Adaptive Performance Control of Internet-based Grids in a Dynamic Environment

Paolo Vercesi
ESTECO
Trieste, 34012, ITALY
vercesi@esteco.it

Alberto Bartoli
DEEI
University of Trieste
Trieste, 34127, ITALY
bartolia@univ.trieste.it

Abstract— **Organizations are increasingly aggregating their computing resources to form Internet-based grids specialized in specific application workflows and made available to other organizations. The scheduling of jobs in such grids may clearly have a substantial impact on performance, but finding effective scheduling policies is hard due to the very same nature of this scenario. Performance may greatly depend on a myriad of parameters whose values can hardly be determined in practice. Moreover, the load injected by users is typically unpredictable, performance of Internet links may vary widely during an execution and computing resources at participating organizations could also vary dynamically, perhaps because of additional workloads injected by other competing activities.**

In this paper we propose mechanisms and policies for controlling the scheduling of jobs in such a highly dynamic environment. We attempt to minimize the resource usage at the participating organizations while maintaining the performance delivered to clients at an acceptable level. Our approach consists of a form of admission control at the entrance point of the application workflow that is simple to deploy in practice and does not need any hook from the participating organizations. We simply vary dynamically the maximum number of jobs that can be injected within the grid, based on performance measures taken on line, and delay excess jobs.

We have evaluated our proposal in detail, by simulation, focussing on its ability to adapt automatically to perturbations in the form of substantial and unexpected changes in the amount of computing resources available. We have found that our proposal is indeed capable of finding automatically a suitable trade-off between throughput and resource usage, even in such a dynamic scenario.

I. Introduction

Scientific and engineering applications are increasingly fulfilling their computation and data demands by means of distributed resources owned by collaborating organizations [1]. Typical scenarios in this area consist of the composition of multiple stages of processing, in which each stage is an application code specialized in a specific computation [2]—e.g., finite elements of fluidodynamic analysis. Such *application-level workflows* may also be accessed as a high-level *composite service* whose invokers need not be aware of its internal structure. Thanks to the plenty of protocols enabling programmatic interactions across different organizations connected to the Internet (e.g. web services and grid computing), the individual stages of a workflow may be exported by different organizations and the resulting workflow made available to clients of other organizations [3], [4], [5], [6]. Organizations are thus increasingly aggregating their resources to form an Internet-based grid specialized in a specific workflow and made available to other organizations.

In this work we are concerned with the scheduling of workflow invocations, hereinafter *jobs*. The rate at which jobs are injected in the grid may clearly have a substantial impact on performance—it is crucial to avoid congestion of the grid resources as well as to avoid their underutilization. Coping with this issue in such a multi-organization, multi-tiered and geographically dispersed environment is obviously hard. The load injected by users of the grid is typically unpredictable. Computing resources available for executing jobs at each organization could vary dynamically, perhaps because such resources are shared with other applications. Performance of the Internet links may vary dynamically, due to the very nature of the Internet.

In this paper we propose mechanisms and policies for controlling the scheduling of jobs in such a highly dynamic environment. We take the point of view of the organizations participating in the grid, in the sense that we attempt to minimize the resource usage at the participating organizations while maintaining the performance delivered to clients at an acceptable level. This perspective is crucial in our scenario, where the grid workload consists of many large and resource-consuming jobs that are submitted in batches and do not require interaction with human users. Key feature of our approach is the emphasis on unpredictability of the available resources, which calls for the ability to adapt automatically to their dynamic changes. Such an unpredictability includes the absence of any form of collaboration from the participating organizations. For example, we do not assume that when an organization is about to begin the execution of another competing application, thereby decreasing the available resources, it will notify the other organizations or the scheduling machinery. While this assumption may clearly limit the performance that can be obtained, we believe it is a practical way to aggregate organizations in a single composite service.

