Proceedings of the 3rd Workshop on Emerging Web Services Technology - WEWST, Dublin, IE, 2008/11

Enforcing Advance Reservations for E-Science Workflows in Service Oriented Architectures*

Christoph Langguth and Heiko Schuldt

University of Basel
Department of Computer Science
Database and Information Systems Group
Bernoullistrasse 16
CH-4056 Basel

Abstract. Scientific Workflows have become an important tool to perform complex calculations, especially when individual operations are made available as services in Service Oriented Architectures. At the same time, Quality-of-Service aspects and Advance Reservation of resources by means of Service Level Agreements (SLA) are topics that get ever-increasing attention in order to make best use of available resources in a predictable manner. The support of such SLAs at the level of workflows raises two interrelated issues pertaining (i) to the temporal prediction of reservation start time and duration of individual activities, and (ii) to the actual enforcement of resource commitments at the provider side.
In this paper, we outline our vision of a distributed workflow engine with support for SLAs and Advance Reservations. We focus on reservations addressing processing capabilities, i.e., shares of CPU power. In particular, we present a module of the system that is responsible for the enforcement of such reservations at the individual service providers' nodes, which, by means of a Fuzzy Controller adjusting task priorities, makes sure that the SLAs are met in a fair way.

Key words: Advance Reservation, SOA, Service Grid, Scientific Workflows, CPU share enforcement

1 Introduction

As Service Oriented Architectures (SOAs) are becoming widely deployed in a variety of domains, e.g., in e-Commerce or e-Science, the focus is shifting more and more from mere deployment and integration issues to other, non-functional aspects like Quality of Service (QoS), for example to reserve storage, network or computational resources in advance. A SOA separates functions into distinct units (services), which can be distributed over a network and can be combined and reused to create larger-scale applications (workflows). A widespread

* This work has been partly supported by the Hasler Foundation within the project COSA (Compiling Optimized Service Architectures)

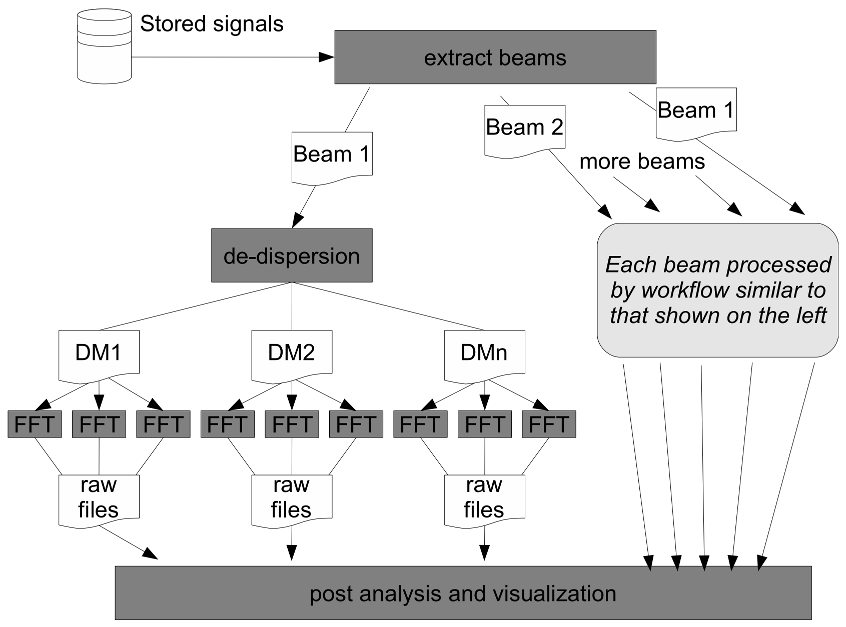
Stored signals
extract beams
Beam 1
Beam 2
Beam 1
more beams
de-dispersion
Each beam processed by workflow similar to that shown on the left
DM1
DM2
DMn
FFT
raw files
post analysis and visualization

Fig. 1. Pulsar astronomy workflow, according to [4]

language used for defining such workflows in WSDL/SOAP-based SOAs is the Business Process Execution Language (BPEL [2]).

For instance, consider the workflow from the Pulsar Astronomy domain, depicted in Figure 1. This workflow, described in detail in [4], is used to discover and visualize radiation caused by pulsars. For each beam extracted from captured radiation signals, a number of computation steps – each of which can be implemented as a web service (WS) – has to be performed, namely several dedispersion measures, followed by multiple Fast Fourier transforms, and a final aggregation and visualization step. Note that the presented workflow is currently not run in a SOA (but in a traditional cluster environment using MPI), however the authors state that a transition to SOA is envisaged [4].

