

Enhancing Scalability and Performance of Mashups through Merging and Operator Reordering

Osama Al-Haj Hassan
Computer Science Department
University of Georgia
 Athens, GA 30602, USA
 hasan@cs.uga.edu

Lakshmith Ramaswamy
Computer Science Department
University of Georgia
 Athens, GA 30602, USA
 laks@cs.uga.edu

John A. Miller
Computer Science Department
University of Georgia
 Athens, GA 30602, USA
 jam@cs.uga.edu

Abstract—Recently, mashups are gaining tremendous popularity as an important Web 2.0 application. Mashups provide end-users with an opportunity to create personalized Web services which aggregate and manipulate data from multiple diverse sources distributed across the Web. However, this increase in personalization also results in new scalability and performance challenges. Surprisingly, there are very few studies on the performance aspect of mashups. In this paper, we propose two novel techniques to enhance the scalability and performance of mashup platforms. The first is an efficient mashup merging scheme that avoids duplicate computations and unnecessary data retrievals by detecting common operator sequences in different mashups and executing them together. Second, we propose a canonical form-based mashup reordering scheme that not only transforms individual mashups to their most efficient forms but also increases the effectiveness of mashup merging. This paper also reports a number of experiments studying the benefits and costs of the proposed techniques.

Keywords-mashup; Web 2.0; personalization; performance

I. INTRODUCTION

Recently, Web 2.0 applications have garnered tremendous popularity. These applications provide end-users a set of powerful tools to query, retrieve and filter Web information according to their specific needs, thereby enabling personalized Web experience. Mashup [1] is one such Web 2.0 technology. Mashups can be thought of as personalized Web services which are designed and developed by end-users. Mashups essentially fetch data from several Web sources, which would then be aggregated, processed, and refined according to the needs of the end-users. Several mashup platforms are available including Yahoo pipes [2], and Intel MashMaker [3]. Mashups, while providing enhanced personalization, also pose distinct scalability and performance challenges. First, since mashups are designed by end-users, a mashup platform may have to potentially host very large number of mashups. Second, mashups fetch data from large numbers of diverse data sources distributed across the Internet. These data sources vary widely with respect to the characteristics of their data - while some data from some sources changes very frequently others may remain

unchanged for considerable durations. Furthermore, the data sources exhibit significant heterogeneity with respect to their popularity, performance and reliability. Fourth, mashups are designed by non-technical end-users who are likely not aware of the efficiency and performance implications of their design. Hence, it is unrealistic to expect mashups to be optimized from a performance stand-point. Due to these unique aspects, the performance enhancement schemes used in traditional Web applications become ineffective for mashups. Surprisingly, these efficiency and scalability issues of mashups have received very little attention from the research community. None, to our best knowledge, has comprehensively studied the performance aspects of mashup platforms nor has anyone systematically addressed the above challenges.

A. Contributions

Towards addressing these issues, this paper presents the design and evaluation of *AMMORE* - an Automative Mashup Merging and Operator *RE*ordering platform. In designing *AMMORE*, we explicitly consider the fact that mashups may have been developed by multiple end-users with varying levels of technical expertise. In order to overcome this challenge, *AMMORE* analyzes each mashup from a performance stand-point and transforms it to achieve high scalability and low execution overhead. Specifically, *AMMORE* incorporates the following unique features.

- With the objective of avoiding wasteful repetitive computations, *AMMORE* efficiently detects common components (operator sequences) from different mashups, executes them only once and uses the results in final computations of various mashups.
- *AMMORE* converts each mashup into its most efficient form by re-ordering the operators such that the operators that reduce data are executed early-on. Converting mashups into a pre-specified form also improves the effectiveness of common component detection.

We have developed a real *AMMORE* prototype. This paper reports a detailed experimental evaluation studying the costs and benefits of the proposed techniques.

