Improving QoS Delivered by WS-BPEL Scenario Adaptation through Service Execution Parallelization

Dionisis Margaris
Department of Informatics and Telecommunications
University of Athens
Athens, Greece
+302107275220
margaris@di.uoa.gr

Costas Vassilakis
Department of Informatics and Telecommunications
University of the Peloponnese
Tripoli, Greece
+302710372203
costas@uop.gr

Panagiotis Georgiadis
Department of Informatics and Telecommunications
University of Athens
Athens, Greece
+302107275219
p.georgiadis@di.uoa.gr

ABSTRACT
Abstract—WS-BPEL scenario execution adaptation has been proposed by researchers as a response to the need of users to tailor the WS-BPEL scenario execution to their individual preferences; these preferences are typically expressed through Quality of Service (QoS) policies, which the adaptation mechanism considers in order to select the services that will ultimately be invoked to realize the desired business process. In this paper, we study the potential to parallelize the execution of the WS-BPEL scenario in order to minimize its response time and/or achieving higher scores in the other qualitative dimensions, such as cost, reliability, etc., at the same time. We also describe, develop and validate a parallelization algorithm for realizing the proposed enhancements.

CCS Concepts
• **Information systems →World Wide Web →Web Services**

Keywords
WS-BPEL, adaptation, parallelization, quality of service, performance evaluation

1. INTRODUCTION
Web Services are the dominant standard for building distributed applications. Service providers make available functionalities that can be invoked through well-defined XML-based protocols, and consumer applications may locate and invoke them using standard representations and interfaces, regardless of internal implementation or infrastructure details on the side of the service provider. WS-BPEL (Web Services Business Process Execution Language) allows for building high-level business processes through orchestrating individual services. WS-BPEL programs (*scripts* or *scenarios*) are made available for execution through deployment on WS-BPEL execution engines. WS-BPEL has been designed to model business processes that are fairly stable, therefore the WS-BPEL programmer specifies at the exact services to be invoked for the realization of the business process at scenario development time. This arrangement however is not adequate in the current web: the functionality offered by services invoked within the scenario (e.g. booking an air flight ticket) are typically offered by numerous providers (different airlines and travel agencies), and different providers may offer this functionality with diverse quality of service (QoS) parameters. Considering this, it would be desirable to enable consumers to adapt the WS-BPEL scenario execution to suit their QoS requirements; according to [2], governance for compliance with QoS and policy requirements is an open issue for the SOA architecture.

To foster this requirement, a number of approaches have been proposed following two main strategies [3]: (i) *horizontal adaptation*, where the composition logic is not modified and the main adaptation task is to select and invoke, on a per-service basis, the service implementation that delivers the requested functionality *and* best matches the client's QoS requirements; the selected services are substituted for either *abstract tasks* (e.g. [3]) or *concrete service invocations* (e.g. [5]) and (ii) *vertical adaptation*, where the composition logic may be changed.

In this paper we propose a transformation-based approach to exploit the potential parallelism in service invocations, so as to minimize the overall WS-BPEL scenario execution time. Exploitation of parallelism can serve as an aid to the adaptation process by broadening the set of alternatives available to the adaptation mechanism: since parallelism reduces the overall execution time, in the parallelized scenario it is possible to choose operations with higher response times but better values in other QoS dimensions (e.g. cost), with the composition respecting the overall WS-BPEL scenario execution time limits, but scoring higher in the other dimensions (e.g. lower costs). We present an algorithm based on data flow analysis combined with side-effect analysis, to identify cases where services for sequential execution in the WS-BPEL scenario can run in parallel; the WS-BPEL scenario is then restructured to utilize the available parallelism. Since the WS-BPEL scenario is restructured, the presented approach does not strictly follow the horizontal adaptation paradigm [3], however the changes to the composition logic are limited and performed in a fashion that enables the exploitation of exception handlers provided by the scenario designer, which may have been elaborately crafted to correspond to the particularities of the business process modeled by the WS-BPEL scenario.

SAC 2016, April 4-8, 2016, Pisa, Italy.

ACM 978-1-4503-3739-7/16/04…$15.00
DOI: http://dx.doi.org/10.1145/2851613.2851805

Finally, we present a middleware-based architecture which enables the realization of the tasks presented above and validate the applicability of the proposed approach through experiments, concerning both the overhead introduced by the adaptation mechanism and the QoS of the formulated adaptations.

The contribution of this work is: (a) the introduction of an algorithm to exploit the parallel execution potentials and (b) the design, development and evaluation of an architecture to perform the QoS-based adaptation, incorporating the parallelization algorithm.

The rest of this paper is structured as follows: Section 2 presents fundamental concepts regarding the QoS. Section 3 presents the parallelization algorithm, and Section 4 outlines the overall architecture used for service execution parallelization and WS-BPEL scenario adaptation. Section 5 evaluates the proposed approach, both in terms of performance and adaptation quality, while Section 6 overviews related work and Section 7 concludes the paper and outlines future work.

