# Dynamic Instrumentation, Performance Monitoring and Analysis of Grid Scientific Workflows [*]

Hong-Linh Truong[†] (truong@dps.uibk.ac.at) and Thomas Fahringer (tf@dps.uibk.ac.at)
*Institute for Computer Science, University of Innsbruck*
*Technikerstrasse 21A, A-6020 Innsbruck, Austria*

Schahram Dustdar (dustdar@infosys.tuwien.ac.at)
*Information Systems Institute, Vienna University of Technology*
*Argentinierstrasse 8/184-1, A-1040 Wien, Austria*

**Abstract.** While existing work concentrates on developing QoS models of business workflows and Web services, few tools have been developed to support the monitoring and performance analysis of scientific workflows in Grids. This paper describes novel Grid services for dynamic instrumentation of Grid-based applications, performance monitoring and analysis of Grid scientific workflows. We describe a Grid dynamic instrumentation service that provides a widely accessible interface for other services and users to conduct the dynamic instrumentation of Grid applications during the runtime. We introduce a Grid performance analysis service for Grid scientific workflows. The analysis service utilizes various types of data including workflow graphs, monitoring data of resources, execution status of activities, and performance measurements obtained from the dynamic instrumentation of invoked applications, and provides a rich set of functionalities and features to support the online monitoring and performance analysis of scientific workflows. Workflows and their relevant information including performance metrics are stored and utilized for comparing the performance of constructs of different workflows and for supporting multi-workflow analysis.

**Keywords:** dynamic instrumentation, Grid computing, Grid service, scientific workflows, performance monitoring and analysis

## 1. Introduction

Recently, increased interest can be witnessed in exploiting the potential of the Grid for scientific workflows. Scientific workflows [31, 37, 42, 43], in contrast to production and administrative business workflows, are normally more flexible and completely automatic. On computational Grids [20], the most common Grid type, scientists usually try to harness and utilize available resources in Grids for their experiments. As the Grid is diverse, dynamic and inter-organizational, it comes out that even with a particular scientific experiment, it requires to have a set of different workflows because (i) one

[*] The work described in this paper is supported in part by the Austrian Science Fund as part of the Aurora Project under contract SFBF1104 and by the European Union through the IST-2002-511385 project K-WfGrid.
[†] Corresponding author

workflow mostly fits to only a particular configuration of the underlying Grid systems, and (ii) the available resources allocated for a scientific experiment and their configuration in the Grid are changed each execution. This requirement is a challenge to the workflow composition and the workflow scheduler because normally they focus on composing and constructing a particular workflow with respect to available resources, and on mapping that workflow into the available resources. It is also a challenge to the performance monitoring and analysis of the workflows because very often clients of the performance analysis service (e.g., users and scheduling systems) want to compare the performance of different workflow constructs[1] with respect to the resources allocated in order to determine which workflow construct should be best matched to which topology of the underlying Grid. Even though numerous tools have been developed for constructing and executing scientific workflows in the Grid, such as [31, 29, 39, 17], there is a lack of tools that support the performance monitoring and analysis of such flexible scientific workflows in the Grid. Most existing work concentrates on developing QoS (Quality of Service) models of business workflows and Web services [28, 16, 35, 8], however, few tools have been developed to support scientists to monitor and analyze the performance of their workflows in the Grid.

Because of the dynamics of the Grid, the performance monitoring and analysis of workflow-based applications (WFAs) has to be carried out in online manner. Firstly, as a workflow (WF) is executed spanning on distributed organizations in the Grid, in monitoring and analyzing the performance of the workflow, we need to collect and process a variety of types of data relevant to the performance of the WFs, for example execution status of WFs from workflow management systems (WfMS), monitoring data of resources on which WF activities are executed, and performance measurements of code regions of invoked applications of workflow activities. These relevant data are not only provided by many sources but they are also diverse and distributed. The performance monitoring and analysis service therefore needs the support from the monitoring middleware in order to obtain, gather, and utilize that diverse data in a unified way. Secondly, to fully understand the performance of a workflow, we need monitoring and performance data of the workflow that are measured at many levels of detail, such as at the whole-workflow, activity and code region level. Without the instrumentation of code regions of workflow activities, we are only able to monitor at the level of activity, thus significantly reducing the ability to detect and correlate performance problems. Most existing WF monitoring tools are limited to the activity level.

In previous work, we have developed a middleware which supports services to access and utilize a variety types of performance data in a single

[1] Basically, a workflow construct consists of a set of dependent activities that are represented in a connected subgraph of the workflow, e.g., loop, sequence and fork-join. Common workflow constructs can be found in [7].

system named SCALEA-G [41]. In this paper, we firstly present a Grid service to support the dynamic instrumentation of Grid applications. The Grid dynamic instrumentation service provides a widely accessible interface to other services/users to control the instrumentation process. The instrumentation service leverages an XML-based Standardized Intermediate Representation for Binary Code (SIRBC) for describing the program structure of executable, and an instrumentation request language (IRL) for specifying code regions of which performance metrics should be determined and controlling the instrumentation process. Secondly, we introduce a Grid service for online monitoring and performance analysis of scientific workflows on the Grid. In order to provide detailed performance status and problems of a workflow, the service collects resources status from the Grid infrastructure monitoring, workflow execution status from the workflow control and invocation services, and performance measurements obtained through the dynamic instrumentation service. It then conducts the online analysis of these data along with the workflow graph. Relevant data to workflows including workflow graphs and performance data are stored. We then develop techniques to support multi-workflow analysis. Refinement constructs of workflows can be specified, and performance of refinement constructs of different workflows can be compared and evaluated for multiple experiments. The work described in this paper has been implemented based on the SCALEA-G framework [41].

The rest of this paper is organized as follows: Section 2 discusses instrumentation techniques for the Grid. Section 3 describes the dynamic instrumentation service for Grid applications. Section 4 details techniques used to implement incremental online profiling. Performance analysis for WFs is presented in Section 5. We illustrate experiments in Section 6. Section 7 discusses the related work. We summarize the paper and outline the future work in Section 8.

## 2. Instrumentation Techniques for Grid Applications

### 2.1. Instrumentation and Measurement Techniques

One of the central elements of the performance analysis of Grid applications is how performance data is measured and collected. Firstly, we have to study different instrumentation mechanisms to efficiently measure different types of performance data. Source code instrumentation provides a simple and efficient way for collecting measurement data, however, it requires the availability of all the source files. The instrumented sources have to be compiled and linked with instrumentation libraries for specific target machines. That is a time consuming effort because each time the application executes the resources allocated may be different, not to mention the allocated resources may not be known in advance. Dynamic instrumentation is complex but well-suited for measuring volatile and long-running applications, and for applications whose source code is not available. The WFA is normally dynamically composed, possibly at runtime, from deployed applications whose source code is not available for instrumentation. The dynamic instrumentation would be an alternative for solving the problems arisen from the selection of instrumentation and measurement system and the compilation of instrumented code fitted to the allocated resources.