Our proposal consists of a simple module to be deployed at the entrance point of the grid. This module implements a form of admission control: it defines an upper bound on the number of jobs that can be running within the grid and delay excess jobs. The number of jobs that can be injected into the composite service is *varied dynamically* with an

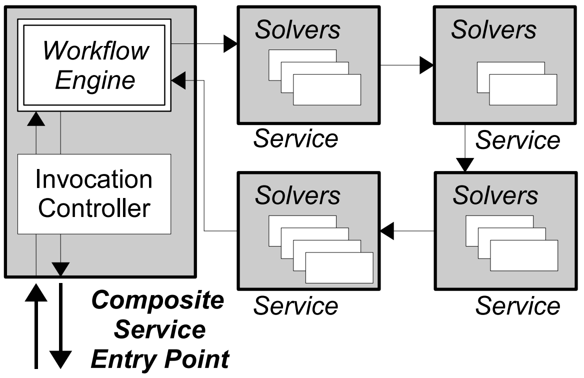

Fig. 1. A composite service composed of a pipeline of four services. All communication between services, workflow engines and clients (not shown here) occurs through the Internet. The invocation controller will be described in Section III-A

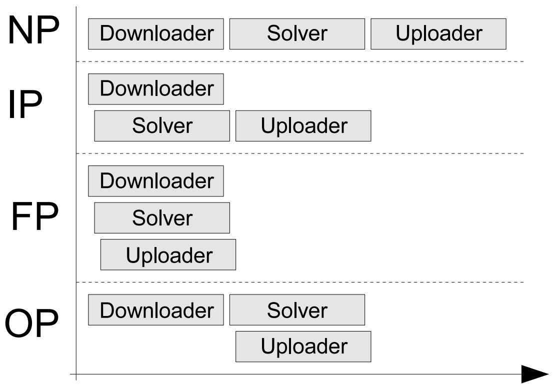

Fig. 2. Data Delivery Patterns

adaptation policy driven by the current estimates of latency and throughput for the composite service as a whole. We remark that our proposal can be implemented simply and, in particular, it handles the composite service internals as a black-box. We evaluated our technique in a broad range of scenarios, by means of a detailed event-driven simulator. These experimental results suggest that our proposal can provide significant benefits.

II. Application Domain

Clients submit jobs to a composite service exposed to the Internet. Execution of a job involves the execution of one or more *services* exported by potentially different organizations connected through the Internet. There are usually data dependencies between services, e.g. the output data produced by a service constitute the input data of another service. For simplicity but without loss of generality we assume that all jobs require the same steps within the composite service, that is, they all require the execution of the same services with the same data dependencies. Our experiments are based on a pipeline of four services. Execution of jobs is orchestrated by an engine that invokes services as appropriate and coordinates the movements of (potentially large amount of) data between services.

Each service encapsulates a *solver*, i.e., an application code specialized in a specific computation, e.g., finite elements of fluidodynamic analysis. A solver is often a commercial licensed software, which means that the maximum number of solver instances that can be run simultaneously is bounded by the license agreement. A solver execution usually proceeds as follows: it obtains an available license, reads the input data, performs the specific computation without any interaction with other entities (in particular, without interacting with other solvers), and finally it produces the corresponding output data, see Figure 1.

The service code wrapped around the corresponding solver takes care of all the necessary communication and synchronization activities, including the transfer of input/output data. We analyzed in our experiments four *data delivery patterns* leading to different degrees of overlapping between computation and communication. With the *NoPipe* pattern, the service starts downloading input data only when the preceding solver has terminated the execution; the solver starts only when the input data is fully available locally. The *InputPipe* pattern is similar, except that the solver starts as soon as some input data is available locally. In both these patterns the transfer of output data can start only when the solver execution has completed. In the next patterns, instead, such transfer begins as soon as some output data are available. With the *OutputPipe* pattern, the service starts downloading input data as soon as some output data are available from the previous solver; the solver starts only when the input data is fully available locally. Finally, the *FullPipe* pattern is similar, except that the solver starts as soon as some input data is available locally. Figure 2 summarizes these concepts. Recall that a solver acquires a license as soon as it starts execution.