2. QoS CONCEPTS

QoS is generally defined in terms of attributes corresponding to non-functional aspects of services [13], with typical attributes being response time, availability, price, reputation, security etc [14]. In this paper, we will limit our discussion to attributes *response time* (rt), *availability* (av) and *cost* (c), for brevity reasons, adopting their definitions from [11]. This limitation does not lead to loss of generality since the extension of the proposed algorithm to include more QoS attributes is straightforward. Taking the above into account, each functionality implementation (realized as a *service operation*) considered in the adaptation process has a known QoS vector $QoS_S=(rt_s, av_s, c_s)$ which is recorded in an appropriate repository (e.g. METEOR-S [19] or WSMO [28]). The same repository should also provide information regarding which operations are equivalent. Within a WS-BPEL scenario, individual functionalities are composed into sequential or parallel flows to implement the business process. Considering the QoS parameters of the individual functionalities invoked and the type of their compositions (sequential or parallel), it is possible to compute the QoS value of the overall composition using the formulas shown in Table I [8]. As we can see from Table 1, the response time (*rt*) of a sequential composition is equal to the sum of its components' *rt*, while the *rt* of a parallel composition is equal to the maximum value. This difference is important in the context of this work, since the exploitation of available parallelization can lead to reduction of the overall *rt*.

Table 1: QoS of composite services

	QoS attribute		
	response time	cost	availability
Sequential composition	$\sum_{i=1}^{n} rt_i$	$\sum_{i=1}^{n} c_i$	$\prod_{i=1}^{n} av_i$
Parallel composition	$\max_i rt_i$	$\sum_{i=1}^{n} c_i$	$\prod_{i=1}^{n} av_i$

In the context of adaptation, selection of the concrete service that will realize some functionality is typically driven by parameters specifying the upper and lower bounds for each QoS attribute. QoS bounds may either be defined as *global constraints* (i.e. express the desired values for the whole WS-BPEL scenario) or as *local constraints* (each such constraint expresses the desired values for a particular service invocation) [9]. When adaptation problems need to address global constraints performance is poor [21], therefore either local constraints are directly used (e.g. [8]) or methods for mapping global constraints to local constraints are employed (e,g, [21]). Complementary to QoS bounds, a *weight* is assigned to each QoS attribute, indicating how important each QoS attribute is considered in the context of the particular business process. Weights always apply to the whole composition, rather than to individual services, since they reflect the perceived importance of each QoS attribute dimension on the process as a whole [18]. In the proposed algorithm, QoS specifications for a service within the WS-BPEL scenario may include an upper bound and a lower bound for each QoS attribute, i.e. for service s_j included in a WS-BPEL scenario, the designer formulates two vectors $MIN_j(min_{rt,j}, min_{av,j}, min_{c,j})$ and $MAX_j(max_{rt,j}, max_{av,j}, max_{c,j})$. Additionally the designer formulates a weight vector $W = (rt_w, av_w, c_w)$, indicating how important each QoS attribute is considered by the designer in the context of the particular operation invocation. The values of the QoS attributes are assumed to be expressed in a "larger values are better" setup, e.g. a service having *cost = 6* means that that it is *cheaper* than a service having *cost = 4* [8],[10].

3. THE PARALLELIZATION ALGORITHM

Although WS-BPEL provides the mechanisms to designate parallel execution of operation invocations, WS-BPEL scenario designers may not fully exploit the potential for arranging operations into parallel execution structures, similarly to the case that programmers typically write programs in a single-threaded fashion [22][23]. This is due to the fact that parallelization is a laborious task and WS-BPEL designers mostly focus on accurately realizing the business logic behind WS-BPEL scenarios, rather than pursue execution time optimizations. To this end, a tool that would detect and exploit the parallelization opportunities available in WS-BPEL scenarios, would deliver the benefits of parallel execution without placing the parallelization burden on WS-BPEL scenario designers.

In our approach, WS-BPEL scenario parallelization is undertaken by a *preprocessor*, which preprocesses the scenario before it is deployed to the WS-BPEL execution engine. Parallelization is be driven by *data flow and dependence analysis* used in instruction-level parallelism [24], supplemented with techniques addressing the particularities of WS-BPEL execution (exceptions, compensations and side-effects) and aspects related to the QoS of the invoked services. The criteria for identifying invocations that can be executed in parallel are detailed in the following paragraphs. In these paragraphs, we will consider that operation invocation op_1 appears in the WS-BPEL scenario before invocation op_2. At the current development stage, only invocations belonging either to (i) the same *sequence* structured activity or (ii) nested *sequence* and *flow* activities, with no intervening conditional (*if)* or loop structured activities (*while, repeatUntil, foreach*) [1] are considered. The development of the necessary techniques for control dependence checking and loop unrolling [26] to foster parallelization among invocations nested in loops and conditionals are part of our future work. The main challenge in this task is to appropriately handle dynamic XPath expressions typically used for accessing array elements used in WS-BPEL loops [1]), while the parallelization potential of control structures must also be coupled with the adaptation of these structures, employing techniques such as those described in [34].

WS-BPEL provides two main control flow structures for composing operation invocations into business processes, namely *sequence* and *flow*, which arrange for sequential and parallel execution of the invocations they contains, respectively [1].

1. Two operation invocations op_1 and op_2 can be scheduled to run in parallel, if they have been designated to be executed in parallel in the original WS-BPEL scenario (as crafted by the WS-BPEL designer).

