators. A Join operator starts with special character SU: starts a join block, followed by the first component, followed by a special character MU: comes in the middle between joined components, followed by the second component, followed by a special character EU: ends a join block. Figure 3 shows an example of a mashup representation. The dashes in index keys are not part of mashup representation, they are introduced as separators between attributes for more clarity.

The index nodes’ entries of the B+ tree are substring of mashups identification strings. They are entered to the index based on their lexicographical order. Consider the following mashup example, Fetch data source “buycars.cars.com”, filter data based on model=”Honda”, sort on
“price”, if we decide to cache after the filter operator is executed, then “11#15-11-04-30-Honda” will be inserted into the index. However, if we decide to cache after the whole mashup execution flow is done, then “11#15-11-04-30-Honda#09-11-06” is inserted into the index. Figure 3 shows the mashup index if we decided to cache after both of the previous 2 points. The numbers in identification strings are IDs of the data sources, operators and attributes forming a mashup, for example, the filter operation (15-11-04-30-Honda) is interpreted as follows, ”15” is the ID of the filter operator, ”11” is the data source ID from which attribute ”04” is taken, ”30” is the ID of the equality operator and ”Honda” is the value on which the attribute ”04” is filtered. In the previous identification strings # represents a special character which works as a separator between operators. Notice that each operators’ identification string reflects the operators which precede it in the mashup workflow, this enables us to index mashups without losing order of execution of mashup operators.

**RANGE QUERIES AND TAXONOMY AWARENESS**

Until now, our design of the indexing scheme is focused on lookups for *exact matches*. However, supporting lookups for inexact matches can considerably increase the reuse of cached data. Consider the case when the cache contains superset of data needed for an incoming mashup. In this case, the data in the cache can be appropriately filtered and reused for the new mashup. Unfortunately, looking up for exact matches would fail to even locate the existence of the superset, thereby precluding the possibility of data reuse.

Towards addressing this issue, we enhance our indexing mechanism to support two specific kinds of inexact matching, namely, range queries and hierarchical taxonomies. Our strategy relies on the fact that the keys in the leaf level of the B+ tree index are sorted, which implies that mashups that are lexicographically close by to one another are stored either in the same index node or in a nearby node.

**Supporting Range Queries**

Suppose that a new mashup requires data with parameter “p” being in the range interval [x,y]. With exact matching, if the mashup index does not contain the precise range [x,y], a cache miss occurs and the new mashup is executed end-to-end. Consider the scenario when the cache contains the results of an earlier mashups that is similar to the new one except that parameter “p” is in the range [a,b]. Now the question is whether this data can be reused for the new mashup? In order to determine this, we need to consider three distinct cases. First, if “x≥a and y≤b”, then this means that range interval [x,y] is fully included within range interval [a,b] which also means that the result of the new mashup is fully contained within the result of the existing mashup. In this case, a cache hit is declared and a local search within the result of the existing mashup is performed to extract the result of the new mashup. Second, if “x≥a and y>b” or “x<a and y≤b” or “x<a and y>b”, then the range interval [x,y] is partially included in the range interval [a,b]. Here, because the result of the new mashup cannot be completely satisfied by the existing cached mashup result, then a cache miss is declared and the new mashup is executed end-to-end. It is noteworthy that this partial inclusion relationship can be useful in two situations. First, the case where partial results can be extracted from cached data and we query for missing data, then the missing data and partial results are combined. Second, the case where the new mashup cannot be executed due to difficulties in communicating with data sources. In such a situation, the data available in the cache is reused to provide the end-user with a valid but incomplete result. For example, if we consider the “x≥a and y>b” scenario, the result satisfying the range interval [x,b] can be provided for the end-user from cached data. Such a result is not complete, but it is still valid and may be useful to the end-user. Third, if “x<a and y<a” or “x>b and y>b”, then the range
[x, y] is totally outside the range [a, b]. In this case, a cache miss is declared and the new mashup is executed end-to-end. The inherent capability of the B+ tree structure to lookup range values can be leveraged for the above purpose. The following example illustrates how this is achieved in the MACE framework. Suppose the following mashup result is cached in the system: Fetch data from “buy cars.cars.com” then filter data based on “price < 5000”. Based on our mashup string representation, this mashup is represented in the cache as “11#15-11-06-32-5000” where the last 4 digits correspond to the value on which data is filtered (5000). Now, suppose an end-user asks for a new mashup which is described as follows, fetch data from “buy cars.cars.com” then filter data based on “price < 4000”, the new mashup is represented as “11#15-11-06-32-4000”. This case represents case 1 where the range interval of the new mashup is fully included within the range interval of the existing mashup. We can see that the two mashups have some common part (11#15-11-06-32) in their string representation. Without using range query improvement, a search for the new mashup representation “11#15-11-06-32-4000” results in a cache miss, this happens because we make exact matching between index keys and the mashup representation we are looking for. When using range query awareness, the search process for previous mashup (11#15-11-06-32-4000) in the tree explores through index levels based on lexicographical order of keys and eventually arrives at the existing mashup key “11#15-11-06-32-5000”. Now, instead of declaring a cache miss, we detect that this key (11#15-11-06-32-5000) and the key we are looking for (11#15-11-06-32-4000) represent the same mashup except that the value on which data is filtered is different. Here, we do not have to execute the new mashup right from its starting point, instead, we search items that the previous key points to and then exclude items with price between 4000 and 5000. Accordingly, we achieve better utilization of our mashup index. As an example of case 2, suppose the new mashup is filtering data based on “price<6000”, here the result of the new mashup is partially included in the existing mashup cached result. Normally, we declare a cache miss and execute the new mashup from scratch, but if the new mashup execution is interrupted due to communication problems, then a cache hit is declared and the end-user is provided with the result of car items cheaper than $5000. One might argue that providing incomplete result is not accurate. Although this is true, the partial result can satisfy end-user demands in many cases, here in the car example, the end-user might find a suitable car within the incomplete list of cars. Note that when MACE caches mashup results, it may so happen that two results sets (of two distinct mashups) overlap without anyone of them being a subset of the other. In such a scenario, we do not make duplicate copies of data that is common to both sets. We maintain a single copy of the common items but store the pointer at two distinct locations in the index. This maximizes the utility of the available storage.