We consider a workload consisting of a specified number of jobs, called a *session*. The number of jobs concurrently being processed is called *batch size*. The *job latency* is the time required for completing a job, from its submission to the composite service to its termination. The *solver time* is the time required for one solver execution: it includes the reading of all the input data of a job from the local disk, their processing and the writing of all the output data for the job on the local disk. The *service time* is the time spent by one of the composing services for completing a job. This time includes the downloading of input data, the solver time, the uploading of output data. Finally, the *composite service time* is the sum of the service times for each service involved in the workflow. In our experiments all the latency and time results refer to the average values computed in a session. We measured *throughput* as the ratio between the number of jobs in a session and the time for completing all of them. We remark that predicting the relation between these indexes is hardly feasible, even in a system where relevant parameters never change during execution—e.g., the transfer of data may overlap with computation and the system may be concurrently executing several jobs.

We take the point of view of the organizations participating in the composite service implementation. We assume that, for each organization, the cost in processing a job is proportional to the service time for that job, i.e., to the time actually spent by the job at that organization. In our experiments we will consider the cost globally incurred by all the organizations, i.e., the composite service time. Our aim is minimizing this time without affecting throughput substantially.

III. Our Approach

We analyzed in earlier works the relation between throughput, composite service time and batch size in the application domain of our interest [7], [8]. The results were as expected. Throughput grows more or less linearly with batch size, until reaching a saturation point starting from which throughput remains constant irrespective of the batch size (depending on the system parameters throughput may collapse for very large batch size values). Composite service time grows linearly, or more than linearly, with batch size. A key finding of such analysis, though, was that the actual shape of the curves—slope, saturation point, maximum throughput and alike—is *greatly* influenced by a myriad of factors—e.g., number of licenses available, bandwidth, size of input and output data, data delivery pattern and alike. It follows that figuring out the particular batch size which, in a particular real environment, maximizes throughput while keeping composite service time as small as possible is very hard—recall that in our framework the composite service time is the main cost index from the point of view of participating organizations. The task becomes practically impossible if one considers that the environment usually changes dynamically and unpredictably due to the very nature of the Internet and of Internet-based services, where fluctuations in load and traffic are rather common events.

Motivated by the above we proposed a solution in [7] in which an *invocation controller* is placed at the entrance point of the composite service. This module selects dynamically a value for the batch size and implements a form of admission control to ensure that the number of in-progress jobs does not exceed that value. Excess jobs are queued by the invocation controller, that will inject them into the composite service as prior jobs complete. In the cited paper we proposed one specific algorithm for controlling the batch size and analyzed its performance in a completely static environment, i.e., one in which the factors that may affect performance never change during execution. In this work we implement two new control algorithms suitable for a *dynamic* environment, and we analyze their performance in the presence of unexpected perturbations taking the form of sudden and transient variations in the amount of computing resources available. This scenario is meant to model the injection of additional workload, due either to other composite services sharing resources with the composite service being controlled, or to internal activities of the participating organizations. Such a form of basically unpredictable dynamic behavior is an intrinsic feature of the Internet and of Internet-based services involving multiple and independent organizations.

