

## Runtime Web-Service Workflow Optimization \*

Radu Sion<sup>1</sup> and Junichi Tatemura<sup>2</sup>

<sup>1</sup> Computer Sciences  
Stony Brook University  
Stony Brook, NY 11794  
sion@cs.stonybrook.edu

<sup>2</sup> NEC Research Labs  
10080 N. Wolfe Rd.  
Cupertino, CA 95014  
tatemura@sv.nec-labs.com

**Summary.** Increasingly, business models build on mechanisms for online interaction. Web Services, business workflows and service orchestrations provide essential tools for staging successful networked business to business (B2B) or business to customer (B2C) transactions.

One of the important online interaction quality metrics is perceived response time. The ability to sustain a high throughput and provide fast service effectively means more business and associated revenue. Down-times and slow transactions can be a strong deterrent for potential customers. The ability to automatically monitor and optimize response performance becomes more important than ever.

In this paper we propose a solution for optimizing web service business workflow response times through dynamic resource allocation. On-the-fly monitoring is combined with a novel workflow modeling algorithm that discovers critical execution paths and builds “dynamic” stochastic models in the associated “critical graph”. One novel contribution of this work is the ability to naturally handle parallel workflow execution paths. This is essential in applications where workflows include multiple concurrent service calls/paths that need to be “joined” at a later point in time. We discuss the automatic deployment of on-the-fly monitoring mechanisms within the resource management mechanisms. We implement, deploy and experiment with a proof of concept within a generalized web services business process (BPEL4WS/SOAP) framework. In the experiments we show the natural adaptation to changing workflow conditions and appropriate automatic re-allocation of resources to reduce execution times.

**Key words:** Web Services, Workflows, Response Time Optimization

### 1.1 Introduction

To achieve maximum performance with a finite set of resources, these need to be (re)allocated preferably for each new instance of a workflow execution. Because it is often hard for human system operators to perform such allocations appropriately, to

---

\* An extended abstract [1] of this paper appeared in ICWS 2005.

achieve maximum utilization and response times, an ability to automatically do so becomes more important than ever. In [2] Gillmann, Weikum and Wonner propose an adaptive workflow management solution integrating service quality awareness with stochastic modeling of workflows, an approach well suited in the case of single-threaded workflows with non-concurrent paths.

There exist scenarios however, where such static stochastic models cannot be applied. These include workflows with parallel threads of (potentially) mutually interacting services. A classical example is the case of a manufacturing company workflow, issuing a request to a supplier *A* for various parts and, *occasionally* also to a supplier *B* (simultaneously). A static stochastic model of the workflow could not possibly accommodate this conditional scenario. Moreover if the two parallel workflow paths *A* and *B* interact (e.g. by messaging), things complicate further.

Here we propose to build upon the stochastic modeling approach introduced in [2] and provide a solution able to handle such dynamic and parallel workflows. Our solution is based on the on-the-fly discovery of workflow critical paths [3] and their analysis within an adaptive resource reallocation framework.

While there exist likely scenarios for single-webservice invocation workflows, here we analyze mainly workflows invoking an average of more than one web - service in their lifetime. This is of interest as we are not as much concerned with individual web services but rather the the ability to perform global workflow optimization through web service resource (re)allocation. We also note that, while we scope our solution for web service business workflows, it arguably extends to arbitrary frameworks.

We also implemented and experimented with a proof of concept within a web services based business workflows (i.e. BPEL [4]) framework. The design is multi-layered: the first layer enables the ability to transparently monitor execution times for the component workflow states (web service calls) and collect associated execution times. The second layer reasons about these collected statistics dynamically, with the aim to optimize global responsiveness (workflow execution times). This layer then originates resource (re)allocation recommendations to the third layer handling resource allocation.

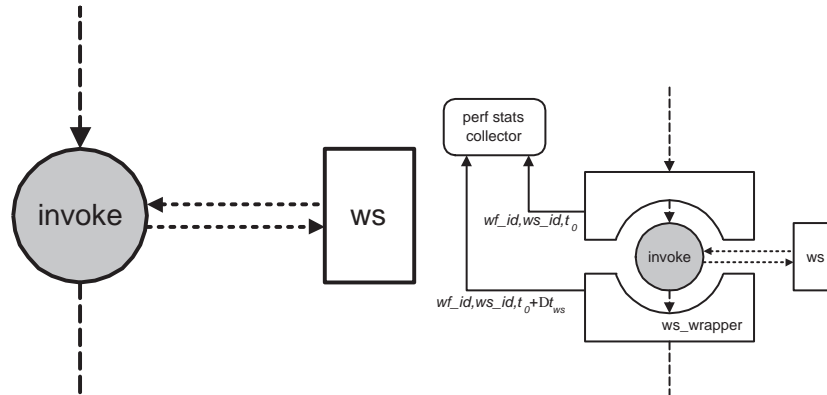
The main contributions of our work include: (i) a solution for on-the-fly optimization of average workflow response time, adaptive to a dynamic environment (e.g., workload patterns, available server resources, network congestion), (ii) a proof-of-concept implementation of monitoring architecture components, enabling run-time profiling and data collection on open-standard Web Service platforms, (iii) a “dynamic” stochastic workflow modeling approach for workflows with concurrent paths, and (iv) the experimental analysis thereof. Moreover, we note that our model is based on a critical path as a construction as opposed to existing models which only consider simple execution paths.

The paper is structured as follows. Section 1.2 introduces the web service execution monitoring framework. Section 1.3 defines the modeling framework and establishes an execution-aware weighting scheme for workflow tasks. Section 1.4 discusses the resource allocation algorithm. In Section 1.5 we explore limits and im-

provements to our solution. Section 1.6 presents experimental results and Section 1.7 explores related work. Section 1.8 concludes.

## 1.2 Web Service Wrapping

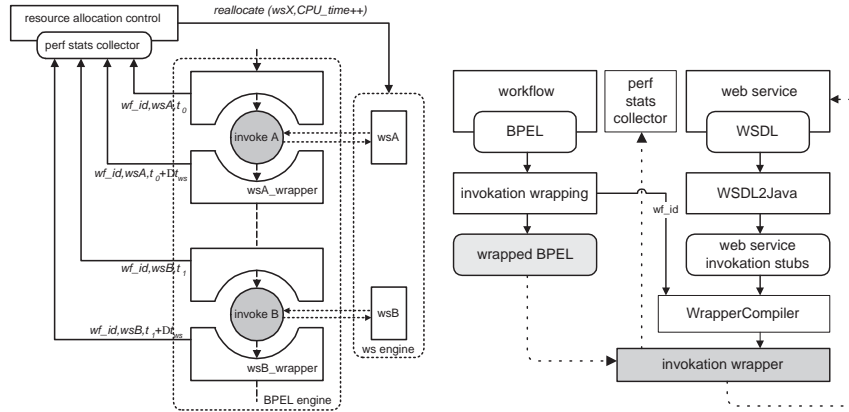
Statistics and local information about individual web service performance are essential in building any solution for evaluating and optimizing of overall orchestration/workflow behavior. The process of collecting these statistics has to be non-intrusive and transparent from both a programming and deployment point of view. More specifically, we are concerned here with the ability to assess local and remote web service response times within an existing, deployed BPEL [4] workflow which invokes a set of external services. One of the main difficulties in designing an automated solution for this derives from the requirement to be able to associate each response time sample with both the corresponding web service and a specific workflow instance.



