Using SchedFlow for Performance Evaluation of Workflow Applications*

Gustavo Martínez[1], Elisa Heymann[1], Miguel Angel Senar[1], Emilio Luque[1], Barton P. Miller[2]

[1]Departament d´Arquitectura de Computadors i Sistemes Operatius
Universitat Autònoma de Barcelona, Spain
gustavo.martinez@caos.uab.es,
{elisa.heymman, miquelangel.senar, emilio.luque}@uab.es,
[2]Computer Sciences Department
University of Wisconsin
bart@cs.wisc.edu

Abstract- **Computational science increasingly relies on the execution of workflows in distributed networks to solve complex applications. However, the heterogeneity of resources in these environments complicates resource management and the scheduling of such applications. Sophisticated scheduling policies are being developed for workflows, but they have had little impact in practice because their integration into existing workflow engines is complex and time consuming as each policy has to be individually ported to a particular workflow engine. In addition, choosing a particular scheduling policy is difficult, as factors like machine availability, workload, and communication volume between tasks are difficult to predict. In this paper, we describe SchedFlow, a tool that integrates scheduling policies into workflow engines such as Taverna, DAGMan or Karajan. We show how SchedFlow was used to take advantage of different scheduling policies at different times, depending on the dynamic workload of the workflows. Our experiments included two real workflow applications and four different scheduling policies. We show that no single scheduling policy is the best for all scenarios, so tools like SchedFlow can improve performance by providing flexibility when scheduling workflows.**

Keywords: **Workflow Management, Scheduling Policies, Distributed Environments.**

I. INTRODUCTION

Workflow-based technologies have become an important aid in the development of complex applications. For convenience and cost reasons, scientists typically run workflow applications [4] [13] in distributed environments such as multiclusters, Grids or Clouds.

A workflow application consists of a collection of jobs to be executed in a partial order determined by control and data dependencies. Although the characteristics of workflows may vary, a simple approach to model a workflow is by means of a directed acyclic graph (DAG).

Executing this type of application requires two components: a workflow scheduling policy and a workflow engine for ensuring the correct execution order of these tasks.

Several workflow engines are currently being used to support the execution of workflow applications on clusters and Grid systems, including Condor DAGMan [1] [2], Taverna [3] [4], Triana [5], Karajan [6] and Pegasus [7].

Traditionally, significant effort has been put into developing scheduling policies for workflows, such as Heterogeneous Earliest Finish Time (HEFT) [8] [9], Balanced Minimum Completion Time (BMCT) [10], Min-Min [11], and DAGmap [12]. However, little attention has been paid to linking workflow scheduling policies with existing workflow engines.

In addition, when studying scheduling policies, most research has been oriented towards evaluating and improving the application's makespan (the time difference between the start and finish of the workflow application). However, that research has not taken into account several factors that may affect the application performance (such as workload size, inaccurate computing and communication times, and machines appearing/disappearing dynamically). And not all of the studied scheduling policies can be reasonably implemented on several of the existing workflow engines.

We used SchedFlow to systematically assess the influence of one of a key factor that affects the application makespan; specifically, we evaluated the effect of varying the initial workload for two different real workflow applications. By varying the initial workload we considered two different initial input data for each of the studied applications (400MB and 1024MB). Not surprisingly, we found that under different workloads, an application will show different behaviors under a given scheduling policy. Therefore, applications can benefit from tools like SchedFlow as they allow applications dynamically to choose different scheduling policies for different applications depending on what is happening at execution time.

SchedFlow allows users to specify a wide variety of workflow scheduling policies through a well-defined API, and integrates the user-defined policy with a workflow engine system. SchedFlow transparently handles the various system events of the underlying workflow engine, and allows the user to focus on the main workflow scheduling algorithm.

Based on a layered architecture, SchedFlow's implementation is portable across different workflow engine

*This work was supported by MEyC-Spain under contract TIN 2007-64974.

systems. Currently SchedFlow works with Condor DAGMan, Taverna, and Karajan.

The remainder of this paper is organized as follows. Section 2 describes the general SchedFlow architecture and Section 3 describes the Task Monitoring system we developed to deal with runtime events. Section 4 presents our experiments and the results we obtained. Section 5 sets out the related work. Finally, Section 6 concludes the paper.

