Reducing Escalation-Related Costs in WFMSs

Euthimios Panagos and **Michael Rabinovich**
AT&T Labs - Research
{thimios, misha}@research.att.com

Abstract

Escalations refer to the actions taken when workflow activities, which constitute business processes, miss their deadlines. Typically, escalations increase the cost of business processes due to the execution of additional activities, the compensation of finished activities, or the intervention of highly-paid workers. In this paper, we present on-going research addressing workflow escalations and the cost associated with them. Our goal is twofold: (a) minimize the number of escalations during process execution and (b) reduce the cost associated with escalations when they cannot be avoided. To reduce the number of executions that result in escalations, we propose to keep track of the slack time accumulated during process executions and use it to adjust the deadlines (i.e., delay escalations) of the remaining activities. To reduce the cost of inevitable escalations, we describe an approach for predicting whether a process is going to escalate at some future point and, then, deciding whether and when to *force* escalation at an early stage of the process execution. Escalating early is often cost-effective because it allows more time for remedial actions and reduces the work and resources wasted for finished activities that would have to be compensated. Preliminary experimental results show the effectiveness of our techniques.

1 Introduction

Today, organizations use workflow management systems (WFMSs) to streamline, automate, and manage business processes that depend on information systems and human resources (e.g., provisioning telephone services, processing insurance claims, and handling bank loan applications). WFMSs provide tools to support the modeling of business processes at a conceptual level, coordinate the execution of the component activities according to the model, monitor the execution progress, and report various statistics about the business processes and the resources involved in their enactment [GHS95, KS95].

A workflow is an abstraction of a business process. It consists of *activities*, which correspond to individual process steps, and *agents*, which execute these activities. An agent may be an information system (e.g., a database system), a human (e.g., a customer representative) or a combination of both (e.g., a human using a software program). The workflow specification may also specify execution durations, referred to as *deadlines*, for both the activities and the process. When an activity misses its deadline, special actions, referred to as *escalations*, may be triggered. Escalations increase the operational cost of a business process since, otherwise, they would have been specified as part of the normal execution path.

Escalations may have local or global scope. A local escalation affects the triggering activity (i.e., the one that triggered the escalation), and it may: (a) restart the triggering activity, (b) execute a new activity and then resume the execution of the triggering activity, or (c) replace the triggering activity with a new activity. A global escalation affects the whole business process, and it may: (d) abort the process and compensate all finished activities, or (e) stop the normal flow of execution, compensate some of the finished activities, and execute an alternative sequence of activities to complete the process. In all cases, invoking escalation results in an increased cost for a business process due to the additional activities that may have to be executed, or because completed work is rolled back, or because intervention of highly-paid workers is required.

In this paper, we concentrate on ways to reduce operational costs due to escalations. Of course, the most obvious way to achieve this is by assigning very long deadlines or increasing the available resources (e.g., the number of agents), to ensure that escalations are rare events. However, this may increase the operational costs during the execution of normal, non-escalated, processes and may not be always possible. Therefore, our goal is to try to reduce the escalation-related costs by better management of business process during their execution by a WFMS, assuming that business processes and the allocated resources are decisions outside our control. In particular, our approach combines two inter-related mechanisms: *dynamic deadline adjustment* and *early escalations*.

The dynamic deadline adjustment mechanism, initially outlined in our position paper [PR96], reduces the number of escalations during the execution of workflow processes by keeping track of any available slack time accumulated during a partial process execution and using this slack to extend the deadlines (i.e., delay escalation) of the remaining activities. Various algorithms for distributing the slack are possible. The main observation behind the deadline adjustment mechanism is that deadlines are usually assigned to activities based on the *estimated* execution times of these activities, the need to meet the overall business process deadline, and to ensure that corrective measures (i.e., escalations) are invoked in a timely manner. On the other hand, the *actual* execution times of activities vary from one instance to the next due to variations of load, work conditions, etc. Therefore, if an activity in a given workflow execution finishes faster than its estimated execution time, the remaining activities can be given extra time before escalation is invoked.

The early escalations mechanism attempts to reduce the escalation cost when escalations cannot be avoided. It is based on the observation that, when escalations are inevitable (e.g., when the resources needed for the enactment of an activity are not available for an extended time period), it may be beneficial to escalate during the execution of some *earlier* activity, which has not missed its deadline yet. Indeed, early escalations leave more time for contingency plans, resulting in reduced operational costs. Often, escalations involve the compensation of finished activities and, hence, invoking them early entails less work for executing and then compensating activities that are rolled back, and fewer wasted resources.

Based on the above observation, the early escalations mechanism attempts to predict whether a process is going to escalate at some future point and, then, decide whether and when to force escalations before they actually occur. Our approach exploits information about escalation costs, which are supplied by the business analyst, as well as information provided by existing workflow systems, such as statistical measurements based on past process executions and current status data (e.g., agent load and availability).

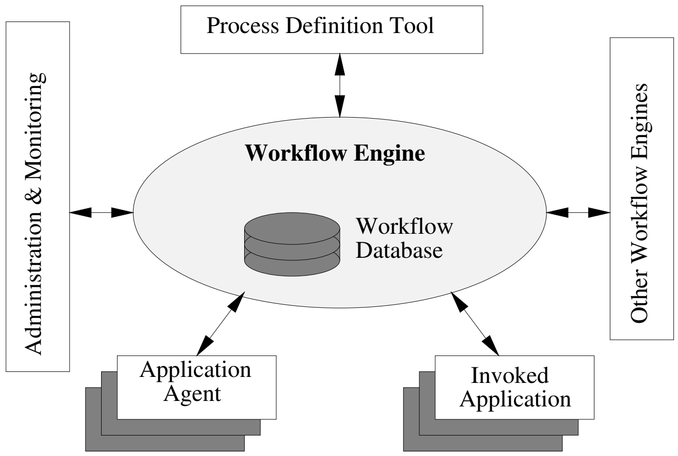

Figure 1: Workflow Management Coalition: Reference Architecture

Our preliminary experiments show that our approach reduces escalation costs. This is achieved without any modification to the process models or increase of allocated resources and, thus, it is completely transparent to the users of the workflow system. A limitation of our approach is that it assumes that, during the execution of a process instance, its process model stays the same. We thus exclude *ad hoc workflows* [GHS95] where the schema of the process for a given process instance is altered during the instance execution. Instead, our approach targets *production workflows* systems, where escalation-related costs constitute a significant amount and an important concern. Our approach can also be applied to *administrative workflows* to the extend they do not dynamically change workflow specifications of workflow instances in progress.

The remainder of the paper is organized as follows. Section 2 offers an introduction to workflow concepts and describes the workflow process model we use in this paper. Section 3 presents our dynamic deadline adjustment algorithms. Section 4 describes our basic algorithm for invoking early escalations. Section 5 presents preliminary experimental results. Section 6 compares our work with related work and, finally, Section 7 concludes our presentation.

2 System Model

Figure 1 shows the WFMS reference model provided by the Workflow Management Coalition (WfMC) [Gro94]. The WFMS consist of an engine, application agents, invoked applications, a process definition tool, and administration and monitoring tools. The process definition tool is a visual editor used to define the specification, i.e., the *schema*, of a workflow process. The same schema can be used for creating multiple *instances* of the same business process at a later time. The workflow engine and the tools communicate with a workflow database to store and update workflow-relevant data, such as workflow schemas, statistical information, and control information required to execute and monitor the active process instances.

Each application agent has an activity queue associated with it. Activities submitted for execution are inserted into the activity queue when the agent is busy. The agent follows its own local policy for selecting from its queue the next activity to execute. The most common scheduling policies are priority-based and FIFO.