2. Operations op_1 and op_2 are analyzed for existence of data dependence between them. Four types of dependencies may exist between operation invocations [25]:
   a. *True (or flow) dependence:* op_2 uses a parameter that is either directly returned by op_1 as its result, or computed using the result of op_1. In this case, clearly op_2 cannot be executed before op_1 concludes its execution, since the value of some input parameter of op_2 is yet unknown.
   b. *Anti-dependence:* op_2 modifies a variable V by assigning to it its result value, and the same variable V is used as an input parameter to op_1. In this case the operations cannot be executed in parallel because if op_2 concludes before op_1 is processed, variable V will be modified and thus the parameter passed to op_1 will not have the correct value.
   c. *Output dependence:* both operations store their result to the same variable V. In a sequential execution, after op_2 has concluded the value returned by op_2 will be stored in V. If however op_1 and op_2 are scheduled to be executed in parallel, the value of the operation invocation that concluded last will be finally stored in V; therefore in the case that op_1 concludes after op_2, the execution result will be erroneous.
   d. *Input dependence:* both operations share an input parameter.

   If true dependence, anti-dependence or output dependence is identified between two invocations, then they cannot be scheduled to run in parallel; input dependence does not preclude parallel execution of the involved operation invocations [26].
3. Operations op_1 and op_2 cannot be scheduled to be executed in parallel if the invocation of op2 either (a) incurs some cost or (b) has some side-effect (e.g. creating a session, booking a ticket etc.) [31], unless the results of the invocation of *op2* are undoable, through a compensation handler [1] provided in the WS-BPEL scenario. This criterion targets the case in which an exception is raised during the invocation of op_1: if op_1 failed due to an exception and op_2 were scheduled to run after op_1, then op_2 would not be executed at all (and thus the associated cost would not be incurred and/or the relevant side-effects would not be created) since either the scenario would be terminated or control would be transferred to the appropriate fault handler. If however the invocations were executed in parallel, op_2 would run and therefore the associated cost would be incurred, which is undesirable; nevertheless, if the WS-BPEL scenario included a compensation handler for op_2 it would be possible to execute the services in parallel and provide a fault handler which would arrange for invoking op_2's compensation handler to recuperate the cost stemming from the invocation of op_2 and/or undo the created side-effects.
4. Two invocations op_1 and op_2 cannot be scheduled to run in parallel if op_1 creates a side-effect (e.g. creation of a session, sending goods) and op_2 depends on the existence of the side-effect.
5. In all other cases, op_1 and op_2 are able to run in parallel, since conditions analogous to those used for coarse-grain parallelism detection hold between op_1 and op_2 (lack of data dependence *and* lack of (non-undoable) side effects) [33].

Criteria 1 and 2 in the above list, as well as the existence of the compensation handler stated in criterion 3 can be directly evaluated by analyzing the WS-BPEL scenario. The existence of a cost associated with the invocation of a service mentioned in criterion 3 can be directly retrieved from the service repository (e.g. METEOR-S [19]). Finally, side-effects either created by the service (criteria 3 and 4) or needed by the service (criterion 4) can be retrieved from a repository such as WSMO [27]. Obviously, instead of using two distinct repositories, the information needed may be stored into a single, comprehensive repository; in our implementation we have used a unified repository. When two (or more) operation invocations that were initially designated to run sequentially are restructured to run in parallel, their QoS limits regarding the response time can be relaxed. For instance, consider the case that a WS-BPEL scenario comprises of operation invocations O_1 and O_2 that are designated to be executed sequentially, with an upper bound on the response time 3 and 7 seconds, respectively; therefore the upper bound on the scenario execution time would be 10 seconds. If the scenario is restructured so that O_1 and O_2 are executed in parallel, then the upper bound of both operations' execution time can be set to 10 seconds, a setting which provides guarantees that the WS-BPEL scenario will conclude in 10 seconds, but it also broadens the pool of operations that the adaptation mechanism can choose from to realize O_1 and O_2. Generalizing, if operations O_1, O_2, …, O_n were initially restructured to run sequentially and are restructured to run in parallel, then the upper bounds of their response time are set to $\sum_{i=1}^{n} U_i^{RT}$, where U_i^{RT} is the initially set upper bound for the run time of operation O_i.

Taking the above criteria into account, the preprocessor analyzes the structure of the WS-BPEL scenario and determines which invocations can be parallelized. Operations within a sequential structure that are found to be parallelizable, are organized in a *flow* construct. Consider for instance the WS-BPEL scenario fragment illustrated in listing 1 (for conciseness purposes, only the relevant parts/attributes of the scenario are shown), which arranges for getting a quote for a hotel room and booking it, renting a car and then paying for both items. The invocations are arranged in a sequential structure, however in this sequence, we can identify that invocations to *getRoomQuote* and *rentCar* may proceed in parallel, since they (a) have no interdependencies and (b) *rentCar* has an associated cost (the cost of invoking the service e.g. a commission; the actual fee for renting the car is paid later through *finalizeReservation*) and a side-effect (recording the car rental in the service provider's database), however a compensation handler exists, therefore any incurred costs and/or side effects are undoable by invoking this compensation handler.