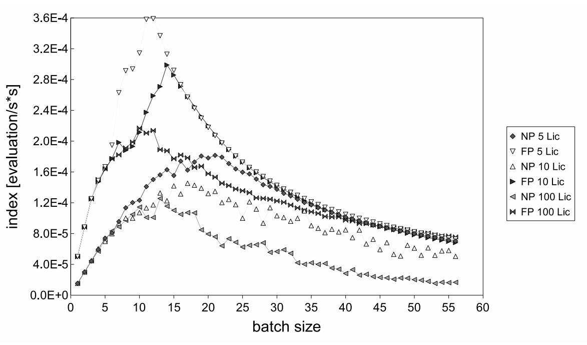

Fig. 3. Objective with constant batch size

Our control algorithms are based on two key observations. First, latency is proportional to the composite service time. This is important because we do not assume any hooks from services, thus we cannot measure the time that a job spends at each organization. Although the proportion between latency and composite service time may vary widely, minimizing the former implies minimizing the latter. Thus we can use latency—that we can measure—as a substitute for composite service time—that we cannot measure. Second, the ratio $\frac{throughput}{job\ latency}$ (that we call *objective index*) usually varies with batch size according to a bell-shaped curve (Figure 3). The actual curve depends on many factors, but its shape is of the form in the figure. The crucial property of the objective index is that its peak occurs at a batch size value very close to the cost-optimal value, i.e., the batch size value that exhibits minimum composite service time amongst those that exhibit maximal throughput. Based on these observations, our control algorithms evaluate the objective index by measuring latency and throughput. They vary the batch size dynamically so as to maximize the objective index, thereby keeping the batch size very close to its cost-optimal value.

We remark again that our approach treats the system as a black box. The invocation controller does not need any prior or run-time knowledge about the static or dynamic factors affecting performance: it simply observes, through its sensors, start time and completion time of each job. Note also the the composing services does not need to provide any hook to the invocation controller. Finally, we note that it is not possible to control the batch size by reasoning on either throughput or job latency taken in isolation. The curve throughput vs. batch size exhibits a flat or nearly-flat region beyond the saturation, thus maximizing throughput would lead to very wide oscillations of the batch size value. The job latency instead grows more or less steadily, thus minimizing this index would invariably keep the batch size to zero.

A. Adaptive Invocation Controllers

The *TIMED controller* works with the aid of two dedicated sensors. The latency sensor measures the job latency averaged across the last batchSize jobs that have completed. The throughput sensor measures the throughput across the same set of jobs. The controller runs periodically, with a period equal to the current job latency (as measured by the sensor). At each run it updates the $batchSize$ as follows. The update algorithm determines the current value for the objective index as provided by the sensors and determines the relative difference with the prior value (rel). The algorithm also maintains a boolean state variable ($delta$) that is true when the batchSize is growing and false otherwise. To follow the bell-shaped curve of the objective index, $batchSize$ is increased when rel is positive and $delta$ is true, or when rel is negative and $delta$ is false. Conversely, $batchSize$ is decreased in the two other cases. The amount of the variation applied to $batchSize$ is proportional to rel (10 times in our experiments) and bounded by 5. In order to filter out changes in the measured objective index that are too small to be truly significant, we actually modify $batchSize$ only when the absolute value of rel is greater than 10% of the current objective index value.

The *TCP like controller* incorporates some ideas of the TCP congestion control algorithm. The controller includes a state variable called *threshold* which determines the amount of the increment. If the current batch size is below the threshold, the controller increments the batch size by one (which mimics the slow start phase of TCP congestion control). If the current batch size is above the threshold, the controller increments the batch size by $1/batchSize$ (which mimics the congestion avoidance phase of TCP congestion control).

Whenever a job starts, the controller associates a timeout with the job, whenever the number of jobs not completing whitin the associated timeout exceeds 50% of the current bach size the controller halves the batch size and it sets the threshold to the current batch size. Initially, the batch size is 1 and the threshold is set to maximum batch size allowed by the admission controller (100 in our experiments). In early experiments we adopted an approach closer to that of TCP, i.e., one single timeout was enough to trigger a batch size decrease, but we found this approach to be excessively conservative. In other words, the batch size decreases when too many of the jobs that have been injected do not complete by their timeout.

The timeout choice is done as follows. This controller also works with a latency sensor and a throughput sensor. In this case, however, the former measures the latency of the last completed job whereas the latter provides a throughput estimate that is updated whenever a job completes. When a job J completes, the current throughput estimate becomes the batchSize that was current when J started divided by the latency experienced by J.

