Optimizing Pipe-like Mashup Execution for Improving Resource Utilization

Jingbo Xu, Hailong Sun, Xu Wang, Xudong Liu, Richong Zhang
School of Computer Science and Engineering
Beihang University
Beijing, China 100191
Email: {xujingbo, sunhl, wangxu, liuxd, zhangrc}@act.buaa.edu.cn

***Abstract*—Mashup is usually created by end-users to provide new services by combining data or functionality from multiple sources on the Web. Given that a mashup may have millions of concurrent user access, it is essential to work out a framework to optimize the runtime engine hosting the running of a myriad of pipe-like mashups. According to our analysis, memory is the primary resource to run mashups and we work out a metric named PMT to measure the memory consumption. A scheduling framework is then put forward consisting of mashup decomposition and a PMT-aware scheduling policy, which is named "lazy-start" designed to improve memory utilization. A set of experiments are performed to show the effectiveness and efficiency of this framework.**

***Keywords*-mashup; web 2.0; web service; personalization; performance optimization**

I. Introduction

A mashup is a web application that combines data or functionality from several sources to provide a new service. Many industry pioneers like Google, IBM and Yahoo have launched their own mashup platforms. Yahoo! Pipes, for example, has done an extraordinarily good job with over thousands of individual mashups being created and executed over 5,000,000 times each day[1]. There can be a large number of users to run their mashups concurrently, meanwhile the infrastructure resources (e.g. memory) are usually limited. Therefore, optimizing mashup execution performance and improving resource utilization is of great importance for a mashup runtime engine.

In this work, we first observe that most mashups are developed to combine contents from various web sources (e.g. RSS feeds) and there is little computing-intensive processing during the execution. This means that in most cases the mashup execution mainly consumes memory resource instead of much CPU capacity. Based on this observation, we define a metric called PMT (Product of Memory and Time) to measure memory consumption and propose a scheduling framework to optimize memory resource utilization. We are especially concerned with pipe-like mashups (just as Yahoo! Pipes), which are usually in the form of inverse trees. In this framework, a mashup tree is decomposed into a set of components called mashlets, which are the basic scheduling units. Second, we find out that mashlets belonging to one *Merge* operator have to wait for each other at *Merge* operator. With this observation, we design a lazy-start scheduling policy to avoid useless memory holding. Major contributions of this work are as follows:

- We identify execution of most mashups mainly consumes memory resources and propose to use PMT to measure the memory consumption;
- We propose a PMT-aware scheduling framework for runtime engine so as to improve memory utilization, in which a pipe-like mashup is decomposed to mashlets for efficient scheduling;
- On the basis of analysis of mashup structure, we design a "lazy-start" scheduling strategy to reduce the memory holding time of mashlets.

The rest of this paper is organized as follows. Section II introduces related work; in Section III, we formalize the performance optimization problem studied in this work; Section IV presents the scheduling framework and our lazy-start scheduling policy; in Section V, we present experiment evaluation results; finally Section VI concludes this work.

II. Related Work

Since its advent and popularity, mashup has been drawing a lot of attentions of researchers. A significant body of research on mashup is performance optimization. One approach to enhancing mashups efficiency is to use a cache. Study in [2] proposes a dynamic cache framework specifically designed for mashup. The cache framework stores results at intermediate stages of mashup workflows. Another study in [3] proposes a common component detection which is used to reduce the delay resulting from executing repeated components. And they propose an approach to increase the probability of detecting common components by reordering the operators. The study in [4] focuses on identifying code smells indicative of the deficiencies in mashups. Authors introduce refactoring targeted the code smells, reducing the complexity of the mashups and standardizing their structures to fit the community development patterns. A thread management strategy for server supporting the execution of event driven mashups is discussed in [5][6]. They classify the Web APIs into three categories and designed three service proxy structures correspondingly, which can be used as a reference when implementing new basic mashable components.

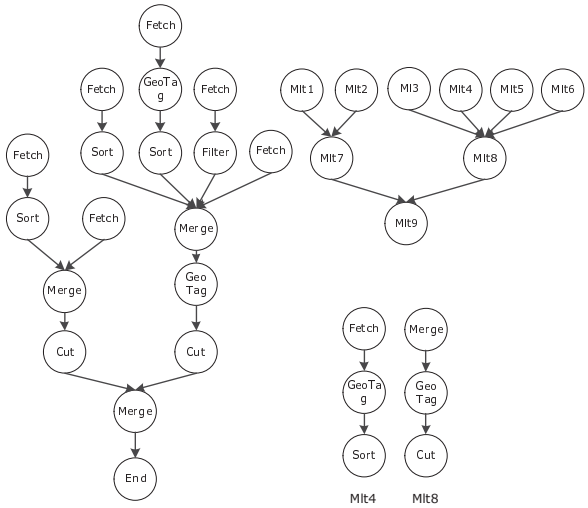

Figure 1. Decomposing a mashup tree to mashlets, where the left is the original mashup tree, and the upper right shows the mashlet representations and lower right Mlt4 and Mlt8 as two mashlets.