Contrary, the invocation to *reserveRoom* must strictly be performed after the invocation to *getRoomQuote* has concluded, since *reserveRoom* uses variable *quote* as its input, which is produced by *getRoomQuote* (direct dependency). Similarly the invocation to *finalizeReservation* should follow the conclusion of both *getRoomQuote* and *rentCar* because variable *paymentInfo* (the input of *finalizeReservation*) is indirectly dependent on the output of *rentCar* (variable *carRentalInfo*) and *getRoomQuote* (variable *quote*), since the *copy* construct in listing 1 uses the *carRentalInfo* and *quote* variables to calculate the value to be assigned to (a part of) *reserveRoom*'s input *paymentInfo*. A more subtle dependence exists between services *reserveRoom* and *finalizeReservation*, which cannot be determined by analyzing the scenario code alone: *finalizeReservation* can be performed only when a room has been reserved; this is a required side-effect for operation *finalizeReservation*, and this side-effect is produced by operation *reserveRoom*, hence *reserveRoom* must have concluded before *finalizeReservation* is invoked. The information regarding the side effects is drawn by the preprocessor from the service repository, where it is recorded that *reserveRoom* creates the side effect and *finalizeReservation* depends on it. After the dependence analysis results have been computed, the WS-BPEL scenario is restructured to accommodate the available parallelism, as shown in listing 2

```
<sequence>
 <invoke operation="getRoomQuote" outputVariable= "quote"
    inputVariable="roomTypeAndPeriod" name= "getQuote"/>
 <invoke operation="reserveRoom" inputVariable= "quote"
    outputVariable="reservationInfo" name= "reserveRoom"/>
 <invoke operation="rentCar" inputVariable= "carTypeAnd
    Period" outputVariable="carRentalInfo" name="rentCar">
  <compensationHandler>
    <invoke operation="cancelRentCar" inputVariable=
      "carRentalInfo">
  </compensationHandler>
 </invoke>
 <assign>
  <copy>
    <from expression="$quote.price + $rentalInfo.price" />
    <to variable="paymentInfo" part="amount" />
  </copy>
 </assign>
 <invoke operation="finalizeReservation" name="doReserve"
    inputVariable="paymentInfo" outputVariable="receipt" />
</sequence>
```

Listing 1: Excerpt of sequential WS-BPEL scenario

(only the first part which has changed is shown; the part that has remained intact has been omitted for brevity; [29] includes graphical representations of the two scenario excerpts). Regarding the upper response time bound of the services that are restructured to be executed in parallel, the preprocessor arranges for designating that the upper response time bound of *each of the invocations* to *getRoomQuote* and *rentCar* is equal to the sum of the individual invocations, with the sum being normalized to the [1, 10] scale.

An issue that needs to be addressed regarding these transformations, is the fact that one of the criteria for determining whether operations are parallelizable, and in particular the criterion examining whether the involved service incurs some cost (criterion 3 above) is based on the service repository contents. However, the service repository contents may change regarding this dimension i.e. either (a) a provider may begin charging a previously free service, hence operation invocations that were previously parallelizable cease to be so, or (b) a provider may stop charging a previously non-free service, in which case two invocations that were previously non-parallelizable can now be scheduled to be executed in parallel. A similar issue exists for side-effect creation and requirement. To tackle this issue, the preprocessor takes the following two measures:

1. to guard against selecting a non cost-free service, the preprocessor arranges for setting the upper bound for the cost of the particular invocation to zero (normalized to the [1, 10] scale).
2. in all cases, the preprocessor establishes *redeployment triggers*, which consist of monitoring updates to the repository that fall into the previously described categories (cost, side-effect creation and side-effect requirement). When such a change is detected, the affected WS-BPEL scenarios are identified and a preprocessing and redeployment action is initiated for them, so that the preprocessor takes into account the updated contents of the repository (c.f. Fig. 1).

```
<sequence>
 <flow>
  <invoke operation="getRoomQuote" inputVariable=
      "roomTypeAndPeriod" outputVariable="quote"
      name="getQuote" >
    <compensationHandler>
      <invoke operation="cancelRentCar"
          inputVariable="carRentalInfo"/>
    </compensationHandler>
  </invoke>
  <invoke operation="rentCar"
    inputVariable="carTypeAndPeriod"
    outputVariable="carRentalInfo" name="rentCar" />
 </flow>
 <sequence>
  <invoke operation="reserveRoom" inputVariable="quote"
    outputVariable="reservationInfo" name="reserveRoom"/>
  <assign>
    …
 </sequence>
</sequence>
```

Listing 2: Excerpt of transformed WS-BPEL scenario

4. THE ADAPTATION ARCHITECTURE

The adaptation architecture, illustrated in Fig. 1, adds to the standard SOA architecture three additional modules, the *preprocessor*, the *adaptation layer* and the *redeployment triggers*.

The *preprocessor* performs transformations on the original WS-BPEL scenario by (a) restructuring service invocations to be performed in parallel under the conditions described in Section 4 above (b) arranging for passing appropriate data to the adaptation layer to drive the adaptation and (c) redirecting service invocations to the adaptation layer, so as to be sent to the service implementations best matching the QoS specifications. The preprocessing step produces an *enhanced WS-BPEL scenario*, which is then deployed to the WS-BPEL orchestrator. Activities (b) and (c) are performed in a similar way to [20] and are omitted in this paper due to space limitations; details on these steps can be also found in [29].