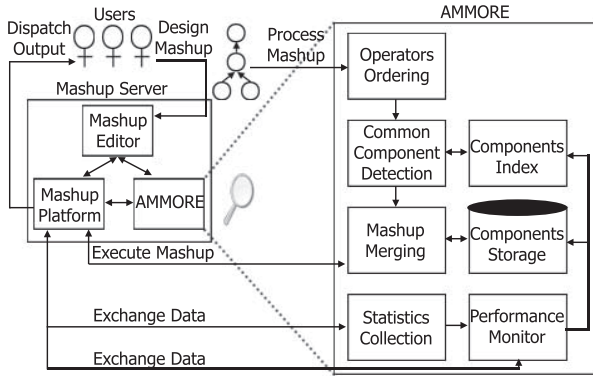


Figure 1. System Architecture showing *AMMORE* components and mashup platform

B. Motivation

A mashup platform has to execute mashups repeatedly. This places a huge burden on mashup platforms. One optimization that can enhance mashup execution is to detect common components across mashups. We represent mashups trees as strings, which means that common components across mashups appear as common subtrees with common substrings in mashup representation. When mashups are executed without common component detection, the mashup platform executes every occurrence of common components which is clearly unnecessary. If common component detection (CCD) is used, some mashups can be merged with other mashups and that minimizes the number of operators a mashup platform has to execute. For example, consider having two *identical* mashups, the two mashups fetch data from sports.yahoo.com then filter data based on topic equals Tennis. Four operators exist in these two mashups. Now, if common component detection (CCD) is used, these two mashups will be merged into one mashup resulting in only two operators. As we can see, the advantages of common component detection (CCD) are two fold. First, the number of components needed to represent mashups in the system is minimized which saves memory space. Second, redundant execution for mashup components is avoided which in turn minimizes delay of mashup execution.

A mashup execution depends on the order of operators forming that mashup. Consider two mashups, the first mashup fetches data from sports.yahoo.com then performs a filter operation followed by a sort operation. The second mashup is exactly the same as the first one except that the sort operator precedes the filter operator. The two mashups are equivalent because they generate the same output. If order of execution of operators is not considered here, then these two mashups would be treated as two different mashups. Another example of the importance of operators ordering is related to order of siblings in mashup trees. Sibling nodes in mashups can be found in only one case which is the case of having a

union operator where siblings are the two inputs of the union operator. Consider two mashups, the first mashup fetches data from sports.yahoo.com and then combines it with data fetched from news.google.com. The second mashup performs the same operation but in reverse order, that is fetching data from news.google.com then combining it with data fetched from sports.yahoo.com. These two mashups can exist because end-users do not follow any specific rules when designing mashups. Although these two mashups look different, they are equivalent. Therefore, the order of siblings for union operators has to be taken into account in order to detect equivalent mashups sequences.

C. Background and Architecture

A mashup takes a form of tree structure, such that mashup execution starts at leaf level by the fetch operators which fetch data from several sources over the Web. Mashup execution ends at the root level by the dispatch operator which sends result to the end-user. Several operators in between the root level and the leaf level exist which process data and refine it based on end-user needs, such as filter and sort operators. Each component in a mashup has a string representation that defines the component and it is constructed by concatenating the representation of component's attributes. For example, consider a filter operator that filters data coming from sports.yahoo.com such that topic equals Tennis. The representation of this operator is 15|10|07|30|Tennis where 15 is ID of the filter operator, 10 is the ID of the data source sports.yahoo.com, 07 is the ID of the attribute topic, 30 is the ID of the operation equals, and *Tennis* is the value on which topic values are filtered. Similar representations are used for the operators employed in our system, namely, fetch, union, filter, sort, unique, subelement, tail, truncate, count, and reverse.

A mashup is represented by concatenating the representation of each of its components. For example, 10#15|10|07|30|Tennis#17|05 is the representation of a mashup consisting of 3 components, the mashup starts by the component 10 which fetches data from sports.yahoo.com. That operator is followed by 15|10|07|30|Tennis which represents the previously explained filter operator. After that, the mashup concludes by the component 17|05 which corresponds to a truncate operator that keeps the first 5 elements of the data and eliminates the rest.

Figure 1 shows the components of *AMMORE* which is co-located with mashup platform in the mashup server. The Figure illustrates how our system interacts with the mashup platform. End-users typically design mashups using a mashup editor. Those mashups in addition to existing mashups are inputs for *AMMORE*. These inputs are taken by *AMMORE* to apply operators reordering on them. After that, the reordered mashups are fed to our common component detector which utilizes a mashup components index to detect common components across mashups. After common components are

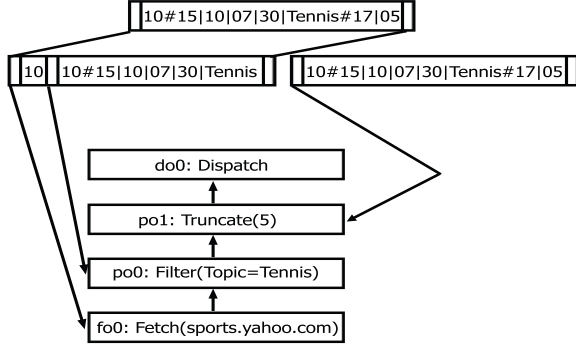


Figure 2. A Mashup example and how it is stored in components index

detected, they are taken as an input for our mashup merger where common components are merged. Following mashup merging, the mashup platform executes merged mashups by coordinating with *AMMORE*. The mashup platform and *AMMORE* exchange housekeeping information to monitor and maintain system performance.

