

MACE: A Dynamic Caching Framework for Mashups

Osama Al-Haj Hassan, Lakshmesh Ramaswamy and John A. Miller

Computer Science Department

University of Georgia

Athens, GA 30602, USA

{hasan, laks, jam}@cs.uga.edu

Abstract—The recent surge of popularity has established Mashups as an important category of Web 2.0 applications. Mashups are essentially Web services that are often created by end-users. They aggregate and manipulate data from sources around the World Wide Web. Surprisingly, there are very few studies on the scalability and performance of mashups. In this paper, we study caching as a vehicle for enhancing the scalability and the efficiency of mashups. Although caching has long been used to improve the performance of Web services, mashups pose some unique challenges that necessitate a more dynamic approach to caching. Towards this end, we present MACE - a cache specifically designed for mashups. In designing the MACE framework this paper makes three technical contributions. First, we present a model for representing mashups and analyzing their performance. Second, we propose an indexing scheme that enables efficient reuse of cached data for newly created mashups. Finally, this paper also describes a novel caching policy that analyzes the costs and benefits of caching data at various stages of different mashups and selectively stores data that is most effective in improving system scalability. We report experiments studying the performance of the MACE system.

Keywords-mashup; Web 2.0; Web Services; Personalization; Caching

I. INTRODUCTION

The advent of Web 2.0 marks a paradigm shift in the World Wide Web. It envisions an enhanced level of participation from the end-users by empowering them with tools for creating and sharing novel services. Mashups [10] have emerged as a popular class of Web 2.0 applications. Conceptually, mashups are Web services that are created by end-users. This enables them to offer significantly better personalization than traditional Web services (throughout this paper, the term Web services refers to traditional Web service model in which a service provider creates and deploys Web services). A typical mashup obtains data from one or more sources on the Internet, performs various operations upon them and ships the results to the end-user. Recently, several editors have been developed for creating and editing mashups including Yahoo pipes [18] and Microsoft Popfly [9].

Mashups, while enhancing personalization and end-user participation, also introduce new scalability and performance challenges. Unfortunately, these issues have received little attention from the research community. None, to our best

knowledge, has studied the performance characteristics of mashup platforms or proposed techniques for improving the same.

Caching has been a proven technique to enhance the scalability and efficiency of various Web applications, including content delivery and Web services [17]. Several caching techniques have been specifically developed for Web services [14], [15]. However, the existing Web services caching techniques cannot be directly applied to the mashup domain. Many traditional Web service caching techniques store end results of Web services (a few store intermediate results at pre-specified stages of the Web service workflow). Most of these approaches are blind to the structural composition of the Web services. However, a typical mashup platform has to support large numbers of mashups in comparison to Web service portals. Furthermore, mashups are created by a large, diverse set of end-users, many of whom are not professional developers. Due to the above factors, the current Web services caching techniques would fail to achieve reasonable degrees of data reuse if employed in the mashup domain (please see Section II for a brief discussion in this regard). Thus, we need a dynamic approach to caching that analyzes internal structures of mashups and stores the data that yields maximum performance benefits.

A. Paper 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 present a formal model for mashups. In this model, each mashup is represented as a collection of operators forming a tree. This model serves as the basis for analyzing the costs and performance of mashups.
- This paper proposes a dynamic cache point selection scheme that estimates the benefits and costs of caching data at different stages of the mashup tree. Our approach selects a set of points that collectively maximize the benefit-to-cost ratio of caching data at those points.
- We design a scheme for indexing cached data which

enables MACE engine to efficiently discover whether any of the currently cached data can be reused in the execution of a newly created mashup.

We have conducted a series of experiments to study the performance of the MACE framework. The results demonstrate the effectiveness of the proposed techniques in improving the scalability of the mashup platform.

II. 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.

A. Motivation

Most Web service caches store the final results of Web service workflows or at pre-specified stages of the Web service processes¹. We contend that this *static* caching strategy would not be effective for mashups due to the inherent differences between mashups and Web service processes.

As exemplified by Yahoo pipes [18], mashup platforms typically host several thousand distinct mashups, whereas the number of *distinct* Web services in a typical Web services portal is relatively small. The frequency of 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 orders of magnitude 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.