This controller attempts to follow the bell-shaped curve of the objective index in a way quite different from the previous one. The value of $objIndex$ at step i, denoted $objIndex_i$, is $\frac{batchSize_i}{lat_i^2}$ (this follows immediately from the working of the sensors). If one requires that $objIndex_{i+1} > objIndex_i$, straightforward calculations lead to $batchSize_{i+1} > objIndex_i \times lat_{i+1}^2$, thus to $lat_{i+1} < \sqrt{\frac{batchSize}{objIndex_i}}$. The timeout for the i-th job is set to the calculated lat_{i+1}. The reason is because if the job has not completed by that deadline then the objective index will not increase. In order to increase the stability of the controller, the above formulas actually make use of a low-pass filtering of the objective index estimates.

TABLE I
LINK PARAMETERS

link speed	bandwidth mean [KB/s]	bandwidth std dev	latency mean [ms]	latency stddev
high	977	97.7	0.1	0.01
low	97.7	9.77	1	0.1

TABLE II
WORKING POINT

Parameter	Value
Buffer Cache Size	100000
Minimum Disk Read Time [s]	0.001
Maximum Disk Read Time [s]	0.002
Minimum Disk Write Time [s]	0.0002
Maximum Disk Write Time [s]	0.0004
Sector Size [B]	8192
Initial Data Size [B]	1000000
Start Up Time [s]	30
Elaboration Speed [B/s]	65536
Output/Input Ratio	1.1

IV. EXPERIMENTS & RESULTS

We modeled the system as described below and simulated it with DESMO-J, a discrete event simulation framework written in Java [9]. The setting of the simulation is the same as in [7], we summarize it here for completeness. Each process runs on a host and each pair of hosts is connected by a communication link. Each service provider runs on a different host. Invocation controller and workflow engine run on the same host and this host is not shared with any service. Each service consists of an uploader, a downloader and a solver. A solver elaborates input data at a rate specified by its *elaboration speed*. A service outputs an amount of data given by the size of input data multiplied by the *output/input ratio*. Hosts have unlimited memory and, as commonly found in practice, a number of CPUs equal to the number of licenses for the solvers. The number of jobs concurrently being executed by each service is bounded by the number of available CPUs. Processes access local data via files. The modelling of file access is described in detail in [7].

We modeled link delay and bandwidth as random variables with Gaussian distribution. We associated with each link an additional parameter called the number of *active connections*, i.e., connections feeding data into the link. The bandwidth of a link is fairly distributed among all the active connections. Each downloader/uploader creates a connection over a specified link in response to a file transfer request (issued by the workflow engine) then exchanges data along that connection with the other endpoint. The size of each link I/O operation is equal to the size of a disk sector.

We considered a workflow schema composed of four identical services connected as a pipeline — the output data of a service are the input data for the next service. Each experiment is characterized by the following parameters: (i) data delivery pattern, either NoPipe (NP), InputPipe (IP), FullPipe (FP) or OutputPipe (OP) (Section II); (ii) number of licenses, either 10, 20 or 100; (iii) link quality, either Normal, or High (Table I.) The remaining parameters are summarized in Table II, buffer cache size is expressed in number of blocks. Each experiment refers to a session composed of 10000 jobs. We assume that, at any time, there is always at least one job submitted at the composite service and waiting to be executed (except when the session is about to complete).

The focus of our work is the ability of controllers to react and adapt to changes in the environment, i.e., to unexpected changes to parameters that may affect performance. To this end we subdivided each experiment (session of 10000 jobs) in three intervals and introduced a substantial *perturbation* in the mid interval, from the submission of the 3333-th to the termination of the $6667th$ job. We considered a *one-stage* perturbation, in which we decreased the number of licenses available at the second service in the pipeline to 10% of its initial (and final) value. For example, an experiment with 20 licenses is divided in three equal-sized intervals and in the mid interval the number of licenses at the second stage suddenly drops to 2. This perturbation models the injection of an additional competing workload at that stage, perhaps for internal activities of the corresponding organization or perhaps because that organization is sharing its resources with other application-level workflows.