These works have made certain achievements on the performance improvement. However, they made mashups more effective merely by refactoring them and changing the original structure, which inevitably conducts misunderstanding of users' intent. Moreover, it takes more time to analysis mashups and cannot meet the real-time demand in execution, Even some works above are implemented in offline mode.

III. Problem Statement

A. Mashup Model

Mashup can be seen as a web application that fetches multiple data sources over the Internet, and processes the fetched data and dispatches the results to end-users. A mashup platform has some basic fetching and processing operators, including *Fetch*, *Filter*, *Sort*, *Merge*, *Cut*, *End*, etc. Formally, a mashup, Mp, can be represented by a tree $Mp = \{N, E\}$, where N denotes the set of operators in the mashup platform and E is the edge set which shows the input and output dependencies between operators. Specifically, intermediate results are generated by *Fetch* operators and are manipulated by corresponding operators as they flow across the nodes. Multiple intermediate results are merged together when they meet a *Merge* node and are finally dispatched by the *End* operator to the end user as the final results.

We propose a data structure "mashlet" as scheduling unit. Mashlets are fragments of mashup and produced by splitting the mashup on every *Merge* operator. A mashlet can be represented as a quadruple, $mlt_i = \{id, ISet_i, O_i, oprSeq_i\}$, means a mashlet mlt_i contains several input flows $ISet_i = \{I_i^1, I_i^2 ...\}$, only one output dataflow O_i as its result, and a sequence of operators $oprSeq_i$ manipulating the data. Thus the execution of a mashlet is to process data from input flows with the operators one by one and transmit result to the next mashlet through $output$. A mashup returns a final result when the last mashlet ends processing.

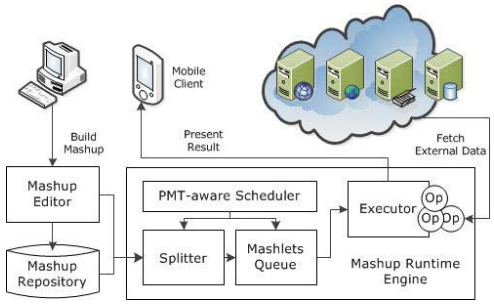

Figure 2. Overview of PMT-Aware Scheduling Framework

We can estimate the execution time of a mashlet based on historical statistics. T_{opr}^i denotes the average consuming time of opr_i. The predicted run time of mlt_i can be denoted as $prTime_i = \sum T_{opr}$, where opr is in $oprSeq_i$. In addition, we introduce a concept named "Sibling Mashlets" to represent the relationship of mashlets have the same parent node in the mashlet tree. Sibling mashlets may be interdependent with each other during the execution.

B. Performance Metric

In our study, we find that mashups mainly consumes memory resources on the platform. The data stream is filled up on Fetch operators at the beginning and has minor changes afterwards, while operators process the dataflow with less CPU usage. Thus the memory allocation for mashlets is easier to meet bottleneck compared to other resources.

We introduce a metric named PMT, which means Product of Memory and Time, to measure memory resource consumption of a mashlet in time and space dimensions.

Our goal is to design a scheduling policy to minimize the accumulated PMT of a single mashup in its whole life cycle, so as to archive a high effective rate of utilization of the memory resources of the platform.

IV. PMT-aware Scheduling Framework

A. Framework Overview

Figure 2 shows the architecture for our mashup platform. The platform mainly contains 4 parts, namely *Mashup Editor*, *Mobile Client*, *Repository* and *Mashup Runtime Engine*. Users can develop a mashup with *Mashup Editor* and use a mashup with the *Mobile Client*. The *Mashup Repository* is designed to store mashups for reuse. Figure 2 also illustrates four critical modules in the *Mashup Runtime Engine*. When a mashup is sent to the engine, *Splitter* analyzes its structure and decomposes it to several mashlets. The mashlets are added to the *Mashlets Queue*. *PMT Scheduler* is implemented repeatedly runs to refresh the PMT value of mashlets and determines which one to be scheduled to *Executor* next. The *Executor* maintains many operators. Every operator runs

its own functionality such as fetching or manipulating data according to certain rule.