Another workflow, from the Earth Observation Domain, is described in [6]. These processes feature both attributes that are generally used to distinguish so-called *Scientific Workflows* from Business workflows: vast amounts of data, along with computationally expensive processing steps. While QoS may be a requirement for some applications, e.g., when results are needed in real-time or generally “as fast as possible”, any kind of service execution can benefit from the predictability that such QoS contracts (or Service Level Agreements, SLAs) can provide. Assuming that the agreement covers, for example, computational power, i.e., CPU times or shares, service consumers can weigh cost against speed of execution, based on the individual requirements. Service providers may be able to achieve an (economically) optimal resource usage by careful negotiation.

Suppose that a user wants to run the abovementioned workflow taking advantage of QoS criteria, where SLAs with the service providers are established by Advance Reservations (AR). While this task is still relatively easy for individual service invocations scheduled to start at a given point in time, to use ARs in a composed service workflow which consists of a partially ordered set of service invocations, one needs to answer the following two questions:

1. For how long should a reservation for a particular service be made? Since the service implementation is on the provider side, it is generally the service provider that has to make this information available. Note that the provider also has to take measures to enforce this prediction, such as controlling CPU usage.
2. When should a reservation for a particular service start? In a workflow setting, individual service calls will usually depend on the execution or output of previous operations – so anticipating the start time in turn resolves to answering the previous question.

Our objective is to develop and evaluate a system, called DWARFS (Distributed Workflow engine with Advance Reservation Functionality Support), that can support QoS at workflow level. In this paper, we address the first question above, i.e., enforcing CPU usage levels, which is of fundamental importance to proceed with our overall vision. We show that, by using a fuzzy controller to dynamically adjust thread priorities, it is possible to closely confine tasks to the CPU percentage committed to in the SLA, and that one can obtain relatively accurate runtime predictions for future reservations by extrapolating from the runtime and the observations made during the enforcement.

The remainder of this paper is structured as follows: Section 2 shortly introduces the overall DWARFS system. Sections 3 and 4 present our approach to CPU usage enforcement and runtime prediction, as well as an evaluation of first results. Section 5 gives an overview of related work. Finally, Section 6 concludes.

2 Overview of the DWARFS system

The vision of the DWARFS system (Distributed Workflow engine with Advance Reservation Functionality Support) is an advanced BPEL orchestration engine, particularly tailored to e-Science workflows, that is:

– **Fully decentralized.** Whereas any workflow execution is by definition decentralized in the sense that the operations take place at various independent providers, our goal is to also distribute the orchestration engine itself, eliminating the need for a central component controlling the workflow execution. To name just a few assets, a decentralized system helps avoid bottlenecks, hot-spots and single points of failure that a centralized execution engine could potentially create. In addition, especially in scientific workflows where large data volumes are transported during the orchestration, overall performance also may benefit from having the execution engines in proximity to the target services, by jointly selecting the providers of the workflow's activities and the providers to store instance data in a way that allows to minimize data transfer during workflow execution.

– **WS-Agreement capable.** Ultimately, a workflow execution should be subject to SLAs just like a "normal" WS execution can be. This means that the workflow engine, from a customer perspective, is a service (and agreement) provider, while in essence it is acting as a proxy that itself has to take the customer role for negotiating agreements with the providers of the target services.

For the distributed execution engine part, we can revert to previous experiences from implementing similar systems based on OSIRIS [7, 18], which will be enhanced and extended. For the WS-Agreement (WS-A, [3]) part however, entirely new components have to be developed. This poses a lot of challenging questions including, but not limited to, the (semantic) evaluation of the agreement terms and matchmaking of the possible providers, re-negotiation strategies especially for failure handling, possibly redundant reservation strategies for extremely important processes, etc.

Since we are mostly interested in timing issues (because these are crucial for the agreement establishment for entire workflows), this leads to a natural focus on the *prediction* and therefore *control* of (wall-clock) runtime. As for CPU-intensive tasks, the runtime is directly related to CPU usage, we use a basic model where clients negotiate a reservation for a particular share of the available CPU with a service provider. The service provider gives an estimation of the (maximum) expected runtime for the provision of the service, and its degree of confidence that the estimation will be met. Whereas several implementations for the negotiation of WS-A exist, to our knowledge currently none exists that is able to *enforce* the abovementioned requirements at runtime.