II. COMMON COMPONENT DETECTION

As explained in Section I-C, mashup operators are represented as strings. As a result, mashup trees are represented as strings which also means that mashup subtrees are represented as strings as well. Detecting common components is equivalent to detecting subtrees in mashups. We are looking to find the longest common operator sequences, because this causes mashup tree representation to be more compact and saves unnecessary computation when executing mashups. Since operator sequences are represented as strings, then our problem is to find the longest common substring in mashups string representations. The following subsection introduces index structure that helps to make common component detection more efficient process. The subsection following that formalizes the common component detection problem. The last subsection proposes our common component detection technique.

A. Mashup Components Index

Detecting common components across mashups requires performing operator matching over mashup sets in the system which is an expensive process. In order to make the process efficient, we use a B+ tree index to index mashup components. The index keys are component's string representation. The index keys at the leaf level point to mashup components. Notice that each key in the index reflects its corresponding component and the components that precede it. This is important because we need to maintain the order of mashup components. Figure 2 shows a mashup example and its components stored in the index.

B. Mashup Common Component Definitions

Based on the previous challenges listed in Section I-B, we describe the following definitions to make common component detection problem for mashups more concrete. In order to detect common components across mashups, we need to define the term *COMMON* for two mashups.

Mashup platform has a set of mashups $\{Mp_0, Mp_1, \dots, Mp_{N-1}\}$. Each mashup Mp_i tree consists of set of nodes $\{nd_x^l, nd_y^l, \dots, nd_{Q-1}^l\}$. Each node corresponds to a specific operator. Two nodes nd_x^0 and nd_y^1 are *COMMON* across mashups Mp_0 and Mp_1 if and only if:

- Condition1: Mashup Mp_0 contains node nd_x^0 and mashup Mp_1 contains node nd_y^1 .
- Condition2: Node nd_x^0 has attribute list $AtList - nd_x^0 = \{At_0^{nd_x^0}, At_1^{nd_x^0}, \dots, At_{J-1}^{nd_x^0}\}$. Node nd_y^1 has attribute list $AtList - nd_y^1 = \{At_0^{nd_y^1}, At_1^{nd_y^1}, \dots, At_{J-1}^{nd_y^1}\}$, such that $At_0^{nd_x^0} = At_0^{nd_y^1}, At_1^{nd_x^0} = At_1^{nd_y^1}, \dots, At_{J-1}^{nd_x^0} = At_{J-1}^{nd_y^1}$.
- Condition3: The input for node nd_x^0 is data sources list $DsList - nd_x^0 = \{Ds_0^{nd_x^0}, Ds_1^{nd_x^0}, \dots, Ds_{K-1}^{nd_x^0}\}$ and the input for node nd_y^1 is data sources list $DsList - nd_y^1 = \{Ds_0^{nd_y^1}, Ds_1^{nd_y^1}, \dots, Ds_{K-1}^{nd_y^1}\}$ such that $Ds_0^{nd_x^0} = Ds_0^{nd_y^1}, Ds_1^{nd_x^0} = Ds_1^{nd_y^1}, \dots, Ds_{K-1}^{nd_x^0} = Ds_{K-1}^{nd_y^1}$.

Condition1 and condition2 guarantee that mashups Mp_0 and Mp_1 contain the same node while condition 3 makes sure that nodes nd_x^0 and nd_y^1 have the same data sources as input. A list of consecutive nodes in mashup Mp_0 $CoList - Mp_0 = \{nd_f^0, nd_{f+1}^0, \dots, nd_{H-1}^0\}$ and a list of consecutive nodes in mashup Mp_1 $CoList - Mp_1 = \{nd_z^1, nd_{z+1}^1, \dots, nd_{R-1}^1\}$ are considered common if and only if: - Condition4: nd_f^0 *COMMON* nd_z^1 , nd_{f+1}^0 *COMMON* nd_{z+1}^1 , \dots , nd_{H-1}^0 *COMMON* nd_{R-1}^1 .

- Condition5: nodes forming $CoList - Mp_0$ appear in the same order as nodes forming $CoList - Mp_1$.

C. Common Component Detection Technique (CCD)

This section describes our technique for detecting common components. The technique is targeted towards finding the longest common components sequences across mashups. The longer the sequence, the more components are considered as common across mashups which leads to more savings. To the best of our knowledge, subtree matching algorithms focus on matching subtrees of a pair (or a small set) of trees. However, they do not consider situations where a system would contain large numbers of trees and subtrees. This raises significant efficiency and scalability issues. Our system consists of potentially large number of mashups because of the personalization property of Web 2.0.