B. PMT-aware Scheduling Policy

Let us look at the life cycle of a mashlet. It is filled up with input data at the beginning and frees itself after transmitting the data flow to the subsequent mashlet. As previously mentioned, the memory consumption can be measured with PMT.

We have a guiding principle for the design of scheduling policy. A mashlet with a high PMT value has a high priority. The significance of this rule is to run the most resources consuming mashlet and free its holding memory.

Based on this principle, we give a native Max-first PMT-aware scheduling policy works on the infrastructure depicted in Figure 2. Algorithm description is presented as follows.

Algorithm 1: PMT-aware Scheduling Policy

Require: select some mashlets to execute. where executing pool and scheduling queue are both singlton.
Input : mashup execute requests

```
for mashup in requests do
    TempSet ← Splitter(mashup)
    for mashlet in TempSet do
        CalculatePMT(mashlet)
        AddToQueue(mashlet)
    end
end
while queue ≠ 0 and pool.isNotFull do
    t ← pool.remainSize
    for mashlet in queue do
        UpdatePMT(mashlet)
    end
    pool ← SelectMashlets(queue, t)
end
```

1) Mashups are committed to the platform and decomposed to mashlets by the Splitter.
2) At first, PMT Scheduler associates a roughly PMT value to a mashlet approximately based on historical statistics.
3) Mashlets are inserted into Scheduling Queue. Then their PMT values are updated repeatedly by the PMT Scheduler along the waiting.
4) Mashlets with the highest PMT value will pop up from Scheduling Queue and be executed.

PMT Scheduler calculates PMT value in stage 2 and 3. In the calculation, memory size occupied by a mashlet in its running time basically is the sum of its input data streams. To take note of a phenomenon, mashlets with leaf-nodes in the original mashup tree has no input data and they begin with *Fetch* operators. We treat data, executed after *Fetch*, as input data without loss of generality.

We use PMT_{run} to represent PMT value of a mashlet during its execution, in which the $ISize_i^k$ denotes the *k*th input memory size of $mashlet_i$.

$$PMT_{run} = prTime * \sum_k ISize_i^k \tag{1}$$

C. Lazy Start-based Scheduling Algorithm

We investigate the execution of the mashup as shown in Figure 1. Four mashlets start at the same time but may not arrive at *Merge* operator simultaneously because mlt_4 takes more time to extract geo information. mlt_3, mlt_5 and mlt_6 have to wait for mlt_4 to complete its processing with their own results ready for merge. Since mashlets have to wait for synchronization at the *Merge* node, It is a natural optimization to reduce the memory consumption during this block time.

Here we introduce a concept named "lazy-start" to describe this optimization. Lazy-start is a method to start mashlets of a mashup asynchronously so that data flows can arrive at the *Merge* points simultaneously and blocks between sibling mashlets are eliminated.

But the effect of lazy-start is affected by two constraints in practical scheduling. First, because of the limited resources and load capacity, mlt_i may not be scheduled to run at t_0^i exactly. Second, the deviation introduced by estimation may affect t_0^i calculation. By taking these into account, we give a formula to illustrate the cost caused by a mashlet delay from its assigned start time. The cost comes from its sibling mashlets and its precursor mashlets waiting for processing with their result size. Similarly, we use $OSize^i$ to represent the output memory size of mlt_i.

$$PMT_{delay} = (t - t_0) * (\sum_p OSize_{sibl}^p + \sum_q OSize_{prec}^q) \tag{2}$$

Then we use calculator of PMT to $PMT = PMT_{run} + PMT_{delay}$. PMT denotes the chain cost of a delay and urgency degree of mashlets, the scheduling policy based on this dynamic factor becomes more effective.

PMT embodies some properties. First, PMT_{delay} increases over time, which means the priority of a mashlet grows along with delay and starvation has been avoided. Second, the chain effect of delay ensures the scheduling of a mashup as a whole. After a mashlet has been executed, the PMT_{delay} of its siblings or successors increases, which accelerate the whole mashup to be executed.

V. Experiments and Results

A. Experimental Setup

In our experiment, we develop a mashup platform providing 8 distinct featured modules, namely *Fetch*, *Sort*, *Filter*,

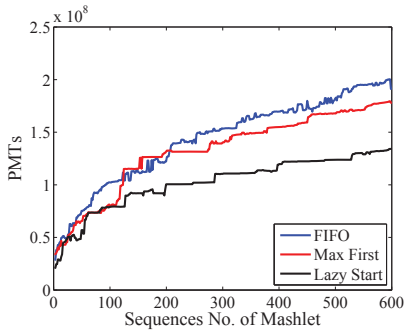

Figure 3. PMTs of mashlets at 200 concurrent mashup requests.