Because of these traits, 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.

¹The cache is explicitly configured to store results at a certain point of the workflow.

B. Mashup Model

In this section, we develop a formal model for mashups. A mashup platform can be thought of as a system that 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. $MpSet = \{Mp_0, Mp_1, \dots, Mp_{N-1}\}$ represents the mashups existing in the mashup platform at a given point in time.

The mashup platform includes a set 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 operators. The *fetch* operator corresponds to the function of retrieving data required for a mashup from an external or an internal source, and a *dispatch* operator represents the function of dispatching the mashup results to the end user. $OpSet = \{op_0, op_1, \dots, op_{M-1}\}$ denotes the set of operators available in the mashup platform. Without loss of generality, operators op_{M-2} and op_{M-1} correspond to the fetch (represented as *fo*) and dispatch (*do*) operators, respectively. The rest of the $OpSet$ elements are data processing operators. Each operator 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 always produces the same type of output. For example, the sort operator 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.

Mashups comprise of a set of operators chosen from the $OpSet$. Every mashup contains one or more instance of the fetch operator and one dispatch operator. Specifically, a mashup is modeled as a *tree* with each node corresponding to a mashup operator. In this tree, the output of an operator node forms (one of the) inputs of its parent node. Furthermore, The dispatch operator always forms the root of the tree, and each leaf node corresponds to a fetch operator. $\{nd_0^l, nd_1^l, \dots, nd_{Q-1}^l\}$ represent the nodes in the tree of the mashup Mp_l , where each node corresponds to an operator from $OpSet$. 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 the ones that update them.

Each operator in $OpSet$ is associated with two functions. The *cost function*, represented as $CF^{op_j}(s_0, s_1, \dots, s_{q-1})$ for operator op_j represents the cost of performing the operation. The parameters s_0, s_1, \dots, 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

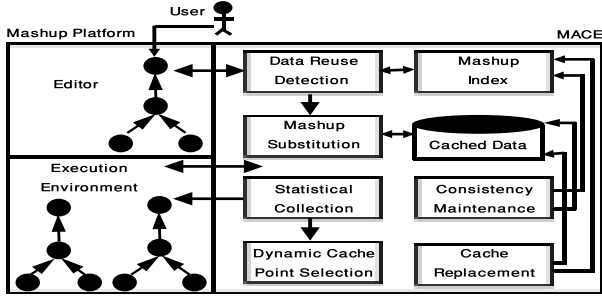


Figure 1. MACE Architecture

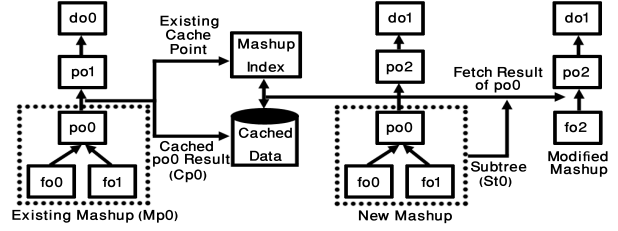


Figure 2. Cached Data Reuse in MACE

operation. In this paper, we quantify the cost of an operator through its latency. The output size estimation function, represented as $OSF^{op_j}(s_0, s_1, \dots, s_{q-1})$ captures size of the output of the operator op_j , where s_0, s_1, \dots, s_{q-1} are the sizes of the inputs. The *cost value* of a node nd_i^l (denoted as $CV^{nd_i^l}$) in the mashup Mp_l is the value of the cost function of the corresponding operator on the specific inputs indicated in the mashup tree. Similarly, the output size value $OSV^{nd_i^l}$ of the node nd_i^l is output size function of the corresponding operator evaluated on the inputs specified by the mashup tree.

The total cost of executing the mashup Mp_l is the sum of the cost values of all its operators. Similarly, the output size of mashup Mp_l is OSV of its root node.

III. MACE 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 cost 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* ($PcSet$), and individual nodes in this set are referred to as *potential cache points*.

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.

A. Cache Indexing for Efficient Data Reuse

Determining points of data reuse in new coming 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 an incoming mashup Mp_l if and only if the subtree represented by Cp_h *exactly matches* a subtree in Mp_l . 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_l are found to match existing cache points in MACE system, Mp_l 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_l , 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.

