Top-k Automatic Service Composition: A Parallel Method for Large-Scale Service Sets

Shuiguang Deng, Longtao Huang, Wei Tan, *Senior Member, IEEE*, and Zhaohui Wu, *Member, IEEE*

***Abstract*—Quality-of-Service (QoS)-aware web service composition is of great importance to assemble individual services into a composite one meeting functional and nonfunctional requirements. Given a large number of candidate services, automatic composition is essential so as to derive a composite service efficiently. Most existing methods return one solution that is optimal in some given criteria. This is somewhat rigid in terms of flexibility. In case some component service in the optimal composition becomes unavailable, the composition algorithm has to run again to find another optimal solution. Also, in a lot of circumstances users prefer multiple alternatives over a single one. Therefore, providing top-k service compositions according to their QoS is becoming more desirable. On another aspect, from the perspective of computation efficiency, due to the explosion of the searching space, single-threaded methods are usually not capable of handling a large number of candidate services. This paper tackles these two issues together, i.e., large-scale, QoS-based services composition yielding top-k solutions. The composition algorithm is based on the combination of backtrack search and depth-first search, which can be executed in a parallel way. Experiments are carried out based on the datasets provided by the WS-Challenge competition 2009 and China Web Service 2011. The results show that our approach can not only find the same optimal solution as the winning systems from these competitions, but also provide alternative solutions together with the optimal QoS.**

***Note to Practitioners*—To address the challenge of the automatic composition of web services in a large scale, this work proposes a novel approach to find top-k solutions with optimal global QoS. The problem of service composition is transformed into a graph searching problem. Each composition solution is represented in the form of a directed acyclic graph. The approach is divided into two stages, i.e., the run-up stage and the composition stage. Run-up stage deals with data preprocessing to parse service repository, which can be executed offline. In the composition stage, top-k compositions in response to users' queries are generated in parallel.**

***Index Terms*—Parallel implementation, quality-of-service (QoS), service composition, top-k.**

Manuscript received October 17, 2013; revised January 06, 2014; accepted February 06, 2014. Date of publication March 06, 2014; date of current version June 30, 2014. This paper was recommended for publication by Associate Editor C.-H. Chen and Editor H. Ding upon evaluation of the reviewers' comments. This work was supported in part by the National Key Technology Research and Development Program of China under Grant 2013BAD19B10 and the National Natural Science Foundation of China under Grant 61170033. *(Corresponding author: L. Huang.)*

S. Deng, L. Huang, and Z. Wu are with the College of Computer Science and Technology, Zhejiang University, Hangzhou, Zhejiang 310027 China (e-mail: dengsg@ zju.edu.cn; hlt218@zju.edu.cn; wzh@ zju.edu.cn).

W. Tan is with the IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598, USA (e-mail: wtan@us.ibm.com).

Digital Object Identifier 10.1109/TASE.2014.2306931

I. Introduction

SERVICE-ORIENTED Architecture (SOA) and its key implementation, web services, provides a promising solution for the seamless integration of single-functional applications to create new large-granularity and value-added services [1]. However, with the diversity and complexity of user requirements, it is increasingly difficult for a single service to fulfill a user's requirement. [2], [3] Therefore, Automatic Service Composition (ASC) technology has been proposed to automatically discover, select, and compose multiple individual services to satisfy a user's query (i.e., a request for a composite service) [4].

Most researches have transformed the automatic service composition problem into the AI planning problem where services correspond to operators in the planning domain and the composition corresponds to the plan [5]. Many approaches based on different kinds of AI planning techniques were used to solve the composition problem, including situation calculus [6], [7], state-space search [8]–[11], problem deduction [12], [13], and automatic theorem proving [14], [15]. Furthermore, Quality-of-Service (QoS) has drawn more attention when web services are selected from multiple candidates [16]. As a result, it is a challenge for composition systems to locate and select a proper component service quickly from so many candidate services on the Internet. In order to guarantee the overall QoS of the result composite services, many approaches have been proposed to generate composition service with QoS as well as function requirements [13], [17]–[20].

In general, the aforementioned QoS-aware service composition techniques have different features and can be applied in different situations. However, most of the existing approaches are based on single-thread computation. Once the number of services becomes large, these methods behave quite inefficiently because of the searching space explosion. Therefore, there is a need to develop a method to handle large-scale datasets efficiently.

In addition, most of the existing approaches are designed to return only the optimal composition with the best QoS. This has several limitations and may cause users some inconvenience. For example, in case some service in the optimal composition becomes unavailable, the whole composition is void and a recomposition is needed. Besides, returning only the optimal result cannot satisfy users' preferences for more alternatives. Hence, providing top-k service composition solutions that enjoy top-k QoS values, respectively, among all feasible solutions, can avoid these limitations and achieve benefits as follows.

1545-5955 © 2014 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

1) Feasibility. In case some component services in the optimal service composition are invalid, we can select another composition among the top-k as an alternative. This can improve the feasibility of the result set.
2) Diversity. As illustrated in cognitive science [21], users are willing to believe what they have been told most often by the greatest number of different of sources. For example, some users like cheaper compositions, while others prefer faster compositions. Thus, a composition result with small response time and higher cost will not satisfy both category. If top-k compositions with best response time are returned, a price-sensitive user can pick up cheaper ones from the top-k list. Furthermore, some users may have preferences that cannot be measured by QoS. For example, some users may be Google's fans and prefer to choose a service composition with Google's services even though the service composition's QoS is not the best. Therefore, providing top-k QoS compositions can provide diversified alternatives for users to satisfy their various preferences.
3) Load balance. Another benefit of providing top-k service compositions rather than only the optimal result is that this can avoid the optimal service composition to be overloaded and degrade.

Based on these motivations, top-k QoS composition is highly desirable. However, it is more difficult than those providing the optimal solution only. This is because service compositions with different structure may bear the same QoS. Under the same structure, there may be multiple candidate services with different QoS, which may affect the overall QoS of the composition. This makes it much harder to calculate top-k results when handling the complex structures of compositions. In order to tackle the problem of top-k QoS service composition, the China Web Service Cup [1] (CWSC2011) was held in 2011. We participated in CWSC2011 and won First Place. Our proposal aims at tackling the problem of making top-k QoS services composition from a large number of services, focusing on the following issues.

1) *QoS-awareness*. QoS refers to various nonfunctional properties of web services, such as response time, throughput, and availability, as well as some domain specific QoS attributes [22]. In this study, our goal is to return compositions with better QoS. For simplicity, in this paper we take response time as the only QoS criterion but it is easy to extend to other criteria later.
2) *Top-*k *compositions.* Almost all the existing methods aim at making the single optimal composition. Our composition algorithm is designed to return the best k compositions according to the QoS behavior.
3) *Scalability*. Most of the existing methods have met the performance bottleneck when dealing with large-scale datasets. To address the challenge of state-space explosion, we divide the composition process into several subtasks that can be executed in parallel.

In this study, we transformed the problem of service composition into a graph searching problem. Each composition is represented in the form of a directed acyclic graph (DAG). In addition, semantic annotations have been applied to help better match the queries and services. The proposed approach has two stages: run-up stage and composition stage. In the run-up stage, there is some preprocessing which can be executed offline. In the later stage, top-k compositions in response to users' queries are created in parallel. To evaluate the performance, we conducted a series of experiments based on the large-scale datasets from the WS-challenge 2009[2] and CWSC2011. Results show that the method can respond to composition requirements quickly and efficiently even with a large-scale dataset.

The contributions of this paper are as follows.

1) We propose a parallel method to address the problem of top-k QoS service compositions. This method is inspired by MapReduce that has promised massive performance gains for large-scale computations. Traditional service composition methods are mainly based on centralized mechanisms, which can hardly guarantee high scalability. As the number of web services on the Internet is increasing heavily, scalability has been identified as one of the most important challenges in the area of web service compositions. In particular, the top-k QoS service composition may take much computation work when the candidate service set is large. Therefore, determining how to improve the scalability is a key issue for top-k QoS service composition.
2) Based on the idea of MapReduce, we divide the top-k composition problem into several mutually independent subtasks that can be executed in parallel. This problem division can not only reduce the communication among parallel subtasks but also guarantee the validity of integrating results from subtask at the same time.
3) We use the dataset from China Web Service Cup 2011 (CWSC2011) to validate our solution. The experimental results show that our framework can perform with relatively high efficiency and accuracy. We do not have the solutions from other participants, however, our system has won the first place both on efficiency and accuracy in this competition.

The rest of this paper is organized as follows. Section II illustrates our motivation through a motivating scenario. Section III introduces the formal models used in the remainder of the text. Section IV describes the parallel framework for top-k service composition. Section V describes evaluation experiment and its results. Section VI reviews the related studies. Section VII concludes this research and discusses future directions.

II. A Motivating Scenario

In this section, we use an example to illustrate why finding top-k solutions is important. The example is about file processing services in document translation. Suppose that there are three categories of services related to file processing: language translating, format transforming, and file merging. The details of all available services including functions and QoS, are presented in Table I. Suppose that an Arabic student finds two references in English and French, respectively. In order to get better understanding, he wants the two references to be translated into Arabic and merged into a pdf file. Then the user request is, input a doc file *fileA* in English and a Latex™ file *fileB* in French, and

[1]http://debs.ict.ac.cn/cwsc2011/

[2]http://ws-challenge.georgetown.edu/wsc09/

TABLE I
WEB SERVICES ON FILE PROCESSING

Web Services	Function	Response Time (ms)
Fr2En_1	Translate language from French to English	80
En2Ar_1	Translate language from English to Arabic	100
En2Ar_2	Translate language from English to Arabic	120
En2Fr_1	Translate language from English to French	60
Latex2Pdf_1	Transform format from Latex™ to doc	150
Pdf2Doc_1	Transform format from pdf to doc	160
Doc2Pdf_1	Transform format from doc to pdf	180
Doc2Pdf_2	Transform format from doc to pdf	200
MergePdf_1	Merge pdf files	30
MergePdf_2	Merge pdf files	20
MergeDoc_1	Merge doc files	40

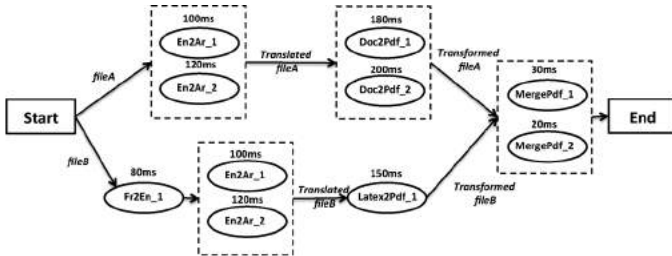


Fig. 1. A file processing system.