In order to support SLAs at the workflow level, all individual service providers taking part in the workflow must support the respective SLAs as well. As a first step, we therefore developed a component which is able to control CPU usage, restricting it to a given value, and to derive the information needed for a correct runtime estimation. DWARFS uses a Java-based implementation of all components, because of the widespread use of Java in the SOA field and its well-known advantages such as portability and potential for re-use. While we opted for a Java implementation, the approach is not limited to a Java environment. In fact, we show that even in an environment where direct access to the system scheduler is not available, it is possible to accurately control the CPU consumption of tasks. In other environments, the techniques to control the CPU usage may be different, but the conclusions drawn regarding the prediction of future runtimes remain valid – thus, the approach, or a variation of it, can be extended to legacy implementations found in the eScience domain. The basic functionality of the components is as independent as possible from concrete container implementations, thus easing ports of the prototype implementation targeted at a Globus Toolkit 4 (GT4) container. In addition, this component is meant to be as non-invasive as possible: while it may require certain adjustments to the container or its configuration, it does not require any changes to the OS, the JVM, or – most importantly – the actual service implementations.

Figure 2 presents an enactment scenario where the DWARFS components have been deployed in several of the WS containers providing the target services for the workflow. While the presence of the Process Execution (PE) module is not mandatory on all nodes, as service calls can equally well be made remotely, to deliver the added value of QoS we assume the Advance Reservation (AR) module (or a substitute providing its functionality) to be present.

3 Enforcing CPU share based SLAs

In this section, we describe the architecture and logic of the AR module of the DWARFS system. The module is comprised of the three main components depicted in Figure 3, namely the Agreement agent, which uses previously gathered statistical data to negotiate agreements and authorizes WS-A-bound service invocations; the Supervisor, which monitors the execution of operations (called tasks), enforcing the requested minimum QoS level by accelerating or slowing down execution of individual tasks; and a Fuzzy Controller [9], used for actual decision-making based on a set of configurable rules. In what follows, we summarize our basic assumptions and focus on the two latter components.

3.1 Model and Basic Assumptions

First and foremost, the goal of predicting the execution time of an operation is actually proven to be unachievable in the general case, as it projects to the halting problem [16]. However, assuming that service operations do deterministically provide a result, we argue that a prediction is in many cases possible based on the extrapolation of past results. This leads to the second assumption that such an extrapolation is possible and reasonable, without considering the actual input

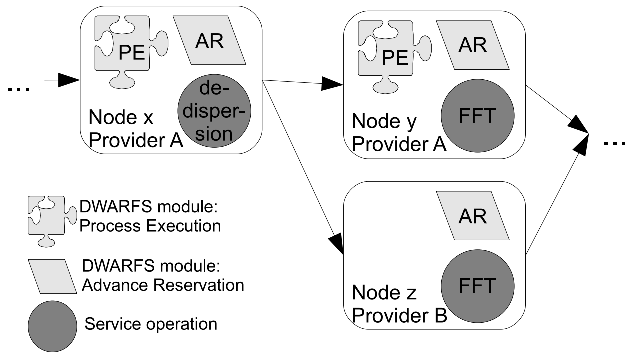

Fig. 2. Sample enactment of a part of the pulsar astronomy workflow using DWARFS

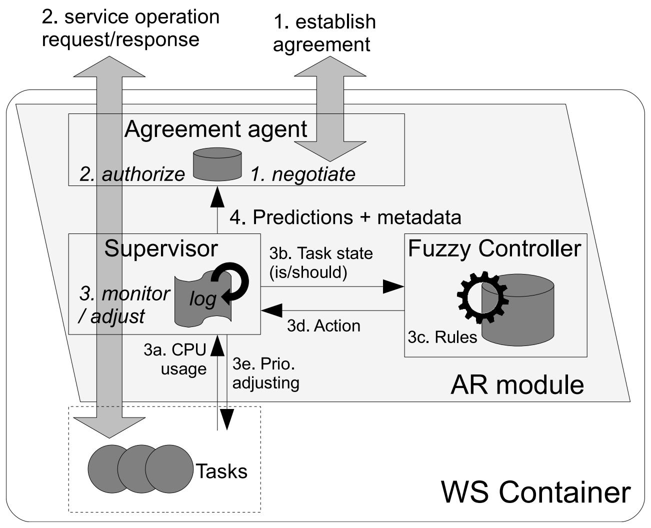

Fig. 3. CPU share controller

data. This might be a limiting factor, and overcoming it or dampening its impact is left for future research. Other factors we do not consider (yet) include the effect of I/O-bound operations (as opposed to the CPU-bound ones we assume), and effects of synchronization and locking in multi-threading calculations.