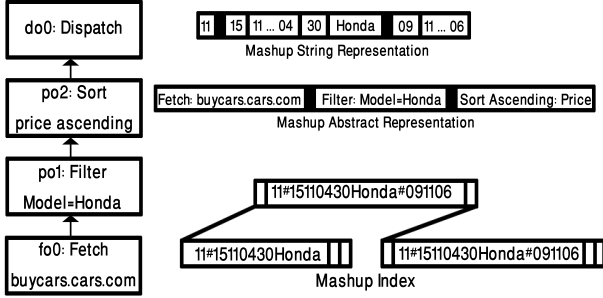


Figure 3. Mashup representation and index

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. Join operators 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 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#15110430Honda" will be inserted into the index. But if we decide to cache after the whole mashup execution flow is done, then "11#15110430Honda#091106" 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 (15110430Honda) 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.

IV. DYNAMIC CACHE POINT SELECTION

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.

A. 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 terminologies and notations that are employed in the problem formulation.

Potential cache point set ($PcpSet = \{Pcp_0, Pcp_1, \dots, Pcp_{M-1}\}$) 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 represents 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 ($CCV^{Pcp_k} = CV^{Pcp_k} + \sum_{Pcp_h \in Descendent(Pcp_k)} (CV^{Pcp_h})$).

Cost-Benefit Analysis

The benefits of caching the results at a particular potential cache point Pcp_k is that the cached data would be re-used 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 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 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 re-computed, 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 is directly proportional to the size of the output (OSV^{Pcp_k}). However, notice that the storage costs only matter 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 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_k} 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_k} be a $\{0, 1\}$ variable denoting whether Pcp_k is selected as a cache point (X^{Pcp_k} is 1 if Pcp_k is chosen and 0 otherwise). Therefore, the optimization criterion would be to assign X^{Pcp_k} values to each potential cache point $Pcp_k \in PcpSet$ such that $\sum_{Pcp_k \in PcpSet} X^{Pcp_k} \times CBT^{Pcp_k}$ 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 $Pcp_k \in Descendant(Pcp_i)$ or vice-versa, $X^{Pcp_k} + X^{Pcp_i} \leq 1$.

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 $\{X^{Pcp_0}, X^{Pcp_1}, \dots, X^{Pcp_{(M-1)}}\}$ corresponding to the potential cache points $\{Pcp_0, Pcp_1, \dots, Pcp_{M-1}\}$ such that $\sum_{Pcp_k \in PcpSet} X^{Pcp_k} \times CBT^{Pcp_k}$ is maximized while ensuring that the following constraints are not violated: (1) $X^{Pcp_k} \in \{0, 1\}, \forall Pcp_k \in PcSet$; (2) $\forall \{Pcp_k, Pcp_i\}$ such that $Pcp_k \in Descendant(Pcp_i) || Pcp_i \in Descendant(Pcp_k), X^{Pcp_k} + X^{Pcp_i} \leq 1$; and (3) $\sum_{Pcp_k \in PcSet} X^{Pcp_k} \times OSF^{Pcp_k}(ips) \leq Sg$, 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.

B. 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 CBT^{CSP} as $RF^{CSP} \times CCV^{CSP} - UF^{CSP} \times CCV^{CSP}$. We then compare the value of CBT^{CSP} 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.² 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

²The search may also terminate when we reach the end of the mashup tree (in either direction)