II. SCHEDFLOW ARCHITECTURE

In this section, we describe the architecture and available interfaces in SchedFlow. We also explain how users can integrate new scheduling policies into SchedFlow without changing either the workflow engine architecture or the user application. Figure 1 shows the three main modules (*Controller*, *Observer*, and *Scheduler*), that allow the execution of a workflow's tasks. SchedFlow has two interfaces, one dedicated to the integration of scheduling policies, and another one to link our system to a workflow engine (*Adaptors*).

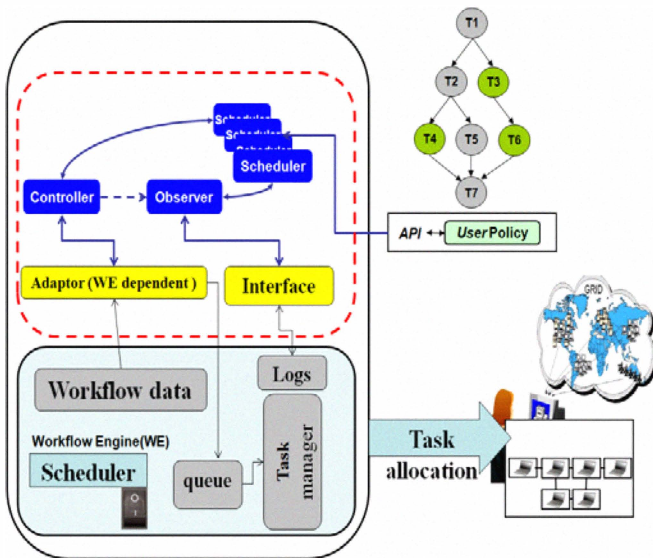

Figure 1. SchedFlow General Architecture.

Controller: This module is responsible for storing workflow tasks in the *task_list*, and sending these tasks to the Scheduler module for planning. The Controller module remains active waiting for the Scheduler module to send the result of mapping the tasks. Once received, tasks mappings are sent to the workflow engine to be executed.

Observer: This module is responsible for managing the resource list used by the Scheduler module to schedule tasks. Additionally, this module monitors the logs in order to get information on the events that affect the tasks that are being executed, and informs the Controller module of the tasks that have finished correctly.

After getting that information, another task is sent to the workflow engine if the scheduling is static or to the Scheduler if the scheduling policy is dynamic. If there is a failure in the execution of a task, the Observer will signal the Scheduler to remap the whole failed task's sub-tree in the static case, or only remap the failed tasks in the dynamic case.

Scheduler: This module is responsible for mapping tasks onto machines. As shown in Figure 1, the *Scheduler* uses the scheduling policies defined by the user. The scheduler returns a list of scheduled tasks according to the scheduling algorithm selected to run the workflow. This mapping is sent to the *Controller* so it can send the scheduled tasks to the workflow engine to be run.

API: SchedFlow provides an API to the user that allows the integration of scheduling policies for workflows without the need to modify the workflow engine. The user has to implement their scheduling policy (or policies) as a C++ function that will call the appropriate methods provided by SchedFlow. This user scheduling policy is loaded as a dynamic library by the Scheduler module, which will invoke it as needed. Both static and dynamic policies can be integrated into SchedFlow.

Adaptors: These modules provide an interface to allow the connection of different workflows engines, taking advantage of the services provided by those workflow engines (such as launching a task for its execution). So far SchedFlow includes adaptors for three workflow engines, namely DAGMan, Taverna, and Karajan.

The *adaptors* link SchedFlow to the different workflow engines. Figure 2 shows the three main adaptor services that are used to obtain the information necessary to SchedFlow.

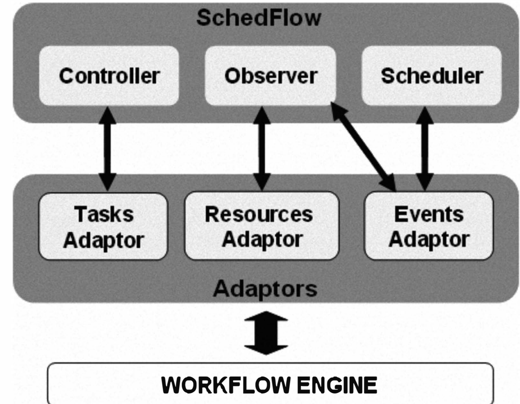

Figure 2. Workflow Engine Interface.

We now explain the services implemented by the *Adaptors*, and shown in Figure 2.