From a Java program, interactions with the system scheduling are rather limited: to retrieve information about CPU usage, one can only query the number of CPU time all considered threads have used. Similarly, to influence the scheduler, one can only set thread priorities to one of the 10 Java priorities (or, in extreme cases, suspend and resume threads). This results in a rather coarse granularity of possible actions to influence the scheduling.

Our experiments have shown that the actual CPU shares – i.e., the percentage of processing power that threads running at different Java priorities get – are heavily depending on the Operating System scheduler and largely varying between different OS's. Figure 4 shows a representative part of these experiments, where three threads were run in parallel, with the priority of the first thread fixed to 8, and the other two threads taking all possible combinations of priority values from 1 to 8. The resulting CPU shares are depicted textually and graphically, where each thread is represented by a different color. The fact that not all possible requested share combinations can be accomodated by a fixed combination of Java priorities (thus requiring adjustments at runtime), and the rather big differences in behavior of the schedulers among different OS's were the main motivation for using a fuzzy controller to dynamically adjust the priorities at runtime.

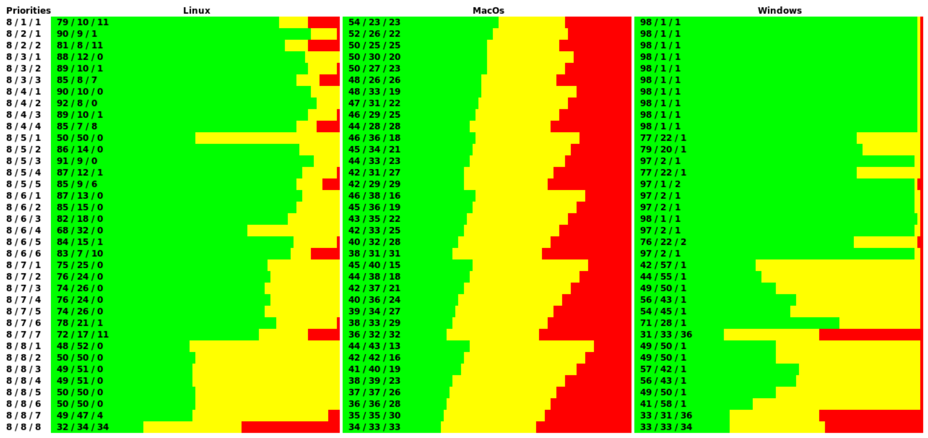

Fig. 4. Mapping of Java thread priorities to effective CPU shares on different OS's

The way of gathering information about CPU usage in Java has another implication: to determine the effective CPU percentage of a thread, one has to sum up all threads' used CPU times to determine the 100% ratio, and then to calculate the actual shares.[1]

Figure 5 depicts this relationship and the overhead introduced by various other parts of the system. Fig. 5 (a) represents 100% of the physical CPU available. In Fig. 5 (b), the overhead introduced by the OS, the JVM, and the AR module itself are depicted. Finally, Fig. 5 (c) shows what the supervisor would see as 100% if the system was not fully loaded – this is caused by the way the calculations are performed, as explained above. However, because we only rely on relative shares, the reasoning is still correct regardless of the actual load on the system. Note that the figure is not drawn to scale, but purely illustrational – while we cannot reliably measure the OS and JVM overhead, we expect them to be rather low, and our measurements have shown that the overhead of the supervisor, in terms of CPU usage, is negligible.

3.2 Maximum Task Share Calculation

Each operation call will result in one or more threads running, which we define as constituting a *task*. With multi-CPU machines, an additional factor has to be taken into account: On a machine with P processors, the maximum achievable share of a task t with n threads is $s^t_{max} = min(1, \frac{n}{P})$. If the system allowed reservations for more than s^t_{max}, the task could not be able to achieve the expected share, resulting in an erroneous slowdown of other simultaneously running tasks. For instance, on a dual-CPU machine with two threads running at full speed, each thread will run on one CPU – a reservation combination of 70%/30% will

[1] Shares are represented as real numbers in the range $[0, 1]$ in the model. However, to ease the understanding, we mostly use the equivalent percentage representation.