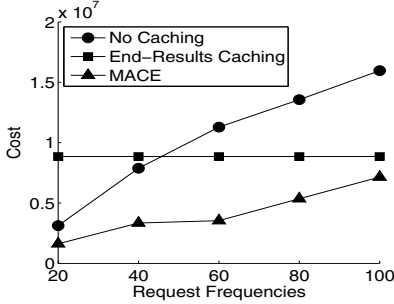


Figure 4. Total cost when request frequency is variable

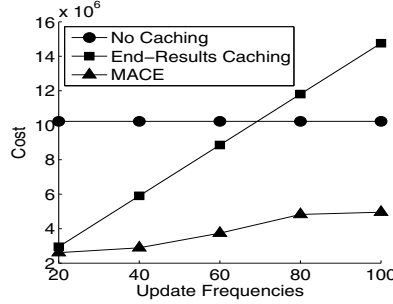


Figure 5. Total cost when update frequency is variable

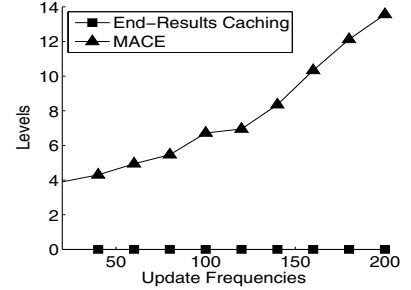


Figure 6. Level of Cache Points when update frequency is variable

at this step may exceed the available storage (S_g). The second stage of the algorithm performs additional level of pruning as follows. For each cache point P_{cp_k} in the CPSet produced at the end of first step, it calculates the ratio $BCS^{P_{cp_k}} = \frac{CBT^{P_{cp_k}}}{OSV^{P_{cp_k}}}$. This ratio quantifies the per-byte cost savings obtained by caching the results of P_{cp_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.

V. EXPERIMENTS AND RESULTS

The objective of our experiments is two fold, namely studying the impact of MACE’s dynamic cache point selection on the performance of the mashup platform and evaluating the benefits and overheads of the proposed cache point indexing scheme. First, we briefly describe the experimental setup.

A. Experimental Setup

Our experimental setup simulates a mashup environment with a mashup server, several data sources and many more end-users spread out on the Internet. The mashup server in our setup is, to a considerable extent, based upon the Yahoo Pipes environment [18]. Similar to Yahoo pipes our mashup platform contains 11 distinct operators. In order to come up with realistic cost functions and the output size functions for the various operators we performed 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. The number of distinct mashups existing at the platform varies with the experiment, and it ranges from 1000 to 5000. The network topology employed in our experiments is based upon the

measurement by DIMES [13] on the actual Internet in 2008. We use BRITE [8] and BRITE extension [16] to transform DIMES data into a more convenient form. Our topology has 378444 nodes.

B. 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 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 request 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). A Zipfian distribution with $\alpha = 60$ is used to model the popularity variations among the individual mashups. The mean of the update frequencies of the data sources (henceforth referred to as update frequency) is set to 60. In this experiment, 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 4 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 through out 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 changes³.

³First-time mashup executions also contribute towards the total costs in End-results caching but these costs are comparatively very small.

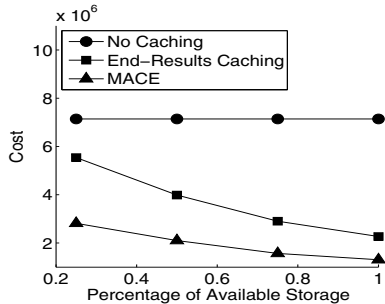


Figure 7. Total cost when available storage is variable

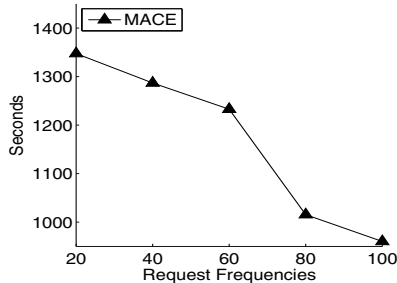


Figure 8. Total index access time when request frequency is variable

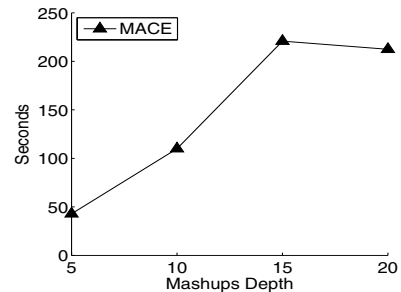


Figure 9. Total index access time when mashup depth is variable

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.

In the second experiment (Figure 5), we study 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 frequencies of all data sources is varied from 20 to 100 per unit time. Again, we see that the MACE system yields the 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.

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 6 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 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. LRU cache replacement is employed for all schemes. As Figure 7 demonstrates, MACE results in better performance by selecting cache points that

provide higher per-byte cost savings.

C. Mashup Index Analysis

In the second set of experiments, we 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 8 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 9 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.