The *adaptation layer* intervenes between the WS-BPEL orchestrator and the actual web service implementations, arranging for formulating the WS-BPEL scenario *execution plan*, i.e. to choose for each operation invocation designated in the executing scenario the most appropriate implementation with respect to the QoS policy defined for the current execution. The adaptation layer uses integer programming to determine the optimal execution plan for the specific WS-BPEL scenario execution, subject to the QoS policy specified by the consumer, and stores this execution plan to the *session memory*. Subsequently intercepts service invocations performed in the context of the WS-BPEL scenario execution and redirects them to the chosen service implementations. Execution plan computation and service invocation redirection are performed in a similar way to [20] , while details on these steps can be also found in [29].

Finally, *redeployment triggers* periodically check whether changes have occurred to the data within the repository on the basis of which decisions regarding parallelization capability have been made. This includes (a) cost of services (b) creation of side-effects by services and (c) requirement of side-effects by services. When such a modification is expected, the affected WS-BPEL scenarios are identified and, for each of them, the preprocessor is invoked to perform the applicable transformations, considering the updated service repository contents. Redeployment of the new preprocessed file is performed without affecting currently running instances of the scenario, exploiting the hot redeployment feature of contemporary WS-BPEL orchestrators (e.g. [30]).

5. EXPERIMENTAL EVALUATION

In this section, we report on our experiments aiming to substantiate the feasibility of the proposed approach, both in terms of execution

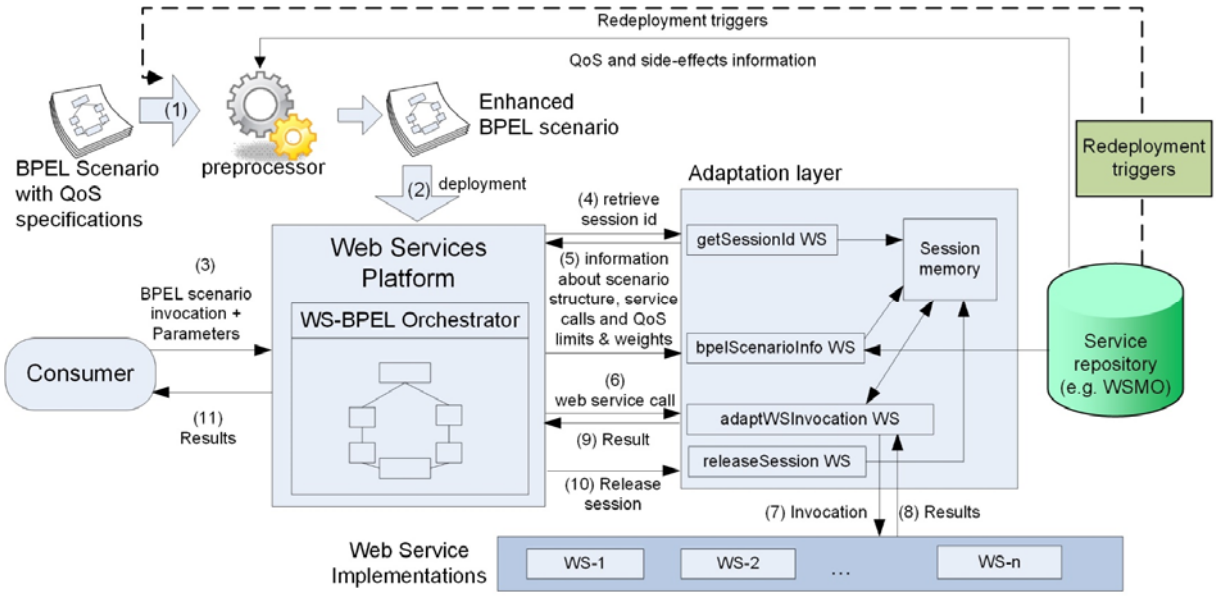

Figure 1: The adaptation architecture

time (quantifying the introduced overhead and performance gains) and solution quality. For our experiments we used two machines: (a) a workstation, equipped with one 6-core Intel Xeon E5-2620@2.0GHz CPU and 16 GB of RAM, which hosted the preprocessor and the clients and (b) a workstation with identical configuration to the first, except for the memory which was 64GBytes, that hosted the WS-BPEL orchestration engine (Apache ODE 1.3.6), the adaptation layer, the target web services deployed on a Glassfish 4.1 application server and the service repository. The machines were connected via a 1Gbps LAN. The service repository was implemented as in-memory hash-based structure, which proved more efficient than using a separate (memory or disk-based) database. Preprocessing time is not included in the overheads, since this is performed in an off-line fashion and does not penalize the WS-BPEL scenario execution performance. In all experiments, the service repository was populated with synthetic data having an overall size of 1,000 web services; each web service included 3-8 operations and each operation was offered by a number of alternative providers, ranging from 5 to 50. Each service had at least 5 other services equivalent to it (i.e. having equivalents for *all* its operations). QoS attribute values in this repository were uniformly drawn from the domain [0, 10]. The WS-BPEL scenarios used in the experiments were synthetically generated by randomly drawing operations from the repository, and the performance evaluation tests were run for each of the generated scenarios; 1,000 scenarios were generated in total. We resorted to synthetic data due to the lack of a real-world test suite. In the scenario generation process, two consecutive functionality invocations were selected to be executed sequentially (*sequence* construct) with a probability of 0.7 and in parallel (*flow* construct) with a probability of 0.3. In our first experiment, we quantify (a) the time needed to formulate the WS-BPEL scenario execution plan, for varying degrees of concurrency (incurred once per execution), (b) the overhead imposed by the middleware intervention during service invocation (incurred for each invocation; the diagram illustrates the overhead sustained for *all* invocations within the scenario executions) and (c) the overall overhead per WS-BPEL scenario execution (Fig. 2). We can observe that all overheads remain relatively low, even for high degrees of concurrency, (an overall penalty of 250 msec for 200 concurrent invocations) and scales linearly with the concurrency degree.