Supporting Hierarchical Taxonomies

The range query technique presented in the previous section works well for numeric parameters. However, in many cases, end-users can create mashups that extract general information, while others might create mashups that extract more specific information with respect to parameters that are non-numeric. Consider the case when the mashup platform caches the results of a mashup that extracts all sports related stories from “sports.yahoo.com”. Now, suppose another end-user creates a mashup that extracts all stories related to tennis from “sports.yahoo.com”. Clearly, the results of the new mashup are more specific and constitute a subset of the results of the existing mashup. The previous range query mechanism cannot be used for this case as the parameter is keyword-based. We have developed a mechanism to detect these types of generic/specific relationships among mashups in terms of keyword parameters. The central idea is to build a hierarchical taxonomy that defines relationship between various keywords or categories. The assumption is that a
keyword at a higher level in the hierarchy subsumes all keywords which reside underneath it. This hierarchical taxonomy is used to enhance data reuse and minimize cache misses. When a new mashup is created, our strategy is not to just search the index for earlier mashups with exactly matching keywords, but also to look for existing mashups that have ancestors of the keywords in the incoming mashup. Specifically, suppose an existing mashup filters data based on non-numeric keyword-valued parameter “p” being equal to “X”, and suppose this result is cached. Later, suppose a new mashup which is identical to the existing mashup except that “p” equals “Y” is created. Normally, a cache miss is declared and the new mashup is executed end-to-end. However, providing that a hierarchical taxonomy exists, we can use it to look for a possible relationship between X and Y. If X is an ancestor (direct or indirect parent) of Y, then the result of new mashup is a subset of the cached result. However, the cached results cannot be directly used for the new mashup. The cached results have to be locally filtered to the actual result of the new mashup. As an example, suppose the cache contains the result of the mashup that fetches data from “sports.yahoo.com”, then filter data based on the criterion “category = sport”. The index has the key “09#15-09-02-32-sport” corresponding to this data in the cache, where “sport” is the value on which data is filtered. Now, suppose an end-user creates a mashup to fetch data from “sports.yahoo.com”, then filter data based on “category = tennis”. The key for the new mashup is “09#15-09-02-32-tennis”, where “tennis” is the value on which data is filtered. The search process for the new mashup in the cache starts by exploring the index until we reach the level containing the key “09#15-09-02-32-sport”. If we are going to use exact matching to look for the new mashup in the cache, we will end up with a cache miss. Instead, we detect that the part “09#15-09-02-32” is common between the new mashup and the cached mashup, so we extract the value on which the cached mashup is filtered (sport) and we extract the value on which the new mashup is filtered (tennis), then we consult the taxonomy to look for a possible relationship between these two keywords. Since “tennis” is a child of “sport” in the taxonomy, we conclude that the result of the new mashup is contained in the result of the previously cached mashup. Consequently, the result of the new mashup can be found by locally filtering the cached data of the first mashup. Figure 4 illustrates the above example.