In order to interpret our performance results we determined a baseline for each experiment, based on an hypothetical controller using a statically defined and immutable batch size value. To this end we ran a session of 10000 jobs for each combination of parameters, by keeping the batch size constant during the entire session and by injecting the perturbation as above. For each combination of parameters, thus, we obtained a curve throughput vs. batch size and composite service time vs. batch size. These curves enabled us to identify the throughput-optimal batch size value for that combination of parameters. We remark that, in practice, this value cannot be known, as it would require the knowledge of the perturbation that will affect the session. We also remark that the performance baseline is not necessarily the best that can be achieved: it is the best choice only when the batch size is selected statically.

A. Results for One Stage Perturbation

Our approach implements a trade-off between throughput and composite service time. Throughput may be lower than the maximum that could be achieved, in order to minimize the time spent at each organization. In other words, we execute the same amount of work by consuming less resources, at the price of a longer completion time for a session. We expect from the experiments, thus, results showing a (hopefully small) decrease in throughput and a (hopefully large) decrease in composite service time.

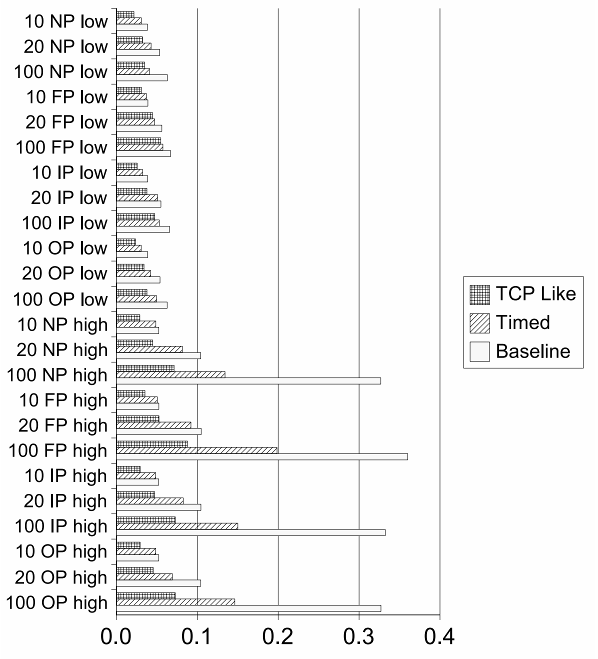

Fig. 4. Throughput for 2nd stage perturbation: low quality links (up), high quality links (down).

Figure 4 and 5 show throughput and composite service time in all the 24 scenarios that we have analyzed. These scenarios are all the combinations of: link quality (low, high), available licenses (10, 20, 100), and the data delivery patterns (NP, FP, IP, OP).

It can be seen that, broadly speaking, the controllers are indeed capable of delivering reasonably good throughput even without any apriori knowledge about the system parameters. Moreover, they do so in spite of the substantial perturbation injected into the system.

With regard to throughput (Figure 4) we recognize that the TIMED controller obtains always greater throughput than TCP like controller for all scenarios. In many cases the TIMED controller throughput is comparable with the baseline and it reachs the 95% of the baseline for 10 FP for both low and high quality links. For low quality links TCP like controller throughput is always above 60% of the baseline. It can be seen that the benefits of our proposal are less significant for high quality links and when the number of license is high.

Concerning the composite service time, Figure 5 shows that for low quality links the TCP like controller composite service time is similar to the TIMED controller composite service time, with the exception of 10 FP they are always lower than the baseline. For high quality links the composite service times obtained by the Timed and the TCP like controller are considerably lower than the baseline, ranging from 10% to 70% of the baseline.