Fig. 3 compares the QoS of the execution plan formulated for a number of representative trial cases and on average by (i) the simple QoS-based algorithm described in [8] and (ii) the approach proposed in this paper. The average shown in the diagram has been computed considering all 1,000 WS-BPEL scenarios used in the experiment, while the representative trial cases were chosen so as to include different number of operation invocations (scenarios 1-3 contain 3 invocations, scenarios 4-6 contain 6 invocations and scenarios 7-10 contain 8 invocations), varying settings regarding parallel flows (scenarios 1, 2, 4 and 7 contain no parallel flows, scenarios 3, 5, 8 and 9 contain one parallel flow and scenarios 6 and 10 contain two parallel flows) and different numbers of data-dependent invocations (from one to seven; some data dependencies formed chains e.g. *s1 is dependent on s2* $\wedge$ *s2 is dependent on s3*, while other data dependencies were unconnected, e.g. *s1 is dependent on s2* $\wedge$ *s3 is dependent on s4*).

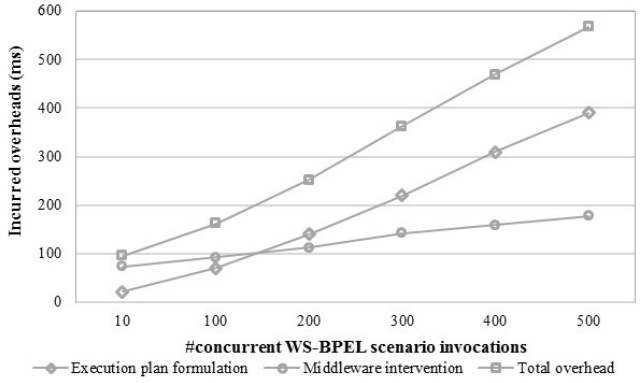

Figure 2: Execution plan formulation overhead

We chose to compare the proposed approach against the one described in [8], since the latter handles parallel flows and is exhaustive, always thus locating the optimum solution. The lower and upper QoS bounds for operation invocations were randomly drawn

from the domains [0,4] and [6,10, respectively]. The weights of the QoS attributes were randomly selected from the domain [0,1]. In all cases, a uniform distribution was used. The diagram shows that the algorithm proposed in this paper achieves solutions whose QoS is on average higher by 22% than the corresponding solutions formulated by the algorithm described in [8]. This is due to the parallelization of operation invocations, which (a) lead to reduced response time and (b) due to the relaxation of the response time constraints allowed by the parallelization, the set of alternatives available to the adaptation mechanism is broadened (through allowing for selection of implementations with higher execution times than would be possible in the original scenario with sequential execution); this in turn provides opportunities for formulation of better execution plans, in the cases that the implementations that can now be selected score better in the rest QoS dimensions.

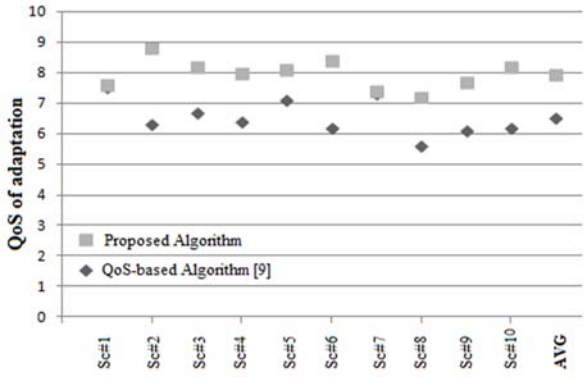

Figure 3: QoS of solutions formulated by the proposed approach and the algorithm described in [8].