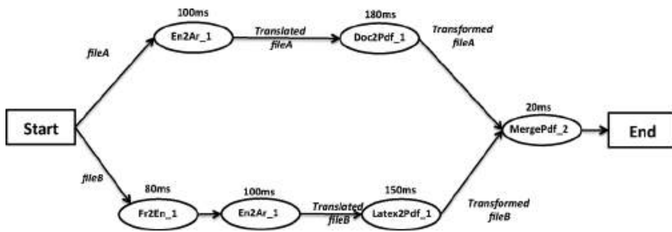


Fig. 2. The optimal composition result.

generate a merged pdf file in Arabic. For this request, we generate a dependency graph which contains all feasible solutions for the request (shown in Fig. 1). In Fig. 1, each node represents a service. The directed edges indicate the dependency among the component services. For example, an edge from service a to service b means that the output of a can cover some inputs of b.

Then, our goal is to choose the optimal solution (with smallest response time) from all feasible solutions. After calculating the response time of each feasible solution, the optimal service composition result is retrieved and presented in Fig. 2. We use a Directed Acyclic Graph (DAG) to describe the optimal result. Its overall response time is $\max((100+180);(80+100+150))+20=350\ \mathrm{ms}$.

For the optimal composition in this example, if the service *En2Ar_1* happens to be unavailable for some unpredicted reason such as network problem, server crashing, etc., then the whole composition cannot return the expected output to the user. In that case, the composition request has to be reissued and the composition system should replan an optimal composition which excludes the service *En2Ar_1*. If top-k composition results are returned to the user, the user can simply choose another composition without the service *En2Ar_1* in the top-k list when the optimal one is unavailable.

On the other hand, some users may have some special preferences which are not measured by QoS, such as brand bias and individual habits. For example, one user may prefer to choose *MergePdf_1* other than *MergePdf_2* because *MergePdf_1* is his favorite brand. Thus, the response time will be 360 ms, which is a little longer (10 ms) than the optimal one. This situation implies that returning only the optimal composition does not provide more alternatives for users to satisfy their various preferences.

Hence, providing top-k service composition results rather than the optimal result can bring more benefits. But it is not easy to solve the problem of top-k QoS service composition [55], [62], [63].

First, it is computationally infeasible to obtain top-k compositions by enumerating and ranking all feasible solutions, especially when the searching space is large. Let us take Fig. 2 as an example, if each service has 9 other alternatives, there will be 106 feasible solutions.

Second, current approaches for optimal service composition cannot be extended to top-k service compositions in a straightforward way. Intuitively, the basic method to get top-k results based on the current optimal approaches is as follows. First, the optimal composition result is achieved by these approaches. Then, the services participating in the optimal result are removed from the candidate service set and the next optimal result is generated with the remaining service set recursively. However, this cannot get the real top-k results. Because some services in the optimal result may also participate in other top-k results. If these services are excluded already, the top-k results will be inaccurate.

Furthermore, different composition patterns makes the problem more complicated. In this simple example, all possible service composition results share the same composition pattern in Fig. 1. It only needs to consider the candidates of each task to find top-k results. However, if there exists another service that can translate French to Arabic, other composition patterns can be generated. Then, it should consider different compositions in different composition patterns to get the final top-k results.

III. FORMAL MODEL FOR SERVICE COMPOSITION

We first define the key concepts used in service composition and the top-k problem. The formal framework for service composition in [46] is adopted and extended for top-k service composition.

A. Model Entities

Definition 1. (Web Service): A web service is defined as a triple $ws=\langle I,O,QoS\rangle$, where:

1) I is the set of input parameters.
2) O is the set of output parameters.
3) QoS is an n-tuple $\langle q_1,q_2,\ldots,q_n\rangle$, where each q_i denotes a QoS property of ws such as cost, response time, throughput, or availability.

Definition 2. (Ontology Tree): An ontology tree is defined as a 2-tuple $OT=\langle C,R\rangle$, where:

1) C is the set of concepts represented by nodes in the ontology tree.

TABLE II
EXAMPLE OF WEB SERVICES AND SEMANTIC WEB SERVICES

Web Services	Semantic Web Services
WebService1:	SemanticWebService1:
I: automobile	C_I: vehicle
O:price	C_O: money
QoS:<response time, accuracy>	QoS: <response time, accuracy>
WebService2:	SemanticWebService2:
I:car	C_I: vehicle
O:cost	C_O: money
QoS:<response time, accuracy>	QoS: <response time, accuracy>

2) R is the set of direct inheritance relationships represented by edges in the ontology tree.

In fact, an ontology tree represents the relations among semantic concepts. If a concept c_1 inherits from another concept c_2 directly or indirectly, it can be denoted as $c_1 \rightarrow c_2$.

Definition 3. (Semantic Web Service): Given a web service ws, the sematic form of ws is modeled as a triple $sws = \langle C_I, C_O, QoS \rangle$, where:

1) C_I is the set of the semantic concepts in an ontology tree, each of which corresponds to a parameter of $ws \cdot I$.
2) C_o is the set of the semantic concepts in the ontology tree, each of which corresponds to a parameter of $ws \cdot O$.
3) QoS is the same as Definition 1.

Table II shows several examples of web services and semantic web services. *WebService1* and *WebService2* are two services that provide the functionality of "querying the cost of cars." *SemanticWebService1* and *SemanticWebService2* are their semantic forms, respectively. The two services are different in terms of syntax. Without the help of semantics, it is difficult to find both services when searching with the keyword "car." In fact, "automobile" and "car" have the same semantic concept "vehicle"; they actually provide the same (semantic) functionality. Therefore, utilizing semantics can improve the recall for service discovery and composition.

Definition 4 (Semantic Dependency): Given two sets of concepts M and N, if $\forall m \in M(\exists n \in N(m \rightarrow n \vee m = n))$, then we say N semantically depends on M, denoted as $M \Rightarrow N$.

Definition 5. (Web Service Composition): A service composition is defined as a 4-tuple $wsc = \langle S, A, I, O \rangle$, where:

1) S is the set of services.
2) $A = \{a(s_i, s_j) | s_i \in S, s_j \in S\}$ is the set of dependencies between services in S. $a(s_i, s_j)$ represents that the inputs of s_j depend on the outputs of s_i semantically.
3) I is the set of concepts that are needed for wsc to be initialized.
4) O is the set of concepts that wsc outputs.

B. QoS Computation for Web Service Composition

According to [23], QoS properties can be categorized into two classes. One is *negative*, which means that the higher the value is, the lower the quality. Examples in this category include response time and price. The other is *positive*, that is, the higher the value is, the higher the quality. Examples include throughput and availability.

In this paper, we use *response time* as the only QoS parameter to explain our approach in the remaining parts of this paper. We may study the composition of multiple QoS properties in the future.

TABLE III
COMPUTATION RULES FOR QOS OF WEB SERVICE COMPOSITION

QoS property	F_1	F_2
cost	$\sum$	$\sum$
response time	$\sum$	Max
throughput	$\sum$	Min
availably	$\prod$	$\prod$

Given a service composition wsc, the QoS of wsc is the aggregation of the QoS of its component services. We adopt the computation rules in [23] to get the overall QoS of wsc, shown in Table III, where F_1 is the computation function for QoS of services in a sequence execution path. F_2 is the computation for QoS of multiple parallel paths. For the notations in the table, we only use their intuitive mathematic meanings. For example, "$\sum$" means summation, "$\prod$" means product, "$\max$" means maximum, and "$\min$" means minimum. The optimal QoS of a composition is the best value obtained from the computation rules from Table III. For multiple dimensions of QoS, the common solution is to add a weight to each QoS property and then get a weighted summation value.

The optimal service composition is the one with the optimal QoS. There may be multiple service compositions with the same optimal QoS. For simplicity, we only consider an one-dimensional QoS value (response time) in this paper. Still, it is easy to extend to other criteria by aggregating the overall QoS value of the service composition through the mentioned computation rules. If an efficient aggregating function of multiple QoS properties is provided, our proposal can also handle multiple-dimension QoS value.

C. Problem Definition

The aim of our approach is to make top-k QoS service compositions from a large-scale number of services. The problem can be defined as follows.

Definition 6. (Top-k QoS Service Composition): Given a set of web services S, a semantic ontology tree OT, and a user's request $R = \langle I, O \rangle$, where I is the set of requested input concepts and O the set of requested output concepts. The objective is to make top-k service compositions according to the global QoS. The set of compositions is denoted as SC, each of which (denoted as sc_i) should satisfy the following conditions.

1) $IR \cdot I \Rightarrow sc_i \cdot I \wedge sc_i \cdot O \Rightarrow R \cdot O$.
2) $|SC| = k$.
3) $\forall sc_i \in SC \neg(\exists sc' \in (SC_{all} - SC) \wedge sc' \cdot QoS \succ sc_i \cdot QoS)$.

where $\succ$ means "better than" (e.g., the response time is smaller), SC_{all} is the set of all possible service compositions that can satisfy the user request.

That means, the top-k composition results should actually rank from number 1 to number k w.r.t. QoS values in all feasible results. That is, the number of composition results is k. When $k = 1$, top-k service composition is equivalent to finding the

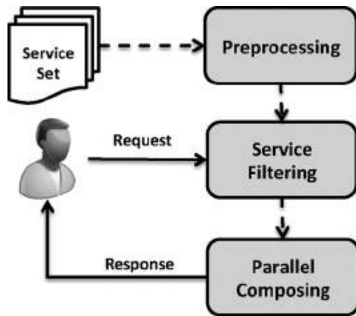


Fig. 3. Overall composition procedure.

optimal composition. For example, in Fig. 1, there are 16 composition results and their associated response time are: 4 results in 350 ms, 4 results in 360 ms, 4 results in 370 ms, and 4 results in 380 ms. Then, the top-5 results are the 4 results in 350 ms plus any one result in 360 ms.