Existing WFMSs maintain audit logs that keep track of information about the status of the various system components, changes to the status of workflow processes, and various statistics about past process executions. This information can be used to provide real-time status reports about the state of the system and the active workflow process instances, as well as various statistical measurements such as the average execution time of an activity belonging to the particular process schema. These statistical measurements can be used for predicting the timing characteristics of the active workflow instances.

2.1 Workflow Schema

A workflow schema specifies activities that constitute the workflow and dependencies between them. Dependencies determine the execution sequence of activities and the data flow between these activities. Activities can be executed sequentially or in parallel. Parallel executions may be unconditional, i.e., all activities are executed, or conditional, i.e., only activities that satisfy a condition are executed. In addition, activities may be executed repeatedly, and the number of iterations may be determined during the execution.

The point where control splits into multiple parallel activities is referred to as *split point*. The point where control merges into one activity is referred to as *join point*. A join point is called AND-join when the activity immediately following this point starts execution only when all the activities preceding the join point finish execution. An unconditional split, or a conditional split such that it can be statically (before execution) determined that all branches are taken is called AND-split. A split for which is can be statically determined that exactly one of the branches will be taken will be called OR-split.

Many of the existing WFMS process definition tools allow the specification of arbitrary dependencies between activities. Used without a discipline, these tools may result in schemas with a "spaghetti" of interwined precedence relationships, which is hard to debug, modify, and reason about. The situation is analogous to the state of software engineering before the invention of structured programming and top-down refinement. This problem has been recognized in the recent report on challenges in workflow management, where the problem of finding a more structured way of defining precedence relationships was posed [SGJ+96].

While we do not place any restrictions on workflow types that are allowed in the system, some of our algorithms for dynamic deadline adjustment are applicable only to "well-structured" workflows. A well-structured workflow consists of m sequential activities, $T_1 \dots T_m$. Each activity T_i is either a primitive activity, which is not decomposed any further, or a composite activity, which consists of n_i parallel conditional or unconditional sub-activities $T_i^1, \dots, T_i^{n_i}$. Each sub-activity may be, again, primitive or composite.

Thus, when a workflow instance is initiated, the system determines which DDA algorithms it can apply. An interesting side effect of this is if the same process is represented as a non-well-structured and an equivalent well-structured schema (where equivalent schemas are those that produce traces

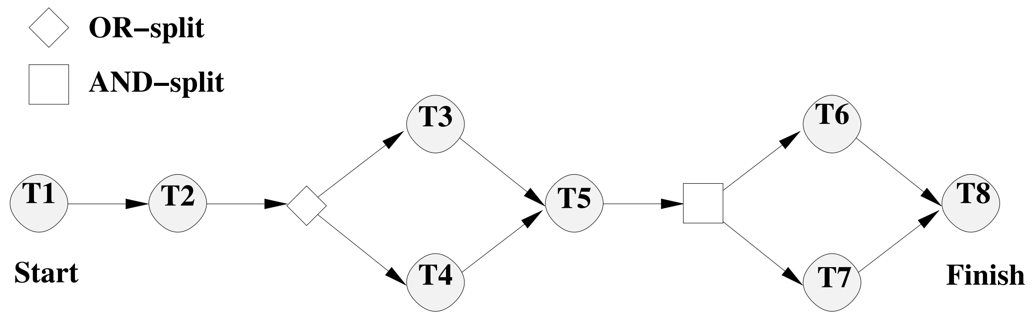

Figure 2: Service Provisioning Workflow Process

of process instances with the same chains of executed activities), executing well-structured schema may result in lower operational costs.

The schema also specifies which agents can execute each workflow activity, which is normally done by associating *roles* with activities. Roles serve as "descriptions" for human skills and software system services required for the enactment of activities. Formally, a role is a set of agents; each activity has an associated role, that determines which agents can execute this activity. The notion of a role facilitates load balancing among agents and can flexibly accommodate changes in the workforce and the computing infrastructure of an organization, by changing the set of agents associated with roles.

The following example, illustrated in Figure 2, shows a well-structured telephone service provisioning workflow that we use in the remainder of the paper.

Example: An instance of the telephone service provisioning workflow is created when an operator collects from a customer the information needed to carry out the service, such as name, address, kind of service and requested options. $T1$ is executed to verify that the information is accurate, check if the requested service can be provided, and create a service order record. $T2$ uses the service order record created by $T1$ and consults the facilities database to determine whether existing facilities can be used for establishing the service. If existing facilities can be used, $T3$ is executed. Otherwise, a human field engineer is selected to execute $T4$, which may involve manual changes to some switch and the establishment of a new telephone line. Once $T3$ or $T4$ is completed, $T5$ is executed to verify that the installation was successful. Then $T6$ and $T7$ are executed in parallel to update the telephone directory database and generate billing information, respectively. Finally, $T8$ generates some summary data about the provisioning of the new telephone service. □

2.2 Deadlines and Escalations

A workflow schema can be used to assign *deadlines* to activities and the entire process [LR94, XSo, Flo]. The schema may specify time-triggered activities, *escalations*, which are executed when the process or some of its activities miss their deadlines. For the entire process, the deadline is specified as the allowable execution time. When a process instance is initiated, this deadline is translated into an absolute time based on the starting time of the instance. For an individual activity, the deadline may be expressed as the allowable execution time, or the time (relative to the starting time of the process instance) by which the activity must complete.

In the example shown in Figure 2, deadlines specified as allowable execution times would stipulate that, e.g., activity $T1$ must complete within 3 time units, $T2$ must also finish within 3 units after it starts, etc. When defined in terms of completion times relative to the start of the process, the deadlines would specify that $T1$ must complete no later than 3 time units after the workflow instance started, $T2$ must complete no later than 6 time unites, etc.

Existing existing WFMSs provide capabilities for specifying activity deadlines in either one or both of these ways. We will show that specifying the deadline as completion time relative to process start is equivalent to one particular method of deadline adjustment. Moreover, our performance study shows that this method does not always offer the best performance. Thus, we argue that even when activity deadlines are specified relative to the process starting time, it is beneficial to internally translate each activity deadline into its maximum allowable execution time, and then choose the right algorithm to adjust the latter. From now on, we assume that deadlines correspond to allowable execution times.

2.3 Assumptions

We now summarize our assumptions about the WFMS and the information that is available about the activities constituting the business processes.

- Scheduling of the activities that are ready for execution is not based on the deadlines assigned to them.[1] To the best of our knowledge, the above assumption is true for all commercial WFMSs and existing research prototypes.

- Each activity T has a deadline, $deadline(T)$. The deadline specifies the allowable execution time for the activity, and is assumed to be specified by the business analyst when defining the schema. When only the overall process deadline is specified, techniques similar to those proposed in [KGM93a, KGM93b] can be used to assign deadlines to individual activities at runtime, based on the overall process deadline and the load on the agents. Finally, both the process and activity deadlines can be assigned dynamically based on the current load of the system [PR97].

- An estimate of the average completion time for each activity T, $avg\_completion(T)$, is available. This estimate corresponds to the time it takes the agent, which will execute the activity, to complete the activity after the activity is submitted for execution, including any queuing time at the agent's work queue. Such estimates can be extracted from the audit logs maintained by existing WFMSs.

- Each activity T is assigned an escalation cost, $escal\_cost(T)$. Typically, WFMSs allow user-defined attributes to be associated with activities and, thus, one of these attributes could be

[1]Under the *earliest deadline first* scheduling policy, the scheduling priority of a task decreases as the task's deadline increases. Consequently, our dynamic deadline adjustment algorithms may increase the number of escalations because they extend deadlines.

used for specifying escalation costs. Escalation costs are assigned by business analysts based on the effects of escalation should an activity miss its deadline.