**Fig. 1.1.** (a) A remote web service is accessed in a traditional BPEL invocation. (b) Transparent monitoring entails automatically modifying the BPEL flow description with calls to invocation wrappers instead of direct web service invocations.

For the purpose of optimizing overall workflow behavior, we need the ability to monitor composing individual web service behavior directly for each execution instance, with a minimal overhead. In our solution, we propose to enable this through a novel call-tracing mechanism, namely *invocation wrappers*. An invocation wrapper is a software component (e.g. Java class) associated with a specific web service that acts as a proxy for it (within a workflow execution engine, e.g. invoked from the associated BPEL file) while also collecting execution time statistics into a common repository. In Figure 1.1 (a) a traditional BPEL web service invocation/call is

depicted. Figure 1.1 (b) illustrates the same call, wrapped within an associated invocation wrapper (*ws\_wrapper*). In summary, invocation wrappers are used for dynamic and transparent response time profiling.



**Fig. 1.2.** (a) When multiple web services compete, the resource allocation is greatly improved if on-the-fly workflow performance awareness is built in. (b) The invocation wrapper is automatically produced by the wrapper compiler from the java invocation stubs generated by a WSDL to java converter and a unique identifier for the current workflow.

Naturally, most workflows invoke multiple web services throughout their lifetime. When multiple web services are coming into play, an optimization of the global workflow execution/response time can be achieved by an adaptive resource allocation scheme, considering each invoked component, see Figure 1.2 (a).

For (mainly) economic reasons, invocation wrappers need to be non-intrusive from both a programmer's and runtime deployment point of view. Legacy applications are to be handled and assuming a runtime profiling aware programming model is not always natural. Thus, the monitoring framework has to provide for automatic transparent invocation wrapping for targeted web services within a BPEL process.

Achieving this goal entails two dimensions. First, the workflow specification file (BPEL, omitted due to space constraints) needs to be augmented with invocation wrapper calls instead of direct web service invocations. A traditional call to a web service method *getValue(Stringname)* is wrapped by invoking the wrapper instead. An additional ("wf\_id") parameter is added to its signature, initialized with a per-workflow unique identifier. This parameter is used to associate the collected web service execution statistics with their corresponding workflow. Additionally, the wrapper partner will have an associated WSDL [5] file derived from the original web service description file augmented with the new method signature(s), passing workflow identification information to the wrapper. The second stage in automatically wrapping a web service invocation provides for each wrapped web service an invocation wrapper. The invocation wrapper is to be automatically generated (see Figure

1.2 (b)) by a specialized wrapper compiler (an extension of WSDL2Java) that takes as input a unique current workflow identifier and a web service invocation stub and generates the invocation wrapper. The wrapper is in effect a modified version of the web service invocation stub, augmented with code that profiles the current execution and stores the associated data in a specialized (system-wide) statistics collector bean.

### 1.3 Workflow Modeling

Once the monitoring framework is in place, our solution deploys a novel inference mechanism able to model and predict workflow behavior and associated optimizing resource (re)allocations for the composing tasks. Here we outline this model.

#### 1.3.1 Stochastic Workflow Modeling

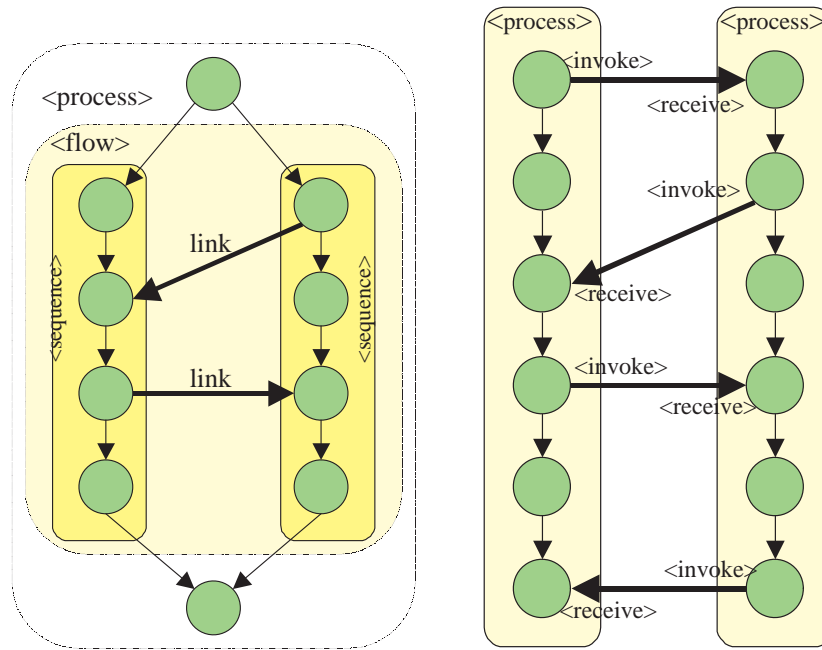
In [6] and [2] Gillman et al. motivate and discuss the issue of service quality guarantees in the framework of workflow management for e-services. A solution based on static stochastic modeling of workflow behavior is provided.

The core algorithm starts by defining the concept of a “flow process”, a “stochastic process with a finite set of states”. Certain conditions and assumptions are then associated with the transitions between these states, including: (i) “the time until the flow process performs the next transition is randomly distributed”, (ii) there is a given (static) probability of the flow process entering a certain state, and (iii) this probability is independent of the time the process spent in the current state and also (iv) “independent of the process’s earlier history”.

A stochastic model is built for a given workflow using the assumptions above. It is then coupled with a “performance model” in which each service call (server) is modeled by “considering only its mean service time per request and the second moment of this metric”. By estimating these values (through “collecting and evaluating online statistics”) and feeding them into the stochastic model, resource allocation requirements are inferred for optimizing execution times.

**Limitations.** The static stochastic approach is certainly suited for a number of workflow scenarios under given assumptions. One of the most restrictive assumption is the requirement that the Markov model transition probabilities are history independent (point (iii) above, “independent of the time period that the flow process spent in ... [a previous state]”).

There are scenarios however (see Figure 1.3), where such static stochastic models cannot be applied. These include workflows with parallel threads of (potentially) mutually interacting services. A classical example could be the case of a manufacturing company workflow, issuing a request to a supplier *A* for various parts and, *occasionally* also to a supplier *B* (simultaneously). A static stochastic model of the workflow could not possibly accommodate this conditional scenario. Moreover, input data-dependency in workflow transitions is often occurring and this dependency is mostly related to the past history of the flow, thus directly conflicting with point (iv) in Section 1.3.1 above.