Task's Adaptor: This service interacts with the Controller module, sending out the tasks mapped to the corresponding workflow engine.

a. In the case of Condor DAGMan, the *condor_submit_dag* command is used.

b. In the case of Triana, the *run_workflow* command is used.
c. In the case of Taverna, the *job_submission* command is used.

Resource Adaptor: This service allows the Observer module to obtain the resources available from the execution environment as seen by the each workflow engine.

a. In the case of Condor DAGMan, the *condor_status* command is used.
b. In the case of Triana, the *TrianaV3Resources* directory is used.
c. In the case Taverna, the *g_resource* command is used.

Event adaptor: This service is required for the Observer and Scheduler modules to reschedule tasks when an unexpected event occurs in the machine running them (these events are detailed in Section 3); this information is provided for each workflow engine accordingly.

a. In the case of DAGMan, events are obtained from the information *log* indicating the task's status.
b. In case of Triana, we use the *show_properties* tool of the task's that provides their status at runtime.
c. In the case of Taverna, the *log_book_data* content is used.

III. Task Monitoring

In this section, we describe the task monitoring carried out by the *Observer* to allow SchedFlow to react to different events occurring while the tasks of the workflow are running. The *Observer* has the *log_monitor()* function which allows dealing with dynamic scheduling policies and performing rescheduling. If a task finishes with a state different than *Done,* the *Observer* checks what event happened (events are described later in this section) and reacts accordingly.

In the previous section, we defined the requirements for a workflow engine, and now we describe the flow that each task must follow to be executed. We also describe what happens when an event occurs at runtime.

Runtime events, such as task suspensions, are monitored by SchedFlow. When a task belonging to the workflow is ready to be executed (*ready state*) as its dependencies are satisfied, SchedFlow assigns the task to a computing resource through the Scheduler module, putting this task on the list of mapped tasks (*mapped_list*).

Then, using the Controller module, the task is submitted to the workflow engine, so that it will be run (Run state). When a task completes these steps, it will successfully finish (*done state*) and be put on the *done_list*. Our system also manages unexpected events, by reacting dynamically.

We basically detect, throughout the *Observer* module, two different events:

Suspended Tasks: a task sent to a computing resource may be interrupted and suspended when local processes are run on the executing machine, so that the task is suspended for a random period of time.

The consequence is a delay in the conclusion of the whole application. To support this event, SchedFlow uses the Observer module, so when this event occurs, the task might be rescheduled to a new resource, using the *unmap()* function. The Scheduler is responsible for performing this corrective action.

Failed Task: a task that was sent to a computing resource and running might fail due to a variety of reasons (evicted from the system, due to a machine or network failure, for example). In all these cases, the task will be considered as failed because SchedFlow can no longer wait for it to finish successfully. SchedFlow manages this event by rescheduling the task to a new resource, using the *unmap()* function.

These actions are carried out by SchedFlow in a transparent way and no user intervention is needed. It is worth noting that this Task Monitoring option can be switch off by the user, but that would mean that the user should be aware of the task that fails and resubmit it. Given that workflows may have thousands of tasks, manual error handling can be difficult.

While each workflow engine has its own notation for representing workflows, ShedFlow is capable of reading the these different notations by using the corresponding adaptor (by using the *set_wofkflow()* function). SchedFlow will translate the workflow into an internal structure that contains all the details about workflow tasks and their dependencies. Similarly, the resources, specific to each Workflow Engine, are obtained with the *get_resource()* function, which will translate the information provided by each workflow engine into an internal structure.

Scheduling algorithms for workflows usually need accurate estimates of the execution time of the task on each potential machine, as well as accurate estimates of the communication time between each pair of connected tasks in the workflow. This data is needed because execution time is commonly a function of the size and properties of the input data. And communication times depend on the volume of data transferred.

In the homogeneous case, it can be assumed that each task performs identically on each target machine. Therefore, a single estimate of the execution time of each task is required, and is fairly easy to obtain. This situation, however, is not true for heterogeneous and dynamic systems since an execution time estimate is required for each task-machine pair, and there are many factors unique to heterogeneous systems that can affect the execution time. SchedFlow includes two methods that can be used to include estimates of execution and communication times, (*get_com_t() and get_comm_t()*), respectively.