We believe that instrumentation for the Grid should employ both methods. We can instrument sources of WF control and invocation service in order to gather execution status of WFs because execution status information is normally simple and small. However, for instrumentation of Grid scientific applications, we believe that dynamic instrumentation would be more suitable. While source code instrumentation for Grid applications is widely supported, e.g. in [12, 24], dynamic instrumentation in Grids has not got much attention, even though dynamic instrumentation has a long history in clustering and parallel computing [32, 18].

Secondly, we have to carefully select the granularity of the measurement for Grid applications, namely profiling or tracing mechanism. Many tools support tracing of Grid applications, e.g. [34, 24]. However, as Grid performance monitoring and analysis must be carried out in online manner, tracing is not suited because it generates a huge volume of trace data which has been transfered on the fly to analysis components. On the other hand, traditional profiling is not suited for online monitoring and analysis because profiling data can only be obtained at the end of the execution of applications. Therefore, incremental mechanisms in which profiling data is updated or requested and retrieved incrementally at runtime would be more suitable. For example, in CrossGrid project monitoring data can be periodically retrieved during the runtime [14].

## 2.2. Instrumentation Service

While execution status of workflows and monitoring data of resources may be obtained from WfMS and infrastructure monitoring, respectively, the current situation is that the user has to manually instrument his code in order to obtain performance measurements of code regions of workflow activities, which are executed on multiple Grid sites, because existing instrumentation systems are only appropriate for a single Grid site (within a single organization). While existing Grid toolkits (e.g., Globus[22]) provide core services for job submission and resource discovery, similar Grid services for instrumenting Grid application do not exist.

Currently, in most cases the instrumentation of Grid workflows must be carried out manually by the end user. Consider the diversity and dynamics of the Grid. On the one hand, if the user wants to instrument his code, the user has to know in advance the Grids he submits jobs to, and has to select the right instrumentation tool for each Grid site. As a result, the user has to do a daunting task in order to instrument his code. Moreover, the selected instrumentation tool may not work with the monitoring middleware deployed in the selected Grid site. On the other hand, instrumentation techniques are typically bound to specific languages and systems. Therefore, it is possible that we need many different instrumentation systems just for instrumenting an application executed on the Grid.

We argue that the instrumentation service should be a core service of a Grid. This approach gives many advantages. Firstly, an instrumentation service is bound to a specific Grid site, which normally consists of (homogeneous) computational resources that are controlled by a single security policy and exchange data through a local network. Thus, the instrumentation service can be better developed, can efficiently exploit features on that site, and is better to be coupled with the supportive monitoring middleware deployed in that site. Secondly, as an instrumentation system is a service, the user does not need to worry about how to select a suitable instrumentation system. Instead, he just discovers the service and uses it. Each Grid site may provide an instrumentation service that allows the user or the high level tools to control the instrumentation. To this end, the instrumentation service hides all the low-level details of the instrumentation process while the client of the instrumentation service just simply specifies its requests. To follow this idea, the instrumentation service must support widely accessible interfaces, e.g., Grid/Web service operations, and protocols, e.g., APART SIR and MIR [36]. Nevertheless, with such generic Grid instrumentation service, we have to accept some limitations, e.g., instrumentation of arbitrary code regions may not be possible.

## 3. Grid Dynamic Instrumentation Service

Figure 1 presents the architecture of our dynamic instrumentation service for Grids. There are four main components residing in different locations that involve in the instrumentation process: *Instrumentation Requester* (IR), *Instrumentation Mediator* (IM), *Mutator Service* (MS) and *Instrumentation Forwarding Service* (IFS). The IR controls the instrumentation process. The MS, executed on the computation node where the application processes execute, is responsible for performing the dynamic instrumentation of application processes. It attaches the application processes and inserts *application sensors* into the application processes. In the middle of the IR and the MS are

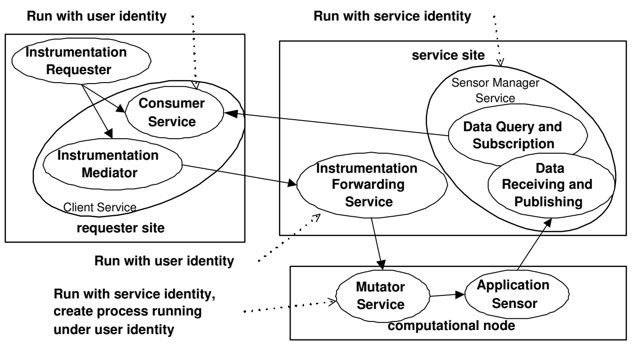

Run with user identity
Run with service identity
service site
Instrumentation Requester
Consumer Service
Instrumentation Mediator
Client Service
requester site
Sensor Manager Service
Data Query and Subscription
Data Receiving and Publishing
Instrumentation Forwarding Service
Run with user identity
Run with service identity, create process running under user identity
Mutator Service
Application Sensor
computational node

*Figure 1.* Architecture of the Grid service of dynamic instrumentation.

the IM and IFS which bridge and aggregate requests and responses between the IR and the MS. IM and IFS are needed because the IR cannot always directly communicate with the MS, e.g. due to the firewall. Moreover, IR works at a high-level at which it considers the execution of an application as a whole. Therefore, IR may conduct the instrumentation spanning multiple Grid sites. However, MS works at the lower level at which its objects are application processes. As a result, IM and IFS are used to transfer and aggregate requests and responses between the high-level view and the low-level one. An IFS instance is responsible for forwarding requests to multiple MSs executed on computational nodes. The above architecture is a service-oriented model based on two languages. The first language named SIRBC (Standardized Intermediate Representation for Binary Code) allows the instrumentor (MS) to describe instrumented applications in a neutral representation and to provide that representation to IR; SIRBC is an implementation of simplified SIR [36]. The second language named IRL (Instrumentation Request Language) allows IR to define what portions of an application should be instrumented and what performance metrics should be collected. Both SIRBC and IRL are XML-based. Details of SIRBC and IRL can be found in [41].