IV. Parallel Service Composition

A. Overview

The architecture of our composition method involves three steps as outlined in Fig. 3. The first step is to preprocess the raw data of the large-scale dataset and transform the services set to a rule repository. The rule repository is a kind of in-memory data structure, which can be accessed fast and efficiently when responding users' requests. The second step is issued by a user's request. When receiving a user's request, it begins to fetch services from the Rule Repository and filter the services that are impossible to present in the final results according to the request and the rules in Rule Repository. Meanwhile the filtered services are layered in order for fast processing in the next step. The final step, which is the core of the whole architecture, is responsible for finding top-k composition result in parallel.

B. Preprocessing

In the preprocessing step, services in the service set are transformed into rules and a rule repository is constructed. Here, we involve a term "rule" which is defined as follows.

Definition 7. (Rule): A rule is defined as a 4-tuple, $r = \langle s, C_I, C_O, QoS\rangle$, where:

a) s is the service that generates the rule.
b) C_I is the set of the semantic concepts the rule needs as inputs.
c) C_o is the set of the semantic concepts the rule outputs.
d) QoS refers to the quality properties of the rule.

Each web service s can be transformed into a rule $r \cdot r \cdot C_I$ equals $s \cdot C_I$, and $r \cdot C_o$ is the conjunction of $s \cdot C_o$ and $s \cdot C_o$'s ancestor concepts. This means that the rule not only can generate the declared concepts of the service but also their ancestors. After all the services are transformed into rules, a rule repository is built. The rule repository stores the data in memory rather than in a database, which provides fast and efficient access when making compositions. Table IV presents an example of a rule repository.

In order to provide fast access to the rule repository, we build an inverted index for it. The rules are indexed by the concepts they can output. Table V shows the inverted index of the rule repository in Table IV. When we search for the concept E, we can find the rules $r3$ and $r9$ quickly.

TABLE IV
An Example of Rule Repository

Rule	s	C_I	C_O	Response Time (ms)
r1	w1	A	D	20
r2	w2	A	F	70
r3	w3	D	E	30
r4	w4	E,F	J	40
r5	w5	B	C	60
r6	w6	B	G	5
r7	w6	C,G	K	30
r8	w8	F	K	200
r9	w9	D	E,L	130
r10	w10	H	,G	20

TABLE V
Example of Inverted Index for Rule Repository

Concept	Rules
C	r5
D	r1
E	r3 ,r9
F	r2
...	...

C. Service Filtering

Since choosing appropriate services from a large-scale service set sharply increases the processing time, it is necessary to reduce the scale of the candidates. To this end, it is important to filter out useless services at the very beginning.

Hennig *et al.* have introduced an approach to filter out services when making compositions [28]. The process flow can be summarized as follows.

a) Generate a set "*INPUTS*;" the initial elements of the set are the inputs of the user's request.
b) Find all the services that can be triggered by the elements in "*INPUTS*." Then, the outputs of these services are added to the set "*INPUTS*" and the services are kept.
c) Repeat step b) until no more services can be triggered with the elements in "*INPUTS*." That is, the set "*INPUTS*" is not extensible. Then, all the services found are the filtered services.

Using the method above, services that cannot be triggered by the user's request will be filtered out. This largely reduces the number of candidates for compositions.

In this study, we make an extension for the filtering approach above to filter the large-scale service set. In the extension, we use the rules to find services triggered by the elements of "*INPUTS*," which can support semantic matching. We also make

TABLE VI
FILTERING AND LAYERING ALGORITHM

```
ALGORITHM1: FILTERING AND LAYERING
Input: S – the original service set
RR – the rule repository parsed from S
r – the user's requests
Output: FS – filtered service set
01: Initialize FS = {}, INPUTS= r.I, layer = 0.
02: while(INPUTS is extensible){
03:   for each rule in RR{
04:   if(rule can be triggered by the elements in INPUTS && rule.s
        hasn't appear in INPUTS){
05:   |   put rule.C_O into INPUTS;
06:   |   put rule.s into FS
07:   |   set the layer of rule.s as layer
08:   }
09:   layer++;
10: }
11: return FS;
```

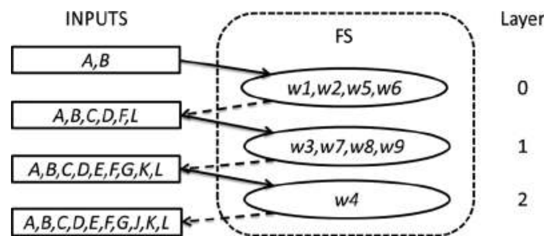


Fig. 4. An example of service filtering.

layers of the filtered services in order to avoid cycles when making compositions. After the execution of this algorithm, useless services are filtered out, and the remaining services are layered. The proposed extended filtering algorithm is illustrated in Table VI. Suppose that the total number of services is N, since each service is checked only once during the filtering process, the time complexity of this algorithm is $O(N)$. If the final "*INPUTS*" contains the concepts that the user's requested output concepts semantically depends on, then there must exist solutions for the user's request. Otherwise, there are no solutions for the user request, and then the following procedures needn't to be executed.

As an example, take the initial set $\text{INPUTS} = \{A, B\}$ and the rule repository in Table IV. For the first iteration, $r1$, $r2$, $r5$, and $r6$ can be triggered, and $INPUTS = \{A, B, C, D, F\}$. The filtered services set $FS = \{w1, w2, w5, w6\}$ with the level as 0. Then, the filtering algorithm continues until no rules can be triggered. The final filtering result is shown in Fig. 4. The service $w10$ is filtered out.

D. Parallel Composing

Parallel composing is the core step in our method. We adopt the basic idea of MapReduce to build a parallel framework for top-k service composition. MapReduce is a framework for processing parallelizable problems across huge datasets using a large number of nodes. MapReduce has proved to be a very successful way to solve large-scale computation and applied widely (e.g., Google file system, web access log analysis, document clustering, etc.). The issue of top-k service composition should find solutions from large-scale dataset and may take much computation work. Therefore, introducing the idea of MapReduce to solve the top-k service composition problem can produce sharp improvements in terms of efficiency and scalability. What we do in this proposal is to map the top-k service composition into multiple tasks that can be executed in parallel and guarantee that the results from all the tasks can form the final results. We also provide algorithms that can be applied for parallel execution.

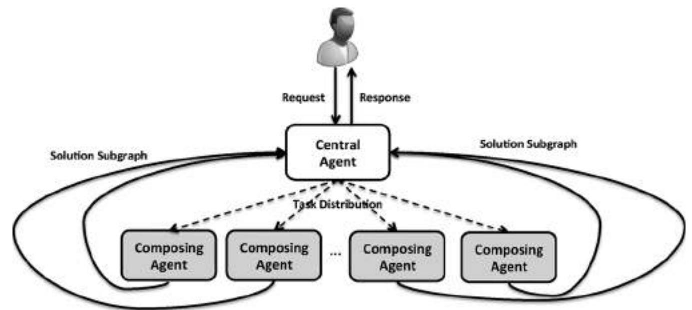


Fig. 5. Parallel composition framework.

Inspired on MapReduce, the parallel composition framework consists of a central agent and several composing agents (shown in Fig. 5). First, the central agent (corresponding to the master in MapReduce) processes the user request and generates several tasks that are mutually independent and can thus be executed in parallel, then distributes the tasks to each composing agent. Afterward, each composing agent (corresponding to the worker in MapReduce) starts to generate solution subgraphs for their given tasks. The generated solution subgraphs are then sent to the central agent to generate the final top-k results. This parallel composition framework is a generic framework of embarrassingly parallel. In implementation, it can leverage a wide spectrum of parallelization techniques, such as multi-threads, multi-processors, Hadoop, etc. For our experiments, we utilize multi-threads to implement the framework.

In order to illustrate our method clearly, we use an example through this subsection. The rule repository is shown in Table IV, and the user request is $R = \langle \{A, B\}, \{J, K\} \rangle$.

1) Task Generation: The first thing to do is task generation, which results in multiple tasks that can be executed in parallel. The formal definition of a task is as follows:

Definition 8. (Task): A task for a composing agent distributed by the central agent is represented as a 3-tuple $t = (c, RR, FS)$, where:

a) c is the target concept that this task needs to accomplish.
b) RR is the rule repository.
c) FS is the set of filtered services. The output of a task are the solution subgraphs for the target concept of the task.

The tasks are generated according to the user's requested output. Given the user request $R = \langle \{A, B\}, \{J, K\} \rangle$, two tasks are generated. The first is to generate solution subgraphs for J. The first is to generate solution subgraphs for K. Since the tasks are embarrassingly parallel, they can be executed without communicating with one another. We use parallelization degree metric to describe the parallel process capability of the parallel composition framework. The parallelization degree pd equals

the number of tasks executed in parallel, which can be calculated as follows:

$$pd = \min(n_a, n_t) \tag{1}$$

where n_a is the number of composing agents, n_t is the number of the tasks generated by the central agent.

2) Parallel Generation of Solution Subgraphs: The main responsibility of each composing agent is to generate the initial solutions on which the final results are based. The initial solutions are called solution subgraphs defined as follows.

Definition 9. (Solution Subgraph): A solution subgraph for a concept C is a directed graph from the user's input concepts to C through a series of rules, which can be modeled as a 4-tuple $\langle v_0, v_t, V, E\rangle$, where:

a) v_0 is the starting vertex of the solution graph representing the user's input concepts.
b) v_t is the terminate vertex of the solution graph representing the target concept C.
c) V is the set of other vertexes of the solution subgraph. Each vertex of V represents one service or a set of services with the same functionality.
d) E is the set of edges of the solution subgraphs. An edge from v_{from} to v_{to} means that v_{from} can provide some concept that v_{to} requires. For each vertex, its inputs must be satisfied by other vertexes' outputs based on the rule repository.

So the output of each parallel task are the generated solution subgraphs. The way to generate solution subgraphs is based on the backtracking algorithm, which has been introduced in several existing methods [29]–[32]. The generation of a solution graph starts from the terminate vertex v_t.