**Fig. 1.3.** Scenarios where a direct stochastic approach cannot model workflow behavior. (a) Concurrent sequences within a process (b) Concurrent processes

Yet another important aspect that cannot be handled by a purely stochastic approach is the scenario of parallelism, e.g. inter-thread or inter-process asynchronous communication within the same workflow. (in the previous example, if the two parallel workflow paths *A* and *B* interact, e.g. by messaging) For this aspect, an apparent solution would be to augment the semantics of transitions with synchronization primitives, however this is highly likely leading to an un-intuitive, complex model. In the following we show how “dynamic” stochastic modeling over the space of graphs defined by workflow-instance critical paths can be used to naturally handle these issues.

### 1.3.2 Instance Critical Paths

Starting from the need to address the limitations outlined above, we propose an alternative to simple stochastic modeling, more suited to handling these and other specific workflow-modeling challenges. Our solution relies on the concept of workflow *critical paths*. Critical paths aim to capture the most time-consuming activities within a workflow. In other words, the question that is answered is: which of the composing workflow tasks have the most impact on the overall workflow execution time? Traditionally [7] [8], a workflow critical path is defined as the *longest execution path*

from the initial task to the final task of the workflow. Here we use a by-construction runtime definition, designed specifically to leverage data available from the monitoring framework. Our definition computes critical paths for workflow instances on the fly by processing runtime execution statistics. In the following we are using the term *critical path* or *instance critical path* to denote a path within a workflow *instance* (not schema).

Let there be a workflow composed of a set of states/tasks  $\mathbb{S} = \{S_0, S_1, S_2, \dots, S_n\}$  and transitions  $\mathbb{T} \subset \mathbb{S} \times \mathbb{S}$  (i.e.  $(S_i, S_j) \in \mathbb{T}$  iff. there exists a workflow transition from task  $S_i$  to task  $S_j$ ). Let  $O(S_i) = \{X | \forall (S_i, X) \in \mathbb{T}\}$ , the set of all potential “target” (child) transitions of  $S_i$  and  $I(S_i) = \{X | \forall (X, S_i) \in \mathbb{T}\}$  the set of all “incoming” transitions of  $S_i$  (see Figure 1.4 (a)).

Let there be a set of predicates  $\mathbb{P}$  defining the semantics of workflow execution, in particular its transition schedule. Each task  $S_i$  with a single incoming transition ( $|I(S_i)| = 1$ ) is considered to immediately start executing. For each task, the execution behavior is defined by a conditional “incoming transition evaluation” expression,  $SYNC_{S_i} : I(S_i) \rightarrow \{true, false\}$  that govern the transition execution.  $S_i$  only starts executing if  $SYNC_{S_i}()$  becomes true.  $SYNC_{S_i}$  can evaluate an arbitrary boolean formula on the state of the incoming transitions. We assume that  $SYNC_{S_i}$  is expressed in a normal conjunctive form.

Considering a certain workflow instance, for each task  $S_i$ , let  $\tau_i$  be the time when  $S_i$  starts executing and  $\tau_i + \delta_i$  the time when  $S_i$  finishes (i.e. is removed from  $A()$ , the set of active tasks, see definition below). If for example  $SYNC_C(AC, BC) = (AC \text{ and } BC)$ , then task  $C$  is not executing until both transitions  $AC$  and  $BC$  are done by the workflow, i.e. when both tasks,  $A$  and  $B$  finish executing. To be noted that in this particular case,  $C$  starts executing at time  $\tau_C = \max(\tau_A + \delta_A, \tau_B + \delta_B)$ . For each task  $S_i$ , the outgoing transitional behavior is modeled by a specific function,  $TRANS(S_i)$  which determines which of the transitions in  $O(S_i)$  are to be pursued upon completion. These semantics of business workflows allow for multiple parallel “threads”. If for example  $TRANS(C) = \{D, E\}$  then, after completing  $C$ , the workflow is “split” into two parallel execution threads starting in  $D$  and  $E$ . Because of this, at each point in time, several tasks in  $(S)$  can be “active”. Let  $A(t) \subset \mathbb{S}$  (or shorter  $A()$ , for current time) be the set of active tasks of the given workflow at time  $t$ .

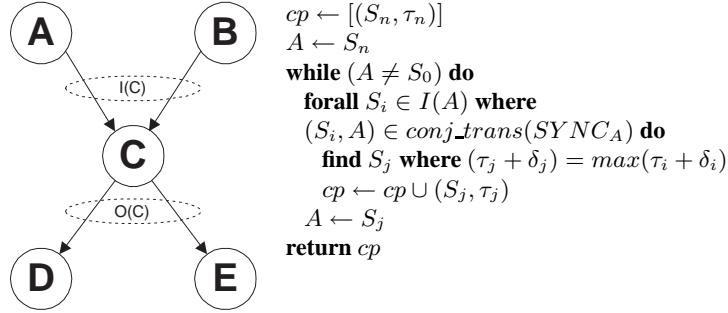
Let  $S_0$  and  $S_n$  (by convention) be the initial, respectively final task of the workflow. In the following we are only concerned with workflow instances that are terminating. In other words, the flow execution starts in state/task  $S_0$  ( $A(0) = \{S_0\}$ ) at time  $\tau_0$  and ends when task  $S_n$  is reached ( $S_n \in A()$ ) at a time  $\tau_n$ <sup>3</sup>. A business workflow is then completely described by  $\mathbb{S}$ ,  $\mathbb{T}$  and  $TRANS_{S_i}() \forall S_i \in \mathbb{S}$ . For the purpose of evaluating and, more importantly, optimizing runtime behavior, this defining description needs to be augmented with the ability to model workflow performance by handling more “measurable” quantities. For a workflow instance, the monitoring framework provides execution and transition statistics. For each task

<sup>3</sup> Naturally, one main goal for workflow optimization is minimizing the execution time,  $(\tau_n - \tau_0)$ .

$S_i \in \mathbb{S}$ , let  $TS(S_i)$  be the set of all pairs of corresponding  $(\tau_i, \delta_i)$  times. Let  $TS = \cup_{i \in (0, n)} (TS(S_i))$ . Also, let  $TR(S_i, t) \subset O(S_i)$  be the set of transitions “fired” after task  $S_i$  completes at time  $t$ . These statistics are collected by the monitoring framework.  $TR()$  effectively models  $TRANS(S_i)$ . Let  $TR = \cup_{i \in (0, n)} (TR(S_i))$ .

**Sequential Workflow.** If, for any two task executions  $i, j$  with  $(\tau_i, \delta_i), (\tau_j, \delta_j) \in TS$  we have  $(\tau_i + \delta_i) \leq \tau_j$  or  $(\tau_j + \delta_j) \leq \tau_i$ , (i.e.  $(\tau_i, \tau_i + \delta_i) \cap (\tau_j, \tau_j + \delta_j) = \emptyset$ ) then the workflow effectively behaves sequentially<sup>4</sup>. In this case we define the workflow critical path  $c$  as an ordered (by start times  $\tau_i$ ) sequence of  $(S_i, \tau_i)$  pairs such that: for any two pairs  $(S_i, \tau_i), (S_j, \tau_j) \in c$  there exists a transition  $(S_i, S_j) \in TR(S_i, \tau_j)$  and  $(S_0, \tau_0), (S_n, \tau_n) \in c$ .