6. RELATED WORK

Insofar, WS-BPEL scenario adaptation has received considerable research attention, with proposed adaptation methods following either the *horizontal* or the *vertical* adaptation approach [3]. VieDAME [6] considers QoS parameters to adapt WS-BPEL scenario execution; pluggable modules, attached to the WS-BPEL orchestrator, provide support for the QoS parameters and the selection strategy. VieDAME monitors the execution of WS-BPEL scenarios and arranges for dynamic replacement of web services that fail to meet the desired QoS levels [20]. VieDAME also uses a monitoring mechanism, using observed service QoS levels to predict future performance of services; these predictions are used to improve adaptation quality. Work in [7] allows for specification of QoS constraints, which drive the adaptation process; the adaptation process aims to minimize an objective function for the entire orchestration employing either brute force (*OPTIM_S*) or heuristic (*OPTIM_HWEIGHT*) algorithms. [4] introduces AgFlow, which performs QoS-based adaptation using a middleware approach. AgFlow has two modes of operation: *global planning*, where QoS constraints are designated for the composite service as a whole rather than to individual tasks, and integer programming is used to compute optimal plans for composite service executions and *local optimization*, where QoS constraints are set to individual tasks; in local optimization, execution plan optimality is not guaranteed, but execution plan formulation is more efficient. Moreover, AgFlow allows for execution replanning, to tackle issues such as services becoming unavailable or changing their predicted QoS. The work in [16] uses constraint optimization techniques to formulate the optimal execution plan, allowing for human user intervention to enhance the solving process. [5] integrates QoS-based adaptation with exception resolution; the ASOB middleware introduced in this work intercepts service invocation failures and distinguishes business logic faults from system faults, remedying the latter category through replacing failed services by "next best" solutions; exceptions stemming from business logic faults are left to the WS-BPEL scenario designer to resolve through appropriate handlers, since they cannot be addressed automatically. [8] extends the work in [5] by being able to adapt scenarios that include <*flow*> constructs.

The issue of performance when composing (or adapting) large composition structures has lately received research attention. [17] uses mixed integer programming to decompose large-scale composition structures into small-scale composition segments, and subsequently determines a QoS-optimal composite solution for each small-scale composition segment, reducing thus the time needed for solving the composition problem.

Regarding the parallelization of WS-BPEL scenario execution, although both business and scientific workflows have been identified to be highly parallel [15], little work has been reported. [12] presents a formal model for semantic-driven service execution; by examining data flows, this model provides potential for parallelizing service execution, based on the data dependencies among services. This approach requires however extensive semantic information (information, functional, nonfunctional and behavioral descriptions of the web services, complementary to their WSDL descriptions), while it also does not consider QoS aspects and adaptation. Execution parallelization techniques have however been studied extensively in other domains, notably in instruction-level parallelism [24], and the data flow and dependence analysis used in this domain can be exploited for parallelizing the execution of WS-BPEL scenarios.

7. CONCLUSION AND FUTURE WORK

In this paper we have presented a transformation-based approach to exploit the potential parallelism in service invocations, so as to minimize the overall WS-BPEL scenario execution time. Besides improving execution time, this exploitation provides more choices to the adaptation mechanism, enabling it to formulate execution plans of better quality. We have also described an architecture for realizing the parallelization and adaptation. The proposed algorithm has been experimentally validated regarding (i) its performance and (ii) the quality of execution plans generated.

Our future work will focus on extending the algorithm to handle more efficiently conditional and iteration constructs in the WS-BPEL scenario, as well as developing the relative framework; this can be supported through branch prediction and loop unrolling techniques [26][32], as well as by gathering statistical information from prior scenario executions and using it as input to the adaptation process. This information will quantify aspects regarding the behavior of control constructs in the scenario, e.g. the probability that a conditional branch is executed or the distribution of the number of executions of a loop [32].

8. REFERENCES

[1] OASIS WSBPEL TC. WS-BPEL 2.0. http://docs.oasisopen.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

[2] MP. Papazoglou, P. Traverso, F. Leymann, "Service-Oriented Computing: State of the Art and Research Challenges", IEEE Computer vol. 40, no 11, 2007, pp. 38-45.

[3] V. Cardellini, V. Di Valerio, V. Grassi, S. Iannucci, F. Lo Presti, "A Performance Comparison of QoS-Driven Service

Selection Approaches", Proceedings of ServiceWave 2011, Abramowicz W et al. (Eds.): 2011, pp. 167–178.

[4] LB. Zeng, AHN. Benatallah, M. Dumas, J. Kalagnanam, H. Chang, "QoS-aware middleware for web services composition". IEEE Transactions on Software Engineering, vol. 30, no 5, 2004.

[5] C. Kareliotis, C. Vassilakis, S. Rouvas, P. Georgiadis. "QoS-Driven Adaptation of BPEL Scenario Execution", Proceedings of ICWS 2009, pp. 271-278.

[6] O. Moser, F. Rosenberg, S. Dustdar, "Non-Intrusive Monitoring and Service Adaptation for WS-BPEL", Proceedings of WWW 2008, China, 2008, pp. 815-824.

[7] Y. Xia, P. Chen, L. Bao, M. Wang, J. Yang, "A QoS-Aware Web Service Selection Algorithm Based on Clustering", Proceedings of ICWS11, 2011.

[8] D. Margaris, C. Vassilakis, P. Georgiadis, "An integrated framework for QoS-based adaptation and exception resolution in WS-BPEL scenarios", Proceedings of the ACM Symposium on Applied Computing, 2013, Portugal.

[9] G. Canfora, M. Di Penta, R. Esposito, ML. Villani, "An Approach for QoS-aware Service Composition based on Genetic Algorithms", Proceedings of the 2005 conference on Genetic and evolutionary computation, 2005, pp. 1069-1075.

[10] D. Margaris, C. Vassilakis, P. Georgiadis, "An integrated framework for adapting WS-BPEL scenario execution using QoS and collaborative filtering techniques", Science of Computer Programming 98, 2015, pp. 707–734.