Traditional backtracking algorithms for service composition are mainly based on breadth-first-search, which may cost too much space and have low efficiency with a large-scale service set [46]. Hence, we develop a new backtracking algorithm for service composition based on depth-first-search (BTSC-DFS). Before illustrating the algorithm in detail, we introduce the basic principle of backtracking based on depth-first-search in BTSC-DFS.

3) Basic Principle: The target of BTSC-DFS is to generate a solution subgraph for a concept. For a concept con, we first search the inverted index of the rule repository and get the services that can output the concept con. For each round of backtracking, instead of choosing all these services to continue to backtrack, BTSC-DFS only chooses parts of them (may be one or multiple). If services that provide the concept con require the same inputs, they will be regarded as one common service chosen to backtrack. That is why the nodes in solution subgraphs may be a set of services. Fig. 6 shows an example using the rules repository in Table IV. The starting point of backtracking is the concept J, which is one concept of the user's requested outputs. From the rule repository, we can find that the service $w4$ can provide J and needs E and F as its inputs. For the next round of backtracking, we find that $w2$ can provide F, while $w3$ and $w9$ can provide E. Meanwhile, we find that $w3$ and $w9$ require the same inputs D. Thus, we can merge it as one common service so it can be followed by two alternatives of backtracking for $w2$ or $\{w3, w9\}$. Also, only one will be executed at each time according to the principle of depth-first-search.

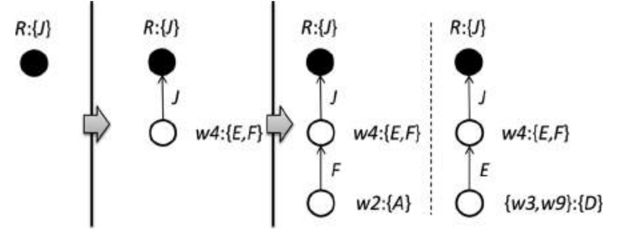


Fig. 6. An example of backtracking based on DFS.

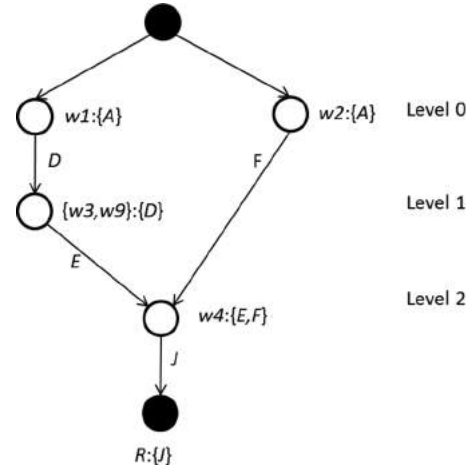


Fig. 7. An example of the critical path and noncritical paths.

For each solution graph generated by BTSC-DFS, there is at least one critical path and several noncritical paths in it. The critical path is based on the thoughts of layered services. For a service at layer k, at least one of its inputs must only be provided by some services at layer $k-1$. Then, there must be at least one path in which each service depends on the services at one upper layer and the last service accepts the user's request inputs. Such a path is called the critical path, and other paths in the solution subgraphs are noncritical paths. The critical path decides the pattern of the solution graph in a large part. The critical path is defined formally as follows.

Definition 10. (Critical Path): A critical path is an order sequence of nodes, $CP = \{n_1, n_2, n_3 \ldots, n_k\}$, in which each node represents one service or a set of multiple services as explained previously. For each pair of neighboring nodes, $\langle n_i, n_j\rangle$ $(i < j)$, at least one input of n_i can be provided by n_j, and the layer of n_i is 1 larger than n_j.

Fig. 7 shows an example of the critical path and noncritical paths. There are two paths in the solution graph. The path $\{w4, (w3, w9), w1\}$ is a critical path. Thus, all of its services depend on some associated services at the neighboring layer. The other path $\{w4, w2\}$ is a noncritical path.

BTSC-DFS consists of two steps. First, find all possible critical paths that output the required concept. Then, add noncritical paths to each critical path to form a solution graph such that all the inputs of services are provided by either the user or other services in the solution graph.

TABLE VII
ALGORITHM TO GENERATE CRITICAL PATHS

```
ALGORITHM2: GCP
Input:c –the required concept
RR – the rule repository
FS– filtered service set
Output: CPs–critical paths
01: Initialize CPs= {},cp= {}, inisvcs = {}.
02: for each rule in RR {
03:   if (rule. C_O contains c)
04:   |  put rule.s into inisrvs
05: }
06: for each svc in inisvcs {
07:   set tStackas a stack;
08:   set tmp = svc;
09:   tStack.push(tmp);
10:   put tmp into cp;
11:   while(tStackis not empty) {
12:   |  if(the layer of tmp is not 0) {
13:   |  |  i= the input of tmp that can be only provided by the
              services at the one upper layer.
14:   |  |  upperservices = the services that can provide i and at
              the one upper layer
15:   |  |  for each visited service vs in upperservices)
16:   |  |  |  for each cp in CPs
17:   |  |  |  |  if(cp contains vs)
18:   |  |  |  |  |  CPs.add(services in tStack + subpath of cp
              from vs)
19:   |  |  if(there aremore than 1 unvisited upperservices)
20:   |  |  |  give tmp a tag
21:   |  |  t = one of upperservices that has never been visited;
22:   |  |  tStack.push(t);
23:   |  |  if (all the upperservices has been visited)
24:   |  |  |  remove the tag of tmp
25:   |  |  tmp = t;
26:   |  }
27:   |  else {
28:   |  |  set cp = services in tStack;
29:   |  |  put cp in CPs
30:   |  |  set tmp = tStack.top;
31:   |  |  while(tmp hasn't a tag){
32:   |  |  |  tStack.pop(tmp);
33:   |  |  |  tmp = tStack.top;
34:   |  |  }
35:   |  }
36: }
37:return CPs;
```

The algorithm GCP is developed to generate critical paths, as shown in Table VII. We also use the example in Fig. 7 for a more detailed explanation. For the first step, the algorithm finds the service $w4$ that provides the required concept J and puts it into the critical path. Then, it searches the services that provide each input of $w4$. Among them, $(w3, w9)$ can provide E, $w2$ can provide F. However, only $(w3, w9)$ is one layer higher than $w4$. Therefore, $(w3, w9)$ is added to the critical path, as well as $w1$ is put into the critical path. Then, the critical path $\{w4, (w3, w9), w1\}$ is generated. The algorithm to generate noncritical paths (NGCP) is similar. The difference is that when it tries to find precursors of a service, it will search services at any layers rather than only at one upper layer.

Since algorithm BTSC-DFS consists of two algorithms (GCP and NGCP) and their complexity is identical, we only need to analyze the time complexity of GCP. In GCP, a stack is maintained to store critical paths. Assume that the filtered services

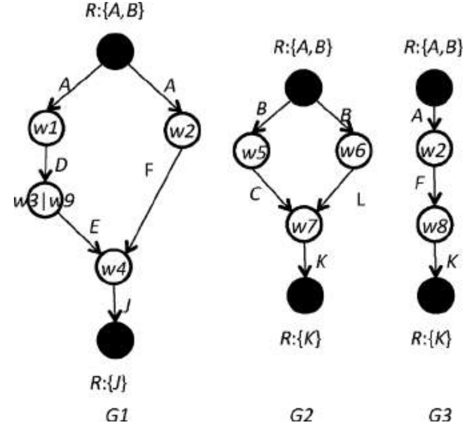


Fig. 8. Solution subgraphs for the given user request.

construct a DAG. Each node (service) is only linked to its *upperservices* (defined in line 15). Let n be the total number of filtered services, and m be the number of all edges in the DAG. For each service at the top of the stack, it will take $O(e_i)$ to decide which service should be put into the stack next, where e_i is the number of its *upperservices* (i.e., the number of its outer edges). Since each service is added to the stack at most once, the total time complexity is $O(n+m)$. As a result, not only the number of candidate services, but also the complexity of the service set which decides the dependencies between services, can influence the performance of BTSC-DFS. The completeness of BTSC-DFS which we prove later, guarantees that all possible solution subgraphs can be found.

The completeness of BTSC-DFS means that given a required concept, BTSC-DFS can find all the solution subgraphs as long as there exist solutions for the concept.

Proof of Completeness: In Algorithm 1, all enabled services are kept and layered, and services definitely useless for the solution are filtered out. Let us assume that there exists one solution subgraph for the required concept but BTSC-DFS cannot find it. Since it is impossible that this solution subgraph contains disabled services, all the services in the solution must be enabled ones.

Meanwhile, algorithm BTSC-DFS can check all the enabled services by lines 19–22 in Algorithm 2. Moreover, all possible paths are recorded and available for new solution subgraphs by lines 15–18. As a result, all edges between services are extracted, making any solution subgraph that contains any edge impossible to exist. Hence, the assumption does not hold.

4) Solution Generation: After all the solution subgraphs generated by compositing agents are returned to the central agent, the central agent starts to merge the solution subgraphs generated for each concept of the user's requested outputs and then generate the final top-k compositions.

In order to generate all possible solutions, the Cartesian product is utilized to compute the patterns of merging solution subgraphs. Given the user request as above, we can get the final solution subgraphs for the required outputs J and K, respectively (shown in Fig. 8). The generated solution subgraphs are: J-$\{G_1, \}$, K-$\{G_2, G_3\}$. Then, the merging graphs will be $\langle G_1, G_2\rangle$, $\langle G_1, G_3\rangle$ (shown in Fig. 9).

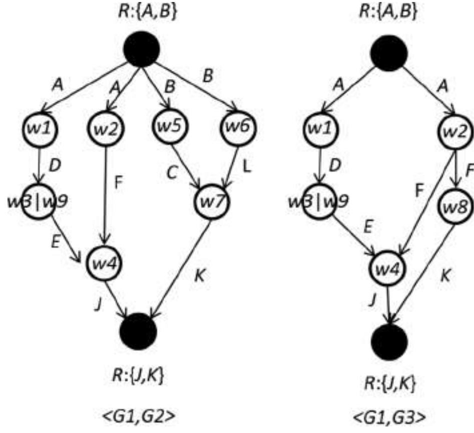


Fig. 9. Merging graphs for the given user request.