The MS is a Grid service which is implemented based on gSOAP, a C++ Web Service toolkit with GSI-plugin [23]. Figure 2 shows interactions between IR, MI, IFS, and MS instances when conducting requests for instrumenting an application. At the requester side, the IR specifies requests and passes these requests to IM. Based on the requests, the IM locates existing IFSs which can forward the requests to MSs executed on the same computational nodes of application processes; if no such IFSs exist, IM makes a request of creating new IFS instances. IM then sends IRL requests to IFSs. When an IFS receives a request, it will search MS instances which can fulfill the request. If there is no MS instance for instrumenting application processes of a user in a computational node, IFS makes a request of creating a new MS instance for the user on that node. IFS will send the requests to MSs which

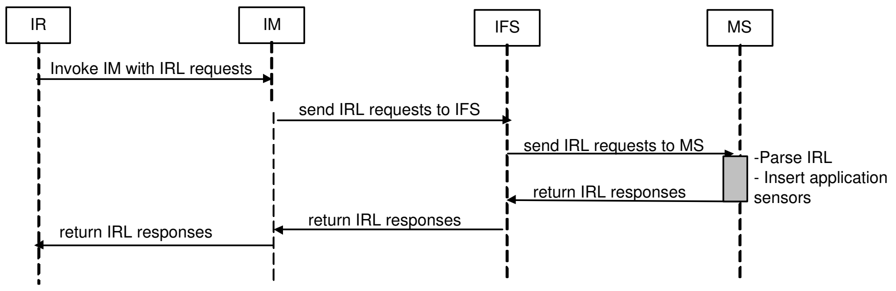


*Figure 2.* Steps in conducting a request for instrumentation.

in turn forward the requests to corresponding MSs. The MS will parse the IRL request and then perform the instrumentation of application processes. The MS inserts application sensors into application processes. The dynamic instrumentation techniques are facilitated by Dyninst [15]. The application sensors perform the monitoring and measurement of application processes. Performance measurements will be sent to Sensor Manager Service (SM), which is a part of the supportive monitoring middleware, or be collected through MS. Note that the role of SM is to collect monitoring data from application sensors. Therefore, in principle, SM can be replaced by similar infrastructures that can communicate with application sensors. Existing infrastructures for collecting monitoring data, such as R-GMA [6] and Mercury [11], are not suitable for our purpose because they are using relational database while our application sensors produce XML-based monitoring data and use XML-based messages in the communication with SM.

The MS provides the application structure to the requester in SIRBC format. Based on SIRBC, the IR can decide which code regions should be instrumented. With the high-level encapsulation and high interoperability, interfaced through service operations, IRL and SIRBC, the dynamic instrumentation service is widely accessible to other services.

## 3.1. Service Interface

The implementation of MS is based on the *factory model*. The MS consists of a Mutator Factory (MF) and Mutator Instance (MI). An MF is a persistent service deployed in each computational node. The MF provides a main operation named `createMutatorInstance` for creating MIs when requested. The MI is responsible for attaching application processes and instrumenting these processes.

Information about MF is published to the supportive monitoring middleware. When IRF receives an instrumentation request, it finds MIs on corresponding computational nodes which can instrument application processes of the calling user. If no such a MI exists, the IFS calls the MF on the corresponding node to create a new MI. When a MI running, it connects to a SM, notifies its existence to the SM and waits for control from requesters. MI provides the following main operations:

– `performIRL`: to process IRL requests. The MI will react with appropriate functions such as attaching the application process, instrumenting and deinstrumenting, or detaching the application process.

– `getProfilingData`: to return profiling data collected to the requester.

– `destroyInstance`: to end the execution of this instance. When this operation is called the MI frees resources it occupies, and finishes its execution.

In addition, MF and MI provide two auxiliary operations: `ping` operation to support ping service, and `getUserProcess` to obtain user processes executed on a computational node.

## 3.2. Practical Issues in Building SIR and Instrumenting Applications

When processing different binary codes compiled by different compilers, we observed that depending on specific compilers and architectures, SIR for an executable is quite different from that of the other. It contains many internal functions that the user may not want to instrument. SIR however is designed for C/C++/Fortran/Java sources, thus, it does not define filters that can be used to exclude these irrelevant information when building the SIR from applications. We extend IRL to allow the IR specifying filters into `getsir` requests. Filters include code region names that the instrumentation service should exclude, and the function scope in which the instrumentation service should limit its traversal.

Due to the dependence of executable structures on the compilers and platforms, the SIR of different processes of the same program may be different when the program is compiled and executed on different platforms. Thus, a SIR is associated with a process, not with a program. In some cases, the same code region has different identifiers in different SIRs. Therefore, when using identifiers to specify selected code regions, the IR has to process each SIR of a process individually. Consider a large number of processes, it is a time-consuming task for IR, if IR wants to instrument a code region in all processes. To avoid that, we can specify only the code region name and the program unit in instrumentation requests. The instrumentation service will instrument all functions which have that name within a given program unit.

### 3.3. Security Model

The security in the dynamic instrumentation service is based on GSI [44] facilities provided by Globus Toolkit (GT) [22]. As shown in Figure 1, the security model employs both transport and message level security, using delegation, authentication/authorization, and run-as mechanism [1]. Except MS uses transport level security, the interactions among the remaining components are based on message level security. Message level security employs GSI secure conversation mechanism [1].

IR and IM run with the security identity of the user. IFS service methods are set to run with the security identity of the client. When IM requests an IFS service to create an instance, the instance will be run with the security identity of the user. MF runs with the service identity in a none-privilege account. However, if MF is deployed to be used by multiple users, it must be able to create its instances running in the account of calling users. The MI created by MF upon requests of IFS will be run as user identity. MF uses a grid-map file to authorize its requesters. As MI executes with the security identity of the user, it has permission to attach user application processes, and is able to perform the dynamic instrumentation. Delegation is performed from IM to IFS to MI.

In push mode, application sensors send measurements to SM. When subscribing and/or querying data provided by application sensors, data requester's identity will be recorded. Similarly, before application sensor instances start sending data to the SM, the SM obtains the security identity of the requester who executed the application. Both sources of information will be used for authorizing the requester in receiving data from application sensors. In pull model, performance measurements collected by applications sensors will be returned to the requester by MI. MI uses self-authorization mechanism to check the requester. Requests for obtaining performance measurements sent by IR will be delegated from IM to IFS to MI. As a result, only the owner can be able to access performance data.

In our system, service management, instrumentation control, and data query and subscription tasks are conducted through service-based operations whereas monitoring data is transfered through TCP-based streams [41], therefore, the implementation of message level security has very little impact on the performance of our system.