**Parallel Workflow.** For each execution instance of a parallel workflow (i.e. where there exist  $i, j \in (0, n)$  such that  $(\tau_i, \tau_i + \delta_i) \cap (\tau_j, \tau_j + \delta_j) \neq \emptyset$ ) we define the critical path  $cp$  as an ordered sequence of  $(S_i, \tau_i)$  pairs, by construction as described in Figure 1.4 (b).  $conj\_trans(SYNC_A)$  is defined as follows. If  $SYNC_A = (XA \vee YA) \wedge (ZA \vee WA) \wedge \dots$  (normal conjunctive form on truth-values of incoming transitions), then for each pair of expressions  $(XA \vee YA)$ ,  $conj\_trans(SYNC_A)$  contains the one expression corresponding to the “faster” firing transition (the expression becomes true faster). For example if  $(\tau_X + \delta_X) < (\tau_Y + \delta_Y)$ , then  $XA$  fires first (if  $XA \in TR(X, (\tau_X + \delta_X))$ ) and  $XA$  is included in  $conj\_trans(SYNC_A)$ .

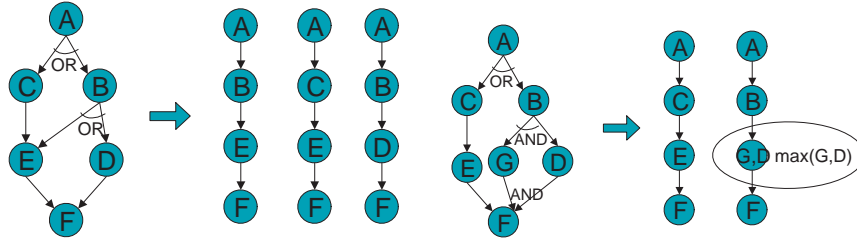


**Fig. 1.4.** (a) Incoming and outgoing task transitions (for task C) (b) Critical path construction algorithm

Thus, by construction, a critical path (see Figures 1.5 (a), 1.5 (b)) aims to capture the true workflow service bottlenecks from a parallel execution perspective. At each step we select the transition corresponding to the task that is the “latest” to finish among the transitions *required* for the continuation of the execution. As will be seen in Section 1.5.2 inter-thread asynchronous messaging can also be naturally expressed by critical path semantics.

<sup>4</sup> Its (potentially) parallel behavior is “masked” by the non-“overlapping” nature (in terms of execution) of all participating tasks.





**Fig. 1.5.** (a) Potential critical paths for a workflow behaving sequentially. (b) Potential critical paths for a parallel workflow.

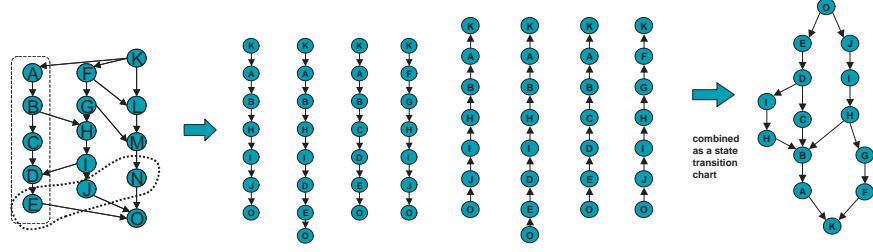
### 1.3.3 Critical Graph

Minimizing a given workflow instance response time is naturally equivalent to minimizing the execution time of its corresponding critical path. Optimizing workflow execution over multiple instances however, is a more challenging task, especially considering history and input data-dependent transitions. In the following we devise a method that aims to accurately and adaptively model average future workflow response time behavior using previously seen multiple execution instances. This model is then to be used in recommending tasks for resource re-allocations with the final goal to optimize global response time.

Let there be a set of  $m$  observed past workflow execution instances and their corresponding critical paths  $C = \{c_1, \dots, c_m\}$ . For each critical path  $c_i \in C$ , let  $t_i$  be the starting time of its initial task ( $S_0$ ). Because all the critical paths are ultimately composed of subsets of the original workflow graph (and contain by nature at least the initial ( $S_0$ ) and final ( $S_n$ ) tasks), intuitively we “group” them into a *critical graph*  $g(t)$ <sup>5</sup> by colluding all common tasks of each critical path into a graph “node” as shown in Figure 1.6 (b). Each such node  $S_i$  corresponds to an observed critical path task. Because of the collusion effect, each  $S_i$  is naturally augmented with a set of timestamps ( $TS(S_i)$ ) corresponding to the  $\tau$ -value on each critical path where the associated task appeared. Some nodes in the critical graph are going to be more “hot/active” than others. For example in Figures 1.6 (a) and (b) task  $B$  is occurring more frequent compared to  $D$  for example. Thus, intuitively, in order to optimize overall performance it could be helpful to rank the tasks accordingly and then use this ranking to recommend appropriate allocation of resources.

The first idea that comes to mind is the use of a stochastic Markov model to augment the critical graph and build a probabilistic weight of each occurring graph state (task) according to its occurrence frequency. Thus, for each task  $S_i \in g$ , we count its number of observed executions  $|TS(S_i)|$ , normalize over all tasks in  $g$  and use this as a frequency weight:

<sup>5</sup> Because of its dynamic nature, the critical graph will “look different” at different points in time, hence the  $g(t)$  notation. For simplicity reasons in the following we are using  $g$  instead to mean the graph at the current time.



**Fig. 1.6.** (a) Four sample observed critical paths in different execution instances of the given workflow. (b) Composing multiple observed critical paths into a critical graph.

$$w(S_i) = \frac{|TS(S_i)|}{\sum_{S_j \in g} |TS(S_j)|} \quad (1.1)$$

There are several drawbacks to this formula. Because of its static nature, it does not adapt to workflow behavior changes easily. Hot current tasks are being potentially out-weighted by hot tasks in the far past. To fix this we propose an idea similar to the memory-aware Markov model described in [9], namely the use of time-locality in the weight computation. For each task  $S_i$ , we can determine its occurrence time from the collected statistics available in  $TS(S_i)$ . Let  $times(S_i) = \{\tau | \forall \delta, (\tau, \delta) \in TS(S_i)\}$ . Thus, if  $t$  is the current time, (1.1) becomes <sup>6</sup> (see also [9]):

$$w(S_i) = \frac{1}{\sum_{S_j \in g} |TS(S_j)|} \times \sum_{\tau \in times(S_i)} e^{-(t-\tau)} \quad (1.2)$$