**CACHE POINT SELECTION TECHNIQUE**

In this section, we describe our dynamic cache point selection technique. We formulate the dynamic cache point selection as an optimization problem following which we provide efficient algorithms for cache point selection.
Problem Formulation

This section formulates the cache point selection as a cost-benefit optimization problem. We provide two flavors of the cost-benefit optimization problem. The first one models a scenario wherein the storage-space availability at MACE is unlimited and the second corresponds to the scenario in which the MACE system has limited storage capacity. We begin by introducing terminology and notation that are employed in the problem formulation.

Potential cache point set \( \{Pcp_0, Pcp_1, ..., Pcp_{M-1}\} \) represents the unique potential cache points corresponding to the mashups existing in the \( MpSet \). Recall that every operator node in a mashup except the root is a potential cache point. The members of \( PcpSet \) are unique in the sense that the potential cache points that represent subtrees which exist in multiple mashups are included only once. The sum of the cost values of all the descendant nodes of a potential cache point \( Pcp_k \) including the cost value of \( Pcp_k \) is called the cumulative cost value of \( Pcp_k \) \( (CCV_{Pcp_k} = CV_{Pcp_k} + \sum_{Pcp_{\alpha\text{Descendent}(Pcp_k)}}(CV_{Pcp_{\alpha}})) \).

Cost-Benefit Analysis

The benefits of caching the results at a particular potential cache point \( Pcp_k \) is that the cached data would be reused for any future requests of all mashups that \( Pcp_k \) is part of, thus avoiding the re-executions of \( Pcp_k \) and all of its descendant nodes. Let request frequency of \( Pcp_k \) (represented as \( RF_{Pcp_k} \)) denote the number of times \( Pcp_k \) needs to be executed per unit time to satisfy end-user requests if the output of \( Pcp_k \) is not cached. Note that \( RF_{Pcp_k} \) is the total sum of the request frequencies of all the individual mashups that the subtree under \( Pcp_k \) is part of. Thus, the benefits per unit time obtained by caching at \( Pcp_k \) is \( RF_{Pcp_k} \times CCV_{Pcp_k} \).

Caching at a potential cache point \( Pcp_k \) involves two distinct costs, namely consistency costs and storage costs. Consistency costs are the costs involved in maintaining the consistency of cached data in the face of updates to the data from external sources that are used in computing the output of \( Pcp_k \). Notice that the data cached at \( Pcp_k \) becomes invalid, and would need to be updated anytime the data obtained through any of the fetch operators below \( Pcp_k \) changes. Each time the output of \( Pcp_k \) needs to be recomputed, \( Pcp_k \) and all of its descendant nodes need to be re-executed. Thus, the consistency costs per unit time of caching at \( Pcp_k \) can be quantified as \( UF_{Pcp_k} \times CCV_{Pcp_k} \), where \( UF_{Pcp_k} \) represents the sum of the update frequencies of all the external data sources fetched by the operators below \( Pcp_k \).

The storage costs of caching at \( Pcp_k \) are directly proportional to the size of the output \( (OSV_{Pcp_k}) \). However, notice that the storage costs only matter when available storage is limited. Furthermore, storage costs and consistency costs are inherently different, and cannot be combined into a single equation in meaningful way. We model the storage costs as constraint rather than optimization criterion.

\( RF_{Pcp_k} \times CCV_{Pcp_k} - UF_{Pcp_k} \times CCV_{Pcp_k} \) is called the cost-benefit trade-off for \( Pcp_k \) (represented as \( CBT_{Pcp_k} \)). \( CBT_{Pcp_k} \) quantifies the net cost-savings obtained by caching at \( Pcp_k \). Note that in this formulation of \( CBT_{Pcp_k} \), the computational overheads incurred at the
time of serving end-user requests and those incurred to maintain consistency of cached data, are of equal importance. In scenarios where one is more important than the other, the two terms of $CBT^{Pcp}$ have to be appropriately weighted to reflect their relative importance.

Scenario 1 —— No storage limitations: As stated earlier, the objective of the dynamic cache point selection scheme is to select a set of cache points such that the benefit-cost tradeoff is maximized. Let $X^{Pcp}$ be a \{0,1\} variable denoting whether $Pcp_k$ is selected as a cache point ($X^{Pcp}$ is 1 if $Pcp_k$ is chosen and 0 otherwise). Therefore, the optimization criterion would be to assign $X^{Pcp}$ values to each potential cache point $Pcp_k \in PcpSet$ such that $\sum_{Pcp \in PcpSet} X^{Pcp} \times CBT^{Pcp}$ is maximized. However, notice that the optimization problem, as it stands, can lead to duplicate-caching (caching same or interdependent data multiple times thus wasting resources). In order to avoid this, we introduce the following constraint. For any pair of potential cache points \{Pcp_k, Pcp_i\} such that $Pcp_k \in Descendent(Pcp_i)$ or vice-versa, $X^{Pcp} + X^{Pcp_i} \leq 1$.