Since the nodes in a merging solution graph are not always single services as mentioned before, there may be several candidates for a node (e.g., the node of "$w3|w9$" in the Fig. 9). If we generate all the possible compositions for each merging solution graph, the number of compositions may result in prohibitive memory and computational costs. Therefore, it is better to select those services with better QoS to generate top-k compositions for each merging solution graph. Finally, it selects the final top-k QoS compositions among the top-k compositions of each merging solution graph.

We develop a method to generate top-k QoS compositions for merging solution subgraphs. The brief procedure is as follows.

a) Find all paths in the merged solution subgraphs.
b) For each path, build a bucket for each node. The candidate services are put into the buckets in order of the QoS behavior. For example, the service with fastest response time will be put first.
c) Take out all the services at the first place of each bucket. These services will then form a path with the best QoS.
d) For each bucket, calculate all the difference values between QoS of the remaining services and that of the service which has been removed from the bucket. Then, build a queue and include these services, ensuring that the ones with smaller difference values are added first.
e) Begin to use the services to substitute the services in the best path to form a new path. The process of substitution terminates when the number of generated paths reaches k or all possible paths have been generated. Then, at most k ranked QoS paths are generated. The rank of the paths is just the order of generation.
f) After every path in the merged solution subgraphs is processed, synthesize all the generated paths to make the final top-k QoS compositions.

For the given user request, we can get four final solutions: $s1 = \{w1, w2, w3, w4, w5, w6, w7\}$, $s2 = \{w1, w2, w4, w5, w6, w7, w9\}$, $s3 = \{w1, w2, w3, w4, w8\}$, and $s4 = \{w1, w2, w4, w8, w9\}$. According to the rule repository in Table IV, we can calculate the response time of each composition, which are 110, 190, 280, and 280 ms, respectively. Then, we can return the final results in order of the response time. Our proposed method can guarantee to obtain the composition result with the top-k QoS value.

Proof of Optimality: The composition results are derived from merging solution subgraphs of all the required concepts. Then, all possible patterns for the top-k query are generated. For each pattern, top-k composition results are obtained according to their QoS value. Then, the composition with the optimal QoS can be found after synthesizing all top-k results of each pattern. Assuming there exists a solution with a better QoS than those returned by our method. That is, there exists a rank-i ($1 \leq i \leq k$) result with a better QoS value than the corresponding rank-i result obtained by our method. That means, there exists either a better result that is selected from the composition patterns constructed by the generated solution subgraphs, or some patterns that are not found by our method. The first situation cannot happen since all candidate services for substitution are checked to get the top-k results; similarly if the second situation happens, it will violate the completeness property of BTSC-DFS (proved earlier in Section IV-D2). Therefore, the assumption does not hold.

Regarding to the large-scale candidate services, we make some optimization to reduce the number of possible patterns of merging solution subgraphs. If one solution graph can provide multiple concepts, this graph will not be merged with other solution subgraphs which also provide these concepts. For example, given three concepts A, B, and C, the solution subgraphs are A-$\{G_1, G_2\}$, B-$\{G_1, G_3\}$, and C-$\{G_4, G_5\}$. G_1 can provide both A and B, so it will be redundant to merge G_1 with either G_2 or G_3 which also provides A and B. Through this reduction, the final set of merging graphs is $\{\langle G_1, G_4\rangle, \langle G_1, G_5\rangle, \langle G_2, G_3, G_4\rangle, \langle G_2, G_3, G_5\rangle\}$, instead of

$$\{\langle G_1, G_4\rangle, \langle G_1, G_5\rangle, \langle G_1, G_3, G_4\rangle, \langle G_1, G_3, G_5\rangle, \langle G_2, G_1, G_4\rangle, \langle G_2, G_1, G_5\rangle, \langle G_2, G_3, G_4\rangle, \langle G_2, G_3, G_5\rangle\}.$$

This optimization can improve the efficiency at the cost of missing some correct composition results. We will experimentally evaluate the effect of the compromise on the scalability and accuracy.

V. Evaluation and Analysis

We conducted experiments to evaluate the behavior of our service composition mechanism based on the methods proposed in this paper. The experiments were conducted with large-scale datasets. We have evaluated the performance of the method with different affecting factors. In addition, the accuracy of the proposed method is assessed by observing the variance between the actual top-k results and calculated results. The datasets are acquired from WS-Challenge 2009 and CWSC2011. WS-Challenge is a famous worldwide automatic composition competition, which test sets have been have been used for assessment by many researches. CWSC2011 is the first competition on web service composition in China. The test sets of CWSC2011 were extended from WS-Challenge 2009 and with larger number of candidate services. Using the datasets from the two competitions for assessment of the proposed

TABLE VIII
DATASETS OF GENERATOR

	Number of services	Number of concepts	Solution Depth	K
Group-1	2,000~20,000	10,000	10	10
Group-2	10,000	2,000~20,000	10	10
Group-3	10,000	10,000	5~20	10
Group-4	10,000	10,000	10	1-100

method can be more convincing. For the experiments, the parallelization is implemented by multi-threads. The composing agents of the Composition Processor are generated dynamically according to the number of the tasks. Thus, all the tasks generated by the central agent can be executed simultaneously. The execution environment is: Intel Core2 P7370 2.0 GHz with 4 GB RAM, Windows 7, jdk1.6.0. In the following, we provide the details of our experiments.

A. Datasets and Setup

In order to evaluate different properties of the proposed method, we choose three types of datasets.

First, we use the tool *Generator*, provided by WS-Challenge 2009, to generate datasets to check the performance with different factors including the number of services, the number of concepts, the solution depth, and the number of return solutions. We generate four groups of datasets, shown in Table VIII. The solution depth is the length of the longest path in the solution. K is the number of the best solutions which are required to return to the user.

Second, we use the datasets in WS-Challenge 2009 to compare the performance of our method with the winning methods in the competition. These datasets are generated to test the efficiency and accuracy of different methods when returning the optimal QoS (response time and throughput, respectively) composition for users. For each challenge-set, the competition also provides the standard optimal QoS (e.g., smallest response time), the accuracy of different methods is evaluated according to the difference between the QoS of returned solution and the optimal value.

Finally, we use the datasets of CWSC2011 to evaluate the accuracy when solving top-k problems. CWSC2011 provides five large-scale datasets. The goal is to compose top-k QoS-aware service compositions with smallest response time according to requirements from the challenge client.

B. Performance Evaluation and Analysis

1) Scalability With Respect to the Number of Services: In this stage of our evaluation, we evaluate the effect of the proposed approach with varied numbers of candidate services. The datasets of Group-1 in Table VIII are used. We fix the concept number at 10,000 and the solution depth at 9. Then, we adjust the services from 2000 to 20,000 and try to make top-10 compositions from the datasets for each request. As Fig. 10 shows, the proposed approach can finish work at a millisecond level. In addition, the time cost of our approach linearly increases with the number of services. This can also validate the time complexity analysis in Section IV-D2.

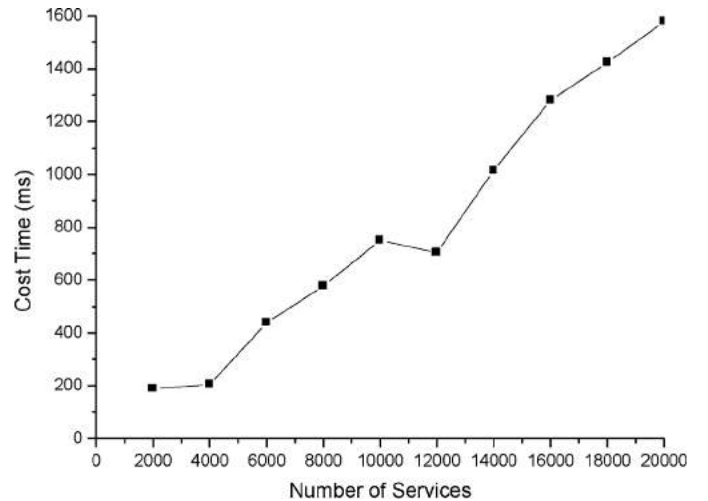


Fig. 10. Effect of the number of services.

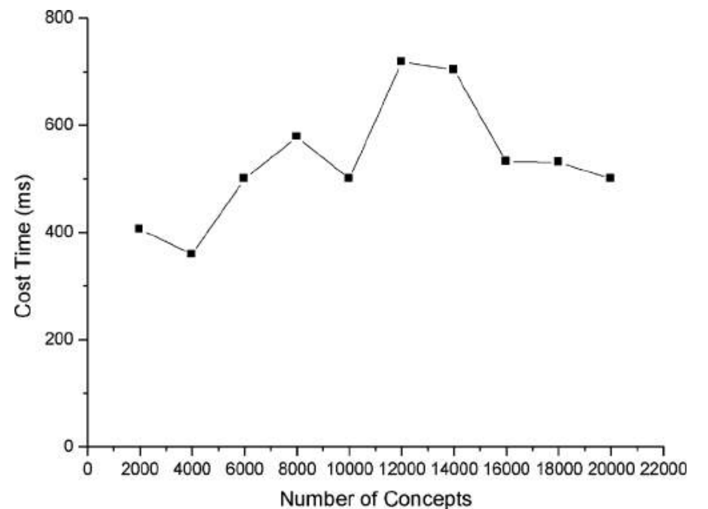


Fig. 11. Effect of the number of concepts.

2) Scalability With Respect to the Number of Concepts: To test the effect of the number of concepts on our approach, we use the datasets of Group-2 in Table VIII. For the datasets, we vary the concept number from 2000 to 20,000, and fix the service number and solution depth as 10,000 and 10, respectively. Again, top-10 results are computed. We plot the computation time of each trial in Fig. 11. It shows that the computation time fluctuates with the number of concepts. The fluctuation is due to the fact that *Generator* creates a data set for each number of concepts, but the complexity of the dataset generated each time cannot be guaranteed to be similar.