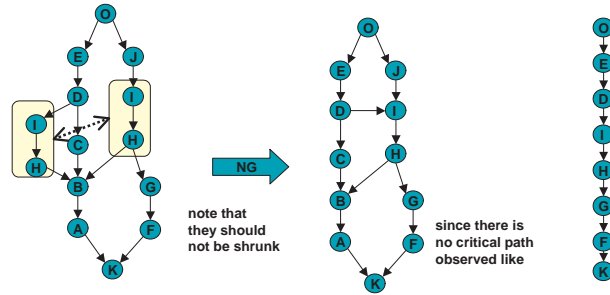
Equation (1.2) does a far better job in adapting to workflow changes by its inherent time-locality awareness. Nevertheless, because of the data-dependent nature of workflow tasks, there arises another problem, namely the ability to handle different execution times for different observed instances of a given task. That is, task  $S_i$  can have a varying impact on the resulting workflow performance if each of its execution instances behave differently. For example, instances of a task that were quite frequent but fast executing in the far past might be less important overall than more recent instances of the same task that are not as frequent but much slower executing. We propose to somehow consider also observed execution times in the computation of the task's weight. Thus (1.2) becomes:

$$w(S_i) = \frac{1}{\sum_{S_j \in g} |TS(S_j)|} \times \sum_{(\tau, \delta) \in TS(S_i)} \delta e^{-\alpha(t-\tau)} \quad (1.3)$$

Equation (1.3) associates a weight to a task, by considering both its time locality as well as its execution duration. A recent task is weighted more than an older one. A hot task is also weighted more than one less frequently occurring. A slow executing task

<sup>6</sup> For illustrative purposes, this is not normalized to 1.0 but can be easily brought to a normalized form if required.

is also assigned a higher weight than a fast executing one. The global parameters  $\alpha \in (0, 1]$  is defining the importance of the start-time and execution duration respectively in one task’s weight. There is a trade-off to be observed between a prevalent time-locality aware behavior and a more execution time dependent weighting. See also Section 1.6.



**Fig. 1.7.** Pruning the critical graph for efficiency is not trivial. Care should be taken to not introduce additional incorrect critical paths in the result (e.g.,  $O - E - D - I - H - J - F - K$ ).

It is worth noting that, apparently, for efficient parsing and storage reasons, the critical graph seems to allow an additional pruning/canonical reduction step meant to reduce its size while maintaining its entire transition structure. For example, in the graph in Figure 1.7, apparently the transition from  $I$  to  $H$ , occurring multiple times, can be “collapsed” into one single instance. If such optimization techniques are to be deployed however, care should be taken to not introduce additional incorrect critical paths in the resulting “optimized graph”. In this particular case, the proposed optimization is not sound, in that it introduces a critical path ( $O - E - D - I - H - J - F - K$ ) that was never observed in the initial construction process.

### 1.4 Resource Allocation

The weights defined by equation (1.3) define an ordering on a set of tasks, with the recent, hot, slow executing tasks being weighted higher. This ordering features a certain degree of adaptability, also tunable from the  $\alpha$  parameter. As tasks become slower and/or less frequent the weights naturally adjust. Intuitively, these weights provide an “importance” metric for executing tasks, that could be used in the resource (re)allocation process to speed up the global workflow execution. Thus, the next step in the optimization process sorts the tasks appearing in the critical graph according to their weight and proposes a weight-proportional (re)allocation of additional resources to the higher-weighted ones. This aims to reduce the task execution time and thus the global workflow execution time (as the weight models the importance of this task in the global workflow).

### 1.4.1 Resource Model

Let us model the resource allocation process in more detail. Let there be the notion of a resource (e.g. CPU time) that can be quantified and allocated to each workflow composing task. Let  $r_i$  be the allocated amount for considered task  $S_i$ . Intuitively, in many scenarios there (arguably) exists a direct dependency between  $r_i$  and the task execution time  $\delta_i$ . The more resources a task receives, the less time it should take executing. For illustrative purposes let this dependency be (see Figure 1.8 (b)) increasing and such that there exists a certain point,  $\min(\delta_i)$ , below which the execution time of  $S_i$  does not improve, even if more resources (i.e. than  $\max(r_i)$ ) are allocated. Also, let  $\min(r_i)$  be the minimal amount of resources that  $S_i$  can function with. Thus, for any task  $S_i$  we have  $r_i \in (\min(r_i), \max(r_i))$ .

Given this model, let us construct a resource allocation algorithm that would optimally handle a fixed amount of total resources  $r$  to a given set of tasks. For simplicity purposes, let us make yet another assumption. Once a workflow starts executing, each task has continuously allocated an amount of resources of at least  $\min(r_i)$  (we do not model out-swapping). Thus  $r > \sum_i \min(r_i)$ .

If  $r > \sum_i \max(r_i)$  each task could be allocated a maximum required amount of resources, thus functioning at its best potential. In this over-provisioning scenario, resource allocation can not improve workflow execution time. Thus, we are mainly concerned with the case when

$$r \in (\sum_i \min(r_i), \sum_i \max(r_i))$$

This is the case where all tasks are functioning (with minimal resources  $\min(r_i)$ ), but resource allocation can improve execution times for the highest weighted tasks.

### 1.4.2 Algorithm

We propose a resource allocation algorithm (see Figure 1.8 (a)) as a continuously running separate thread with control ability over resources and the execution environment of the composing workflow tasks (i.e. webservices).

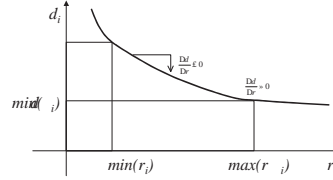
The algorithm starts by assessing the “distance” between the current and the previously (at the time of the last allocation) computed task weights. If this difference exceeds a certain threshold ( $(\Delta_W(W_A, W_B) > \theta)$ ), resource reallocation is initiated. Tasks are allocated the remaining resources, according to their respective weights.  $\Delta r_i(x) = \frac{w_i}{\sum w_j} \times x$ , represents the chunk to be allocated to task  $S_i$  out of the total remaining resource amount  $x$ . The thread then sleeps for a certain amount of time before resuming execution<sup>7</sup>.

**Increased Adaptivity.** What if  $\Delta_W(W_A, W_B) > \theta$  is too restrictive for the given workflow and will never be satisfied? In other words, what if the task weights vary

<sup>7</sup> Alternately, this delay between allocation evaluations could be replaced by, or combined with an active notification mechanism activated by changes in the workflow response time for example

```

WA ← null
while (true) do
  WB ← compute weights according to (1.3)
  sort WB
  if (ΔW(WA, WB) > θ) then
    res ← (r - ∑j min(rj))
    forall wi ∈ WB (in sorted order) do
      ri ← min((min(ri) + Δri(res)), max(ri))
      res ← res - (ri - min(ri))
  WA ← WB
  sleep delay
    
```



**Fig. 1.8.** (a) Resource allocation algorithm (running thread). (b) Proposed task execution time – resource dependency.

but to a degree that is insufficient to exceed the threshold. While the threshold value was introduced to fight the scenario of continuous re-allocations without much benefit, workflow behavior is hardly possible to bound ahead of time. It might be useful to “force” a re-allocation to be performed if too much time has elapsed since the last one. This idea can be coupled with the threshold mechanism in a natural way by changing the re(allocation) triggering condition into:

$$\Delta_W(W_A, W_B) + \varphi \Delta t > \theta$$

where  $\Delta t$  is the time elapsed since the last re(allocation) operation and  $\varphi \in (0, 1)$ .