VI. RELATED WORK

Research in the area of mashups is still in its nascent stages. MARIO [12] is a recent mashup editing tool in which

mashups are built from tags and executed using a planning algorithm. DAMIA [4] is a data integration service for situational applications in the enterprise domain. Kulathuramaiyer [7] describes a mashup for digital journals which enables its users explore digital libraries using semantic-rich meta-data. Subspace [6], adopts the sandboxing principle to isolate applications into trust layers.

On the other hand, Web content caching in general, and caching for Web services in particular have received considerable research attention [5], [17], [11], [19]. Issues such as caching granularity, caching architectures, consistency maintenance and data placement and replacement strategies have been extensively investigated. In WReX [14], a caching middleware architecture for caching XML web services responses is proposed. Terry et.al. [15] discuss caching XML web services for mobile clients. 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. Web services, in general, has been a hot area of research in which aspects such as description, discovery, composition, and efficiency of Web services are addressed [3], [2], [1].

VII. 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 experiments showed that MACE can significantly improve performance and scalability of mashup platforms.

REFERENCES

- [1] M. Aoyama, S. Weerawarana, H. Maruyama, C. Szyperski, K. Sullivan, and D. Lea. *Web services engineering: promises and challenges*. International Conference on Software Engineering, pages 647648, 2002.
- [2] D. Ardagna and B. Pernici. *Adaptive service composition in flexible processes*. IEEE Trans. on Software Engineering, 33(6):369384, June 2007.
- [3] B. Benatallah and H. R. Motahari-Nezhad. *Servicemosaic project: modeling, analysis and management of web services interactions*. Asia-Pacific conference on Conceptual modelling, 53:79, 2006.
- [4] IBM Corp. Damia. <http://services.alphaworks.ibm.com/damia/>, 2007.
- [5] A. Iyengar and J. Challenger. *Improving web server performance by caching dynamic data*. In USENIX Symposium on Internet Technologies and Systems, pages 4960, 1997.
- [6] C. Jackson and H. J. Wang. *Subspace: secure cross-domain communication for web mashups*. In WWW, pages 611620, New York, NY, USA, 2007. ACM Press.
- [7] N. Kulathuramaiyer. *Mashups: Emerging application development paradigm for a digital journal*. Journal of Universal Computer Science, 13(4):531542, April 2007.
- [8] A. Medina, A. Lakhina, I. Matta, and J. Byers. *Brite: an approach to universal topology generation*. In International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, pages 346353, August 2001.
- [9] Microsoft Corp. *Microsoft popfly*. <http://www.popfly.com/>.
- [10] Programmable Web. <http://www.programmableweb.com>.
- [11] L. Ramaswamy, A. Iyengar, L. Liu, and F. Douglass. *Automatic Fragment Detection in Dynamic Web Pages and Its Impact on Caching*. IEEE TKDE, 17(6):859874, 2005.
- [12] A. Riabov, E. Bouillet, M. Feblowitz, Z. Liu, and A. Ranganathan. *Wishful search: interactive composition of data mashups*. In WWW, pages 775784. ACM, 2008.
- [13] Y. Shavitt and E. Shir. *Dimes: let the internet measure itself*. ACM SIGCOMM, 35(5):71–74, May 2005.
- [14] J. Tatemura, O. Po, A. Sawires, D. Agrawal, and K. S. Candan. *Wrex: A scalable middleware architecture to enable xml caching for web-services*. Middleware, pages 124143, 2005.
- [15] D. B. Terry and V. Ramasubramanian. *Caching xml web services for mobility*. ACM Queue, 1(3):7078, May 2003.
- [16] M. Wahlsch, T. C. Schmidt, and W. Spat. *What is happening from behind? - making the impact of internet topology visible*. Campus-Wide Information Systems, 25(5):392406, 2008.
- [17] J. Wang. *A survey of web caching schemes for the internet*. ACM SIGCOMM, 29(5):3646, October 1999.
- [18] Yahoo Inc. *Yahoo pipes*. <http://pipes.yahoo.com/>, 2007.
- [19] J. Yin, L. Alvisi, M. Dahlin, and A. Iyengar. *Engineering Web Cache Consistency*. ACM Trans. Internet Techn., 2(3):224–259, 2002.