Unfortunately, current Workflow Engines do not provide such estimates (only some synthetic performance information is provided for available machines). Therefore, execution and

communication time estimates must be computed by external mechanisms. SchedFlow currently includes a simple mechanism based on historical information from past executions and we are currently working on the integration of a more sophisticated method based on nonparametric regression techniques [19].

In next section before describing the experimentation carried out, we comment on some of the factors that have effect on the scheduling policy to choose for a workflow application.

IV. EXPERIMENTAL DESIGN AND RESULTS

In this section we describe our experiments and the results that we obtained.

Different factors may affect the choice of scheduling policy for specific set of tasks belonging to a specific workflow. One of these factors is the initial workload supplied to the tasks. In the experiments that we performed, we confirmed that the size of the workload, strongly influence which scheduling policies performed better, i.e., resulted in a lower makespan. One of our objectives was to quantify this effect by executing two different applications with the same workload engine and similar execution environments, and changing the initial workload and considering different scheduling policies. The environment was one in which 90% of the machines are the same for all the executions, while the other 10% may change dynamically.

We used SchedFlow to run our experiments on a real cluster so these experiments went beyond the classical results obtained by simulation. With SchedFlow the user-defined scheduling policies are applied by the desired workflow engines. Without SchedFlow the user would have to provide a scheduling policy workflow engine interface, and it would be difficult to try a scheduling policy under different engines. So it is common that the user ends up just using the default scheduling policy provided by a workflow engine, and therefore pays a price in performance.

A. *Experimental Environments*

Our experiments were carried out on an opportunistic and non-dedicated environment, composed of 140 Intel-based computing nodes running Linux Fedora Core 5. According to the data benchmark provided by Condor, machine performance in this environment ranged from 0.25 to 0.75 GFlops.

We used SchedFlow for running different scheduling policies feeding different workflow engines. The scheduling policies implemented were Random, Min-Min[11], HEFT [8, 9], and BMCT [10]. The workflow engines used were Taverna, Karajan, and Condor DAGMan. We ran our experiments with two applications, Montage (with 53 tasks) and LIGO (with 81 tasks), and varied the input workload (400MB and 1024MB) and studied the effect of the scheduling policies depending on the workload.

As we had hypothesized, we found that no single scheduling policy worked better for all the scenarios, therefore SchedFlow can be a useful tool that allows scientists to obtain better performance when executing their workflows, allowing it to choose the scheduling policy to be used depending on dynamic factors of a particular run.

In our experiments we used the rescheduling option of SchedFlow, which means that in case of unexpected events (suspensions or failures) the associated task is rescheduled and executed on a different machine.

B. *Workflow Applications*

Montage [17] is an astronomical toolkit for assembling flexible image transport system (FITS) images into custom mosaics. The Montage workflow is divided into levels (as seen in Figure 3). Levels 1, 2 and 5 have 12, 23 and 12 nodes, respectively, while other levels have only one node.

To execute this application it is necessary to include the input images in FITS format (this is the standard format used by the astronomical community), with e a header file that specifies the mosaic type that is to be built.

This workflow operates in three steps. First, input images are the re-projected; second, re-projected images are refined; and last, of the re-projected images are superimposed and refined to obtain the mosaic in JPEG format.

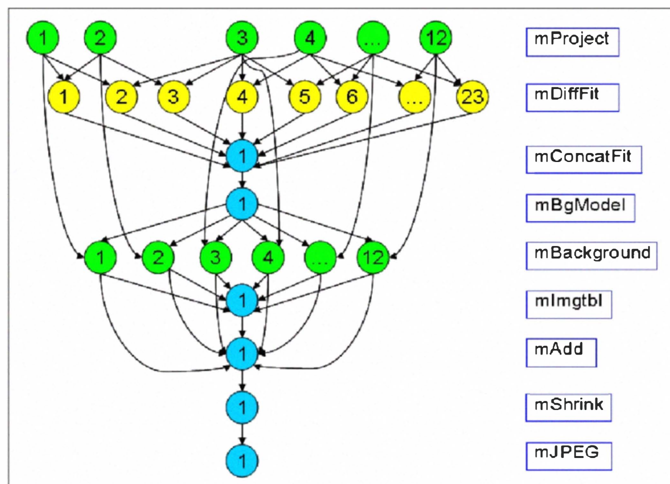

Figure 3: Montage's Workflow.

LIGO (Laser Interferometer Gravitational Wave Observatory) is aimed at detecting gravitational waves produced by violent events in the universe, such as the collision of two black holes or the explosion of supernovae [20].