## 1.5 Discussion

### 1.5.1 Allocation Stability

The resource allocation algorithm (Figure 1.8) suffers from a potential instability problem. If some resources get reallocated from a task  $A$  to a task  $B$  because currently,  $w(A) < w(B)$ , it is likely that in the next execution instance,  $\delta_A$  is going to increase (as less resources are available to  $A$ ). This in turn will yield an increase in  $w(A)$ . This might result in  $w(A) > w(B)$ , thus prompting resource allocations in the opposite direction, from  $B$  to  $A$ .

The effect of this instability is controllable however. Reallocation only happens when the distance between the weight vectors exceeds  $\theta$ . Thus, small changes within individual weights might not be of impact right-away. Also, further fine-tuning can be achieved by choosing an appropriate  $\beta$  (lowering the impact of execution time in the weight computation).

The algorithm can be modified to provide a complete solution to this issue as follows. Keep a history for each task  $S_i$ , remembering its two past differing associated weights (for past reallocation cycles)  $w^1(S_i)$ ,  $w^2(S_i)$ .  $w^2(S_i)$  is the current

weight of the task. Lets assume that  $w^1(S_i) > w^2(S_i)$  (the task was allocated more resources as a result of  $w^1(S_i)$  and then its weight dropped, possibly because of an associated speedup). Let  $w^3(S_i)$  be the future weight that would be associated with this task if resources would be (de)allocated according to the current weight  $w^2(S_i)$ . The future weight can be estimated by approximating  $\delta_i$  from the past execution times in  $TS(S_i)$  and by assuming no new task execution happens before the next reallocation. If  $w^2(S_i) < w^3(S_i) \approx w^1(S_i)$  (close to the original weight) the de-allocation would obviously result in oscillation; the algorithm should NOT de-allocate any resources that were allocated to  $S_i$  but rather ignore it and proceed to the next task, by reallocating the remaining resources.

In other words, we check if the weight of a task went down because of its increased speed (after allocation). If the future weight of the task is going to be back up (similar to the old weight) this might result in instability, thus needs to be avoided. We argue that this (or a similar) heuristic can significantly reduce or even completely eliminate oscillations and instability. Additionally, a variation of the algorithm could be to only de-allocate an upper-bounded fraction ( $\eta$ ) of the resources of a given task. Considering an assumed relationship between task execution time and allocated resources, this ensures that the task slowdown can be upper bounded, thus the weight variation due to reallocation, limited.

### 1.5.2 Asynchronous Messaging

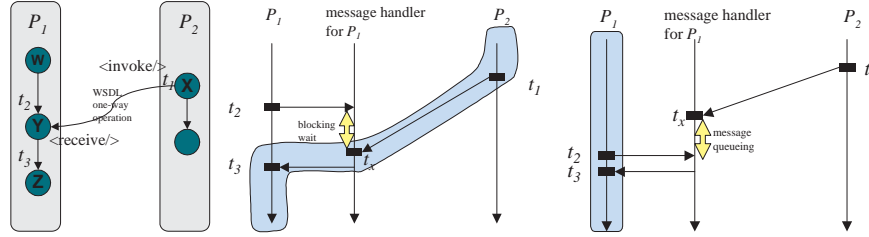
Asynchronous communication between parallel tasks (see Figure 1.9 (a)) needs to be considered when optimizing overall workflow execution times because the messaging constructs and semantics have a direct impact on these times. Fortunately, it turns out that asynchronous messaging can be naturally handled within the critical paths framework with a minimal set of associated algorithm extensions.

For each asynchronous communication instance between two parallel processes (sequences of tasks)  $P_1$  and  $P_2$ , there exists a *invoke* ( $P_1 \rightarrow P_2$ ) and a *receive* ( $P_1 \leftarrow P_2$ ) semantic link. Because of the nature of asynchronous messaging and the BPEL business flows, from the execution flow and times point of view, these links behave effectively like transitions between tasks in  $P_1$  and  $P_2$ . Thus, our main extension to the algorithm involves the critical path construction mechanism in Section 1.3.2. Instead of considering only the explicit transitions in  $\mathbb{T}$  we augment this set with these links and treat them as transitions in the rest of the algorithm. The message initiation via *invoke* is virtually instantaneous (message goes in  $P_2$ 's incoming mailbox), it can be ignored for simplicity. Thus let  $\mathbb{T}^* = \mathbb{T} \cup \text{receive\_links}$ , where *receive\_links* is the set of all pairs of tasks  $(X, Y)$  like in Figure 1.9 (a), where  $Y$  is a receive activity and  $X$  initiates a reply message for  $Y$ .

In the following let us analyze the extended critical path construction in the case of two parallel processes. There are two possible asynchronous messaging scenarios. In the first case (Figure 1.9 (b)), process  $P_1$  initiates an asynchronous message exchange with  $P_2$ , continues executing and eventually blocks waiting for a reply. In other words, the reply arrives *after* (at time  $t_x > t_2$ ) it is expected. In this case, intu-

itively, the workflow execution is impacted by  $t_x$ , thus by the reply activity/task  $X$ . The critical path should thus contain  $\{(W, \tau_W), (X, t_1), (Z, t_3)\}$ .

In the second scenario, the reply arrives *before* (at time  $t_x < t_2$ ) it is expected in  $P_1$  and is queued in  $P_1$ 's message handler. Thus, in this case, the workflow execution is not impacted at all by the reply and process  $P_1$  continues un-interrupted with activity  $Z$ . The critical path construction mechanism should result in a path containing  $\{(W, \tau_W), (Z, t_3)\}$  ( $\tau_W$  not figured).



**Fig. 1.9.** (a) Asynchronous messaging scenario between two workflow processes/threads. (b) Reply arrives *after* it is looked for. Process 1 blocks waiting. The current workflow instance critical path naturally includes the *invoke* activity in process 2 and the blocking *receive* in process 1. (c) Reply arrives *before* it is looked for and becomes queued. Process 1 continues virtually un-interrupted ( $(t_3 - t_2) \approx 0$ ) upon executing *receive*. The critical path includes just tasks within process 1 (as process 2 is faster and “keeps up”, thus not impacting overall execution time).

### 1.5.3 Limited Storage

In Section 1.3.3 a certain assumption was underlying the definition of task weights. A virtually infinite amount of memory was required to keep one task’s execution history in  $TS(S_i)$ . This is sometimes not desirable.