Therefore, we needed an algorithm that efficiently indexes mashups and utilizes the index for scalable subtree matching. The input for our algorithm is a set of mashups $MpSet = \{Mp_0, Mp_1, \dots, Mp_{N-1}\}$ and the output is a structure of merged mashup trees. This structure reflects mashups taking

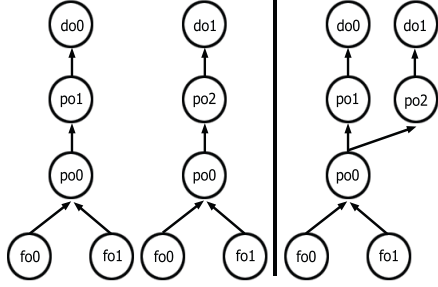


Figure 3. Two mashups and the result of merging them

into consideration common components. Figure 3 is an example of two mashups and the resulting structure of merging them. Each mashup consists of a set of branches such that each branch Br_i starts with a single fetch operator and ends with the single dispatch operator. Figure 4 shows a flowchart describing the common component detection process. The algorithm starts by iterating through all mashups in the system. For every mashup, we start at the beginning of each branch of the mashup. In other words, we start at the fetch operator in every branch of the mashup tree, if the operator’s representation exists in the mashup index, then we move to the next operator (its parent) to check its existence in the mashup index, we keep going for the next operator, because we are looking for the longest sequence of operators shared across mashups. If the operator’s representation does not exist in the mashup index, this means that all subsequent operators in the same branch also do not exist in the index because as explained in section I-C each operator’s representation includes the representation of its children. As a result, the operator and all subsequent operators in the mashup are added to the mashup index. If no further operators are left in the current mashup branch, then we move to the next mashup branch and when all operators within all branches of a single mashup are visited, then we move to the next mashup. The algorithm stops after iterating over all mashups in mashup set $MpSet$.

The common component detection process is efficient because each mashup is merged with existing mashups as the mashup is requested by end-user. For the sake of generality, we described the input of the algorithm as a set of mashups to be merged together. However, mashups are merged on the fly as they are requested by end-users.

To analyze the complexity of our algorithm, consider the case where a new mashup enters the system and the number of operators in the mashup tree is m . We need to traverse all operators in the mashup tree to find out if they exist in our mashup index. In the worst case, none of the mashup operators exist in the system. Traversing the mashup tree is of order $O(m)$. Now, in each traversed operator, we search the B+ tree index once for that operator, searching B+ tree complexity is $O(t * \log_t n)$, where t is the degree of the B+

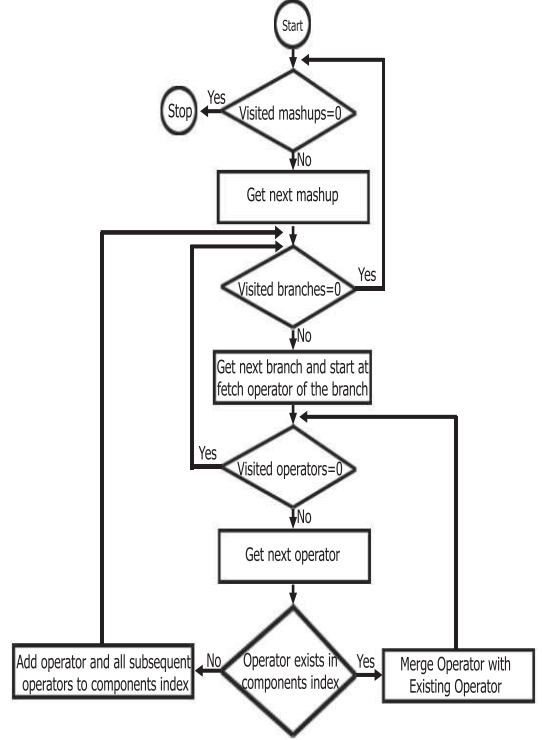


Figure 4. Common Component Detection Description

tree and n is the number of keys in the B+ tree. As a result, the whole algorithm complexity is $O(m * t * \log_t n)$ which is of a quadratic complexity.

III. OPERATOR REORDERING

The objectives of reordering are two fold; (1) Standardize the internal representation of mashups; two mashups that operate on the same data sources, use same set of operators and yield same end-results have identical representations. This improves the effectiveness of common component detection because it increases the probability of detecting common components; (2) Transform the mashup into a form that is more efficient from performance standpoint. For example, executing a filter operation before a sort operation is more efficient than the reverse order of execution.