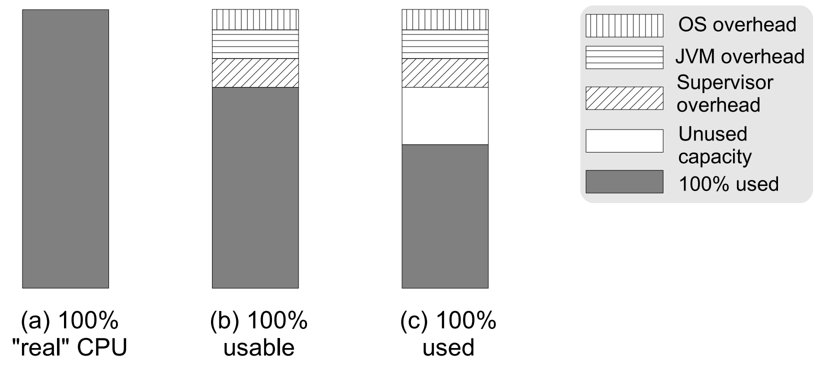

Fig. 5. CPU shares and overhead

result in the first thread never being able to achieve its envisaged goal, but to be blocked at (a maximum of) 50%. The second thread, however, is also not abiding to its 30%, because no matter how low the priority is, the thread will utilize the otherwise unused CPU and run at 50%. It is therefore crucial to know s^t_{max} for a given operation before accepting a reservation request for s^t_{req}, so that these limits can be enforced. This results in the requirement to know the number of threads a given operation will run – which could be provided by the service description, or determined empirically from past executions as well.

3.3 Monitoring and Control of CPU Shares

On task startup, the supervisor gets the necessary metadata (requested CPU share, and number of threads) from the agreement agent. The supervisor, in conjunction with the fuzzy controller, periodically performs the following calculations (labeled *3a* – *3e* in Figure 3) to monitor and control execution for the set T of currently active tasks:

- Calculating the current expected shares (s^t_{cur}) of all tasks, and adjusting them so that $\forall t \in T : s^t_{req} \leq s^t_{cur} \leq s^t_{max} \wedge \sum_{t \in T} s^t_{cur} = 1$. Note that this implies that tasks may get more resources – and thus finish faster – than requested. The objective of the supervisor is to avoid tasks getting too few resources.
- Gathering the CPU usage, and computing the actual share $s^t_{act} \forall t \in T$. So, while s^t_{cur} represents the currently expected share for a task, s^t_{act} is the currently measured share.
- Passing s^t_{act} and s^t_{cur} to the fuzzy controller, and possibly adjusting the thread priority for all of the task's threads in response to the controller output.

Note that the supervisor does not address a "full-fledged" scheduling problem (i.e., it neither has to, nor wants to, assign exactly which tasks have to be run at which moment, which anyway would require to entirely replace the OS or the JVM scheduler). Instead, it merely modifies the priorities of tasks so that their overall CPU consumption matches the requested one.

3.4 Fuzzy Controller Details

The controller used in DWARFS is actually a generic fuzzy controller built for the purpose of, but not limited to usage in, DWARFS. It is completely (re-)configurable at runtime (i.e., all the logic performed, such as getting or setting the system state, fuzzification of parts of it into fuzzy values, fuzzy rule evaluation, and defuzzification, is configured declaratively), and supports detailed logging of the system state. A UI provides users with the ability to perform the configuration, as well as to "replay" and single-step through logs for analyzing them.

In DWARFS, we use 25 rules that evaluate two fuzzy input variables, namely $badness = f(s^t_{act}, s^t_{cur})$, representing the deviation of the actual vs. the expected state, and *tendency*, which reflects the derivation of badness over time. The rule conclusions modify the output variable *action*, which corresponds to the change in thread priorities (-10 to 10) to perform. The following is a textual representation of one of the rules used: `IF badness is overspent_high AND tendency is dropping_slowly THEN action is lower_little`.

3.5 Predicting execution times

After a task has finished, the supervisor aggregates the log information about the elapsed times and CPU usage for the execution and hands this information to the agreement agent, which in turn uses it for future predictions for the operation during agreement negotiation. If task t had run for n intervals with different expected shares (s^t_{cur}), its overall execution E_t can be represented as a set of n time slices $\tau_i = \langle \delta_{\tau_i}, \sigma_{\tau_i} \rangle$, where $\delta_{\tau_i} \in \mathbb{N}^+$ is the duration of the ith slice, and $\sigma_{\tau_i} \in (0, 1]$ is the corresponding actual CPU usage. The predicted execution time for t is then calculated as $PE_t = \frac{1}{s^t_{max}} \sum_{i=1}^{n} \delta_{\tau_i} \sigma_{\tau_i}$. This prediction can be linearly scaled if shares other than s^t_{max} are requested.