Scenario 2 —— Limited storage: The optimization problem for the limited storage scenario is similar to the previous case, but the total storage requirements of cached data should not exceed the storage available in the MACE system. Suppose $Sg$ denote amount of storage available. The optimization problem can be stated as follows. Assign values to decision variables $\{X^{Pcp_0}, X^{Pcp_1},...,X^{Pcp_{M-1}}\}$ corresponding to the potential cache points \{Pcp_0, Pcp_1,...,Pcp_{M-1}\} such that $\sum_{Pcp \in PcpSet} X^{Pcp} \times CBT^{Pcp}$ is maximized while ensuring that the following constraints are not violated:

1. $X^{Pcp_k} \in \{0,1\}, \forall Pcp_k \in PcpSet$
2. $\forall \{Pcp_k, Pcp_i\}$ such that $Pcp_k \in Descendent(Pcp_i) \parallel Pcp_i \in Descendent(Pcp_k)$, $X^{Pcp_k} + X^{Pcp_i} \leq 1$; and
3. $\sum_{Pcp \in PcpSet} X^{Pcp} \times OSF^{Pcp} (ips) \leq Sg$, where the variable $ips$ represent the inputs to the operator at $Pcp_k$ as specified in the mashups. This is a constrained discrete optimization problem solving which requires exhaustive search of the solution space. In the next section, we present a greedy strategy-based algorithm for this problem.

**Cache Point Selection Algorithms**

First, we consider the scenario wherein the storage space is not a constraint. Statistics such as the request frequencies and update frequencies of all potential cache points are collected, and the corresponding cumulative cost values are calculated. For each mashup in the platform, our algorithm searches for the best cache point as follows. The algorithm starts searching from the potential cache point that is shared across many other mashups, and at the same time is located at lower-levels of the mashup tree. This can be achieved by starting at a node that has the maximum value for \( \frac{SMCount}{Height} \), where \( SMCount \) (sharing mashups count) indicates the number of mashups that share the potential cache point and \( Height \) indicates its height in the mashup. The rationale for starting the search at such a node is that it is likely to yield maximum reuse (thereby maximizing the benefits) at low consistency maintenance costs. Suppose the algorithm starts from the potential cache point $Pcp_k$. The node that is currently being searched is called the current
search point (CSP). We calculate $CSP^{CBT}$ as $RF^{CSP} \times CCV^{CSP} - UF^{CSP} \times CCV^{CSP}$. We then compare the value of $CSP^{CBT}$ to the $CBT$ value of its ancestor in the mashup and the $CBT$ value of its descendant in the mashup (if $Pcp_k$ has multiple descendants, we consider the sum of their $CBT$ values). If the $CBT$ value of the ancestor is higher than that of $Pcp_k$, the ancestor is initialized as the new CSP, and the algorithm continues searching upwards from that point. If, on the other hand, the descendant node had a higher $CBT$ value, the descendant is initialized as the new CSP and the algorithm continues searching downwards. If $Pcp_k$ has multiple descendants, the algorithm continues searching downwards from each of them. The search terminates when we reach one or more nodes such that the $CBT$ values of their respective descendants and ancestors are lower than their $CBT$ values. The potential cache point(s) at which the search terminates are chosen as the cache points and included in the cache point set ($CPSet$). The algorithm searches each mashup in a similar fashion to discover all the cache points. This algorithm yields optimal solution to the scenario with no storage limitations. The algorithm is linear in terms of the number of potential cache points in the platform.

We now extend the above algorithm for the limited storage scenario. Recall that discovering optimal solutions for this scenario requires exhaustive search of the solution space. Therefore, our objective is to design an efficient algorithm that yields close to optimal solutions. The algorithm for the limited storage scenario works in two stages. The first stage is exactly similar to the algorithm described above for the scenario wherein the storage space is not a limitation. However, the storage requirements for $CPSet$ obtained at this step may exceed the available storage ($Sg$). The second stage of the algorithm performs additional level of pruning as follows.

For each cache point $Pcp_k$ in the $CPSet$ produced at the end of first step, it calculates the ratio

$$BCS^{Pcp_k} = \frac{CBT^{Pcp_k}}{OSV^{Pcp_k}}.$$ 

This ratio quantifies the per-byte cost savings obtained by caching the results of $Pcp_k$. The cache points are sorted in the descending order of their $BCS$ values. The algorithm then progressively eliminates the cache points from the end of this sorted list (i.e., the cache points with the least $BCS$ values are eliminated first) until the results of the cache points remaining in the $CPSet$ can fit into the available storage. The rationale for this elimination strategy is to retain cache points that provide maximum benefits for the amount of storage space they consume. Once the $CPSet$ is computed, the MACE engine starts storing the outputs of the cache points.