A. Commutable Operators

One approach for standardizing mashup representation is to restrict a specific order on mashup operators such that the semantics of the mashup does not change. This can be achieved by detecting operators that can be interchanged without modifying mashup end-result; these operators are called commutable operators. The following operators are considered to be commutable

- Sort, filter, reverse, and unique are commutable
- Sub-element and sort are commutable; only when both operate on same attribute

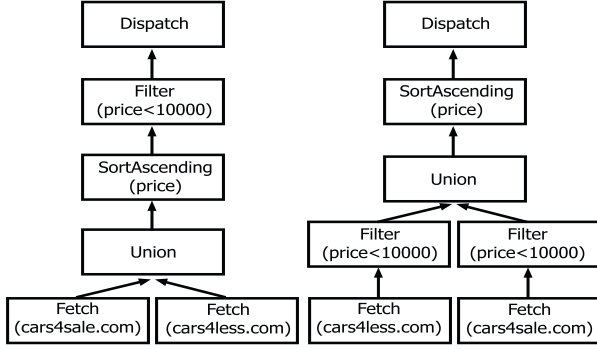


Figure 5. A mashup and its conversion to canonical form

- Sub-element and filter are commutable only when both operate on same attribute
- Sub-element and reverse are commutable

Based on the previous lists of commutable operators, a special order of commutable operators can be set which is described in the next subsection.

B. Canonical Form

Our canonical form defines a set of rules that we apply on mashups to make their design more standard. These rules are analogous to optimizing relational algebra expression trees. Our canonical form has the following rules. In these rules $x \rightarrow y$ means that operator x precedes operator y :

1. Filter \rightarrow Union
2. Filter \rightarrow Sort \rightarrow Reverse \rightarrow Unique
3. Sub-element \rightarrow Sort
4. Filter \rightarrow Sub-element
5. Sub-element \rightarrow Reverse
6. In any union operator, data sources appear in their lexicographical order.

If a union operator merges contents of data sources `sports.yahoo.com` and `news.google.com`, then `news.google.com` should appear to the left of `sports.yahoo.com` after enforcing the canonical form. The reason for making filter operators appear first in the previous rules is that a filter operator usually refines the data it receives as an input and the resulting output would usually be less in size than the input, this makes the operators that follow the filter operator execute faster because the size of their input data becomes smaller.

Figure 5 shows an example of a mashup designed by an end-user and its design after enforcing the canonical form. Canonical form rules are followed to make sure that all mashups abide to a more strict design structure. By applying the canonical form, we increase the probability of detecting common components across mashups and at the same time improve the efficiency of executing mashups. Notice that we cannot rely upon end-users to enforce canonical form, as they may not have the required expertise. After

the end-user completes designing his mashup, the system enforces the canonical form rules on the mashup.

IV. EXPERIMENTS AND RESULTS

The goals of our experiments are two fold; (1) evaluating the impact of common component detection on mashup execution; (2) evaluating the effect of operator reordering on the performance of mashup execution. First, we describe our prototype, then we describe our experimental setup. After that, the characteristics of our data set are described followed by the analysis of common component detection and canonical form techniques.

A. Prototype Development

In our *AMMORE* prototype implementation, we used a package called ROME[4] as the base of implementing our own feed processing operators. ROME is a package designed specifically to handle basic manipulation of Atom and RSS feeds. In our prototype, mashups are described in sets and enter the system as XML files. These XML files contain all mashups, their operators, and the attributes. Data sources information such as URL and popularity are embedded within fetch operators. We have not designed a visual interface for our platform, so end-users send their mashup files to our mashup server who is responsible of parsing the file, executing its mashups, and sending results to end-users.

B. Experimental Setup

Our mashup environment implementation contains one server which hosts a mashup platform, this mashup platform hosts mashups that fetch data from several data sources distributed across the Web. In our experiments, the most popular 1500 data sources in Syndic8 [5] were used for building mashups. Syndic8 is a repository for RSS and Atom feeds. 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 average number of fetch operators per mashup is 2 and the average number of operators per mashup varies from 5 to 20 operators. The number of subscriptions to a data source is representative of its popularity which is also extracted from Syndic8. Data processing operators are distributed randomly on mashups. We perform our experiments with several mashup sets that consist of 2000-10000 mashups. In the following experiments, regular execution corresponds to executing every single component of the input mashup set. Also, the term delay refers to the delay of executing the mashup set in milliseconds.