[11] J. O'Sullivan, D. Edmond, A. Ter Hofstede, "What is a Service?: Towards Accurate Description of Non-Functional Properties", Distributed and Parallel Databases, vol. 12 2002.

[12] T. Vitvar, A. Mocan, M. Zaremba, "Formal Model for Semantic-Driven Service Execution", Proceedings of ISWC 2008, LNCS 5318, 2008, pp. 567–582.

[13] ITU. Recommendation E.800 Quality of service and dependability vocabulary.

[14] J. Cardoso, "Quality of Service and Semantic Composition of Workflows", PhD thesis, Univ. of Georgia, 2002.

[15] Y. Gil, E. Deelman, M. Ellisman, T. Fahringer, G. Fox, D. Gannon, C. Goble, M. Livny, L. Moreau, J. Myers, "Examining the Challenges of Scientific Workflows", IEEE Computer, vol. 40(12), December, 2007, pp. 24-32.

[16] A.B. Hassine, S. Matsubara, T. Ishida, "A Constraint-Based Approach to Horizontal Web Service Composition", Procs. of the 5th International Semantic Web Conference, ISWC 2006, Athens, GA, USA, November 5-9, 2006, pp. 130-143.

[17] L. Qi, X. Xia, J. Ni, Ch. Ma, Y. Luo, "A Decomposition-based Method for QoS-aware Web Service Composition with Large-scale Composition Structure", Proceedings of the Fifth International Conferences on Advanced Service Computing, A. Koschel, J.L. Mauri (eds), May 2013, Spain, pp. 81-86.

[18] X. Fei, S. Lu, "A Dataflow-Based Scientific Workflow Composition Framework", IEEE Transactions on Services Computing 5(1), 2012, pp.45-58

[19] K. Verma, K. Sivashanmugam, A. Sheth, A. Patil, S. Oundhakar and J. Miller, METEOR-S WSDI: A Scalable P2P Infrastructure of Registries for Semantic Publication and Discovery of Web Services, Journal of Information Technology and Management, 6(1), 2005 pp. 17-39.

[20] D. Margaris, C. Vassilakis, P. Georgiadis, "A Hybrid Framework for WS-BPEL Scenario Execution Adaptation, Using Monitoring and Feedback Data", Proceedings of the ACM Symposium on Applied Computing, 2015, Spain

[21] M. Alrifai, T. Risse, "Combining Global Optimization with Local Selection for Efficient QoS-aware Service Composition", Proc.s of the 18th international conference on World Wide Web (WWW '09), 2009, pp. 881-890.

[22] Z.H. Du, C.C. Lim, X.F. Li, C. Yang, Q. Zhao, T.F. Ngai, "A Cost-Driven Compilation Framework for Speculative Parallelization of Sequential Programs", Procs. of ACM SIGPLAN 2004pp. 71-81.

[23] M. Chen, K. Olukotun, "The Jrpm System for Dynamically Parallelizing Java Programs", Proc. of the 30th annual international symposium on computer architecture, 2003.

[24] U. Khedker, A. Sanyal, B. Sathe, "Data Flow Analysis: Theory and Practice", CRC Press, 2009, ISBN-10: 0849328802

[25] G. Goff, K. Kennedy, C.W. Tseng, "Practical Dependence Testing", Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation, 1991, pp. 15-29.

[26] A.J. Bernstein, "Analysis of Programs for Parallel Processing", IEEE Trans. on Electronic Computers" Volume:EC-15(5), 1996, pp. 757–763.

[27] D. Martin, M. Paolucci, S. McIlraith, M. Burstein et al. "Bringing Semantics to WS: The OWL-S Approach", in Semantic Web Services and Web Process Composition, LNCS vol. 3387, 2005, pp. 26-42.

[28] H. Lausen, A. Polleres, D. Roman (eds), " Web Service Modeling Ontology (WSMO)", W3C Member Submission 3 June 2005, http://www.w3.org/Submission/WSMO/

[29] D. Margaris, C. Vassilakis and P. Georgiadis. Preprocessor transformations and adaptation operations for improving QoS delivered by WS-BPEL scenario adaptation through service execution parallelization. http://sdbs.dit.uop.gr/?q=node/287

[30] Red Hat. JBoss Enterprise SOA Platform 5: ESB Services Guide. https://access.redhat.com/documentation/en-US/JBoss_Enterprise_SOA_Platform/5.

[31] S. Kona, A. Bansal, G. Gupta, and T.D. Hite, "Semantics-based Efficient Web Service Discovery and Composition", 2007 The University of Texas at Dallas, Texas, USA, http:// info.asprs.org/publications/proceedings/tampa2007/0019.pdf.

[32] D. Ardagna, B. Pernici, "Adaptive Service Composition in Flexible Processes", IEEE Transactions on Software Engineering, vol. 33, no. 6, June 2007, 369 – 384

[33] M. W. Hall, S. P. Amarasinghe, B. R. Murphy, S-W Liao, M. S. Lam, "Detecting Coarse-Grain Parallelism Using an Interprocedural Parallelizing Compiler", Proceedings of the 1995 ACM/IEEE conference on Supercomputing, 1995.

[34] D. Ardagna, B. Pernici, "Adaptive Service Composition in Flexible Processes". IEEE Transactions on Software Engineering, 33, 6, June 2007