We used a small part of LIGO with 81 nodes, as shown in Figure 4.

C. *Experimentation and Results*

We carried out four sets of experiments using the execution environment above described. In summary, the results from our experiments showed that we were able to effectively use different scheduling policies with different workflow engines in a transparent and flexible way.

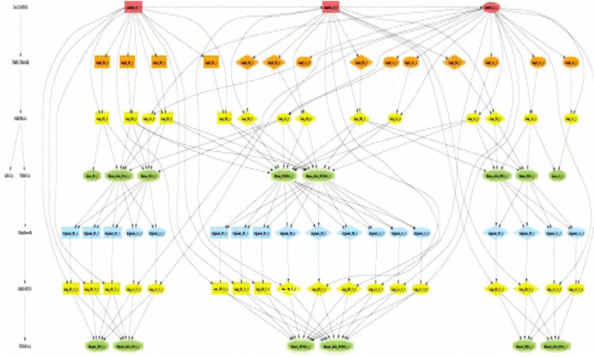

Figure 4: LIGO's Workflow.

Our experiments were designed to study the performance obtained with different scheduling policies when the initial workload supplied to the application is changed. Indeed, we wanted to see the results when changing the initial workload while using the same workflow engine, workflow application and scheduling policy. Our performance measure for these applications makespan, that is, the execution time it takes to run from the first node of the workflow up to the completion of the last one.

Our first scenario consisted of the Montage workflow run on the three different workflow engines considered, Taverna, DAGMan, and Karajan. In this scenario, we used their default- and very simple-scheduling schemes. This was our reference point for comparison when we introduced different scheduling policies and workloads.

For this experiment we used a 400MB workload. We performed 120 executions of a Montage application with 53 tasks. The corrective measures in the presence of failures are those provided by the workflow engine, and the execution environment was stable. That means that we submitted workflow for execution only if 45 computing resources (out of a total of 140) were available for our application.

Figure 5 shows the obtained results in the basic case, when the default scheduling policies of the workload engines were used. The X axis contains shows the workflow engine used, and the Y axis is the average makespan for the 120 executions, shown in seconds (the makespan for the two workloads was in the range from 5000 to 7300, with a standard deviation of around 180).

Our next step consisted of running the same experiments, but this time using SchedFlow to select the scheduling policies, These experiments not only provide a comparison between using a fix and dynamically-chosen scheduling policy; they also provide insight as to which scheduling policy is more effective for a given application.

All other aspects of the scenario are kept the same. We still guarantee 45 machines, and the workload is 400MB. The only change is that now we can choose between different well known scheduling policies (min-min [11], HEFT [8, 9] and BMCT [10]), and the support for rescheduling is managed by SchedFlow. The engines are used for dependency management and monitoring. The information provided by the workflow engine is used by SchedFlow to improve the makespan. Figure 6 shows us the results obtained.

The engines are used for dependency management and monitoring. The information provided by the workflow engine is used by SchedFlow to improve the makespan. Figure 6 shows us the results obtained.

The results show that when using user-supplied scheduling policies (instead of the engine default scheduling policy), the makespan is reduced on all workflow engines. This fact confirms that frameworks like SchedFlow are useful for improving performance, with the benefit of dynamically selecting the algorithm outweighing the cost of choosing or rescheduling.

We also noted that, independently from the workflow engine used, the policy that gives best results with the 400 MB workload is BMCT. The results in DAGMan and Karajan are strikingly similar, which seems to indicate that they manage the applications in a similar fashion, distinctively from that of Taverna. However, the important fact here is that by using more sophisticated policies the makespan is reduced by around 30%.

In the next experiments we varied the input workload from 400MB to 1024MB. Figures 7 and 8 correspond to these results. It can be seen that the results when using the workflow engines and their default scheduling policies are in par with that of the previous ones, of course with much bigger times, given the bigger initial workload.

However, when we perform the comparison with the different scheduling policies and SchedFlow, we can see important changes regarding the results. The first one is that BMCT is no longer the best policy. HEFT schedules better when workloads are bigger, as can be seen in Figure 8.

BMCT delivered lower performance because its scheduling is based in a parameter called rank, which gives priorities to tasks. If this parameter is not properly adjusted, the policy works poorly, leading to a worse makespan.