- At any point, the system can obtain, for each agent A, the current queue length, $cur\_queue(A)$, and the average queue length, $avg\_queue(A)$, based on past history. Most existing WFMSs provide this functionality.

3 Dynamic Deadline Adjustment (DDA)

In this section, we present several algorithms that adjust dynamically the deadlines assigned to workflow activities. The algorithms use a new variable for each activity T, $effective\_deadline(T)$, which is set to $deadline(T)$ when a new workflow instance is created. An activity T escalates when its actual execution time is greater than its effective deadline.

No Adjustment (NOA): This is the default algorithm followed by existing WFMSs. Under NOA, the deadline of an activity is not adjusted at all. NOA is used as the baseline for the experimental results we present in Section 5.

Total Slack (TSL): This algorithm is implicitly followed by WFMSs that specify the deadlines of activities relative to the starting time of the workflow instances containing them [XSo, Flo]. Under this algorithm, the deadline (i.e., allowable execution time) of an activity T_k that is going to be executed next is adjusted by adding the currently available slack to it.

$$effective\_deadline(T_k) = deadline(T_k) + available\_slack$$

The value of $available\_slack$ is computed as follows. The starting activity does not receive any slack since there is no slack available at that point. After an activity T finishes, $available\_slack$ is computed as the difference between T's effective deadline and T's actual execution time. The effective deadline of each activity that can be executed immediately after T is calculated by adding $available\_slack$ to its deadline.

An important point to note is that if T_k is an AND-join, i.e., it has multiple predecessors, its effective deadline reflects the slack produced by the last completed predecessor. This is because T_k's effective deadline is calculated anew each time one of the predecessor activity finishes.

NOA and TSL algorithms can be applied to any workflow model because they do not use any information about the structure of the workflow. The algorithms we present next assume that workflows conform to the well-structured model defined in Section 2. When considering a well-structured workflow schema, it is easy to see that for any given primitive activity T, all successors of T, including T, constitute a well-structured workflow schema, W_T. Therefore, if T_k is a primitive activity that immediately follows T, W_{T_k} consists of several sequential activities $T_k^1 \dots T_k^{m_k}$, each of which is either primitive or composite.

Assuming that the $available\_slack$ is known when T_k is ready to be submitted for execution, the following algorithms can be used for computing its effective deadline.

Proportional Execution (PEX): Knowing the average completion times of the primitive activities constituting W_{T_k}, we can use them to distribute the available slack between T_k and $T_k^1 \dots T_k^{m_k}$

in proportion to their average completion times. The average completion time of a composite activity is the sum of the average completion times of the activities belonging to the critical path within this composite activity.

$$effective\_deadline(T_k) = deadline(T_k) + available\_slack * \frac{avg\_completion(T_k)}{\sum_{j=1}^{m_k} avg\_completion(T_k^j)}$$

Proportional Escalation (PES): This algorithm uses the escalation costs assigned to the primitive activities in W_{T_k} to distribute the available slack between T_k and $T_k^1 \dots T_k^{m_k}$ in proportion to their escalation costs. For composite activities, the escalation cost is set to the maximum of the escalation costs among all of their sub-activities.

$$effective\_deadline(T_k) = deadline(T_k) + available\_slack * \frac{escal\_cost(T_k)}{\sum_{j=1}^{m_k} escal\_cost(T_k^j)}$$

Proportional Load (PLO): Assuming that A_{T_k} denotes the agent that will execute T_k, PLO distributes the available slack in proportion to the loads (expressed as the average queue length) of the agents that are involved in the execution of the remaining activities. For composite activities, these loads are set to be the maximum of the average queue lengths of the agents that will execute their sub-activities. In addition, when there are multiple agents that can execute the same activity, the average load among them is chosen.

$$effective\_deadline(T_k) = deadline(T_k) + available\_slack * \frac{avg\_queue(A_{T_k})}{\sum_{j=1}^{m_k} avg\_queue(A_{T_k^j})}$$

The *available_slack* value used in the above algorithms is computed in the following way. When a workflow instance is submitted for execution, the value of the available slack is 0. When an activity T finishes execution, its slack, which is computed as the difference between T's effective deadline and T's execution time, is added to the available slack. This value becomes the available slack for every W_{T_k}, where T_k is an immediate successor of T. Then, some portion of this slack is assigned to T_k, according to the above algorithms, and the new value of the available slack for W_{T_k} is decremented by this portion.

Finally, AND-joins are handled in the same way as in the TSL algorithm. Upon completion of each predecessor of T_k, the effective deadline of T_k is re-computed, so that when T_k is ready for execution, its effective deadline has received the appropriate portion of the slack produced by the predecessor that finished last.

4 Early Escalations

The algorithms presented in the previous section aim at reducing the number of escalations. However, escalations cannot be always avoided. In such cases, if we were able to *predict* that an escalation is likely to occur during the execution of a future activity, we could use this information to reduce the operational cost of the process containing this activity. Specifically, consider workflows with global escalations. If at some point during the workflow execution we determine

```
EarlyEscalation(W,T):boolean
/* Executed when activity T of workflow W becomes ready */
    RETURN(NextStep(W,T,T);
END;

NextStep(W,T,T_c):boolean
    pred_completion(T_c) = cur_queue(T_c)/avg_queue(T_c) * avg_completion(T_c);
    IF T_c == T AND deadline(T_c) < pred_completion(T_c)
        RETURN("YES");
    ENDIF
    IF T_c ≠ T AND (1 - escal_cost(T)/escal_cost(T_c)) * pred_completion(T_c) > deadline(T_c)
        RETURN("YES");
    ENDIF
    IF number of activities on the path from T to T_c is less than P AND
        T_c not followed by conditional activities AND
        (T == T_c OR escal_cost(T) ≥ escal_cost(T_c))
        FOR each successor T'_c of T_c in W
            IF NextStep(W,T,T'_c) = "YES"
                RETURN("YES");
            ENDIF
        ENDFOR
    ENDIF
    RETURN("NO");
END;
```

Figure 3: The basic algorithm for early escalations

that a future activity is going to escalate, it is often cost-effective to invoke escalation *immediately*, regardless of whether or not the currently executed activity misses its deadline, since this would allow more time for remedial actions and would reduce the cost associated with the compensation of finished activities.

The algorithm for predicting future escalations and deciding whether escalating early is beneficial is shown in Figure 3. The algorithm uses parameter P to limit the prediction scope, and it is invoked every time an activity T becomes ready for execution. Consider a directed graph with nodes corresponding to activities in schema W, where an edge goes from node T_i to T_j if T_j can immediately follow T_i in some execution. The algorithm examines all activities reachable from T by a path of length not greater than P, and such that there are no conditional splits on the path. Conditional splits complicate prediction since the activities that will be executed in the future are not known in advance. The algorithm examines these activities in depth-first order, and stops exploring each path when it finds an activity T_c that has lower escalation cost than T. In this case, even if T_c or any activity following T_c is likely to escalate, it is cost-effective to wait until T_c is ready for execution rather than escalating now.

For an activity T_c, the algorithm computes T_c's predicted completion time, $pred\_completion(T_c)$, as $avg\_completion(T_c)$ times the *load factor* of the system, which is the ratio $\frac{cur\_queue(T_c)}{avg\_queue(T_c)}$, where

$cur\_queue(T_c)$ is the current queue length of the agents that can execute T_c averaged over all such agents, and $avg\_queue(T_c)$ is the average queue length of these agents that corresponds to the load conditions reflected in $avg\_completion(T_c)$. Intuitively, if the average completion time for T_c was observed when the average queue length was 3, and the current queue length averaged over the agents that can execute T_c is 6, we can expect that T_c will take twice as long, provided the current load conditions remain by the time T_c becomes ready for execution.