4 Evaluation

For evaluation and comparison purposes, we repeatedly (15 times) ran the following configuration: The same CPU-intensive operation (repeatedly calculating SHA-512 hashes, as a representative of a purely CPU-bound and expensive calculation) is run as 6 different tasks, started at different times and with varying requested priorities. This setting was chosen since it contains most of the interesting aspects of a real-life setting, i.e., tasks starting at "random" times (also at the same time), high-priority tasks intercepting lower-priority ones, tasks acquiring additional (otherwise idle) CPU resources, etc. All tests were performed on the same computer, running on Ubuntu 8.04 (64-bit) and Windows XP SP2 (32-bit), in a normal, not otherwise loaded configuration. In all cases, a Sun JVM 1.6 has been used, and the control loops were effectuated every 500 ms.

Figure 6 depicts the evolution of the system state, as seen from the controller. For each task t, s^t_{cur} (*should*) and s^t_{act} (*is*) are depicted. An important point is that *should*-values are adjusted as tasks join and leave, defining the slice boundaries and resulting in a stair-case-like *should* curve. The controller tries to keep *is* as close to *should* as possible. The oscillations at the boundary start are caused by the fact that each adjustment of the target values (*should*, or s^t_{cur}) results in the need to take the boundary as the new starting point for share calculation, thus starting the calculations "from scratch". Naturally the resulting coarse granularity of input data, paired with few reference intervals, cause a greater imprecision in the calculations and therefore peaks in the representation. In fact, a more intuitive representation of the system state – and more insight into the effectiveness of the controller – is gained by accounting for the performance during previous timeslices, which is done by calculating *as* and *ai* as the average of all *should* (respectively *is*) values over the complete lifetime of the task. These aggregated values are depicted in Figure 7.

Table 1 presents the evaluation of our measurements. The uncontrolled execution time corresponds to the task being run as a standalone application outside of the controller and serves as a control variable. While we cannot explain the striking difference in execution times between Windows and Linux (possibly

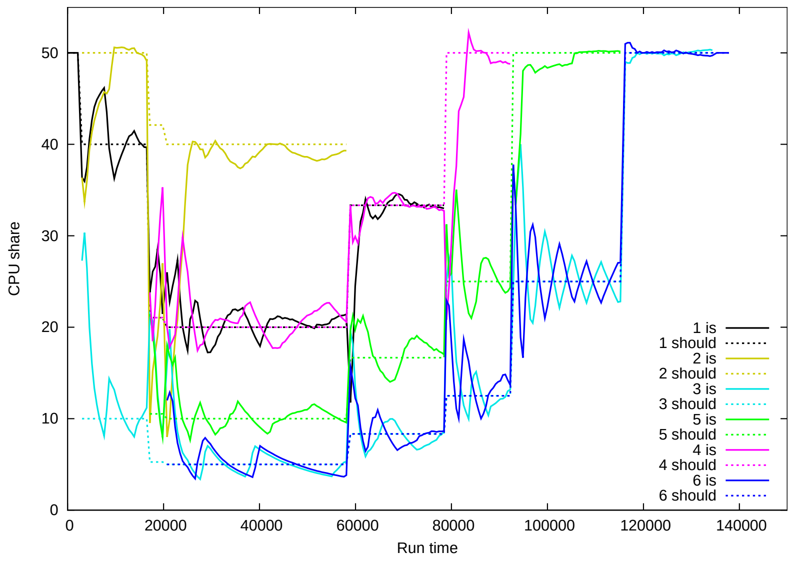

Fig. 6. System state evolution during CPU share controller run

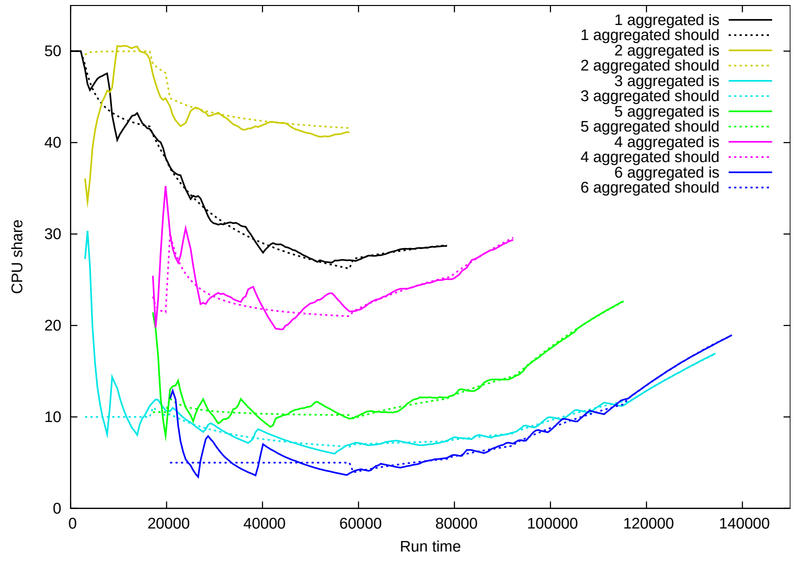

Fig. 7. System state evolution (aggregated shares)

caused by the difference between 32 and 64 bit mode), it is actually helpful for analyzing the effect of longer task run times.