## 4. Incrementally Updating Profiling Data

Traditionally, profiling is performed offline with performance measurements are summarized and available for being analyzed when the application finishes. Thus, this approach is not suitable for online profiling as we have complete summary measurements only when the application finishes. Online profiling requires measurement data to be collected and analyzed during runtime of the application. But if summary data is sent back to the analysis component at the instant the measurement data is updated, a huge volume data will be sent over the network. As a result, the impact of the monitoring on the execution of the application is high.

We develop a mechanism to support online and incrementally updating profiling data. That is, instead of always updating consecutive measurements of code regions, the monitoring system returns only the most-updated measurements in a maximum pre-defined time or upon a request. To profile a code region $r$ the instrumentation service inserts a sensor, composed by a start probe and a stop probe, into the application process as follows:

sis_start($PB_r$)
    $r$
sis_stop($PB_r$)

where $PB_r$ is information used to determine the code region;$PB_r$ is associated with a record storing measurement data of code region $r$. When an activation of $r$ finishes, its measurement data will be updated into the record. Each process keeps a profiling data of all instrumented code regions.

An analysis component can obtain the profiling data through *pull* or *push* mode. In pull mode, profiling data is stored in shared memory. The analysis component calls the `getProfilingData` operation of MI in order to obtain the requested profiling data. In push mode, the most recent updated measurements of $n$ code regions are stored into a *flush buffer* size $n$, $buf_n$. Performance measurements are incrementally sent to Data Receiving and Publishing (DRP) component of SM (see Figure 1). Figure 3 presents the algorithm used to send measurement data to the monitoring middleware. In addition, every $t$ seconds since the last time the buffer is flushed to DRP, the buffer will be flushed if it is not empty. With this algorithm, performance measurements of $n$ last executed code regions are flushed to DRP incrementally in maximum $t$ seconds. As a result, we ensure that the requester receives the newly-updated profiling measurement of a code region no longer than $t$ seconds since the measurement is updated.

We have already implemented the push mode and currently are implementing the pull mode. We are currently investigating to develop application sensors so that they store collected data into shared memory. The task to support pushing or pulling profiling data will be done by MI. Also the `getProfilingData` operation will support requests based on MIR [36].

```
procedure sis_start($PB_r$))

begin
        start the measurement of $r$.
        if (it is first execution of $r$) then
           send $PB_r$ to DRP component of SM.
        end if

end

procedure sis_stop($PB_r$))

begin
        stop the measurement of $r$.
        update performance measurements in $PB_r$.
        if ($PB_r$ is not in $buf_n$) then
           add $PB_r$ into $buf_n$.
        else
           update $PB_r$ in $buf_n$.
        end if
        if ($buf_n$ is full) then
           flush whole $buf_n$ to DRP.
           reset $buf_n$.
        end if

end
```

*Figure 3.* Updating profiling data to DRP.

## 5. Performance Monitoring and Analysis of Grid Workflow-based Applications

Performance monitoring and analysis of Grid WFs should support:

– **inter-activity performance monitoring and analysis**: to monitor and analyze the interactions between activities, the impact of an activity on the performance of the whole workflow or of the workflow construct that the activity participates. To this end, the monitoring and analysis tool has to operate at the whole workflow level and on the whole resources on which the workflow activities are executed.

– **intra-activity performance monitoring and analysis**: to monitor and analyze the performance of the invoked application of the individual

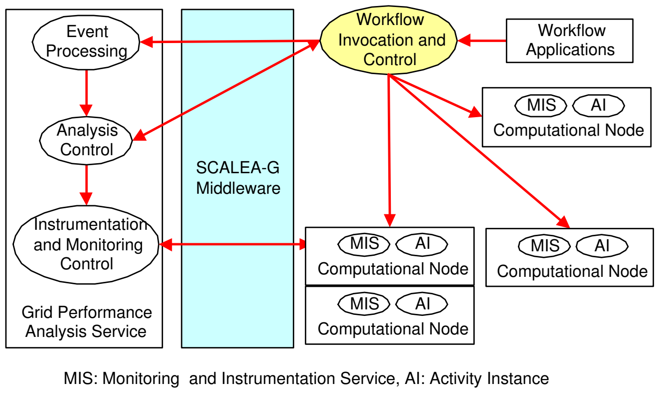

Event Processing
Analysis Control
Instrumentation and Monitoring Control
Grid Performance Analysis Service
SCALEA-G Middleware
Workflow Invocation and Control
Workflow Applications
MIS
AI
Computational Node
MIS: Monitoring and Instrumentation Service, AI: Activity Instance

*Figure 4.* Model of monitoring and performance analysis of workflow-based application.

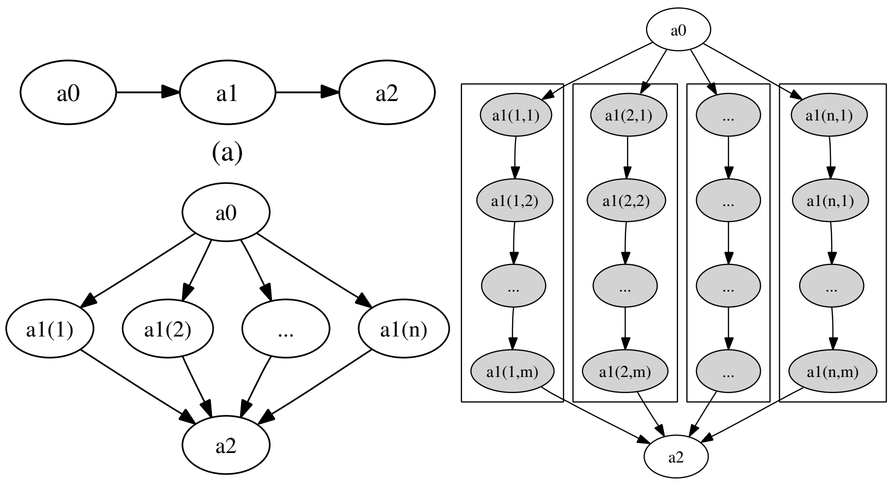
activity. To this end, the monitoring and analysis tool has to operate at the activity level and on the resource on which the activity is executed.

Figure 4 presents the architecture of the Grid monitoring and performance analysis service for WFs. The WF is submitted to the *Workflow Invocation and Control* (WIC) service which locates resources and executes the WF. Events containing execution status of activities, such as *queuing, processing*, and information about resources on which the activities execute will be sent to the monitoring tool. The *Event Processing* processes these events and the *Analysis Control* decides which activities should be instrumented, monitored and analyzed. Based on information of a selected activity instance and its consumed resource, the Analysis Control requests the *Instrumentation and Monitoring Control* to perform the instrumentation and monitoring. Monitoring and measurement data obtained are then analyzed. Based on the result of the analysis, the Analysis Control can decide what to do in the next step.