This can be solved by observing that equation (1.3) is suitable for a natural incremental computation. If we change the notation such that  $w(t)$  denotes the weight of task  $S_i$  as computed at current time  $t$ , then we have a dependency between  $w(t)$  and  $w(t + \Delta t)$  for any  $\Delta t > 0$  as follows:

$$w(t + \Delta t) = \frac{1}{s_{t+\Delta t}} \times [e^{-\alpha\Delta t} s_t w(t) + \sum_{(\tau, \delta) \in TS(S_i) \wedge \tau > t + \Delta t} \delta e^{-\alpha(t - \tau + \Delta t)}]$$

where  $s_t = \sum_{S_j \in g} |TS(S_j)|$  such that  $\forall (\tau, \delta) \in TS(S_i), \tau < t$ . In effect, this reformulation shows the natural ability to compute the task weight dynamically out of a previous task weight incrementally, thus requiring virtually no additional storage.

In practice, when given a limited storage window for each task, another solution is easier to implement and identical from a theoretical point of view. Instead of the

exponential weight in (1.3), which often presents representation accuracy issues, use the following linear version of the weight definition:

$$w(S_i) = \sum_{(\tau, \delta) \in TS(S_i)} \frac{\delta}{t - \tau} \quad (1.4)$$

Then, compute a task weight after seeing a certain fraction of  $m$  (e.g.  $\frac{m}{2}$ ) worth of task instances and store that weight  $w^1(S_i)$  instead of the seen instance stats. Repeat. This process will result in a set of task weights (e.g. one for each  $\frac{m}{2}$  instances seen) that represent a summarized view of the execution history of  $S_i$ . There is only a fixed number of these weights that we remember, and the “older” ones are gradually discarded.

When computing the current task weight, we then consider all the actual time stamps in the remaining window and compute a weight  $w^2(S_i)$ . Then the current weight is going to be defined by

$$w(S_i) = \chi w^1(S_i) + (1 - \chi)w^2(S_i)$$

, where  $\chi \in (0, 1)$ .  $w^1(S_i)$  is then replaced by  $w^2(S_i)$  and the process continues. Depending on how much memory is available, for more accuracy, this mechanism can be extended by storing multiple weights and having multiple  $\chi$  values in computing the final weight.

Yet another optimization that can be deployed derives from the fact that it is likely that some tasks are going to be more active than others. If stats keeping memory is shared, storage can be distributed according to the estimated/previous frequency of each task, such that statistics about very active tasks are going to be more accurate than for less active ones.

## 1.6 Experiments

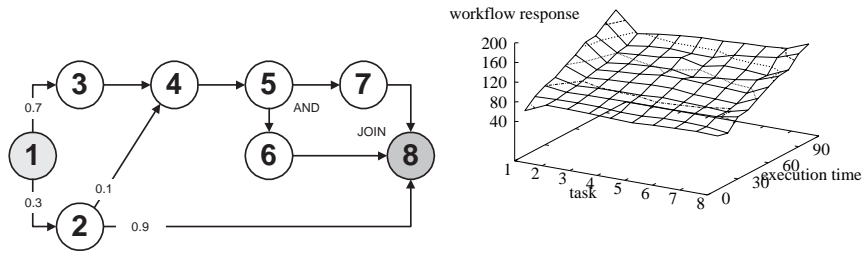
We implemented a proof-of-concept package (wfx.\*) of the workflow modeling and resource allocation algorithms presented. Our implementation allows the definition, virtual execution, monitoring and resource allocation for arbitrary workflows and associated tasks. We also implemented experimental versions of our wrapper technology and tested it with the IBM BPWS4J Engine.

We performed experiments and analyzed the behavior of various classes of workflows. For illustration purposes here we discuss a simple minimal

Here we are discussing a simple, yet illustrative workflow depicted in Figure 1.10 (a). This workflow is simple enough to be easy to comprehend and is composed of an initial traditional stochastic flow (that can be handled by simple stochastic modeling [2]) followed by concurrent execution of tasks 6 and 7 (they start simultaneously, upon completion of 5) and a conjunctive join condition<sup>8</sup> for task 8.

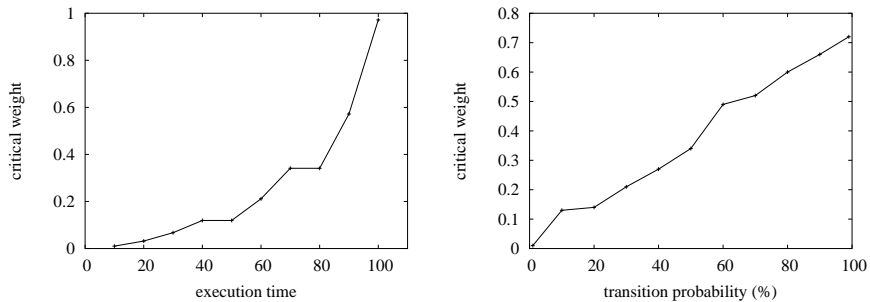
<sup>8</sup> Handling conjunctive joins is one of the important novel use scenarios that our solution handles.





**Fig. 1.10.** (a) Simple parallel workflow. Transitions 5-7 and 5-6 are triggered simultaneously. (b) Average workflow response time as a function of individual service times for the workflow in (a). It can be seen (as intuitively expected) that tasks 1 and 8 have a higher impact on the workflow execution (they are executed every time).

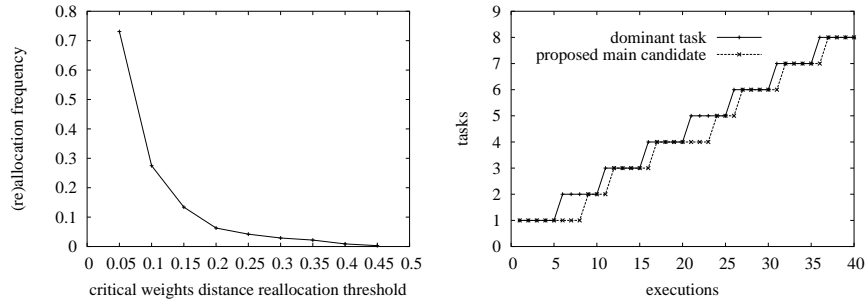
To assess, the initial impact of each task in the global execution times, in Figure 1.10 (b) the overall workflow response time is depicted as a function of the individual task execution times. It can be seen that (as can be inferred from the workflow schema), task 1 seems to correlate stronger (steeper incline) with the global workflow times, than task 2 for example. Task 2 is only triggered 30% of the time which would explain its lower observed weight in the overall times.



**Fig. 1.11.** (a) The critical weights accurately follow increases in task execution times, prompting additional resource allocation. Pictured is task 4 in the workflow in Figure 1.10 (a). (b) As the transition probability increases (for task 2 in Figure 1.10 (a)), the associated critical weight follows almost linearly.

Validating the critical weights is of significant importance if the allocation mechanism is to function properly and recommend appropriate resource allocations. In the experiment depicted in Figure 1.11 (a) gradually increasing task execution times naturally result in higher weights, prompting the allocation of additional resources.

In Figure 1.11 (b) we can see how higher transition probabilities (e.g. for task 2) yield almost linearly following values in the resulting weights. This shows that there exists a natural direct relationship between the weighting process (and additional resource allocation) and the task execution frequency. This in effect ties our work with the work in [2] and shows that it also correctly handles stochastic aspects.