When examining the very first activity, T, (i.e., when $T_c = T$ in the algorithm), early escalation is beneficial if T is likely to escalate, i.e., $deadline(T_c) < pred\_completion(T_c)$. Early escalation in this case means that we force escalation without starting T. For subsequent activities, the condition for early escalation reflects the idea that the less difference in escalation costs the more confident we want to be that T_c will miss its deadline before invoking an early escalation. One way to account for this could be to compute the confidence interval of our estimate for $pred\_completion(T_c)$ for the confidence level that would be set based on the difference in escalation costs. The unclear part here is how to set the confidence level. We plan to explore this method in the future.

Instead, we use a heuristic approach to indirectly measure the confidence in our prediction by how much the predicted completion time of T_c exceeds its deadline. For example, if $pred\_completion(T_c)$ exceeds slightly $deadline(T_c)$, then even a small error in estimating $pred\_completion(T_c)$ will lead to the incorrect prediction that T_c will escalate. Then, if the escalation costs of T and T_c are close, we may not want to escalate early, to avoid an unnecessary escalation. However, if the escalation cost of T_c greatly exceeds that of T, we would want to escalate immediately just because we have a reason to believe that T_c will escalate. We achieve this by introducing the factor $(1 - \frac{escal\_cost(T)}{escal\_cost(T_c)})$, which ranges from 1 when $escal\_cost(T_c) >> escal\_cost(T)$ to 0 when $escal\_cost(T_c) = escal\_cost(T)$, in which case early escalation is not beneficial.

Finally, we show how the early escalation mechanism is used together with any of the DDA algorithms presented in the previous section. For each activity T_c, the early escalation algorithm predicts the value of $effective\_deadline(T_c)$ that would be assigned to it by the DDA algorithm at hand, assuming that each activity T_i on the path from T to T_c takes $pred\_completion(T_i)$ to complete. Then, $effective\_deadline(T_c)$ replaces $deadline(T_c)$ in Figure 3.

This calculation of $effective\_deadline(T_c)$ is very optimistic because it does not take into account AND-joints. Consider, for example, the schema on Figure 2 and assume that we are about to execute $T6$. Assume for simplicity that there is no available slack when $T6$ become ready for execution. The algorithm will obtain $pred\_completion(T6)$, and then calculate $available\_slack$ after $T6$ as $deadline(T6) - pred\_completion(T6)$. It will then use this available slack for calculating the effective deadline of the next activity, $T8$. However, the actual available slack for $T8$ may be considerably smaller, because $T8$ will not be ready for execution until its other predecessor, $T7$, finishes. Thus, the effective deadline of $T8$ may be over-estimated. It is important to note, however, that over-estimating the effective deadline may only reduce the number of times early escalation is found beneficial. Thus, while it may reduce the benefits of predictive escalations, it will not cause erroneous escalations (i.e., the algorithm will not invoke an early escalation for a process that would have completed normally otherwise).

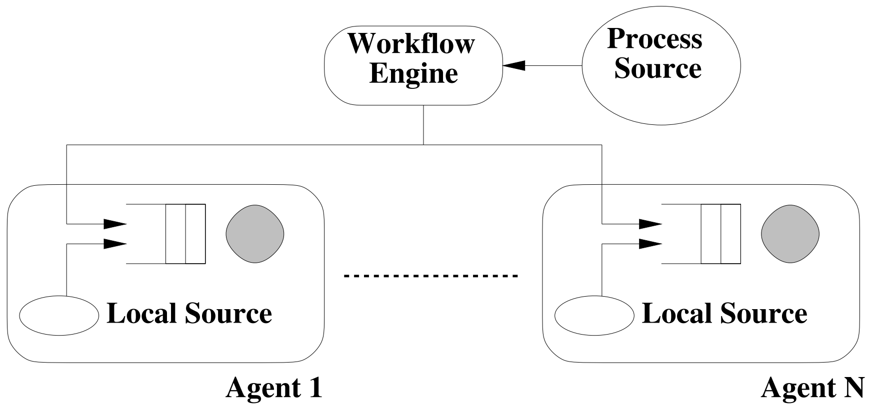

Figure 4: Workflow Simulation Model

5 Experimental Evaluation

To study the performance of the deadline adjustment algorithms and the impact of the early escalation technique presented in this paper, we developed a simulation model and carried out an initial experimental evaluation. In the following sections, we outline the simulation model, present the experimental setup as well as the assumptions made and, finally, analyze the collected results.

5.1 Simulation Model

An event-driven simulation toolkit was used in our study. Figure 4 shows the workflow management system model we used. The model consists of four basic modules: *process source, workflow engine, agent nodes,* and *local sources.* A detail description of each of these modules is given below.

Process Source: Generates instances of business processes and submits them to the workflow engine for execution. Instances of business processes are generated using the schema of each process and the arrival rate specified for each schema.

Workflow Engine: Schedules the activities of every business process instance. The scheduling policy of the workflow engine is very simple; when an activity is ready for execution, the engine selects the agent that can execute the activity and schedules the execution of the activity at that agent. When an activity finishes execution, the engine is notified and, then, it determines the activities, if any, to be scheduled for execution next.

Agents: Execute workflow activities. Agents may belong to different roles, including humans and computer applications.

Local Source: Generates tasks to be executed at an agent. These tasks do not belong to workflow instances and they do not have deadlines associated with them. Local tasks are introduced

Parameter	Setting
Number of agents	8
Workflow schemas	1
Number of activities per workflow schema	8
Instance inter-arrival rate	0.3
Slack factor	1
Local task inter-arrival rate	0.4

Table 1: Baseline simulation setting

into the model for modeling real-life scenarios where existing resources are shared by WFMSs and other applications.

Each business process schema consists of several activities and has an arrival rate, referred to as *instance generation rate*. Each activity belongs to an activity class. Each activity class contains information about the statistical distribution used for computing the execution times of the activities belonging to the class. In addition, activity classes contain information regarding the agent(s) that can execute the activities belonging to them.

We assume that when an escalation occurs, the entire business process instance is aborted and the escalation cost is equal to the cost of the activity that triggered the escalation. We do not model the execution of escalations; the assumption is that escalations are executed by distinct agents and they do not place additional load on regular agents. In addition, we assume that each agent monitors its queue, which contains workflow activities and local tasks, if any, and it maintains a running estimate of the average size of the queue. The workflow engine, on the other hand, is responsible for maintaining the average completion time for every workflow activity.

Before an agent starts executing a workflow activity, it checks if the activity was queued longer than its allowable execution time. If it was, the agent notifies the workflow engine about the missed deadline and escalation is invoked. Otherwise, the activity is executed to completion. In this case, the workflow engine checks upon the completion of the activity whether it missed its deadline, in which case escalation is invoked. When the activity completes without escalation, the activities that can be executed next are computed, their deadlines are adjusted depending on the DDA algorithm used, and they are submitted for execution at the appropriate agents. When an activity escalates, the entire process instance is aborted in a single step.