3) Scalability With Respect to the Solution Depth: We also test the effect of the solution depth on our approach. The datasets of Group-3 in Table VIII are used, in which the numbers of services and concepts are both 10,000, while the solution depth ranges from 5 to 20. Then, we use our approach to find top-10 compositions. The time taken for each solution depth value is shown in Fig. 12. It reveals that our approach performs similarly when the solution depth increases. That is because we utilize the

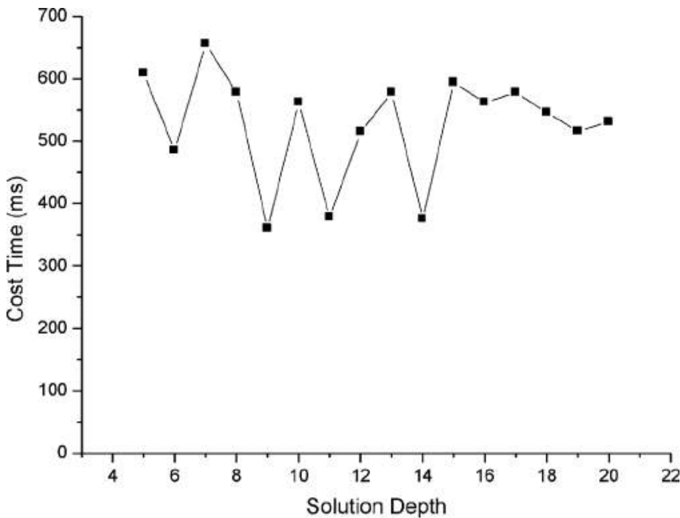


Fig. 12. Effect of solution depth.

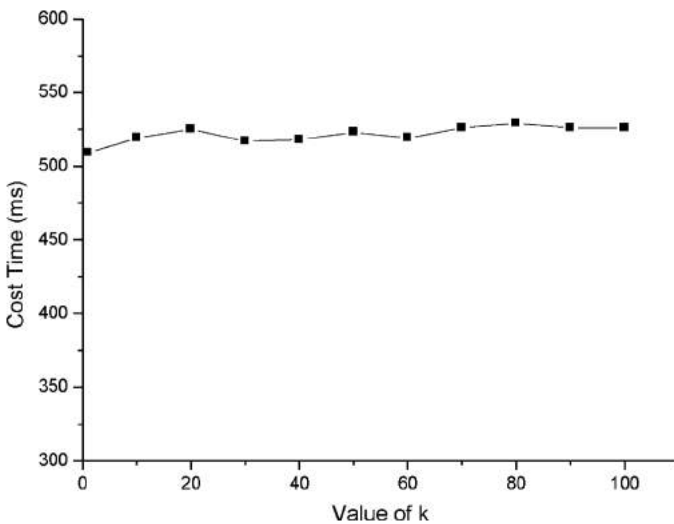


Fig. 13. Effect of k.

TABLE IX
DATASETS OF WS-CHALLENGE 2009

Dataset	Number of Services	Average Number of Inputs	Average Number of Outputs
Challengeset1	572	4.7	4.8
Challengeset2	4,129	4.7	4.7
Challengeset3	8,138	4.7	4.7
Challengeset4	8,301	4.7	4.8
Challengeset5	15,211	4.7	4.7

TABLE X
DATASETS OF CWSC2011

Dataset	Number of services	Number of concepts	Average Number of Inputs	Average Number of Outputs
Dataset1	8,000	4,000	4.7	4.8
Dataset2	10,000	6,000	4.8	4.8
Dataset3	4,000	40,000	4.7	4.8
Dataset4	15,000	100,000	4.8	4.8
Dataset5	40,000	20,000	4.7	4.8

TABLE XI
COMPARISON RESULTS

Dataset	Smallest Response Time		Computation Time (ms)		Solution Number	
	Winner	Ours	Winner	Ours	Winner	Ours
Challengeset1	500	500	47	109	1	157
Challengeset2	1690	1690	205	422	1	4284
Challengeset3	760	760	317	760	1	130
Challengeset4	1470	1470	626	812	1	932
Challengeset5	4070	4070	904	1730	1	10013

idea of depth-first-search and only one possible service is selected for backtracking. It will not increase searching space with the increasing depth. For some methods, the searching space will increase greatly when the solution depth increases since they mostly use the breadth-first-search. Similar to the previous experiment, the varying complexity of the composition structure in each dataset also results in the fluctuation of time.

4) Scalability With Respect to k*:* We also check whether the value of k affects the performance of our approach when making top-k compositions. The datasets of Group-4 in Table VIII are used, in which the value of K ranges from 1 to 100 with the service number 10,000, the concept number 1,000, and solution depth 10. The results in Fig. 13 show that our approach will not spend more composition time when returning more compositions.

C. Accuracy Evaluation and Analysis

1) Top-1 Verification: As proved in Section IV-D3, the proposed method in this paper can guarantee to return the actual top-k solution. This section gives the verification for the optimality of our method by comparing our approach with the winning approach in WS-Challenge 2009 [30] that has been proved to correctly find the top-1 composition with the smallest response time. For our approach, the time taken returning top-1 solution is almost the same with that of returning top-k solutions, so we use our approach to find the top-k solutions. For each dataset, we execute the algorithms for five rounds and get the average time taken. The comparison results are shown in Table XI. The comparison experiments are conducted with five different datasets (Challengeset 1–5 in Table IX). Three metrics, i.e., smallest response time of the optimal solution, computation time to return the solutions, and solution number returned by each approach are compared between the winning approach (Winner for short in Table XI) in WS-Challenge 2009 and ours.

From the comparison results, we can conclude that: 1) for the five challenge set, the optimal composition, i.e., the one with smallest response time computed by our approach is the same with the winning method, which validates the optimality of our approach and 2) when the candidate service number becomes

TABLE XII
COMPARING TOP-K RESULTS

Dataset	Precision		Expected Opportunity Cost		Time cost(ms)	
	Rival	Ours	Rival	Ours	Rival	Ours
Dataset1	80%	100%	20	0	359	640
Dataset2	80%	100%	15	0	453	719
Dataset3	100%	100%	0	0	50,719	719
Dataset4	50%	100%	30	0	108,769	2,031
Dataset5	40%	100%	34	0	3,125	3,734

TABLE XIII
COMPARISON WITH THE PROPOSED OPTIMIZATION.

Dataset	Precision		Expected Opportunity Cost		Time cost(ms)	
	Original	Optimized	Original	Optimized	Original	Optimized
Dataset1	100%	80%	0	15	640	294
Dataset2	100%	80%	0	8	719	359
Dataset3	100%	100%	0	0	719	283
Dataset4	1000%	100%	0	0	2,031	1,451
Dataset5	100%	70%	0	12	3,734	2,059

larger (in the last three challenge sets), our approach spends more time than the other methods. That is because ours returns all the solutions with the smallest response time while others only return one. Although our approach is a couple of times more costly, it yields alternatives while the others do not. Besides, the solutions are sharable among users with the same query.

*2) Top-*k *Accuracy Verification:* China Web Service Cup 2011 is held to solicit algorithm and software for top-k query of QoS-aware automatic service composition problem. Hence, we use the datasets of CWS2011 to verify the correctness of the top-k results retuned by our method. The details of the datasets are shown in Table X. For each dataset, we compute top-10 compositions with the smallest response time; in the meanwhile, the competition also provides the actual top-10 compositions with the smallest response time. The response time of each solution is the aggregation of component services' response time according to the rules in Table III. The results are shown in Table XII. Then, we compare the return results with the actual results provided by the competition to evaluate the accuracy of our proposal. The accuracy was assessed according to the number of hits (computed results that match the actual top-k compositions). The precision is calculated to assess the accuracy

$$\text{Precision} : P = \frac{\text{number of hits}}{k}.$$

We also adopt a second measure of selection quality from [62], the so-called expected opportunity cost E:

$$E = \sum_{j=1}^{k} (J_j - j)$$

where J_j is the actual rank of the rank-j result. This measure penalizes particularly bad choices more than mildly bad choices. For example, when $k = 3$, a result of $\{\text{rank-1}, \text{rank-2}, \text{rank-4}\}$ is better than $\{\text{rank-1}, \text{rank-2}, \text{rank-5}\}$, and both are better than $\{\text{rank-1}, \text{rank-3}, \text{rank-5}\}$. Note that E returns a minimum value of 0 when all the top-k results are correctly returned.

Table XII shows the experimental results compared with the method [64] which is also designed for top-k service composition problem. From the results, we can conclude that our approach can achieve 100% accuracy as proved in Section Section IV-D3. Besides, the results also show that our method outperforms the method in [64] on time cost.

The aim of our method is to find top-k services compositions from large-scale service sets. Therefore, accuracy and scalability of the method are both important. We have made some compromises to achieve high scalability: as seen in Section IV-D3, some redundant composition patterns are removed. Thus, we can save some computation time for generating solutions at the sacrifice of missing some correct results. In this part, we are going to evaluate the effect of the compromise on accuracy and scalability. Table XIII shows the experimental results compared between our original method and our optimized method. From the results, it is clear that the optimized approach does not achieve 100% accuracy. This arises from the fact that the optimized procedures involve reducing the number of possible patterns of merging solution subgraphs as introduced in Section IV-D3. However, the optimized method can save nearly half time cost for each dataset. In general, the results indicate that the optimization in Section IV-D3 can improve the efficiency without much loss of accuracy when processing large-scale service sets.

VI. RELATED WORK

Since our proposal aims at tackling the problem of yielding top-k QoS composition from a large number of services, we review related work from the following three aspects: 1) QoS-aware service composition; 2) top-k service composition; and 3) scalable composition frameworks.

A. QoS-Aware Approaches

Automatic Service Composition (ASC) technology has been proposed to automatically discover, select, and compose multiple individual services to satisfy a user's query. There are two categories of approaches: AI planning [33]–[40] and graph-based search [41]–[47]. However, these early researches on automatic service composition do not consider the nonfunctional attributions of services. They usually return the service composition results with fewer services or less depth.