C. Feeds Characteristics and Trends

In this subsection, we state trends in the distribution of feeds over the Web. Finding these trends provides realistic information that can be used by researchers interested in

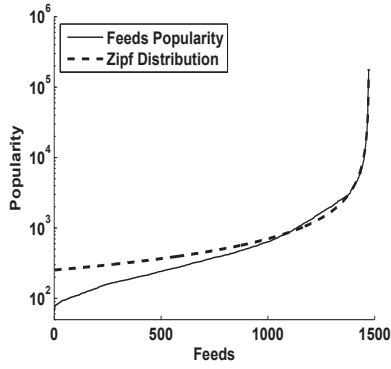


Figure 6. feeds popularity compared to Zipf distribution

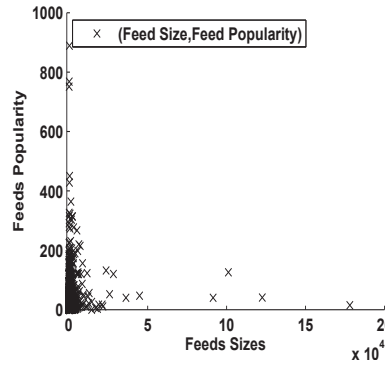


Figure 7. A plot of feeds data size and feeds popularity

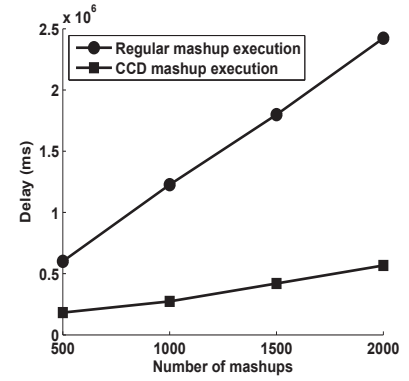


Figure 8. Delay of mashups execution when number of mashups is variable

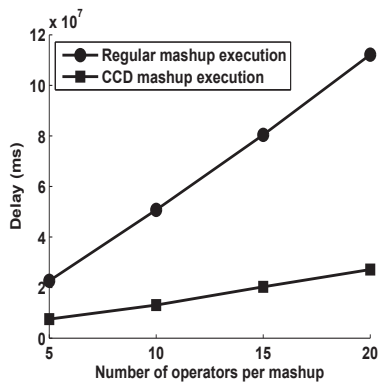


Figure 9. Delay of mashups execution when number of operators is variable

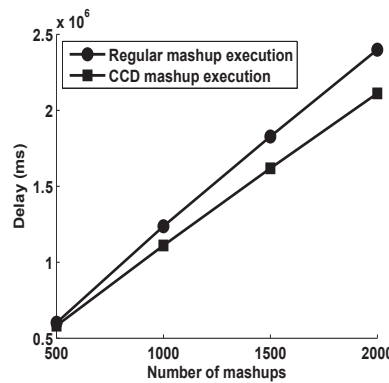


Figure 10. Delay of mashups execution when number of mashups is variable

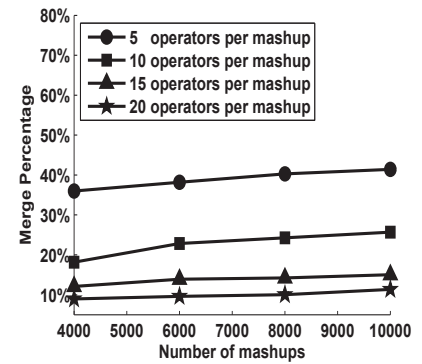


Figure 11. Merge percentage when using CCD

Web 2.0. Feeds in this experiment are the most 1500 popular feeds on Syndic8. Figure 6 represents feed popularity as measured by the number of subscriptions it has on Syndic8. Popularity is compared to a Zipf distribution with α (exponent value) = 0.9. Results show that feeds popularity on the Web closely resembles the Zipf distribution. The relationship between feed popularity and feed size is represented in Figure 7 which shows that most feeds have small data sizes and low popularity.

D. Common Component Detection Results

Executing mashups without common component detection requires the mashup platform to execute every operator in every mashup which is time consuming. In Figure 8, 5 operators are used in each mashup and in Figure 9, 2000 mashups are used. For both experiments, feeds are distributed on mashups based on a Zipf distribution with $\alpha = 0.9$. Both figures show that executing mashups with common component detection results in less delay because a subset of operators are detected as common across mashups. Accordingly, these common components are executed only once for each occurrence in all mashups. As a result, delay is minimized. Figure 8 also shows that delay increases as

the number of mashups increases. This is because more operators exist in the system which causes total mashup execution time to increase. The same behavior appears in Figure 9 because the number of operators is increasing which accordingly increases the time for executing mashups. In Figure 10, we repeat the experiment in Figure 8, but we distribute feeds on mashups according to uniform distribution. The Figure shows that our technique is less effective in this case because selecting feeds randomly minimizes the probability of finding identical feeds across mashups which consequently hurts the common component detection process. However, as indicated by Figure 6, feed popularity on the Web resembles a Zipf distribution which increases the probability of our technique to find common feeds across mashups because few feeds have very high popularity and they are repeated extensively across mashups. The rest of experiments use a Zipf distribution for distributing feeds on mashups. Merge percentage of common component detection (CCD) is plotted in Figure 11. We can notice that merge percentage increases as more mashups enter the system. The increase in number of mashups creates a richer pool of components which can be a potential for detecting common