5.2 Experimental Setup

Typically, organizations have many business processes consisting of different number of activities. These activities can be considerably different in terms of their actual execution times and the resources they require, e.g., a telecommunications service provisioning process may involve human actions, real-time system actions, database application programs, and legacy applications. If we tried to capture all this diversity and complexity in our performance study, the results would have been obscured by too many parameters. Rather, we chose to model just one business process for studying our algorithms. In particular, we chose to model the telecommunications service

Activity	Mean Execution	Deviation	Escalation Cost	Agent
$T1$	2.0	0.5	1	1
$T2$	2.0	0.5	2	2
$T3$	2.0	0.5	4	3
$T4$	10.0	5.0	4	4
$T5$	2.0	0.5	8	5
$T6$	2.0	0.5	16	6
$T7$	2.0	0.5	16	7
$T8$	2.0	0.5	32	8

Table 2: Service provisioning activity attributes

provisioning workflow presented in Figure 2. The workflow system consists of single-threaded agents, which follow a non-preemptive FIFO scheduling policy.

The execution time of activities follows the *erlang* distribution. The execution time of local tasks follows the *exponential* distribution with mean equal to 1.0. The computation of the deadline of an activity is based on the mean execution time of the class containing the activity and the value of the *slack factor*. The slack factor determines how tight the deadline is with respect to the average execution time of the activity. In particular, the deadline of an activity T_i is computed using the following formula.

$$deadline(T_i) = mean\_execution\_time * (1 + slack\_factor)$$

Table 1 summarizes the baseline parameter settings, and Table 2 shows the values for activity parameters used in the experiments. Each simulation experiment consists of three simulation runs, with each run initialized using a different seed. The statistics collected during each of the runs correspond to $100,000$ process instances. In addition, during each run, we started collecting statistics after the simulator had reached a stable state.

5.3 Simulation Results

In this section, we present the results of our initial simulation study. The study investigates the cost benefits of the early escalation mechanism when it is used together with one of the five deadline adjustment algorithms of Section 3. When the prediction scope is 0, the early escalation algorithm is not invoked at all and, hence, the results show the performance of the DDA algorithms alone. As performance measures, we use the percentage of executions that resulted in escalations and the escalation cost.

5.3.1 No Local Tasks

Here, we present the performance results when only workflow activities are executed by the agents. Figures 5 and 6 show the percentage of escalations and the escalation costs, respectively. While

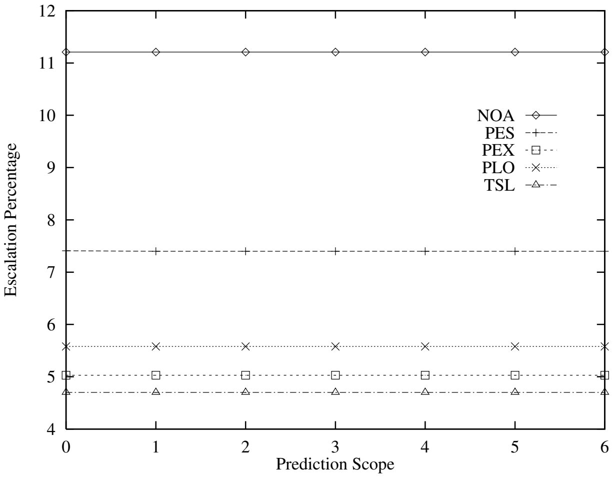

Figure 5: No local load: Escalations

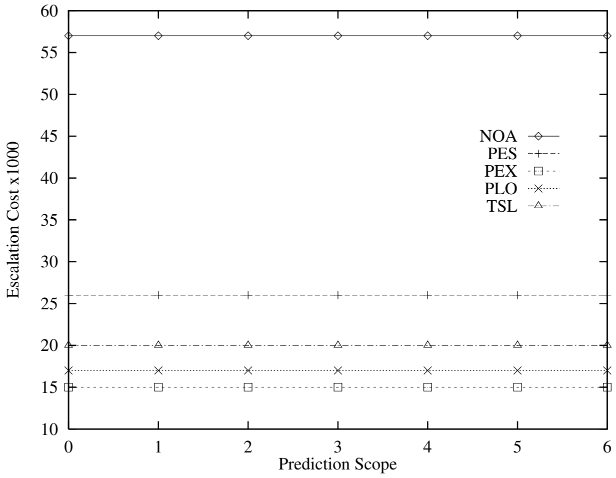

Figure 6: No local load: Escalation cost

these graphs show that dynamic deadline adjustment reduces significantly the number of escalations and the escalation cost of workflow execution, these gains do not depend on the prediction scope of the early escalation mechanism. In other words, early escalation is not beneficial in this case.

The reason for this behavior is that, in the absence of local tasks, the system operates in a steady state, with agent queue lengths not varying dramatically. Moreover, the process arrival rate does not load the system enough to trigger early escalations (early escalations are triggered when the predicted completion time exceeds the deadline by a factor of $(1 - \frac{escal\_cost(T)}{escal\_cost(T_c)}))$. Although some early escalations occur during the execution of $T4$ due to $T4$'s high mean execution and standard deviation values, they account for less than 1% of all escalations and, therefore, they do not affect the results.

Among the different DDA algorithms, Figure 5 shows that TSL outperforms all of them in terms of the number of escalations. However, as seen in Figure 6, TSL is not the most cost effective algorithm. In fact, it is 33% more expensive than PEX and 17% more expensive than PLO. By assigning the entire slack to the next activity, TSL reduces the escalations during the execution of initial activities. However, by letting early activities consume the whole slack, TSL increases the number of more costly escalations during the execution of the remaining activities. On the other hand, even though PEX and PLO cause more escalations than TSL, a greater percentage of their escalations occur during the execution of early activities, resulting in lower overall costs.

Finally, PES performs the worst among all DDA algorithms, both in terms of the number of escalations and escalation costs. It has 57% more escalations than TSL and 32% more escalations than PEX. The reason for this behavior is that, given the difference in escalation costs between earlier and later activities, PES assigns the least slack among all algorithms to activities and, consequently, it causes more escalations to occur. These escalations are too numerous to be compensated by a reduction of escalations at later stages, even though these escalations have higher costs.

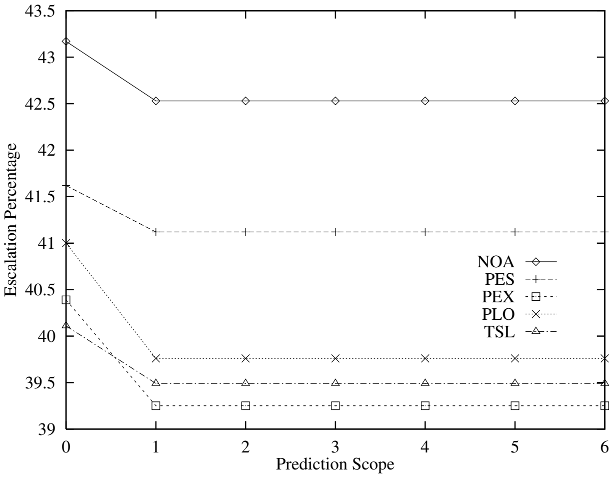

Figure 7: Local load on agent 1: Escalations

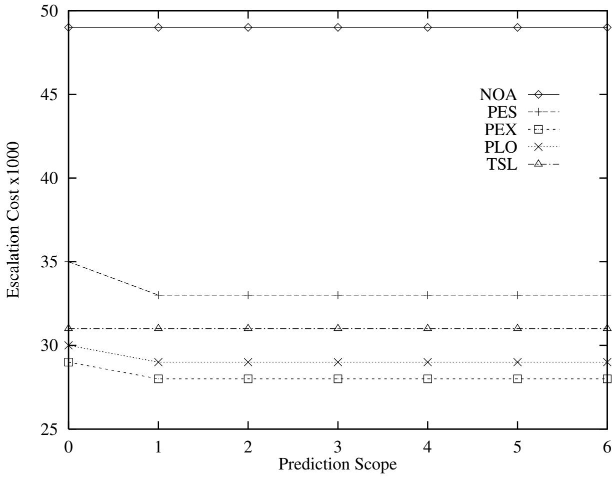

Figure 8: Local load on agent 1: Escalation cost

5.4 Adding Local Tasks

We now turn our attention to the performance of the algorithms when agents execute local tasks as well. Figures 7 and 8 illustrate the percentage of escalations and the escalation costs for the early escalation algorithms when the agent executing $T1$ is loaded with local tasks. These figures show some small benefits due to early escalations, both in terms of the number of escalations and the escalation costs. In particular, early escalations account for about 1.5% of the total number of escalations for all algorithms.