We repeated the same experiments using the LIGO application. Though the makespan for LIGO was approximately five times larger than the makespan for

Montage, the results obtained with LIGO when the workload changed were similar to the ones obtained with the Montage application. The experiments show that as scheduling policies are sensitive to initial input workloads, there is not a single best policy for a given application. With SchedFlow, we were able to quantify the effect of different scheduling policies on different workflow engines, therefore demonstrate the potential of our framework. With SchedFlow the user can use the desired scheduling policies depending on dynamic and application specific conditions in an transparent and flexible way. This can be especially useful in dynamic environments such as Grids and Clouds.

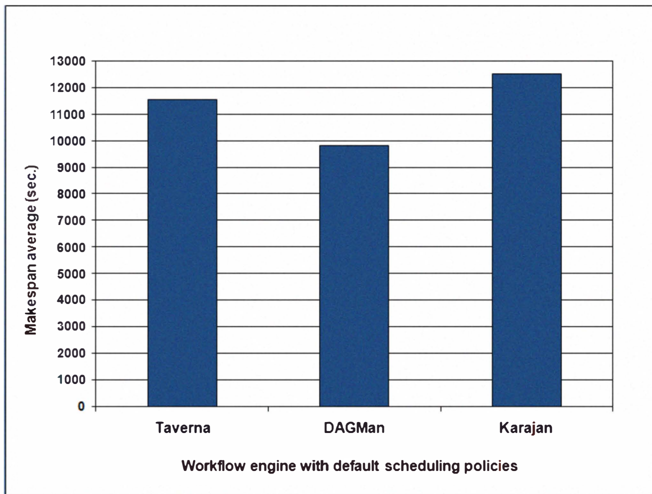

Figure 5. Montage execution using the default scheduling policies with a workload of 400MB.

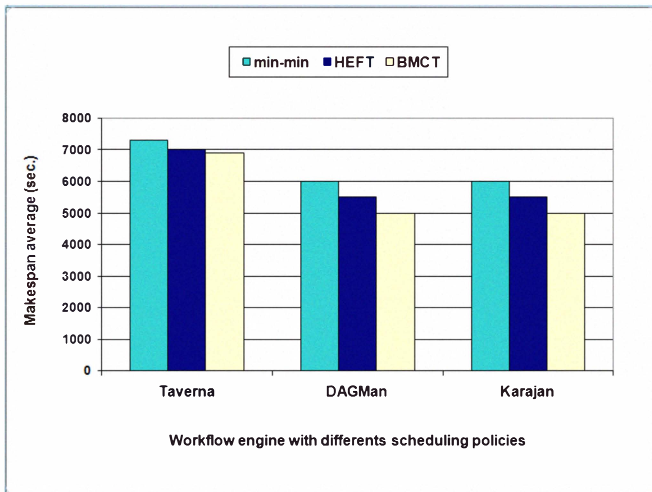

Figure 6. Montage execution using SchedFlow with a workload of 400MB.

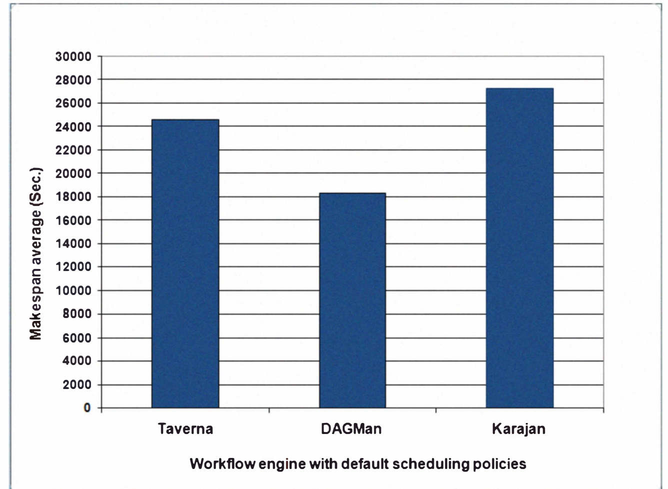

Figure 7. Montage execution using the default scheduling policies with a workload of 1024MB.