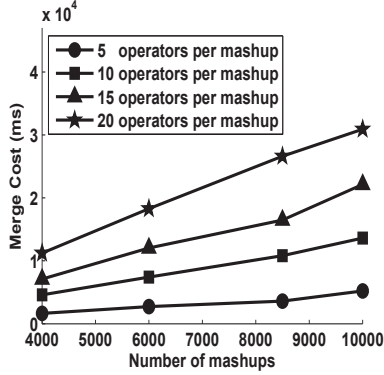


Figure 12. CCD Merge Cost when using different number of mashups and operators

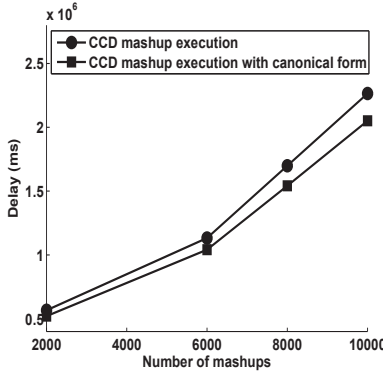


Figure 13. Delay of mashups execution when CCD is used with and without canonical form

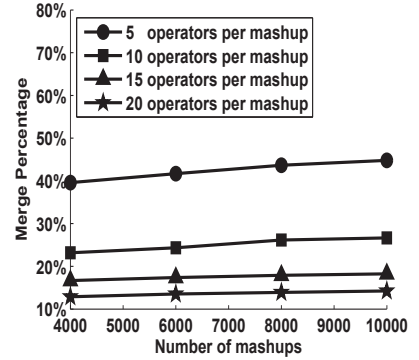


Figure 14. CCD Merge percentage when CCD and canonical form are used together

components across mashups. On the other hand, the merge percentage decreases as number of operators per mashup increases, this happens because mashup depth increases. Recall that one of the conditions for two sequences of components to be considered common is that the operators in these sequences must appear in the same order. When mashups depth increases, depth of sequences of operators increases which accordingly decreases possibility of finding identical sequences of operators. In other words, detecting common sequences of operators consisting of only two operators is more likely to happen than detecting sequences of operators consisting of 8 operators. In Figure 12 we notice that merging cost increases as the number of mashups increases which is a direct consequence of increasing number of operators in the system, but it is considered very small compared to delay savings. One point to mention here is that we expect the throughput of the system to increase with increasing concurrent end-user requests when CCD is used; this is because using common component detection decreases the total time needed for executing mashups.

E. Operator Reordering Results

The goal of having the canonical form is to transform mashups, so they are structured in a specific form. Doing this is expected to increase the effectiveness of common component detection. In Figure 13, 5 operators are used per mashup and the figure shows that delay of mashup execution decreases when CCD with canonical form is used as opposed to the case when only CCD is used. This happens because as Figure 14 shows, merge percentage increases when canonical form is used along with CCD because having operators appear in a specific order in mashup trees implies that we have higher probability of having more operator sequences detected as common across mashups. One point to mention is that when the number of mashups increases, the merge percentage also increases because a richer pool of mashups exists as more mashups enter the system. When number of operators per mashup increases, the merge percentage

decreases because mashup depth increases; when mashup depth increases, longer sequences of operators exist. The probability of detecting common operators in long operator sequences is lower than doing the same in shorter operator sequences. The experiment in Table I is performed on 2000 mashups. One thing the table shows is the delay of executing our mashup set when our system applies common component detection (CCD) only. It also shows the same when our system applies common component detection (CCD) in addition to operators reordering (Canonical Form). We can notice that the delay in the later case is smaller than the delay in the earlier case. This behavior is an indication of the effect of operators reordering in standardizing mashups structure which increases the probability of detecting common components across mashups, and that in turn minimizes delay of executing mashups. Column 4 in the same table shows the delay of applying canonical form on our mashup set; this delay is very low compared to delay savings of executing mashups. The explanation of this is that canonical form is applied separately on each mashup which is a fairly low overhead task. Notice that as number of operators per mashup increases, the delay of executing mashups and applying canonical form increase as well. Having more operators in the system causes common component detection and canonical form to take more time to conclude.