**EXPERIMENTS AND RESULTS**

Our experimental study has three objectives: 1) Study the impact of MACE’s dynamic cache point selection on the performance of the mashup platform. 2) Evaluate the benefits and overheads of the proposed cache point indexing scheme. 3) Test the impact of range queries and taxonomy awareness support on our system. First, we describe the experimental setup.

**Experimental Setup**

Our experimental setup simulates a mashup environment with a mashup server, 80 data sources and 100000 end-users spread out on the Internet. The mashup server in our setup is, to a considerable extent, based upon the Yahoo Pipes environment (Yahoo Inc., 2007). Similar to

1 The search may also terminate when we reach the end of the mashup tree (in either direction)
Yahoo pipes, our mashup platform contains 10 distinct operators namely, filter, sort, join, truncate, count, location-extraction, reverse, subelement, tail, and unique. End-users continuously create mashups which are executed on the mashup server. The mashup server executes the mashup and disseminates the results to the end-user.

We use two datasets for our experiments -- a real dataset and a synthetic dataset. In the real dataset, we build our mashup to closely reflect reality. In this dataset, we have 5000 mashups which pull data from 80 real data sources over the Web. These data sources are extracted from syndic8 (Barr, 2008) feeds repository. This repository is a directory of RSS and Atom feeds existing on the Web. When a mashup is executed, an actual connection to data sources is made to fetch data and the communication time needed to fetch data is measured. We also implemented a set of operators so that they process the real fetched data and therefore their execution time is also measured. In our real mashup set, the mean value for latency to extract data from data sources is 0.6 seconds and the average number of items in these sources is 21 items. The number of subscriptions to a data source is representative of its popularity. In figure 5, we plot the popularities of various feeds. To study the characteristics of the feed popularity, we also show a curve depicting the Zipfian distribution with $\alpha$ (exponent value) = 0.9. The diagram shows that feeds popularity closely resembles the Zipfian distribution. Similarly, Figure 6 indicates that the feed sizes also resemble Zipfian distribution with $\alpha = 0.9$. Figure 7, on the other hand, plots the relationship between feed popularity and feed data sizes. The figure shows that most feeds have small data sizes and low popularity.

Our synthetic dataset contains simulated operators where the execution time for each of these operators is estimated by performing a number of experiments on Yahoo pipes wherein we evaluated the latencies and output sizes of individual operators on XML feeds with sizes varying from 100 KB to 3 MB. As a result, we have realistic cost functions and output size functions. The number of distinct mashups existing at the platform in the synthetic dataset varies with the experiment, and it ranges from 1000 to 5000. The mean value for latency to extract data from data sources is 15 seconds and the average number of items in these sources is 536 items. For our synthetic dataset, the network topology is based upon the measurement by DIMES (Shavitt, 2005) on the actual Internet in 2008. We use BRITE (Medina, 2001) and BRITE extension (Wahlisch, 2008) to transform DIMES data into a more convenient form. Our topology has 378444 nodes.
Evaluation of the Dynamic Cache Point Selection Scheme

In the first set of experiments, we quantify the performance benefits of the dynamic cache point selection scheme. The dynamic cache point selection scheme is compared to two other schemes: *End-results caching* wherein only the end-results of the mashups are cached, and *No caching* wherein the mashup platform does not employ any type of caching. These three schemes are compared with respect to the total cost incurred by the mashup platform in serving the end-user requests. For an individual mashup, the cost is quantified as the associated computational latency at the mashup platform.

In the first experiment, we compare the three schemes as the mean of the request rates of all mashups varies from 20 requests per unit time to 100 requests per unit time. The total number of mashups at the server is 5000 (therefore, the cumulative request rate at the mashup server varies from 10,000 and 500,000). In the synthetic dataset, a Zipfian distribution with $\alpha = 0.9$ is used to model the popularity variations among the individual mashups, while in the real dataset, popularity variations among individual mashups is extracted from syndicate8 feeds repository. The mean of the update frequencies of the data sources (henceforth referred to as update frequency) is set to 60. This experiment is conducted on the synthetic dataset. The cache is assumed to have enough storage to hold the results (intermediate or final) of all mashups. Thus, we use the dynamic cache point selection algorithm for the no-storage limitations scenario. Figure 8 shows the total costs per unit time for the results of the experiments. As the results indicate, the cost incurred by the MACE’s dynamic cache point is lower than the other two schemes throughout the simulated request rate range. The cost incurred by the End-results caching scheme is essentially constant as requests are served using cached data not requiring additional computations. In End-results caching, costs are mainly due to re-calculation of the cached results when one or more inputs used in a mashup changes. At very low request rates, the costs of no-caching scenario are comparable to those of the MACE system. However, the costs of no-caching scenario rises quickly with increasing request rates. It is to be noted here that although the costs of the dynamic cache point selection scheme increases with increasing request rates, it does not rise indefinitely; its curve becomes flat once upon reaching the End-results caching cost levels.