The most important functional quality criteria are the absolute and relative errors, which correspond to an inability of the system to enforce the requested reservations. The results indicate that indeed it is possible to enforce reservations, with the quality of the enforcement and the predictions improving with the duration of a task. The price to pay is a performance penalty, as shown by the predictions. The predictions are generally slower than in the uncontrolled case, as (mostly low-priority) tasks overspending their assigned shares have to repeatedly be suspended so that other tasks meet their target shares. For the sake of reducing this penalty, we tested a configuration that disallowed the suspension of tasks. As shown in the last column, this reduces the overhead, but results in a substantial decrease in quality: most importantly, tasks with higher

Table 1. Evaluation results

Item (averaged over 15 runs)	*Windows*	*Linux*	*Linux (no suspend)*
Uncontrolled execution time (ms)	216829	41361	41361
Coefficient of variation for uncontrolled ex. (%)	0.99	2.19	2.19
Predicted execution time (ms)	221290	48502	44557
Coefficient of variation for prediction (%)	2.05	5.01	2.74
Factor prediction/uncontrolled	1.02	1.17	1.07
Average absolute deviation $\|ai - as\|$ (%)	0.63	0.79	3.26
Average relative error $\frac{\|ai-as\|}{as}$ (%)	5.41	5.96	30.56

priority never achieved their target share, while low priority tasks constantly overspent CPU time.

5 Related Work

Systems targeting the problem of orchestration of resources, in conjuction with QoS criteria, are given a lot of attention predominantly in the Grid community, where the provisioning of resources such as storage or processing capacity is a key aspect. A detailed survey on such systems is presented in [19]. Notably, ASKALON [8] provides a tool set that is focused on measuring, analyzing, and predicting performance aspects of grid applications. The VIOLA project provides support for co-allocation of resources, such as storage, network, and computational resources using SLAs, as described in [13]. Within the GRIDCC project [14], a language for specifying QoS requirements at the level of entire BPEL workflows has been defined. In the context of QoS for workflows, [11] addresses the configuration of the entire system environment, including the dynamic selection and deployment of service instances.

As WS-Agreement seems to emerge as the de-facto standard to describe and negotiate SLAs, some weak points concerning dynamic re-negotiation of agreements (which is particularly relevant for workflows) have been pointed out [1, 17]. [15] proposes modifications to the WS-A specification to support completely dynamic renegotiation of SLAs.

The actual enforcement of the requested QoS criteria – i.e., the assignment of shares of processing power to tasks – is a problem closely related to scheduling, a domain targeted extensively by the Real-Time and Embedded Systems community; an overview of this field is given in [12]. While the DWARFS AR component may well benefit from having an optimized scheduler available at the JVM and/or OS level, the approach of using a Fuzzy Controller on top of the existing scheduler helps us achieve the goal of being both non-invasive (not requiring changes to the underlying system) and flexible (functioning with any kind of underlying OS and JVM scheduler).

Concerning the control mechanisms used to measure and predict CPU usage, the J-RAF framework [5] uses an innovative bytecode instruction counting approach. Whereas this results in accurate measurements, it requires patching of all classes to be executed, and the results are not easily projected to actual wall-clock runtime, which is the target of our work. [10] is targeting the prediction of memory consumption of operations, a topic that, albeit not covered by our approach, would present a useful addition to extend the managed QoS criteria.

6 Summary and Future Work

In this paper, we introduced our DWARFS approach to an AR-supporting decentralized workflow execution engine. In particular, we have presented one of its fundamental modules, which enables us to enforce certain computational QoS criteria at the scheduler level. While these first evaluation results are encouraging, there are still challenging open questions that require further research. This includes the configuration of the fuzzy controller, namely the calculation of the variables and the rulesets, and possibly the evaluation of alternative strategies for controlling combinations of tasks, instead of individual tasks only. Further work will then re-consider the limiting basic assumptions, their practical relevance and possible ways to overcome them or to minimize their impact. At the same time, the implementation and integration of the system will be carried on, so that the focus can be shifted to the larger-scale problems of AR strategies at the level of entire workflows in a distributed setting.