V. RELATED WORK

One way of enhancing the performance of mashup platforms is by using caching [6]. Authors introduce a mechanism to cache the results of executing mashup subtrees. Using caching is effective when feeds are mostly static; meaning that the data of feeds does not change frequently. However, when mashups use continuous feeds that change frequently, caching becomes less effective because of the burden of maintaining consistency of cached data. On the other hand, our system (*AMMORE*) eliminates redundant execution by detecting common components across mashups.

Table I
 DELAY (IN MILLISECONDS) OF APPLYING CANONICAL FORM
 COMPARED TO ITS SAVINGS

Number of Operators	With CCD Only	With CCD plus Canonical	Canonical Delay
5	2,596,498	2,550,254	16
10	3,135,249	3,043,486	22
15	3,701,486	3,544,296	29
20	4,267,846	4,053,295	42

The value of this scheme increases when continuous feeds are used in mashups. This targets the shortcoming of using caching in the case of continuous feeds. Detecting common components in general has been investigated in many systems such as [7] where improvement on flooding technique is proposed to reduce the number of duplicate messages in peer to peer networks. It is also used in mobile broadcast systems [8] to eliminate redundant messages. Common component detection has been used in tree structures. Tree isomorphism for ordered and unordered rooted trees is discussed in [9][10]. Detecting common content in XML document trees is studied in [11]. These techniques are not directly applicable to mashups because they only try to find if a tree A exists as a subtree in tree B which is a valid test for trees in general. But, in mashups, trees are not just a structure of connected components. Connected components in mashups represent input going from one operator to another. Depending on this input, the result of mashup execution differs. This is why two subtrees can be considered identical only when they share the same data sources. Mashup trees are special case of trees which has its own design rules and execution requirements. These rules and requirements are not handled in regular subtree matching algorithms. Moreover, to the best of our knowledge, subtree matching algorithms handle matching subtrees of a pair (or a small set) of trees. However, they do not take into consideration cases where systems contain a large number of trees and subtrees. Such a large number of trees and subtrees affects the efficiency and scalability of those systems. Mashup platforms (including our platform) can host potentially large number of mashups. This is due to the high degree of personalization offered by Web 2.0 which empowers end-users to create their own personalized mashups. Therefore, we need an algorithm that efficiently access mashups. Consequently, we proposed B+ tree structure which is used to access mashup components efficiently. This index is also utilized for scalable subtree matching over mashups.

VI. CONCLUSION

Mashups, while providing improved Web personalization, pose new scalability and performance challenges. In this work, we proposed common component detection which is used to reduce delay resulting from executing repeated

components. We also presented the use of a canonical form which increases the probability of detecting common components. In addition, we performed detailed experiments showing the savings and costs of our techniques. Further, we showed important characteristics of feeds over the Web. Results show that our techniques can significantly improve the performance of mashup execution.

REFERENCES

- [1] Programmable Web, <http://www.programmableweb.com>.
- [2] Yahoo Inc, "Yahoo pipes," <http://pipes.yahoo.com>, 2007.
- [3] Intel Corp., "Mash maker," <http://mashmaker.intel.com/web/>, 2007.
- [4] Sun Microsystems Inc., CollabNet Inc., and Cognisync Ilc, "Rome," <https://rome.dev.java.net/>.
- [5] Barr, Jeff and Kearney, Bill, "syndic8 feeds repository," <http://www.syndic8.com>.
- [6] O. Al-Haj Hassan, L. Ramaswamy, and J. A. Miller, "Mace: A dynamic caching framework for mashups," in *ICWS*, July 2009, pp. 75–82.
- [7] Q. Ly, P. Cao, E. Cohen, K. Li, and S. Shenker, "Search and replication in unstructured peer-to-peer networks," in *International conference on supercomputing*, 2002, pp. 84–95.
- [8] P. Wei and L. Xicheng, "AHBP: An efficient broadcast protocol for mobile Ad hoc networks," *Journal of computer science and technology*, vol. 16, no. 2, pp. 114–125, 2001.
- [9] F. Luccio, A. M. Enriquez, P. O. Rieumont, and L. Pagli, "Exact rooted subtree matching in sublinear time," Universita Di Pisa, Tech. Rep., 2001.
- [10] R. Cole, R. Hariharani, and P. Indyk, "Tree pattern matching and subset matching in deterministic $o(n \log^3 n)$ -time," in *Symposium on discrete algorithms*, 1999, pp. 245–254.
- [11] W. Liang and H. Yokota, "A path-sequence based discrimination for subtree matching in approximate xml joins," in *ICDEW*. Washington, DC, USA: IEEE Computer Society, 2006, p. x116.