2 First-time mashup executions also contribute towards the total costs in End-results caching but these costs are comparatively very small.
In the second experiment (Figure 9) which is also conducted on the synthetic dataset, we study the effect of update frequencies of data sources on the performances of the three schemes. The setup is very similar to that of the previous one except that the mean mashup request rate is fixed at 60 requests per unit time whereas the update frequency of all data sources is varied from 20 to 100 per unit time. Again, we see that the MACE system yields significantly better performance than the other two schemes. However, in this experiment, the costs of the no-caching scenario remain constant. This is because, there is no cached data that needs to be recomputed when the input data changes.

In the third experiment, we aim to compare the results of experiments of the synthetic dataset with the results obtained from the real dataset. So, we measure the total cost per unit time required for executing our mashup set in both cases. First, we fix update rate to 60 updates per unit time and we vary request rate in the range of 20 to 100 requests per unit time. After that we fix request rate to 60 requests per unit time and we vary update rate in the range of 20 to 100 updates per unit time. In Figures 10 and 11, “R” refers to real dataset and “S” refers to synthetic dataset, these two figures show that the patterns we have for the real dataset are similar to the patterns we have for the synthetic dataset, the only thing different is the scale of cost values. The cost values for real data set in figures 10 and 11 reflect the real world and make our experiments more realistic.

The better performance of the dynamic cache point selection scheme is essentially due to its ability to adapt to the changing update and request frequencies by moving the cache point to upper or lower levels of the tree. Figure 12 demonstrates this phenomenon by plotting the average level of the cache points as the update frequency varies from 20 to 100. The mean mashup request rate remains constant at 60. As the results indicate, as the update rate increases, MACE selects cache points that are located at lower-levels of the tree thereby reducing the costs of recomputing the cached results. The End-results caching, on the other hand, always caches at the same level (just before the dispatch operator).

In the next experiment, we use our synthetic dataset to evaluate the three scenarios when the storage available at the caches is limited. In this experiment, we fix the total request rate at 60 and update frequency at 180. The storage availability is varied from 10% to 100% of the storage needed for caching entire result set for the particular caching strategy. Least Recently Used cache replacement is employed for all schemes. As Figure 13 demonstrates, MACE results in better performance by selecting cache points that provide higher per-byte cost savings.
Mashup Index Analysis

In the second set of experiments, we use our synthetic dataset to study the scalability and performance of MACE’s indexing mechanism by measuring the average latency involved in accessing a cache point stored in the B+ tree index. In the first experiment in this set, we evaluate the effects of request rate on the index access time. The mashup server has 5000 mashups with each mashup having 11 operators. The update frequency of all data sources is held constant at 60. As Figure 14 shows, index access time decreases as request frequency increases. This is due to two factors. First, when request frequency increases, MACE tends to select cache points near the roots of the respective mashup trees. As we move closer to the root, the width of the tree shrinks and the number of cache points in the index decreases. Second, MACE analyzes a new mashup starting from its root and goes down the tree looking for matching cache points. At high request frequencies, the cache points are closer to the root, and hence the search for matching cache points concludes faster. This result shows an important strength of our indexing scheme - it responds faster when the request rates are higher thereby improving the mashup platform’s scalability.

Next, we study the effect of the mashup depth on index access time. The server again contains 5000 mashups. The mean mashup request rate and the update frequency are both set to 60. Figure 15 shows the index access times when the depth of the mashups is varied from 5 to 20. Initially, the index access time increases linearly with mashup depth. The reason for this behavior is that probability of selecting cache points from lower levels of the tree increases as the mashup depth increases, and hence the search for matching cache points takes more time to conclude. However, the index access time becomes flat when the mashup depth reaches around 15.