One reason for the benefits in escalation costs being so small is the way we do accounting. Since the loaded agent in this experiment is the one executing the very first activity, early escalation here mostly means escalating *before* the first activity executes as opposed to *after* the first activity misses its deadline. In both cases, we, simplistically, assume the escalation cost to be the same and equal to the cost associated with the first activity. Thus, the cost benefits shown in Figure 8 are due to the reduction in the number of escalations only.

Interestingly enough, although early escalations are supposed to just replace a later escalation with an earlier one, Figure 7 shows a drop in the total number of escalations when the early escalation mechanism is employed, i.e., the prediction scope becomes greater than 0. This is because, when a workflow instance escalates early, it does not contribute to the load of the agents that would have to execute some of its remaining activities. Therefore, these agents can service more tasks from other instances. In other words, escalating early increases the total system capacity in terms of executing non-escalated process instances.

Furthermore, Figure 7 shows that one-step-ahead prediction in the early escalation mechanism yields all the benefits (the graphs flatten beyond prediction scope 1). This is because only the agent that executes activity $T1$ is loaded and, hence, similar to the discussion we had in the previous section, extending the prediction scope increases early escalations marginally since the agents executing the remaining tasks are not loaded.

Comparing Figures 5 and 7, we notice that, while TSL outperforms the rest of the DDA algorithms when there are no local tasks, PEX results in the minimum number of escalations when early escalations mechanism is employed and the agent that executes $T1$ performs local tasks as well. In terms of the escalation costs, PEX is the best in both cases.

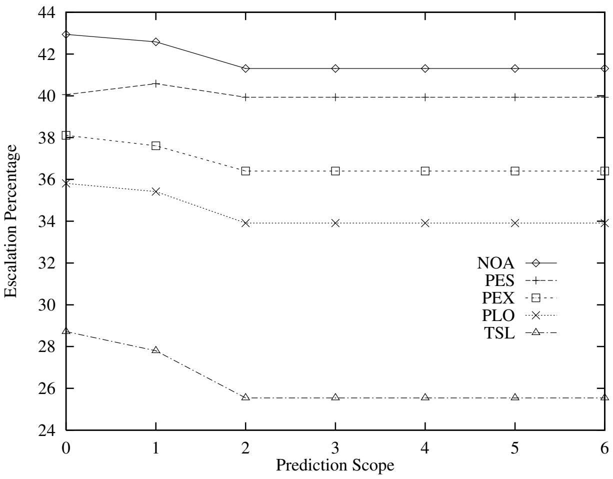

Figure 9: Local load on agent 2: Escalations

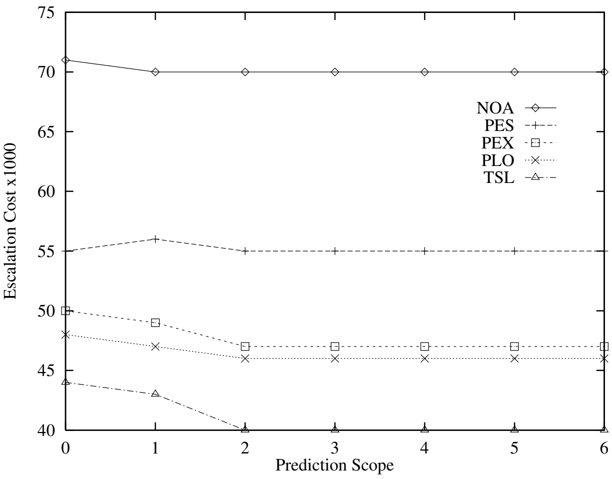

Figure 10: Local load on agent 2: Escalation cost

Figures 9 and 10 show the percentage of escalations and the escalation costs for the early escalation algorithms when the agent executing $T2$ is loaded with local tasks. The graphs exhibit the trends we noticed in the previous experiment, except for the fact that TSL outperforms all DDA algorithms in both the number and cost of escalations. This is because TSL assigns the whole slack produced after $T1$'s execution to $T2$ whereas the rest of the algorithms assign only a portion of it to $T2$ – Notice that in the previous experiment there was no slack to be assigned to the overloaded agent since this agent executed the very first activity. In addition, TSL benefits the most among all algorithms when early escalations take place.

As we could expect from the earlier discussion, the benefits from the early escalation mechanism extend to prediction scope of 2, and are higher than in the previous experiments. In particular, the percentage of early escalations ranges from 6.3 for NOA to 12 for TSL. Finally, in contrast to the previous experiments, PLO outperforms PEX in both the number of escalations and the escalation cost. The reason for this behavior is that PLO assigns more slack to activities than PEX.

When the agent that executes activity $T3$, which is enacted conditionally with frequency 80%, is loaded with local tasks, the performance of our algorithms is shown in Figures 11 and 12. These figures are very similar to the ones for the case where no load is placed on any of the agents. The difference is that, here, TSL has the best performance in both the number of escalations and the escalation cost. In particular, TSL results in 45% fewer escalations than PEX and 53% fewer escalations than PLO. In terms of costs, TSL is 31% less expensive than PEX and 40% less expensive than PLO.

Also, similar to the results shown in Figures 5 and 6, the prediction scope does not matter. The reason is as follows. When we apply the early escalation algorithm on $T1$ and $T2$, the prediction scope does not matter because the agents that execute these activities are not loaded and the algorithm stops after examining $T2$ due to the OR-split. When we examine $T4$ and beyond, the prediction scope does not matter either because the agents are not loaded. When we are about to execute $T3$, the early escalation algorithm is likely to announce that early escalation should be

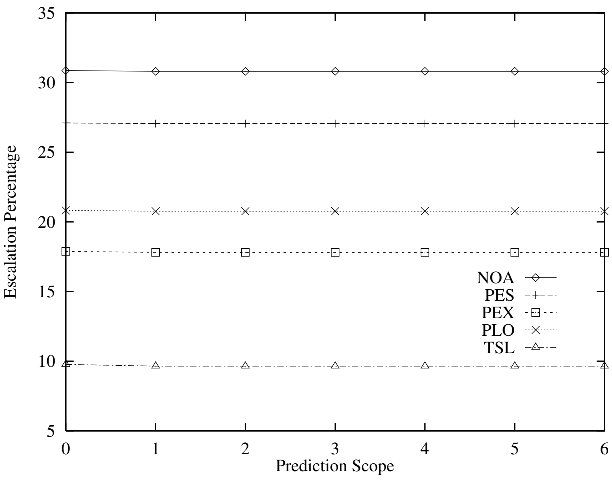

Figure 11: Local load on agent 3: Escalations

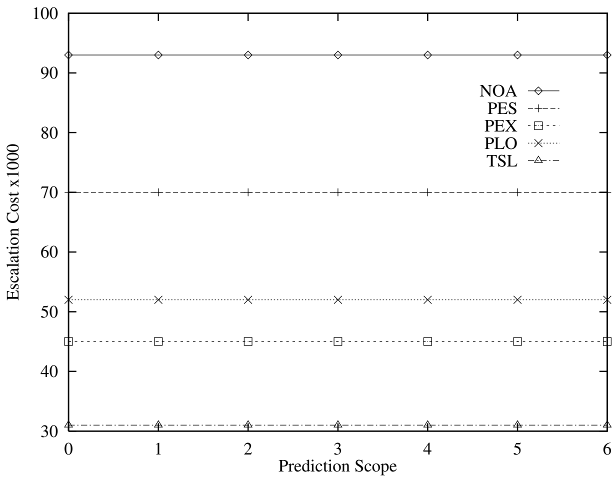

Figure 12: Local load on agent 3: Escalation cost

invoked at this point and will not examine any other activities. However, since the cost of the early escalation in this case is the same as $T3$'s escalation cost (because of our accounting strategy), early escalation does not affect the escalation cost.