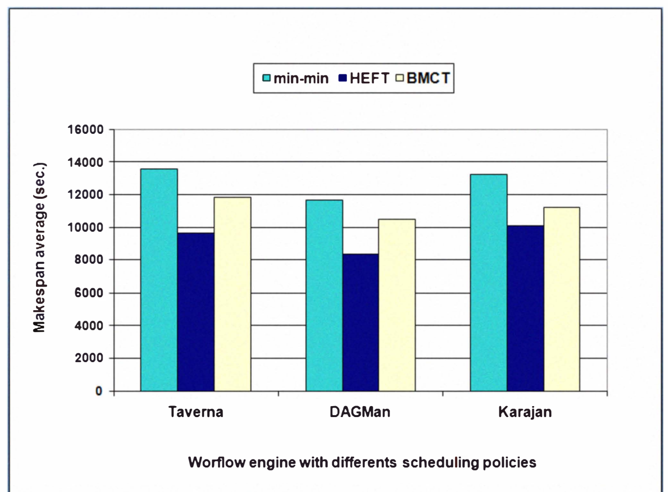

Figure 8. Montage execution using SchedFlow with a workload of 1024MB.

V. RELATED WORK

In this section we review related research in workflow management, and we review how SchedFlow narrows the existing gap between scheduling policies for workflows and workflow engines.

Condor DAGMan [6] is a meta-scheduler for Condor. It manages job dependencies at a higher level than the Condor Scheduler. Condor DAGMan is a dynamic system that uses a random scheduling policy. Additionally, with Condor DAGMan it is difficult to execute tasks in a specific order to minimize the makespan.

In Taverna [10], the workflow manager allows users to construct complex analysis workflows from components located on both remote and local machines, run these workflows with their own data, and visualized the results. The scheduling policy of Taverna is heuristic scheduling based on performance models (CPU speed), which means that tasks are sent to the machines with better performance.

Karajan [6] is an extensible workflow framework derived from GridAnt [14] which supports loops of workflow structures. Its default scheduler cycles through a list of available resources and uses the first resource suitable for the given task. The resource search for the next task begins with the resource immediately following the last resource used in the list. If the end of the list is reached, the search continues from the beginning of the list. Its fault-tolerance scheme is based on task retries in alternating resources. Karajan does not support the integration of new scheduling policies. The resource search for the next task begins with the resource immediately following the last resource used in the list. If the end of the list is reached, the search continues from the beginning of the list. Its fault-tolerance scheme is based on task retries in alternating resources. Karajan does not support the integration of new scheduling policies.

Pegasus [12] is a system that maps an abstract workflow to the set of available Grid resources and generates an executable workflow. An abstract workflow can be constructed by querying Chimera, a virtual data system, is provided by users in DAX (a DAG XML description). Pegasus consults various Grid information services to find the resources, software, and data required for the workflow. The Replica Location Service (RLS) is used to used to locate the replicas of the required data, and the Transformation Catalog (TC) is used to locate the logical application components. There are two techniques used in Pegasus for resource selection: one uses random allocation, and the other uses performance prediction

A complete taxonomy and classification of workflow systems are described [16] according to their main functions and architecture of workflow systems. Existing workflow engines provide very simple scheduling mechanisms. Current experiences with more sophisticated strategies imply introducing important changes to the underlying workflow engine.

SchedFlow is intended to be a general framework that is not targeted to either a specific workflow engine or to a particular application. This generality makes it possible to integrate different policies using a single tool and the end user does not need to change whole architectures to be able to efficiently run a workflow. Furthermore, our tool can be linked with other workflow managers.

VI. CONCLUSIONS

In this work, we showed that for the same workflow applications and workflow engines, the choice of scheduling policy should depend on dynamic and difficult to control factors such as the initial workload supplied to the applications.

As our experimental environment was mostly homogeneous, we predict that, in more heterogeneous and opportunistic environments such as Grids and Clouds, this effect will be magnified. That means that no single scheduling policy will be the most advantageous to use when executing workflow applications.

SchedFlow allowed us to implement and execute different scheduling policies (both dynamic and static), as well as integrate them with different workflow engines in an easy and flexible way, therefore significantly narrowing the current gap existing between scheduling policies and workflow engines.

SchedFlow is available for download at: http://code.google.com/p/schedflow/.

ACKNOWLEDGMENT

Special thanks to Daniel S. Katz for his valuable aid with the Montage application.

REFERENCES

[1] GriPhyn, http://www.griphyn.org.

[2] E. Deelman, C. Kesselman, G. Mehta, L. Meshkat, L. Pearlman, K. Blackburn, P. Ehrens, A. Lazzarini, R. Williams, and S. Koranda, "GriPhyN and LIGO, Building a Virtual Data Grid for Gravitational Wave Scientists," *Proceedings of the 11th Intl. Symposium on High Performance Distributed Computing*, Edinburgh-Scotland, July 2002, pp. 225.