Nowadays, many approaches have been proposed that combine QoS awareness with service composition [20], [22], [23], [30], [32]. The approach QSynth described in [48] achieved first place in the WS-challenge 2009. It provides an efficient algorithm with a scalable runtime performance and QoS awareness. The approach proved to be very efficient and scalable. It realizes preprocessing when effective data structures are built. The data structures are used during the user querying phase to quickly

compose a required composition. But the approach can only return the optimal result to the user, while ours can provide top-k QoS results to the user.

Florian Wagner *et al.* [49] propose a method that combines the AI planning and services selection approaches by leveraging common characteristics of service registries. They utilize a data structure that arranges functionally similar services in clusters and computes the QoS of each cluster. Then, the planning tool composes workflows consisting of these clusters, taking the QoS of the clusters into account. Thus, the utility in general and the reliability of the composed workflows are significantly increased.

Shen *et al.* [50] present an approach to optimize the QoS-aware services composition for multiple concurrent processes for multiple selfish users. They use a multi-issue negotiation protocol among agents based on an auction-based multi-issue negotiation protocol to reallocate the service resources and device resources before and during the execution of concurrent composite services. The approach shows advantages in a dynamic resource-constrained environment. However, it is not efficient when the number of services grows exponentially.

In general, most of the existing researches on QoS-aware automatic service composition focus on returning the optimal service composition result. Different from these researches, our work returns not only the optimal result but also top-k QoS composition results. As we mentioned before, providing top-k composition results can bring more benefits and has a higher level of complexity than returning optimal result.

B. *Top*-k *Service Composition*

Although the problem of top-k service composition has not been studied thoroughly, there exist several studies. Benouaret *et al.* addressed the problem of top-k retrieval of data web service compositions to answer fuzzy preference queries under different matching methods [51], [52]. They present a suitable ranking criterion based on a fuzzification of Pareto dominance and developed a suitable algorithm for computing the top-k data web service compositions. However, the ranking criteria of the top-k results are based on the functional relevance between the composition and the user's query; QoS of services is not considered. In our work, the top-k service compositions are ranked based on the overall QoS.

The work in [53] combines QoS computations into the process of service composition and tries to find the top-k optimal results. The proposed approach is based on a succinct binary tree data structure and QoS is taken into account with a heuristic strategy. However, it only calculates the summation of the QoS of component services as the overall QoS of the composition, but does not consider different QoS properties and composition patterns for the overall QoS of compositions, which makes its QoS computation rules for service composition not always accurate. Since we adopt the computation rules in [23], our proposal can calculate the overall values of different QoS properties with various composition patterns. Zheng *et al.* introduced a web service search engine called WSExpress [54], in which the functional value and the nonfunctional values are aggregated to compute the rank score of a web service. Then, top-k results are ranked for a search query. However, this work is mainly towards atomic services rather than service compositions. Ranking composite services can be seen as an extension to ranking single services, by dealing with more complex structures.

Jiang *et al.* also extend their work [48] to support the query of top-k service composition results [55]. They develop a service composition system called QSynth-Top-K to provide top-k QoS service compositions. A progressive and incremental Key-Path-Loose algorithm is proposed and implemented. But QSynth-Top-K is based on centralized computing, which may need powerful and reliable central servers and a lot of bandwidth for computing, data storage, and communication. Our approach can be implemented in a decentralized way that can reduce the load of the central node and improve the availability of the whole system.

To summarize, the biggest difference between the few works on top-k service composition and our proposal is that we develop a parallel/distributed framework, which divides the top-k composition task into several subtasks that can be executed in parallel. The benefit is that, when the service set is quite large, this framework cannot only improve the efficiency but also reduce the load of central node.

C. *Scalable Composition Frameworks*

Traditional service composition frameworks are mainly based on centralized computation which is not scalable. As the number of web services on the Internet is increasing, scalability has been identified as one of the most important challenges in the area of web service compositions. In particular, the top-k QoS service composition may take much computation work when the candidate service set is quite large. Then, how to improve the scalability is a key issue for top-k QoS service composition.

Currently, the emergence of multicore computing, parallel computing and distributed computing have promised massive performance gains for large-scale computations, which also spark new interest in the complex domain of problem partitioning [57]. These emerging techniques also bring new ideas for web service composition. Some trials to parallelization or distribution for service composition have already been designed [58]–[60].

Pathak *et al.* modeled a choreographer for realizing composite services [58]. The choreographer first selects necessary components to meet a complex composition goal state, and then executes the planning for each individual component in parallel. However, this approach restricts parallelization to portions of the composition graph that can be executed in parallel. The actual synchronization required for such parallelization over the entire composition graph will severely decrease the performance. Since our framework divides the task into multiple subtasks which have nothing to do with each other, there is no need to conduct synchronization process.

Bartalos and Bieliková presented a framework for semantic web service composition which exploits the possibilities of multiprocessor platforms [59]. The composition approach is a combination of three parallel processes operating over the same data

structure: 1) Finding usable services. 2) Finding unusable services,. 3) Backward chaining. Similar to [58], this framework also needs much work on synchronization.

Falou *et al.* proposed a multiagent-based distributed framework for automatic service composition [60]. This framework consists of a core agent and several normal agents. Each normal agent corresponds to a set of services. When the core agent receives a query, it broadcasts the query to all the normal agents. Then, each normal agent make local optimize planning form the initial state to the target state with its services. Finally, the core agent summarizes all local plans and generates the global optimal plans. However, the binding of each normal agent to a specific set of services restricts the flexibility of the framework. This framework may miss a lot of possible results which consists of services from different agents since the target of this work is to find one possible solution not the best solution.

Compared to the above parallel or distributed frameworks for service composition, our framework adopts the idea of MapReduce and maps the top-k composition problem into several subtasks that can be executed in parallel. Furthermore, our framework adopts the principle of MapReduce but not the implementation, so it can be implemented by various parallel techniques such as multithreads, multiprocessers, Hadoop, etc.

VII. Conclusion and Future Work

This paper introduces a parallel framework for top-k QoS service composition from large-scale repositories. In order to reduce the searching space we develop a backtracking composition algorithm based on depth-first-search. To test its performance the approach is applied to a simulated web service environment. The experimental results show that our approach can make top-k QoS service compositions from large-scale repositories accurately and efficiently. In the future, we intend to investigate how to handle multiple QoS properties and take more consideration of users' preferences to make personalized compositions. Also, composition result caching and reusing is interesting to explore.

References

[1] M. P. Papazoglou *et al.*, "Service-oriented computing: A research roadmap," *Int. J. Coop. Info. Syst.*, vol. 17, no. 2, pp. 223–255, Jun. 2008.

[2] W. Tan, Y. Fan, and M. C. Zhou, "A Petri net-based method for compatibility analysis and composition of web services in business process execution language," *IEEE Trans. Autom. Sci. Eng.*, vol. 6, no. 1, pp. 94–106, Jan. 2009.

[3] L. Huang *et al.*, "A trust evaluation mechanism for collaboration of data-intensive services in cloud," *Appl. Math. Inf. Sci.*, vol. 7, no. 1L, pp. 121–129, Feb. 2013.

[4] W. Jiang *et al.*, "Continuous query for QoS-aware automatic service composition," in *Proc. ICWS*, Honolulu, HI, USA, 2012, pp. 50–57.

[5] J. Peer, "Web service composition as AI planning–A survey," Univ. St. Gallen, St. Gallen, Switzerland, Tech. Rep., 2005.

[6] H. Kil and W. Nam, "Semantic web service composition via model checking techniques," *Int. J. Web Grid Serv.*, vol. 9, no. 4, pp. 339–350, Nov. 2013.

[7] M. Phan and F. Hattori, "Automatic web service composition using congolog," in *Proc. ICDCS Workshop*, Lisboa, Portugal, 2006, p. 17.

[8] S. C. Oh *et al.*, "WSPR*: Web-service planner augmented with A* Algorithm," in *Proc. CEC*, Warren, MI, 2009, pp. 515–518.

[9] P. Rodriguez-Mier *et al.*, "An optimal and complete algorithm for automatic web service composition," *Int. J. Web. Serv. Res.*, vol. 9, no. 2, pp. 1–20, Apr. 2012.

[10] B. Wu *et al.*, "AWSP: An automatic web service planner based on heuristic state space search," in *Proc. ICWS*, Washington, DC, USA, 2011, pp. 403–410.

[11] W. Tan *et al.*, "Data-driven service composition in enterprise SOA solutions: A Petri Net approach," *IEEE Trans. Autom. Sci. Eng.*, vol. 7, no. 3, pp. 686–694, Jul. 2010.

[12] I. Paik and D. Maruyama, "Automatic web services composition using combining HTN and CSP," in *Proc. CIT*, Fukushima, Japan, 2007, pp. 206–211.

[13] C. Kun, J. Xu, and S. Reiff-Marganiec, "Markov-HTN planning approach to enhance flexibility of automatic web services composition," in *Proc. ICWS*, Los Angeles, CA, USA, 2009, pp. 9–16.

[14] J. Rao and P. Küngas, "Application of linear logic to web service composition," in *Proc. ICWS*, Las Vegas, NV, USA, 2003, pp. 3–9.

[15] J. Rao, P. Kungas, and M. Matskin, "Logic-based web services composition: From service description to process model," in *Proc. ICWS*, San Diego, CA, USA, 2004, pp. 446–453.

[16] P. C. Xiong, Y. Fan, and M. C. Zhou, "Web service configuration under multiple quality-of-service attributes," *IEEE Trans. Autom. Sci. Eng.*, vol. 6, no. 2, pp. 311–321, Apr. 2009.

[17] S. G. Deng *et al.*, "Parallel optimization for data-intensive service composition," *J. Internet Technol.*, vol. 14, no. 5, pp. 817–824, Sep. 2013.

[18] W. Mayer, R. Thiagaraja, and M. Stumptner, "Service composition as generative constraint satisfaction," in *Proc. ICWS*, Los Angeles, CA, USA, 2009, pp. 888–895.

[19] P. Rodriguez-Mier, M. Mucientes, and M. Lama, "Automatic web service composition with a heuristic-based search algorithm," in *Proc. ICWS*, Washington, DC, USA, 2011, pp. 81–88.

[20] F. Wagner, F. Ishikawa, and S. Honiden, "QoS-aware automatic service composition by applying functional clustering," in *Proc. ICWS*, Washington, DC, USA, 2011, pp. 89–96.