Evaluation of Range Queries and Taxonomy awareness

In this experiment which is conducted on the real dataset, we study the effect of using partial matching on mashup index utilization. The mashup server has 5000 mashups with each mashup having 11 operators. The update frequency of all data sources is held constant at 60. Figure 16
shows that using support for range queries decreases the total cost for executing mashups by 9 percent. This effect happens because mashup index hit rate increases as Figure 17 shows. In the case where range queries are not supported, an index search might result in a cache miss. On the other hand, when range queries are supported, the probability of having a cache hit from searching the index increases. For the experiment of range queries, wherever we have a filter operator that filters data based on a numeric attribute, we use a numeric value between 1000 and 5000 upon which data is filtered (e.g.; price<3000). The numeric value is selected randomly from the range 1000-5000. The effect of taxonomy awareness is shown in Figure 18 where we notice that the total cost for executing mashups decreases when taxonomy awareness is used. This also occurs because of the increase in mashup index hit rate which is shown in Figure 19. An index lookup might result in a cache miss when taxonomy awareness is not used, but when it is used, the probability of having cache hits increases. We build our taxonomy by extracting keywords from Google search-based keyword tool (Google Inc., 2008). This tool classifies the keywords people search for and put it in categories. The tool enables its users to search for keywords as well as download keywords categories and keyword lists as CSV files. When we build our mashups, wherever a filter operator is used, a random keyword or category name from the taxonomy is used as the parameter upon which data is filtered. We believe that the percentage of improvement resulted from using the feature of range queries and taxonomy awareness is affected by the following. First, such a feature can only be applied to a subset of the operators that can be used to build mashups. For example, range queries support can be used for filter, truncate, and tail operators, but it cannot be used for fetch, reverse, and unique operators. Second, the number of keywords end-users can choose from and the number of numeric values the end-users can use in their operators is high; therefore, the possibility of detecting operators where range query and taxonomy support can be used is low. Further improvement can be reached by considering the patterns end-users follow for selecting keywords and selecting numeric values in different domains. For example, if more end-users are interested in car prices below $5000, this can be an indication that most of end-user mashups related to filtering car results might contain “price < 5000”. The same principle is applied to taxonomy awareness, if end-users care most about food, then more mashups are expected to use keywords related to food domain. Incorporating end-user patterns in building mashups is expected to produce more realistic results.
RELATED WORK

Research in the area of mashups is still in its nascent stages. MARIO (Riabov, 2008) is a recent mashup editing tool in which mashups are built from tags and executed using a planning algorithm. DAMIA (IBM Corp, 2007) is a data integration service for situational applications in the enterprise domain. Kulathuramaiyer (2007) describes a mashup for digital journals which enables its users to explore digital libraries using semantic-rich meta-data. Subspace (Jackson, 2007) adopts the sandboxing principle to isolate applications into trust layers. Marmite (Wong, 2007) is a mashup tool implemented as a Firefox plug-in using Java Script and XUL, it enables end-users to aggregate and filter data from several Web contents and services. It also has the capability of directing mashups output to several sources such as websites, text files, or even compliant source code that can be customized. MashMaker (Ennals, 2007) is a mashup Web tool that enables end-users to manipulate data from other websites and define their visualized queries over it; it also provides sharing of mashups as widgets among end-users. Liu (2007) provides a mashup architecture that is based on an extension of the Service Oriented Architecture (SOA). Similar to (provider, broker, and consumer) SOA roles, their system presents roles of mashup component builder, mashup server, and mashup consumer, such that these roles interact together to facilitate data and services composition to end-users through a Web based interface. Karma (Tuchinda, 2008) is another mashup platform that enables end-users to build mashups by example. They argue that using widgets as building blocks of mashups is not convenient for end-users because as a mashup platform grows, the number of widgets grows as well and that will confuse the end-user on which widget he needs to perform his task. Based on that, the authors let end-users extract data they want from websites. The data is stored behind the scenes as DOM trees and it is visualized to the end-user as tables, then the end-user can visually work on the data with the help of their system. Their system allows end-users to fix and integrate data and define rules between attributes of data, these attributes are stored in the system as XPath rules and the system intelligently applies the rules on data containing the previous attributes. SMash (Dekeukelaere, 2008) is a security model that can be integrated in Web browsers in order to make mashup applications secure. The Authors state that current browsers security models are not appropriate for mashups, because mashups require interaction between different data sources.
while regular browsers usually disallow such interactions between links coming from different sources. SMash defines a security model so that scripts coming from different data sources are not allowed to change each others data. At the same time, the model does not allow such scripts to spy on end-user data banks. The model applies component isolation in mashups while at the same time guaranteeing communication between components using authenticated communication channels. The model also defines security rules on which types of interactions are allowed between components. To our best knowledge, the previous mashup platforms do not use caching while in MACE we introduce caching for mashups.