[3] SCECsCM, http://www.scec.org/cme.

[4] J. Annis, Y. Zhao, J. Voeckler, M. Wilde, S. Kent, and I. Foster, "Applying Chimera Virtual Data Concepts to Cluster Finding in the Sloan Sky Survey," *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, Baltimore, MD-USA, November 2002, pp. 1-14.

[5] NPACI, https://gridport.npaci.edu/Telescience.

[6] DAGMan, http://www.cs.wisc.edu/condor/dagman.

[7] E. Ceyhan, G. Allen and Christopher White and Tevfik Kosar, "A grid-enabled workflow system for reservoir uncertainty analysis", *Proceedings of the 6th International Workshop on Challenges of Large Applications in Distributed Environments*, Boston, MA-USA, June 2008, pp. 45-52.

[8] M. Brahm, and H. Pargmann, "**Workflow Management with Sap Webflow: A Practical Manual**", *Springer Verlag*, California-USA, 2004.

[9] Z. Guan, F. Hernandez, P. Bangalore, J. Gray, A. Skjellum, V. Velusamy, and Y. Liu, "Grid-Flow: a Grid-enabled scientific workflow system with a Petri-net-based interface", *Concurrency and Computation: Practice and Experience* **18,** 10, Chichester-UK, August 2006, pp. 1115-1140.

[10] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M.Greenwood, T. Carver and K. Glover, M.R. Pocock, A. Wipat, and P. Li, "Taverna: a tool for the composition and enactment of bioinformatics workflows", *Journal-Bioinformatics* **20**,17 *Oxford Univ Press*, London-UK, 2004, pp. 3045-3054.

[11] I. Taylor, M. Shields, I. Wang and A. Harrison, "Visual Grid workflow in Triana", *Journal of Grid Computing*, Netherlands, **3**, 3-4, 2005, pp. 153-169.

[12] G. Singh, M. H. Su, K. Vahi, E. Deelman, B. Berriman, J. Good, D. Katz, G. Mehta, "Workflow task clustering for best effort system with Pegasus", *Proceedings of the 15th ACM Mardi Gras Conference*, Baton Rouge, LA-USA, January 2008, pp. 1-8.

[13] H. Topcuoglu, S. Hariri, and M. Wu, “Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing”, *IEEE Trans on Parallel and Distributed System*, **13**, 3, 2002, pp. 260-274.

[14] R. Sakellariou, and H. Zhao, “A Hybrid Heuristic for DAG Scheduling on Heterogeneous System,” *Proceedings of 18th International Parallel and Distributed Processing Symposium*, Santa fe, New Mexico-USA, April 2004, pp. 111.

[15] H. Xiaoshan, X. Sun, and G. Laszewski, “QoS guided min-min heuristic for grid task scheduling”. *International Journal of Computer Science and Technology* **18**, 4, Beijing-China, 2003, pp. 442-451.

[16] J. Yu, and R. Buyya. “Taxonomy of Workflow Management Systems for Grid Computing”, *Journal of Grid Computing* **3**, 3, 2005, pp. 171-200.

[17] G. Berriman, A. Laity, J. Good, J. Jacob, D. Katz, E. Deelman, G. Singh, and T. Prince, "Montage: The Architecture and Scientific Applications of a National Virtual Observatory Service for Computing Astronomical Image Mosaics," *Proceedings of Earth Sciences Technology Conference*, Hyattsville, Maryland-USA, June 2006.

[18] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, N. Spring, A. Su, and D. Zagorodnov, “Adaptive Computing on the Grid Using AppLeS”, IEEE *Trans on Parallel and Distributed System*, Vol. **14**, 4, April 2003, pp. 369-382.

[19] Michael A. Iverson, Gregory J. Follen, “Run-time statistical estimation of task execution times for heterogeneous distributed computing”, *Proceedings of the High Performance Distributed Computing Conference*, Syracuse, New York- USA, August 1996, pp. 263-270.

[20] Brown, D.A. and Brady, P.R. and Dietz, A. and Cao, J. and Johnson, B. and McNabb, J. “A case study on the use of workflow technologies for scientific analysis: Gravitational wave data analysis”, *Workflows for e-Science*, 2007, pp. 39-59.