This architecture uses the SCALEA-G middleware as its supportive monitoring middleware. Various types of performance data are published to, stored in and retrieved from SCALEA-G.

### 5.1. SUPPORTING WORKFLOW COMPUTING PARADIGM

Currently we focus on the workflow modeled as a DAG (Direct Acyclic Graph) because DAG is widely used in modeling scientific workflows. In a DAG-based WF, a node represents an activity (task) and an edge between two nodes represents the dependency between the two activities. The invoked application of an activity instance may be executed on a single or on multiple resources. Meanwhile, we focus on activities whose invoked applications are application executables (e.g. MPI program).



(b) (c)

*Figure 5.* Multiple workflows of an workflow-based application: (a) sequence workflow, (b) fork-join workflow, and (c) fork-join structured block of activities.

We particularly concentrate on analyzing (i) *fork-join* model and (ii) *multi-workflow* of an application. Figure 5(b) presents the fork-join model[2] of workflow activities in which an activity is followed by a set of $n$ activities executed in parallel. This model is widely used in many scientific WFs. There are several interesting metrics that can be obtained from this model, such as load imbalance, slowdown factor, and synchronization delay at the synchronization point. These metrics help to uncover the impact of slower activities to the overall performance of the whole structure. We also focus on fork-join structures that contain *structured block* of activities. A structured block is a single-entry-single-exit block of activities [3]. For example, Figure 5(c) presents structured blocks of activities.

A workflow-based application (WFA) can have different versions, each represented by a WF. For example, Figure 5 presents an application with 3 different WFs, each may be selected for executing on specific underlying resources. When developing a WFA, we normally start with a graph describing the WF. The WFA is gradually developed in a sequence of refinement steps that creates a better version or an adapted version fitted to a particular underlying Grid system. This refinement can be done automatically by workflow construction tools or manually by the WF developers. In a refinement step, a subgraph may be replaced by another subgraph of activities, resulting in a set of different WFs. For example, the activity $a1$ in Figure 5(a) is replaced by set of activities $\{a1(1), a1(2), \cdots, a1(n)\}$ in Figure 5(b). (Also we can consider

[2] Also called as *AND-Split AND-Join* [5].
[3] Existing WF constructs are detailed in [7].

set of activities $\{a1(1), a1(2), \cdots, a1(n)\}$ is reduced to $a1$.) In Grids a WF can yield the best result in one particular run but not in the next run because the Grid may be different from run after run. The concept of the best solution is now associated with a particular run. Moreover, since the underlying system changed from experiment to experiment a single WF may not be enough. As a result, different solutions for a WFA, even all of them are just used to solve a specific problem, may equally be important. The key question is which WF construct is best for a given collection of resources. Therefore, multi-workflow analysis, the analysis and comparison of the performance of different WF constructs, ranging from the whole WF to a specific construct (e.g. a fork-join subgraph), is an important feature.

We focus on the case in which a subgraph of a DAG is replaced by an another subgraph in the refined DAG. Let $G$ and $H$ be DAG of workflow $WF_g$ and $WF_h$, respectively, of an WFA. $G$ and $H$ represent different versions of the WFA. $H$ is said to be a *refinement* of $G$ if $H$ can be derived by replacing a subgraph $SG$ of $G$ by a subgraph $SH$ of $H$. $SH$ is said to be a *replaced refinement graph* of $SG$. Note that $SG$ and $SH$ may not be a DAG nor a *connected graph*. For example, consider the cases of Figure 5(a) and Figure 5(b). Subgraph $SG = \{a1\}$ is replaced by subgraph $SH = \{a1(1), a1(2), \cdots, a1(n)\}$; both are not DAG, the first is trivial graph and the latter is not a connected graph. Generally, we assume that there are $n$ connected components of a subgraph $SG$. Each component is either a DAG or a trivial graph.

Graph refinement is a well-established field and it is not our focus. Therefore, we do not concentrate on the determination of refinement graphs in workflows, rather, the workflow developers and/or workflow construction tools are assumed to do this task. Our main goal is that given different solutions for a WFA we study the performance similarity and difference between them.

In this paper, $(a_i, a_j)$ is denoted as the dependency between activity $a_i$ and $a_j$; $a_i$ must be finished before the execution of $a_j$. Let $G = (N, E)$ be given, and select an arbitrary activity $a_i$. *pred($a_i$)* and *succ($a_i$)* are denoted as sets of the immediate predecessors and successors, respectively, of $a_i$.

## 5.2. Activities Execution Model

Each invoked application of an activity instance may be executed on different resources allocated by the WIC. We use discrete process model [38] to represent the execution of an activity $a$. Let $P(a)$ be an activity execution status graph modeling the execution of activity $a$ (hence we call the *execution graph* of an activity). A $P(a)$ is a directed, acyclic, bipartite graph $(S, E, A)$, in which $S$ is a set of nodes representing *activity states*, $E$ is a set of nodes representing *activity events*, and $A$ is a set of edges representing ordered pairs of activity state and event. Simply put, an activity event (e.g. executed) changes

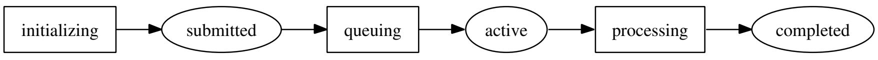


*Figure 6.* Discrete process model for the execution of an activity. □ represents a state, ◯ represents an event.

Table I. Example of event names.

| Event Name | Description |
|---|---|
| active | the activity instance has been started to process its work. |
| completed | the execution of the activity instance has completed. |
| failed | the execution of the activity instance has been stopped before its normal completion. |
| submitted | the activity has been submitted to the scheduling system. |

the activity state (e.g. from queuing to processing), which in turn influences the occurrence and outcome of future activity events (e.g. finished, failed). Figure 6 presents an example of a discrete process modeling the execution of an activity. Note that the real execution model of a WF is more complex, depending on the implementation of WIC. For example, an activity can be re-submitted, aborted and suspended[4].