References

1. M. Aiello, G. Frankova, and D. Malfatti. What's in an Agreement? An Analysis and an Extension of WS-Agreement. In *ICSOC*, pages 424–436, 2005.
2. A. Alves, A. Arkin, S. Askary, C. Barreto, B. Bloch, F. Curbera, M. Ford, Y. Goland, A. Guízar, N. Kartha, C. K. Liu, R. Khalaf, D. König, M. Marin, V. Mehta, S. Thatte, D. van der Rijn, P. Yendluri, and A. Yiu. Web Services Business Process Execution Language Version 2.0. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html, April 2007.
3. A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, T. Nakata, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu. Web Services Agreement Specification (WS-Agreement). http://www.ogf.org/pipermail/graap-wg/2006-July/000457.html, July 2006.
4. J. Brooke, S. Pickles, P. Carr, and M. Kramer. Workflows in Pulsar Astronomy. In *Workflows for e-Science*, pages 60–79. Springer London, 2007.
5. A. Camesi, J. Hulaas, and W. Binder. Continuous Bytecode Instruction Counting for CPU Consumption Estimation. In *QEST '06: Proceedings of the 3rd international conference on the Quantitative Evaluation of Systems*, pages 19–30, Washington, DC, USA, 2006. IEEE Computer Society.
6. L. Candela, F. Akal, H. Avancini, D. Castelli, L. Fusco, V. Guidetti, C. Langguth, A. Manzi, P. Pagano, H. Schuldt, M. Simi, M. Springmann, and L. Voicu. DILIGENT: integrating digital library and Grid technologies for a new Earth observation research infrastructure. *Int. J. on Digital Libraries*, 7(1-2):59–80, 2007.
7. L. Candela, D. Castelli, C. Langguth, P. Pagano, H. Schuldt, M. Simi, and L. Voicu. On-Demand Service Deployment and Process Support in e-Science DLs: the Diligent Experience. In *DLSci06*, pages 37–51, 2006.
8. T. Fahringer, A. Jugravu, S. Pllana, R. Prodan, C. S. Jr., and H. L. Truong. ASKALON: a tool set for cluster and Grid computing. *Concurrency - Practice and Experience*, 17(2-4):143–169, 2005.
9. B. R. Gaines. Fuzzy reasoning and the logics of uncertainty. In *Proceedings of the sixth international symposium on Multiple-valued logic*, pages 179–188, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.
10. O. Gheorghioiu. Statically Determining Memory Consumption of Real-Time Java Threads. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 2002.
11. M. Gillmann, G. Weikum, and W. Wonner. Workflow Management with Service Quality Guarantees. In *In Proceedings of the 2002 ACM SIGMOD Int. Conference*

on Management of Data, pages 228–239, Madison, Wisconsin, June 2002. ACM Press.

12. J. Leung, L. Kelly, and J. H. Anderson. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis.* CRC Press, Inc., Boca Raton, FL, USA, 2004.
13. H. Ludwig, T. Nakata, O. Wäldrich, P. Wieder, and W. Ziegler. Reliable Orchestration of Resources using WS-Agreement. In *HPCC*, pages 753–762, 2006.
14. A. S. McGough, A. Akram, L. Guo, M. Krznaric, L. Dickens, D. Colling, J. Martyniak, R. Powell, P. Kyberd, and C. Kotsokalis. GRIDCC: real-time workflow system. In *WORKS '07: Proceedings of the 2nd workshop on Workflows in support of large-scale science*, pages 3–12, New York, NY, USA, 2007. ACM.
15. G. D. Modica, V. Regalbuto, O. Tomarchio, and L. Vita. Dynamic re-negotiations of SLA in service composition scenarios. In *EUROMICRO-SEAA*, pages 359–366, 2007.
16. P. P. Puschner and C. Koza. Calculating the Maximum Execution Time of Real-Time Programs. *Real-Time Systems*, 1(2):159–176, 1989.
17. R. Sakellariou and V. Yarmolenko. On the Flexibility of WS-Agreement for Job Submission. In *MGC '05: Proceedings of the 3rd international workshop on Middleware for grid computing*, pages 1–6, New York, NY, USA, 2005. ACM.
18. C. Schuler, C. Türker, H.-J. Schek, R. Weber, and H. Schuldt. Scalable peer-to-peer process management. *Int. J. of Business Process Integration and Management*, 1:129–142(14), 8 June 2006.
19. J. Seidel, O. Wäldrich, P. Wieder, R. Yahyapour, and W. Ziegler. Using SLA for Resource Management and Scheduling - A Survey. In *Grid Middleware and Services - Challenges and Solutions*, CoreGRID Series. Springer, 2008. Also published as CoreGRID Technical Report TR-0096.