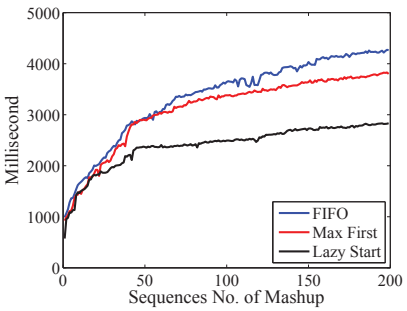

Figure 4. Execution time of mashlets at 200 concurrent mashup requests.

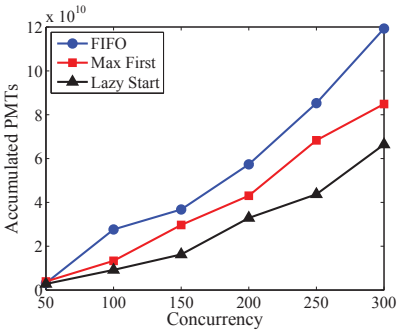

Figure 5. Accumulated PMT of mashlets varing with concurrency.

Cut, *GeoTag*, *Merge*, *Sub-element* and *Unique*, as the atomic operators. We use a mashup containing three mashlets as the test case, which generates combined news from RSS feeds on QQ.com and Sina.com with average size of 50K. The number of concurrent mashup execution requests varies with the experiment, ranging from 50 to 300. We calculate the real result size of a mashlet in bytes, and timing in millisecond.

B. Results and Analysis

Firstly, we quantify the performance of memory utilization. We generate 200 mashup requests concurrently and record the PMT of corresponding mashlets. Figure 3 shows PMT of mashlets as the lazy-start is applied, mashlets start asynchronously and in a more effective order. Thus, PMT reduces significantly by 28.65% through avoid wasting memory consumptions and blocking time.

We also find lazy-start scheduling reduces the execution time of individual mashup. In Figure 4, we draw each mashup execution time in their outcome order. When a mashlet ends execution, the calculation of PMT in lazy-start heighten the priority of its sibling mashlets and the subsequent one. Thus the mashlets coming from the same mashup would all be executed not far apart and bring a continuity, which reduces the execution time of the whole mashup. According to our experimental data, the average execution time of mashups at concurrency of 200 has shorted by 28.53% compared to that with FIFO scheduling.

In our second set of experiment, we evaluate the effects of lazy-start with different concurrency. We set a number of concurrent requests ranging from 50 to 300. As shown in Figure 5, the reduction of mashlet PMT and mashup execution time is affected by the concurrency with a positive correlation, varies from 12% to 44%.

VI. Conclusion

Mashup is a popular development method for end-users to create personalized services or content and the performance of runtime engine is an important issue for a mashup platform. In this work, we aim at improving the execution performance of pipe-like mashups. Taking the structure characterization of mashups into consideration, we propose a new metric PMT to measure the accumulated memory and design a "lazy-start" scheduling policy reduces the memory holding time of a mashlet and improves the memory utilization of the runtime engine.

We will further improve our scheduling policy by adapting the mashup model to a more general acyclic graph situation in the future work. Moreover, we will take other resource (e.g. CPU) into account to improve the utilization of multiple types of resources.

Acknowledgment

This work was supported by China 863 program (No. 2012AA011203), National Natural Science Foundation of China (No. 61103031), Specialized Research Fund for the Doctoral Program of Higher Education (No. 20111102120016) and Beijing Nova Program.

References

[1] Yahoo Inc, Yahoo! Pipes, http://pipes.yahoo.com.

[2] O. Hassan, L. Ramaswamy, and J. Miller, "Mace: A dynamic caching framework for mashups," in *ICWS 2009*. IEEE, pp. 75–82.

[3] O. Hassan, L. Ramaswamy, and J. Miller, "Enhancing scalability and performance of mashups through merging and operator reordering," in *ICWS, 2010*. IEEE, 2010, pp. 171–178.

[4] K. Stolee and S. Elbaum, "Refactoring pipe-like mashups for end-user programmers," in *Proceeding of the 33rd international conference on Software engineering*. ACM, 2011, pp. 81–90.

[5] M. Stecca and M. Maresca, "An execution platform for event driven mashups," in *Proceedings of the 11th International Conference on Information Integration and Web-based Applications & Services*. ACM, 2009, pp. 33–40.

[6] M. Stecca and M. Maresca, "Thread management in mashup execution platforms," in *Proceedings of the 12th International Conference on Information Integration and Web-based Applications & Services*. ACM, 2010, pp. 837–840.