Each state $s$ of an activity $a$ is determined by two events: leading event $e_i$, and ending event $e_j$ such that $e_i, e_j \in E$, $s \in S$, and $(e_i, s), (s, e_j) \in A$ of $P(a)$. To denote an event *name* of $P(a)$ we use $e_{name}(a)$. Table 5.2 presents an example of a few event names used to describe activity events[5]. We use $t(e)$ to refer to the timestamp of an event $e$ and $t_{now}$ to denote the timestamp at which the analysis is conducted. Because the monitoring and analysis is conducted at runtime, it is possible that an activity $a$ is in a state $s$ but there is no such $(s, e) \in A$ of $P(a)$. When analyzing such state $s$, we use $t_{now}$ as a timestamp to determine the time spent on state $s$. We use $\rightarrow$ to denote the *happened before* relation between events.

## 5.3. WORKFLOW INSTRUMENTATION

The monitoring system collects states and events of each activity instance, and builds the execution status graph of that activity instance. WIC in our experiment is currently implemented based on JavaCog [30]. WIC contains a job submission engine which interfaces to GRAM [4] of the Globus Toolkit. Currently, to get execution status of activities from WIC, the job submission engine is instrumented with an event sensor library. Monitoring data of activity execution is described by a well-defined XML representation. The job

[4] Detailed possible states of a workflow can be found in [5].
[5] Detailed possible activity events can be found in [5].

submission engine captures execution status of activities and describes the execution status in the XML representation. The event sensor library is used to send the activity monitoring data to SCALEA-G middleware. The Grid performance analysis service receives monitoring data of activity events and states by using notification mechanism of SCALEA-G and by querying and subscribing activity monitoring data.

## 5.4. Performance Metrics of Grid Scientific Workflows

Performance measurements for a Grid WF are collected at two levels: *activity* and *whole-application* level. Based on monitoring data, performance measurements and WF graphs, the performance of WF is analyzed.

### 5.4.1. *Activity Level*

At activity level, several performance metrics that characterize an activity are provided. Firstly, we dynamically instrument code regions of the invoked application of the activity. We collect performance metrics such as wallclock time, CPU time, hardware counters of instrumented code regions. Performance metrics of code regions are incrementally provided to the user during the execution of the workflow. Based on these metrics, various exploratory data analysis techniques can be employed, e.g. load imbalance, metric ratio. We extend our overhead analysis for parallel programs [40] to WFAs. For each activity, we analyze *activity overhead*. Activity overhead contains various types of overhead, e.g., communication, synchronization, that occur in an activity instance.

Secondly, we focus on analyzing the response time of activities. *Activity response time*, the time an activity takes to be finished, consists of waiting time and processing time. Waiting time can be queuing time and suspending/resuming time and processing time can consist of communication and computation time. For each activity $a$, its execution status graph, $P(a)$, is used as the input for analyzing activity response time. Moreover, we analyze synchronization delay between activities. Let consider a dependency between two activities $(a_i, a_j)$ where $a_i \in pred(a_j)$. $\forall a_i \in pred(a_j)$, when $e_{completed}(a_i) \rightarrow e_{submitted}(a_j)$, the synchronization delay from $a_i$ to $a_j$, $T_{sd}(a_i, a_j)$, is defined as

$$T_{sd}(a_i, a_j) = t(e_{submitted}(a_j)) - t(e_{completed}(a_i)) \qquad (1)$$

If at the time of the analysis $e_{submitted}(a_j)$ has not occurred, $T_{sd}(a_i, a_j)$ is computed as

$$T_{sd}(a_i, a_j) = t_{now} - t(e_{completed}(a_i)) \qquad (2)$$

Each activity $a_j$ associates with a set of the synchronization delays. From that set, we compute maximum, average and minimum synchronization delay at

$a_j$. Note that synchronization delay can be analyzed for any activity which is dependent on other activities. This metric is particularly useful for analyzing synchronization points in a workflow.

### 5.4.2. *Workflow level*

We analyze performance metrics that characterize the interaction and the performance impact among activities. Interactions between two activities can be file exchanges, remote method invocations or service calls. There are various metrics of interest such as average response time, waiting time, queuing time and synchronization delay of activities, load imbalance, communication to computation ratio, and success rate of activity invocation. Correlation metrics, such as number of activities per resource, resource utilization, etc., are also important.

We combine WF graph, execution status information and performance data to analyze load imbalance for fork-join model. Let $a_0$ be the activity at the fork point. $\forall a_i, i = 1 : n, a_i \in succ(a_0)$, load imbalance $T_{li}(a_i, s)$ in state $s$ is computed as

$$T_{li}(a_i, s) = T(a_i, s) - \frac{\sum_{i=1}^{n} T(a_i, s)}{n} \quad (3)$$

We also apply load imbalance analysis to a set of selected activities. In a workflow, there could be several activities whose work are the same, e.g. `mProject` activities in Figure 7, but are not in fork-join model. Load imbalance analysis is useful technique to reveal how the work distribution is conducted.

## 5.5. Multi-workflow Analysis

We analyze *slowdown* factor for fork-join model. Slowdown factor, $sf$, is defined as

$$sf = n \times \frac{max_{i=1}^{n}(T_n(a_i))}{T_1(a_i)} \quad (4)$$

where $T_n(a_i)$ is the processing time of activity $a_i$ in the fork-join WF with $n$ activities and $T_1(a_i)$ is the fastest processing time of activity $a_i$ in the (fork-join) WF of single activity. The slowdown factor analysis can also be applied to fork-join structures that contain structured block of activities. In this case, $T_n(a_i)$ will be the processing time of a structured block of activities in the WF with $n$ blocks.

For different replaced refinement graphs of WFs of the same WFA, we compute *speedup* factor between them. Let $SG$ be a subgraph of workflow $WF_g$ of a WFA; $SG$ has $n_g$ components. Let $P_i =< a_{i1}, a_{i2}, \cdots, a_{in} >$ be a critical path from starting node to the ending node of the component $i$, $C_i$, of

$SG$. The processing time of $SG$, $T_{cp}(SG)$, is defined as

$$T_{cp}(SG) = max_{i=1}^{n_g}(T_{cp}(C_i)), T_{cp}(C_i) = \sum_{k=1}^{n} T(a_{ik}) \tag{5}$$

where $T(a_{ik})$ is the processing time of activity $a_{ik}$. Now, let $SH$ be the replaced refinement graph of $SG$, $SG$ and $SH$ are subgraphs of workflow $WF_g$ and $WF_h$, respectively, of a WFA. Speedup factor $sp$ of $SG$ over $SH$ is defined as follows:

$$sp = \frac{T_{cp}(SG)}{T_{cp}(SH)} \tag{6}$$

The same technique is used when comparing the speedup factor between two workflow $WF_g$ and $WF_h$.