Web content caching in general has received considerable research attention (Iyengar, 1997; Wang, 1999; Ramaswamy, 2005; Yin, 2002). Issues such as caching granularity, caching architectures, consistency maintenance and data placement and replacement strategies have been extensively investigated. Various caching techniques are proposed to optimize several parameters in Web caching. One category of those schemes is cache routing schemes (Thaler, 1998; Valloppilil, 1997) where hashing is used to map files to cooperative caches groups. A second category is multicast-based schemes (Michel, 1998) where communication and coordination between caches is performed using IP level multicasting. A third category is directory-based schemes (Fan, 1998; Tewari, 1999) where a subset of contents of cooperative caches is stored in every cache. Ninan (2003) proposes a lease based scheme called cooperative leasing in which a lease based technique is used to maintain document consistency. Gao (2005) proposes a scheme for edge caching where different levels of object consistency have been taken into account in order to minimize consistency maintenance costs. A dynamic scheme for disseminating data is presented in (Shah, 2004) where each data object has its own dissemination tree which is built based on coherency requirements of caches.

Web services, have been an active area of research in which aspects such as description, discovery, composition, and efficiency of Web services are addressed (Benatallah, 2006; Ardagna, 2007; Aoyama, 2002). Web services caching in particular gained interest from the research community. In WReX (Tatemura, 2005), a caching middleware architecture for caching XML Web services responses is proposed. Terry (2003) discusses caching XML Web services for mobile clients. Caching Web services by utilizing proxy caches is used to minimize communication in mobile ad hoc networks (Artail, 2006). Dorn (2006) proposes a scheme to maintain Web services availability in cases of communication link interruptions; their scheme utilizes a cache proxy to increase the availability of Web services. Ramasubramanian (2004)
explains that WSDL which is used to describe Web services lacks information to support caching, so authors extend WSDL to include information about what operations are cacheable in a way that is transparent to clients and servers. Challenges on caching Web services in PDAs is discussed in (Liu, 2007) where the authors provide a caching scheme that helps alleviate problems resulting from loss of connectivity and their scheme is adaptable to bandwidth changes. However, as we remarked earlier, most of the existing Web service caching schemes store results at fixed stages of the Web services, and hence are less effective for the mashup domain.

CONCLUSION

Traditional Web service caching schemes are not effective for mashup domain due to its unique characteristics. In this paper, we presented the design and evaluation of MACE — a dynamic caching framework for mashups. MACE is based upon a formal mashup model wherein an individual mashup is represented as a tree of operators. MACE’s design includes a dynamic mashup cache point selection scheme which maximizes the benefit of mashup caching. We have also proposed a novel indexing mechanism that supports efficient discovery of mashup cache points. Our indexing mechanism not only supports queries for exact matches but also range queries and hierarchical taxonomy-based queries. Through a detailed experimental study, we showed that MACE can significantly improve performance and scalability of mashup platforms.

REFERENCES


ABOUT THE AUTHOR

Osama Al-Haj Hassan received the BS degree in Computer Science from Princess Sumayya University for Technology (PSUT) and the MS degree in Computer Science from New York Institute of Technology (NYIT). Currently a PhD student in the Computer Science department at the University of Georgia (UGA). His research interests are in distributed systems, Web 2.0, caching and replication techniques, Peer-to-Peer networks and event based systems.

Lakshmish Ramaswamy received the PhD degree in Computer Science from Georgia Tech in 2005. He is currently an assistant professor in the computer science department at the University of Georgia. His research interests are broadly in the area of distributed systems, and more specifically in performance, scalability, security and privacy of Web services, overlay networks, events-based middleware, and mobile systems. He is the recipient of best paper award of the 13th World Wide Web conference (WWW-2004) and the 2005 Pat Goldberg best paper award. He has served on the program committees of several international conferences and workshops. He was the program co-chair of DEPSA-2007 and SENS-2006 workshops. He is a member of IEEE and the IEEE computer society.

John A. Miller is a Professor of Computer Science at the University of Georgia and has also been the Graduate Coordinator for the department for 9 years. His research interests include database systems, simulation, Web services and bioinformatics. Dr. Miller received a B.S. in Applied Mathematics from Northwestern University in 1980 and an M.S. and Ph.D. in Information and Computer Science from the Georgia Institute of Technology in 1982 and 1986, respectively. During his undergraduate education, he worked as a programmer at the Princeton Plasma Physics Laboratory. Dr. Miller is the author of over 140 publications covering all areas of his research interests. He is an Associate Editor for the ACM Transactions on Modeling and Computer Simulation, IEEE Transactions on Systems, Man and Cybernetics and SIMULATION: Transactions of the Society for Modeling and Simulation International as well as an Editorial Board Member for the Journal of Simulation and International Journal of Simulation and Process Modeling.