[21] R. F. Korte, "Biases in decision making and implications for human resource development," *Adv. Developing Human Resources*, vol. 5, no. 4, pp. 440–457, Nov. 2003.

[22] M. Alrifai, T. Risse, and W. Nejdl, "A hybrid approach for efficient web service composition with end-to-end QoS constraints," *ACM T Web*, vol. 6, no. 2, pp. 7–7, May 2012.

[23] L. Zeng *et al.*, "QoS-aware middleware for web services composition," *IEEE T Software Eng.*, vol. 30, no. 5, pp. 311–327, May 2004.

[24] Web Services Description Language 1.1 W3C Note 15 2001.

[25] OWL Web Ontology Language Overview, W3C recommendation 10.2004-03 2004.

[26] H. Ludwig *et al., Web Service Level Agreement (WSLA) Language Specification*. Armonk, NY, USA: IBM Corporation, 2003, pp. 815–824.

[27] M. Záková *et al.*, "Automating knowledge discovery workflow composition through ontology-based planning," *IEEE Trans. Autom. Sci. Eng.*, vol. 8, no. 2, pp. 253–264, Apr. 2011.

[28] P. Hennig and W. T. Balke, "Highly scalable web service composition using binary tree-based parallelization," in *Proc. ICWS*, Miami, FL, 2010, pp. 123–130.

[29] M. S. Kumar and P. Varalakshmi, "A novel approach for dynamic web service composition through network analysis with backtracking," in *Advances in Computing and Information Technology*. Berlin, Germany: Springer-Verlag, 2013, pp. 357–365.

[30] Z. Huang *et al.*, "Effective pruning algorithm for QoS-aware service composition," in *Proc. CEC*, Warren, MI, USA, 2009, pp. 519–522.

[31] S. G. Deng *et al.*, "Trust-based personalized service recommendation: A network perspective," *J. Comput. Sci. Tech-CH.*, vol. 29, no. 1, pp. 69–80, Jan. 2014.

[32] Y. Yan *et al.*, "A QoS-driven approach for semantic service composition," in *Proc. CEC*, Warren, MI, USA, 2009, pp. 523–526.

[33] M. Weiss, B. Esfandiari, and Y. Luo, "Towards a classification of web service feature interactions," *Comput. Netw.*, vol. 51, no. 2, pp. 359–381, Feb. 2007.

[34] F. Lécué *et al.*, "SOA4All: An innovative integrated approach to services composition," in *Proc. ICWS*, Miami, FL, USA, 2010, pp. 58–67.

[35] M. Kuzu and N. K. Cicekli, "Dynamic planning approach to automated web service composition," *Appl. Intell.*, vol. 36, no. 1, pp. 1–28, Jan. 2012.

[36] D. Wu *et al.*, "Automating DAML-S web services composition using SHOP2," in *Proc. ISWC*, Washington, DC, USA, 2003, pp. 195–210.

[37] M. Klusch, A. Gerber, and M. Schmidt, "Semantic web service composition planning with OWLS-Xplan," in *Proc. AAAI Fall Symp. Semantic Web and Agents*, Arlington, VA, USA, 2005, pp. 55–62.

[38] P. Doshi *et al.*, "Dynamic workflow composition using Markov Decision processes," in *Proc. ICWS*, San Diego, CA, USA, 2004, pp. 576–582.

[39] H. Wang *et al.*, "Adaptive service composition based on reinforcement learning," in *Proc. ICSOC*, San Francisco, CA, USA, 2010, pp. 92–107.

[40] S. Y. Hwang *et al.*, "Dynamic web service selection for reliable web service composition," *IEEE Tran. Serv. Comput.*, vol. 1, no. 2, pp. 104–116, Apr. 2008.

[41] A. Goldman and Y. Ngoko, "On graph reduction for QoS prediction of very large web service compositions," in *Proc. SCC*, Honolulu, HI, USA, 2012, pp. 258–265.

[42] A. Zhou, S. Huang, and X. Wang, "Bits: A binary tree based web service composition system," *Int. J. Web. Serv. Res.*, vol. 4, no. 1, pp. 40–58, Jan. 2007.

[43] S. Y. Lin *et al.*, "A cost-effective planning graph approach for large-scale web service composition," *Math. Probl. Eng.*, p. 21, 2012, Article ID. 783476.

[44] Z. Hua, F. Yan, and G. Hui, "A web service composition algorithm based on dependency graph," in *Proc. GCN*, Chongqing, China, 2012, pp. 1511–1518.

[45] S. G. Deng *et al.*, "A method of semantic web service discover based on bipartite graph matching," *Chin. J. Comput.*, vol. 31, no. 8, pp. 1364–1375, Nov. 2008.

[46] S. G. Deng *et al.*, "Automatic web service composition based on backward tree," *J. Software*, vol. 18, no. 8, pp. 1896–1910, May 2007.

[47] S. C. Oh, D. Lee, and S. R. T. Kumara, "Web service planner (WSPR): An effective and scalable web service composition algorithm," *Int. J. Web. Serv. Res.*, vol. 4, no. 1, pp. 1–22, Jan. 2007.

[48] W. Jiang *et al.*, "Qsynth: A tool for QoS-aware automatic service composition," in *Proc. ICWS*, Miami, FL, USA, 2010, pp. 42–49.

[49] F. Wagner *et al.*, "Multi-objective service composition with time-and input-dependent QoS," in *Proc. ICWS*, Miami, FL, USA, 2010, pp. 234–241.

[50] Y. En *et al.*, "Optimizing QoS-aware services composition for concurrent processes in dynamic resource-constrained environments," in *Proc. ICWS*, Miami, FL, USA, 2010, pp. 250–258.

[51] K. Benouaret *et al.*, "Top-$\mathbb{K}$ web service compositions using fuzzy dominance relationship," in *Proc. SCC*, Washington, DC, USA, 2011, pp. 144–151.

[52] K. Benouaret, D. Benslimane, and A. Hadjali, "Top-$\mathbb{K}$ service compositions: A fuzzy set-based approach," in *Proc. SAC*, New York, NY, USA, 2011, pp. 1033–1038.

[53] X. L. Wang, S. Huang, and A. Y. Zhou, "QoS-aware composite services retrieval," *J. Comput. Sci. Tech-CH*, vol. 21, no. 4, pp. 547–558, Jul. 2006.

[54] Y. Zhang, Z. Zheng, and M. R. Lyu, "Wsexpress: A QoS-aware search engine for web services," in *Proc. ICWS*, Miami, FL, USA, 2010, pp. 91–98.

[55] W. Jiang, S. Hu, and Z. Liu, "Top K query in QoS-aware automatic service composition," *IEEE Tran. Serv. Comput.*, to be published.

[56] K. C. C. Chang and S. Hwang, "Minimal probing: Supporting expensive predicates for top-$\mathbb{K}$ queries," in *Proc. SIGMOD*, Madison, WI, USA, 2002, pp. 346–357.

[57] J. J. Joseph*, An Introduction to Parallel Algorithms*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1992.

[58] J. Pathak *et al.*, "Parallel web service composition in moscoe: A choreography-based approach," in *Proc. ECOWS*, Zürich, Switzerland, 2006, pp. 3–12.

[59] P. Bartalos and M. Bieliková, "Semantic web service composition framework based on parallel processing," in *Proc. CEC*, Warren, MI, USA, 2009, pp. 495–498.

[60] M. El Falou *et al.*, "A distributed planning approach for web services composition," in *Proc. ICWS*, Miami, FL, USA, 2010, pp. 337–344.

[61] F. Tao *et al.*, "FC-PACO-RM: A parallel method for service composition optimal-selection in cloud manufacturing system," *IEEE Trans. Ind. Inform.*, vol. 9, no. 4, pp. 2023–2033, Nov. 2013.

[62] M. Theobald, G. Weikum, and R. Schenkel, "Top-$\mathbb{K}$ query evaluation with probabilistic guarantees," in *Proc. VLDB*, Toronto, ON, Canada, 2004, pp. 648–659.

[63] C. H. Chen *et al.*, "Efficient simulation budget allocation for selecting an optimal subset," *Informs J. Comput.*, vol. 20, no. 4, pp. 579–595, May 2008.

[64] S. Deng *et al.*, "Efficient planning for top-$\mathbb{K}$ web service composition," *Knowl. Inf. Syst.*, vol. 36, no. 3, pp. 579–605, Sep. 2013.

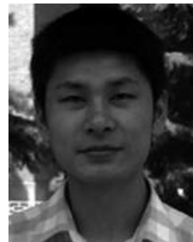
Shuiguang Deng received the B.S. and Ph.D. degrees in computer science from Zhejiang University, Hangzhou, China, in 2002 and 2007, respectively.

Presently, he is an Associate Professor with the College of Computer Science, Zhejiang University. His research interests include service computing, business process management, and data management. Up to now, he has published more than 30 papers in peer-refereed journals and international conference proceedings as the first author or the corresponding author. He holds a number of patents for his many innovations.

Dr. Deng is a Member of the Association for Computing Machinery (ACM). He is the recipient of Microsoft Fellowship Award 2005.

Longtao Huang received the B.S. degree in software engineering from Zhejiang University, Hangzhou, China, in 2010. Currently, he is working towards the Ph.D. degree in computer science and technology at Zhejiang University.

His research interests include service computing and cloud computing.



Wei Tan (M'12–SM'13) received the B.S. degree and the Ph.D. degree from the Department of Automation, Tsinghua University, Beijing, China, in 2002 and 2008, respectively.

He is currently a Research Staff Member at IBM T. J. Watson Research Center, Yorktown Height, NY, USA. His research interests include big data, cloud computing, service-oriented architecture, business and scientific workflows, and Petri nets. He

Dr. Tan is a Member of the Association for Computing Machinery (ACM).

Zhaohui Wu (M'04) received the Diploma degree in computer science from Zhejiang University, Hangzhou, China, in 1988.

He is a member of Sino-Germany jointly trained Ph.D. program from 1991 to 1993. Currently, he is a Professor with the College of Computer Science, Zhejiang University. His research interests cover the range of distributed artificial intelligence, grid computing, biometrics, embedded , etc.