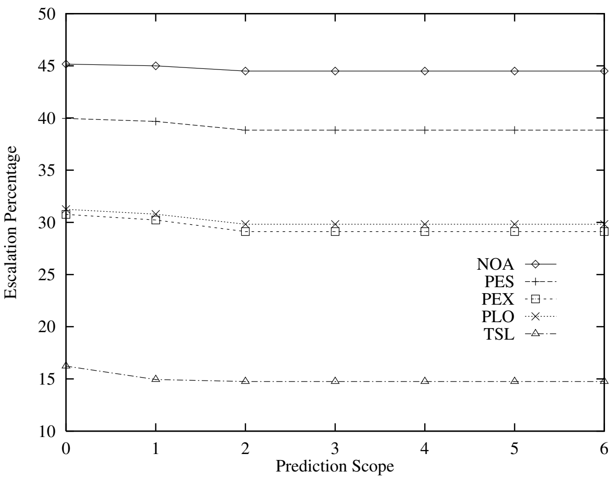

Figure 13: Local load on agent 5: Escalations

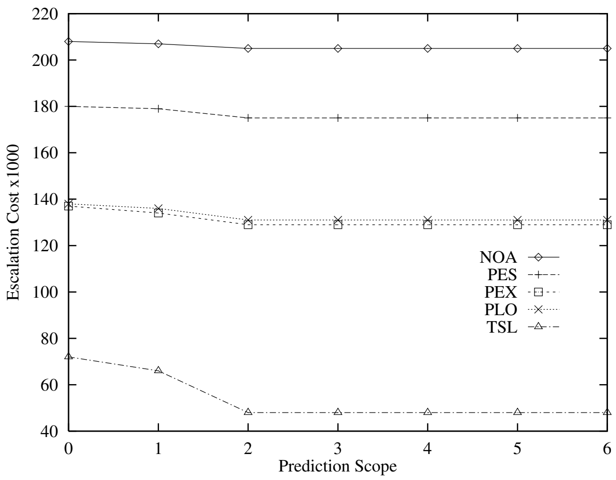

Figure 14: Local load on agent 5: Escalation cost

Turning to the case where the agent that executes $T5$ is loaded with local tasks, the percentage of escalations and the escalation costs are illustrated in Figures 13 and 14. The graphs demonstrate the same trends as the results presented in Figures 9 and 10, respectively. In particular, early escalation is beneficial for all algorithms, and increasing the prediction scope beyond 2 does not produce any additional benefits.

One might have expected that the performance of the DDA algorithms would continue to improve as the prediction scope becomes greater than 2. However, prediction scope greater than two does not have any effect when $T1$ or $T2$ are examined because they are followed by an OR-split and the prediction stops at the split. In addition, the early escalation algorithm will rarely return "YES" when examining $T6$, $T7$, or $T8$ since the agents that execute these activities are not sufficiently loaded. The only situation when early escalation algorithm will return "YES" in the majority of executions is when examining $T5$ because the agent executing $T5$ is loaded. $T5$ is within prediction scope 2 from $T3$ and $T4$.

We do not include the graphs produced for the cases where the agents that execute $T6$ and $T7$ are loaded with local tasks because they are very similar to the graphs when the agent that executes $T5$ is loaded (shown in Figures 13 and 14). However, we should note that in these cases early escalation does not benefit the DDA algorithms any more when the prediction scope exceeds 3. The explanation is the same as the one given for the case where the agent executing $T5$ is loaded; namely, $T6$ and $T7$ are within prediction scope 3 from $T3$ and $T4$.

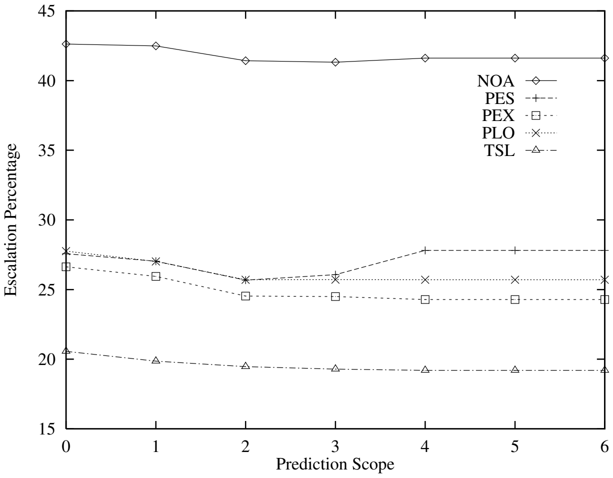

Figure 15: Local load on agent 8: Escalations

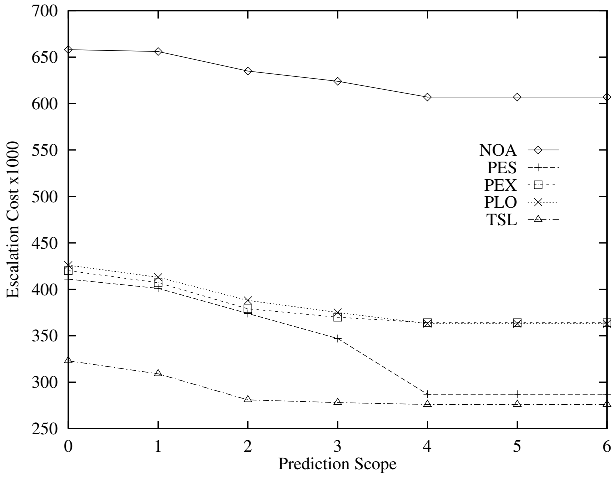

Figure 16: Local load on agent 8: Escalation cost

When the agent that executes the last workflow activity, $T8$, is loaded, the performance of our algorithms differs significantly from the previous cases,as shown in Figures 15 and 16. In particular, PES is not as bad as in the previous experiments. Not only that, but PES outperforms PLO and PEX with respect to the escalation cost. This is because under PES, the percentage of early escalations is the greatest among all PEX and PLO, as shown in Table 3. TSL is still the best overall algorithm.

Table 3 shows the percentage of executions that result in escalations and the percentage of these escalations that are due to early escalations when the prediction scope is between 0 and 4. Beyond escalation scope 4, the percentage of early escalations does not change because the longest path used by the prediction mechanism in the provisioning workflow schema is 4 (remember that prediction always stops at OR-splits). As we see, as the prediction scope increases, early escalations contribute more to the total number of escalations that take place.