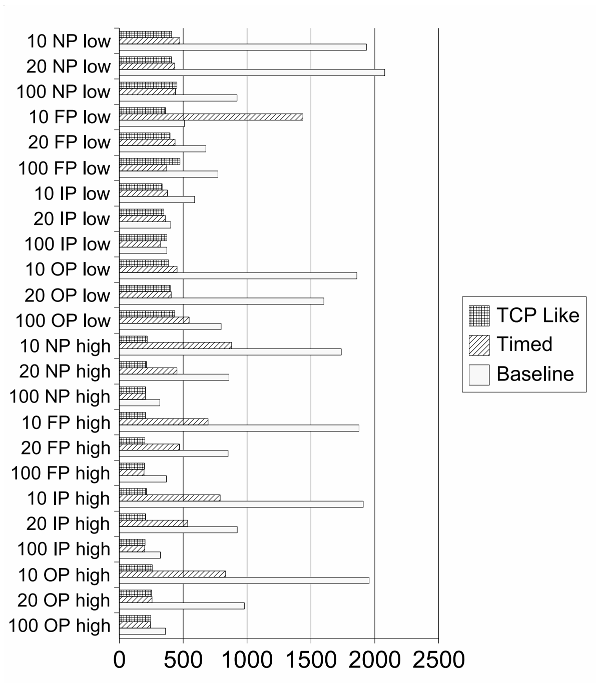

Fig. 5. Composite service time for 2nd stage perturbation: low quality links (up), high quality links (down).

B. Concluding Remarks

The scheduling of jobs in application-level workflows may have substantial effects on performance. Workflows resulting from the Internet-based aggregation of resources exported by multiple organizations are especially challenging from this point of view. The task is complicated further by the very same nature of this scenario, in which there are many performance-critical parameters whose values can hardly be known in advance and that can change dynamically and unpredictably. Additional workloads injected by other competing activities are just one of the many possible causes for this dynamism.

We have addressed the above scenario by proposing two job scheduling policies capable of controlling job scheduling adaptively, based on simple measures on the current status of the system. These policies are based on a mechanism simple to deploy in practice that, in particular, does not need any hook from the organizations participating in the workflow. We have evaluated our proposal in detail, by simulation, and we have found that it is indeed capable of finding automatically a suitable trade-off between throughput and cost, in spite of the fact that both parameters and their variations are unknown. Although we certainly need to broaden our analysis toward further forms of perturbation, we believe that our results are quite encouraging.

REFERENCES

[1] G. Kola, T. Kosar, J. Frey, M. Livny, R. Brunner, and M. Remijan, "DISC: A system for distributed data intensive scientific computing," in *First Workshop on Real, Large Distributed Systems (WORLDS04)*, December 2004.

[2] D. Thain, J. Bent, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. Livny, "Pipeline and batch sharing in grid workloads," in *Proceedings of the 12th IEEE Symposum on High Performance Distributed Computing*, June 2003, pp. 152–161.

[3] F. Casati, S. Ilnicki, L. jie Jin, V. Krishnamoorthy, and M.-C. Shan, "Adaptive and dynamic service composition in eflow," in *Proceedings of the 12th International Conference on Advanced Information Systems Engineering (CAiSE '00)*. Springer-Verlag, 2000, pp. 13–31.

[4] S. Ponnekanti and A. Fox, "SWORD: A developer toolkit for web service composition," in *Proceedings of the 11th International WWW Conference (WWW2002)*, 2002.

[5] B. Srivastava and J. Koehler, "Web service composition current solutions and open problems," in *In Proceedings of ICAP 03*, 2003.

[6] A. Bartoli, R. Jiménez-Peris, B. Kemme, C. Pautasso, S. Patarin, S. Wheater, and S. Woodman, "The adapt framework for adaptable and composable web services," *IEEE Distributed Systems On Line*, September 2005, web Systems Section.

[7] P. Vercesi and A. Bartoli, "Adaptive performance tuning for internet-based workflows," in *Proceedings of 31st Annual IEEE International Computer Software and Applications Conference (COMPSAC 2007)*, July 2007.

[8] ——, "On the performance of inter-oganizational design optimization systems," in *Proceedings of the 38th Winter simulation conference (WSC '06)*, December 2006, p. 29.

[9] *The DESMO–J homepage*, DESMO–J, 2000, http://www.desmoj.de.