**Fig. 1.12.** (a) The stability and frequency of the resource (re)allocation process can be controlled by adjusting the critical weights (re)allocation threshold  $\theta$ . As it relaxes (increases), the (re)allocation frequency drops. (b) As the dominant task in the workflow changes (e.g. from 1 to 8), the resource allocation mechanism closely follows with a minimal latency delay (1-2 workflow executions). This shows the adaptivity to changes in workflow behavior.

An experiment exploring the stability of the resource allocation process is illustrated in Figure 1.12 (a). Stable allocation schemes put less loads on resource managers and also likely minimize resource shifting overheads (not considered here). It can be seen how  $\theta$  can be used as a fine-tuning “knob” balancing this trade-off between high (re)allocation frequency and stability.

The experiment illustrated in Figure 1.12 (b) analyzes the adaptivity of the resource allocation mechanism. This is important to allow for quick turn-around times and stability in the resource allocation mechanisms. Depicted are both the actual dominant task (longest execution time) in various workflow instances and the proposed main additional resource re-allocation candidate. It can be seen that the algorithm adapts quickly to changes in the workflow behavior, with a minimal “following delay” (1-2 workflow instances). This is great news, in that it guarantees that resource allocation converges quickly and improves execution times accordingly.

## 1.7 Related Work

Existing work can be categorized into two parts: (1) QoS-aware service composition; (2) adaptive service composition. QoS-aware service composition has been extensively studied. Existing approaches [10, 11, 12, 13, 14, 15, 16, 17] consider mainly performance at composition time, whereas in our work we optimize performance

at runtime. Moreover, various mechanisms [18, 19, 20, 21] have been proposed to endow a web service workflow with adaptivity to its runtime environment. These proposals mainly focus on how to update a workflow at runtime in a non-intrusive way. Our runtime optimization can be introduced on top of such schemes.

## 1.8 Conclusions

In this paper we proposed a “dynamic” stochastic workflow modeling approach suited to handle workflows with concurrent paths. We then explored this approach experimentally through the design and proof of concept implementation of a novel monitoring component which transparently gathers execution statistics about the composing workflow tasks (web services). These statistics are then used in the resource (re)allocation algorithm to optimize overall workflow performance.

Various issues remain to be explored. Different (potentially composite) quality metrics and alternative wrapping solutions, minimizing overheads could be considered. It might be desirable to augment the solution with effective correlation sets handling. Additionally, deployment of such components in mainstream industry resource managers should be investigated and pursued.

## References

1. Sion, R., Tatemura, J.: Dynamic Stochastic Models for Workflow Response Optimization. In: Proceedings of the International Conference on Web Services ICWS. (2005)
2. Gillmann, M., Weikum, G., Wonner, W.: Workflow management with service quality guarantees. In: Proceedings of the 2002 ACM SIGMOD international conference on Management of data, ACM Press (2002) 228–239
3. Broberg, M., Lundberg, L., Grahn, H.: Performance optimization using extended critical path analysis in multithreaded programs on multiprocessors. *Journal of Parallel and Distributed Computing* **61**(1) (2001) 115–136
4. BEA, IBM, Microsoft: BPEL4WS: Business Process Execution Language for Web Services Version 1.1 (2003)
5. W3C: Web Services Definition Language (WSDL) 1.1 (2001)
6. Gillmann, M., Weissenfels, J., Weikum, G., Kraiss, A.: Performance and availability assessment for the configuration of distributed workflow management systems. In: Proceedings of EDBT. (2000)
7. Son, J.H., Kim, M.H.: Analyzing the critical path for the well-formed workflow schema. In: Proceedings of the 2001 DASFAA, IEEE (2001)
8. Lawrence, P.: *Workflow Handbook*. John Wiley and Sons (1997)
9. Sion, R., Atallah, M., Prabhakar, S.: On-the-fly intrusion detection for web portals. In: Proceedings of IEEE ITCC 2003. (2003)
10. Alrifai, M., Risse, T.: Combining global optimization with local selection for efficient qoS-aware service composition. In Quemada, J., León, G., Maarek, Y.S., Nejd, W., eds.: *WWW, ACM* (2009) 881–890
11. Mukherjee, D., Jalote, P., Nanda, M.G.: Determining qoS of WS-BPEL compositions. In Bouguettaya, A., Krüger, I., Margaria, T., eds.: *ICSOC. Volume 5364 of Lecture Notes in Computer Science*. (2008) 378–393

12. Baligand, F., Rivierre, N., Ledoux, T.: A declarative approach for qoS-aware web service compositions. In Krämer, B.J., Lin, K.J., Narasimhan, P., eds.: ICSOC. Volume 4749 of Lecture Notes in Computer Science., Springer (2007) 422–428
13. Sato, N., Trivedi, K.S.: Stochastic modeling of composite web services for closed-form analysis of their performance and reliability bottlenecks. In Krämer, B.J., Lin, K.J., Narasimhan, P., eds.: ICSOC. Volume 4749 of Lecture Notes in Computer Science., Springer (2007) 107–118
14. Cardellini, V., Casalicchio, E., Grassi, V., Presti, F.L.: Flow-based service selection for web service composition supporting multiple qoS classes. In: ICWS, IEEE Computer Society (2007) 743–750
15. Li, Y., Huai, J., Deng, T., Sun, H., Guo, H., Du, Z.: QoS-aware service composition in service overlay networks. In: ICWS, IEEE Computer Society (2007) 703–710
16. Nguyen, X.T., Kowalczyk, R., Phan, M.T.: Modelling and solving qoS composition problem using fuzzy disCSP. In: ICWS, IEEE Computer Society (2006) 55–62
17. Berbner, R., Spahn, M., Repp, N., Heckmann, O., Steinmetz, R.: Heuristics for qoS-aware web service composition. In: ICWS, IEEE Computer Society (2006) 72–82
18. Moser, O., Rosenberg, F., Dustdar, S.: Non-intrusive monitoring and service adaptation for WS-BPEL. In Huai, J., Chen, R., Hon, H.W., Liu, Y., Ma, W.Y., Tomkins, A., Zhang, X., eds.: WWW, ACM (2008) 815–824
19. Harney, J., Doshi, P.: Speeding up adaptation of web service compositions using expiration times. In Williamson, C.L., Zurko, M.E., Patel-Schneider, P.F., Shenoy, P.J., eds.: WWW, ACM (2007) 1023–1032
20. He, Q., Yan, J., Jin, H., Yang, Y.: Adaptation of web service composition based on workflow patterns. In Bouguettaya, A., Krüger, I., Margaria, T., eds.: ICSOC. Volume 5364 of Lecture Notes in Computer Science. (2008) 22–37
21. Mosincat, A., Binder, W.: Transparent runtime adaptability for BPEL processes. In Bouguettaya, A., Krüger, I., Margaria, T., eds.: ICSOC. Volume 5364 of Lecture Notes in Computer Science. (2008) 241–255