In order to support multi-workflow analysis of WFs, we collect and store different DAGs, subgraphs of the WFA, performance data and machine information into an experiment repository powered by PostgreSQL. Each graph is stored with its associated performance metrics; a graph can be DAG of the WF or a subgraph. We use a table to represent refinement relationship between subgraphs. Currently, for each experiment, the user can select subgraphs, specifying refinement relation between two subgraphs of two WFs. The analysis service uses data in the experiment repository to conduct multi-workflow analysis.

## 6. Experiments

We have implemented prototypes of Grid services for dynamic instrumentation and performance analysis of Grid WFs. JGraph [26] and JFreeChart [25] are used to visualize WF graphs and performance results, respectively. In this section, we illustrate the usefulness of our services by presenting experiments of different workflows of the Montage application in the Austrian Grid [10].

Montage [33] is a software for generating astronomical image mosaics with background modeling and rectification capabilities. Based on the Montage tutorial, we develop a set of WFs, each generating a mosaic from 10 images without applying any background matching. Figure 7 presents experimental workflows of the Montage application. In Figure 7(a), the activity `tRawImage` and `tUncorrectedMosaic` are used to transfer raw images from user site to computing site and resulting mosaics from computing site to user site, respectively. `mProject` is used to reproject input images to a common spatial scale. `mAdd` is used to coadd the reprojected images. `mImgtbl` is used to build image table which is accessed by `mProject` and `mAdd`.

In workflows executed on multiple resources, we have several subgraphs $tRawImage \rightarrow mImgtbl1 \rightarrow mProject1 \rightarrow tProjectedImage$, each

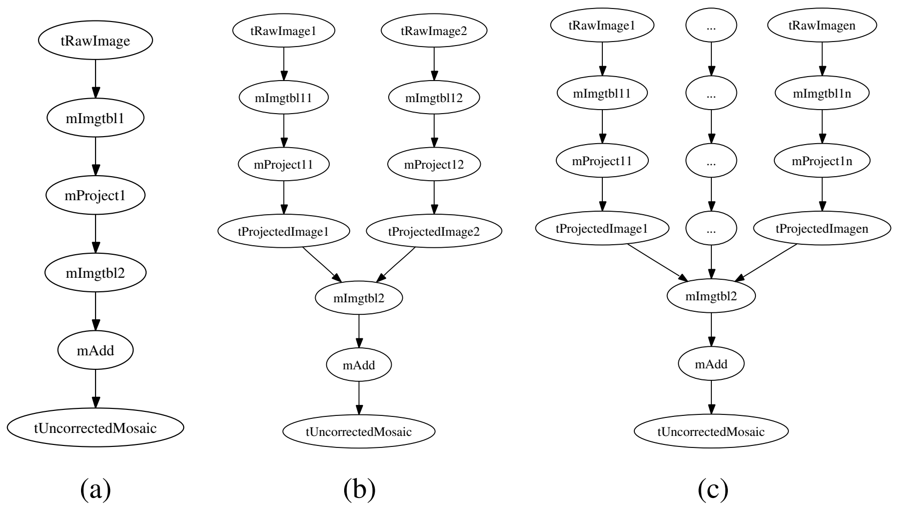


*Figure 7.* Experimental workflows of the Montage application: (a) workflow executed on single resource, (b) workflow executed on two resources, and (c) workflow executed on $n$ resources

subgraph is executed on a resource. The `tProjectedImage` activity is used to transfer projected images produced by `mProject` to the site on which `mAdd` is executed. When executed on $n$ resources, the subgraph $mImgtbl2 \rightarrow mAdd \rightarrow tUncorrectedMosaic$ is allocated on one of that $n$ resources. When executed on Grid resources using the same NFS (Network File System), the task `mProject` can work on fork-join fashion.

We conduct experiments on sites named LINZ (Linz University), UIBK (University of Innsbruck), AURORA6 (University of Vienna) and VCPC (University of Vienna) of the Austrian Grid. The user resides in VCPC and the workflow invocation and control service (WIC) submits invoked applications of workflow activities to VCPC, LINZ, UIBK, AURORA6. Most machines in experiments are non dedicated ones.

## 6.1. Monitoring Execution Status of Activities

Before a WF is submitted to WIC, the performance monitoring and analysis service subscribes notifications of workflow executions to the SCALEA-G middleware. When the WF is executed, events containing execution status (e.g. submitted, active, ..) of activities are reported back to the monitoring and analysis service. Figure 8 shows the *Execution Status* display which monitors the execution status of activities. The left window shows one of Montage workflows. The right window displays execution status of activities of that workflow. We also can examine execution time of states during the runtime.

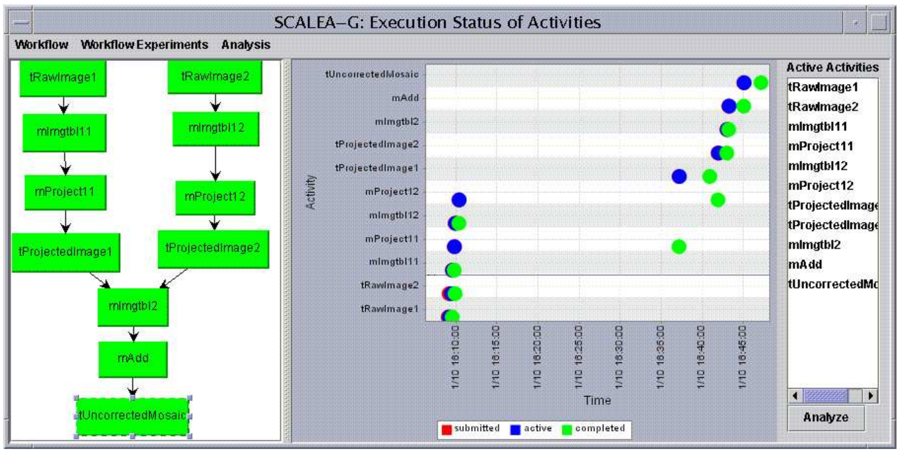


*Figure 8.* Monitoring execution status of a Montage workflow executed on 2 resources.