6 Related Work

We are not aware of any work that directly addresses the issue of reducing costs due to precess escalations in workflow systems. Nevertheless, dynamic deadline adjustment is somewhat similar to real-time scheduling [LL73, AGM88, HSTR89]. Real-time systems use deadlines for scheduling

Prediction Scope		NOA	PES	PEX	PLO	TSL
0	*Total*	42.62	27.57	26.63	27.76	20.57
	Early	0.00	0.00	0.00	0.00	0.00
1	*Total*	42.49	27.03	25.94	27.02	19.86
	Early	1.41	3.99	5.01	4.81	19.23
2	*Total*	41.43	25.68	24.53	25.70	19.47
	Early	7.82	12.38	14.88	13.85	31.99
3	*Total*	41.32	26.07	24.50	25.71	19.29
	Early	9.87	20.25	17.42	17.15	32.45
4	*Total*	41.61	27.81	24.28	25.70	19.20
	Early	14.05	38.40	18.28	20.43	32.81

Table 3: Local load on agent 8: Percentage of total and early escalations

system components such as CPU and I/O and, typically, they target applications consisting of single tasks with known resource requirements and execution times. We view scheduling and dynamic deadline adjustment as complimentary mechanisms.

Our DDA mechanism is related to the work described in [MN95]. In [MN95], the authors present priority-driven CPU scheduling algorithms for transactional workflows. Each workflow process consists of several sequential tasks. Each task is an ACID transaction having an average response time goal. The assignment of priorities is based on the performance of the tasks relative to their original response time goals. In contrast to these algorithms, our DDA algorithms do not concentrate on CPU scheduling. In addition, our algorithms are not restricted to transactional workflows, and they allow both sequential, conditional, and parallel execution of tasks.

In [KGM93a, KGM93b], the authors study the problem of how the deadline of a real-time activity is automatically translated to the deadlines of all sequential and parallel sub-tasks constituting the activity. Each sub-task deadline is assigned just before the sub-task is submitted for execution, and the algorithms for deadline assignment assume that the *earliest deadline first* scheduling policy is used. Similar to our work, the goal of the proposed algorithms is that of minimizing the number of missed deadlines. In contrast to these algorithms, our work assumes that all sub-tasks have been assigned deadlines before the activity is submitted for execution and, thus, we focus on adjusting sub-task deadlines on-line rather than assigning deadlines. In addition, [KGM93a, KGM93b] assume a soft real-time environment where sub-tasks may miss their deadlines without triggering other sub-tasks. In our environment, however, escalations caused by missed deadlines may result in the execution of new activities, and our algorithms take into account the cost associated with these escalations.

In active real-time databases transactions may trigger new transactions during their execution. The triggered transactions are executed immediately or after the original transaction finishes execution. In [SSDTR96], the authors present algorithms that try to minimize the number of triggering transactions that miss their deadlines when a priority-driven scheduling policy is employed. The proposed algorithms assign priorities to the triggered transactions, and they dynamically reassign the priorities of the triggering transactions. In contrast, our DDA algorithms modify the deadlines assigned to activities without altering their priorities, if any. Consequently, our DDA algorithms could be complemented with the scheduling algorithms proposed in [SSDTR96].

7 Conclusions

In this paper, we concentrated on ways to reduce operational costs due to escalations by combining *dynamic deadline adjustment* and *early escalations*. Dynamic deadline adjustment reduces the number of escalations that occur during workflow executions by extending the deadlines of the workflow activities that are going to be executed in the future by using any available slack time. The distribution of the slack time is based on the estimated execution time of the activities, their escalation cost, and the loads of the agents responsible for their enactment.

Our early escalation mechanism tries to reduce the operational cost by invoking escalations early. Our technique uses statistics from past executions as well as current status information, such as current agent loads, to predict if a future activity is going to miss its deadline. It then exploits the knowledge about the costs of escalation provided by the business analyst to decide if it is beneficial to force escalation early, without waiting for the execution to reach the trouble point.

We have also presented preliminary performance results which evaluate the performance of our algorithms under different workload conditions. The results show that early escalation combined with DDA is an effective approach for reducing escalation-related costs. All DDA algorithms resulted in substantial savings compared to the system with no adjustment, both in the number of escalations and the escalation costs. Among the DDA algorithms, TSL and PEX appear to perform well overall. Nevertheless, more work is needed to investigate the trade-offs between the DDA algorithms presented in this paper.

In addition, the results showed that using early escalations in conjunction with DDA offer additional cost reduction of up to 33% (see Figure 14), compared to DDA alone. It even reduced the number of escalations by up to 10% (see Figure 9), due to decrease in the system load. We expect to be able to improve these numbers further by being more aggressive in invoking early escalations. Indeed, our algorithm for early escalations, in its present form, invokes early escalation only when the predicted completion time of a future activity exceeds its deadline by a significant factor, especially if the escalation costs of the current and future activities are close. We deliberately chose this conservative heuristic so that, in our first approach to the problem, we do not cause unnecessary escalations. We plan to explore other heuristics in the future.

References

[AGM88] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions: a performance evaluation. In *Proceedings of the 14th International Conference on Very Large Data Bases*, pages 1–12, Los Angeles, CA, 1988.

[Flo] TeamWare Flow. Collaborative workflow system for the way people work. P.O. Box 780, FIN-00101, Helsinki, Finland.

[GHS95] D. Georgakopoulos, M. Hornick, and A. Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3(2):119–153, 1995.

[Gro94] The Workflow Management Coalition Group. A workflow management coalition specification. Technical report, The Workflow Management Coalition, November 1994. http://www.aiai.ed.ac.uk/WfMC/.

[HSTR89] J. Huang, J.A. Stankovic, D. Towsley, and K. Ramamritham. Experimental evaluation of real-time transaction processing. In *Proceeding of the 10th Real-Time Systems Symposium*, December 1989.

[KGM93a] B. Kao and H. Garcia-Molina. Deadline assignment in a distributed soft real-time system. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 428–437, 1993.

[KGM93b] B. Kao and H. Garcia-Molina. Subtask deadline assignment for complex distributed soft real-time tasks. Technical Report 93-1491, Stanford University, 1993.

[KS95] N. Krishnakumar and A. Sheth. Managing heterogeneous multi-system tasks to support enterprise-wide operations. *Distributed and Parallel Databases*, 3(2):1–33, 1995.

[LL73] C.L. Lin and J. Layland. Scheduling algorithms for multiprogramming in hard real-time environments. *Journal of the Association of Computing Machinery*, 20(1):46–61, January 1973.

[LR94] F. Leymann and D. Roller. Business process management with flowmark. In *Proceedings of the 39th IEEE Computer Society International Conference*, pages 230–233, San Francisco, California, February 1994. http://www.software.ibm.com/workgroup/flowmark.

[MN95] M. Marazakis and C. Nikolaou. Towards adaptive scheduling of tasks in transactional workflows. In *Winter Simulation Conference*, Washington D.C., 1995.

[PR96] E. Panagos and M. Rabinovich. Escalations in workflow management systems. In *DART Workshop*, Rockville, Maryland, November 1996.

[PR97] E. Panagos and M. Rabinovich. Predictive workflow management. Submitted for publication, 1997.

[SGJ+96] A. Sheth, D. Georgakopoulos, S. Joosten, M. Rusinkiewicz, W. Scacchi, J. Wileden, and A. Wolf. Report from the NSF workshop on workflow and process automation in information systems. *SIGMOD Record*, 25(4):55–67, December 1996.

[SSDTR96] R.M. Sivasankaran, J.A. Stankovic, B. Purimetla D. Towsley, and K. Ramamritham. Priority assignment in real-time active databases. *The VLDB Journal*, 5(1):19–34, 1996.

[XSo] A Division of Xero XSoft. Inconcert technical product overview. 3400 Hillview Avenue, Palo Alto, CA 94304. http://www.xsoft.com.