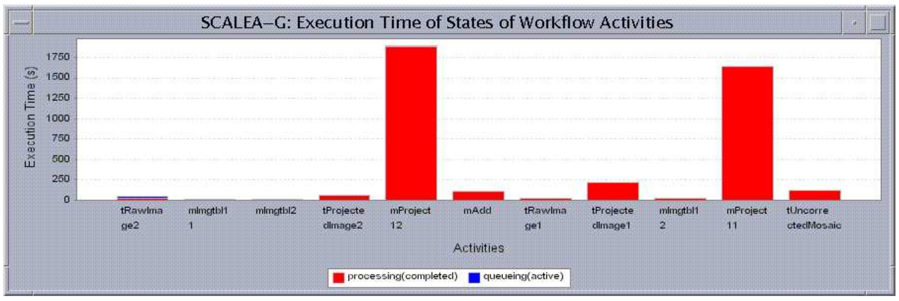
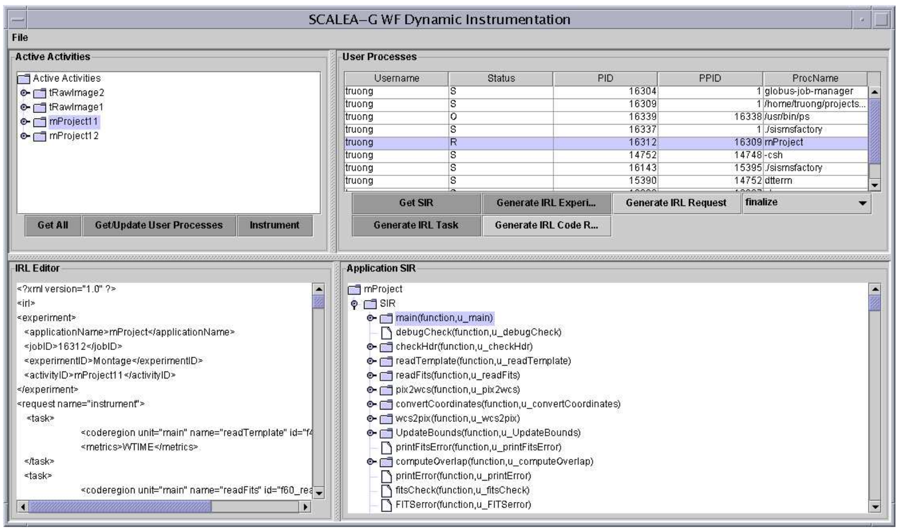


*Figure 9.* Execution time of states of Montage workflow executed on 2 resources.

For example, Figure 9 presents the execution time of states of the experiment presented in Figure 8.

## 6.2. Dynamic Instrumentation

When an activity is executed, its status is shown in the Execution Status diagram. The user then can start to instrument activity instances. Figure 10 depicts the GUI used to control the dynamic instrumentation of activity instances. On the top-left window, the user can choose an activity. For each compute node on which the selected activity instance executed, running processes can be examined by invoking *GetUserProcesses* operation, as shown in the top-right window of Figure 10. For a given process of the invoked application of an activity instance, the detailed SIR can be obtained by clicking *GetSIR* button, e.g. SIR of invoked application of activity `mProject1` is visualized in the bottom-right window in Figure 10. In the bottom-left window is an IRL request used to instrument selected code regions in the `main` unit with a metric `wtime` (wallclock time).



Active Activities
- tRawImage2
- tRawImage1
- mProject11
- mProject12

| Username | Status | PID | PPID | ProcName |
|---|---|---|---|---|
| truong | S | 16304 | 1 | globus-job-manager |
| truong | S | 16309 | 1 | /home/truong/projects... |
| truong | O | 16339 | 16338 | /usr/bin/ps |
| truong | S | 16337 | 1 | ./sismsfactory |
| truong | R | 16312 | 16309 | mProject |
| truong | S | 14752 | 14748 | -csh |
| truong | S | 16143 | 15395 | ./sismsfactory |
| truong | S | 15390 | 14752 | dtterm |

```
<?xml version="1.0" ?>
<irl>
<experiment>
  <applicationName>mProject</applicationName>
  <jobID>16312</jobID>
  <experimentID>Montage</experimentID>
  <activityID>mProject11</activityID>
</experiment>
<request name="instrument">
   <task>
                <coderegion unit="main" name="readTemplate" id="f4
                <metrics>WTIME</metrics>
  </task>
  <task>
                <coderegion unit="main" name="readFits" id="f60_rea
```

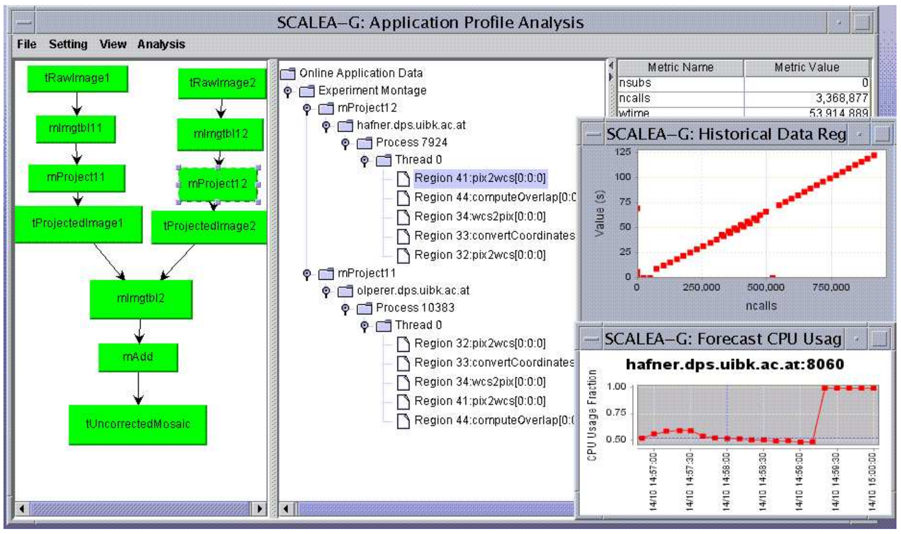
```
mProject
  SIR
    main(function,u_main)
    debugCheck(function,u_debugCheck)
    checkHdr(function,u_checkHdr)
    readTemplate(function,u_readTemplate)
    readFits(function,u_readFits)
    pix2wcs(function,u_pix2wcs)
    convertCoordinates(function,u_convertCoordinates)
    wcs2pix(function,u_wcs2pix)
    UpdateBounds(function,u_UpdateBounds)
    printFitsError(function,u_printFitsError)
    computeOverlap(function,u_computeOverlap)
    printError(function,u_printError)
    fitsCheck(function,u_fitsCheck)
    FITSerror(function,u_FITSerror)
```

*Figure 10.* GUI used to control the instrumentation of activity instances of a workflow.

## 6.3. Performance Analysis

When an invoked application of an activity instance is instrumented, the measurement data collected is analyzed by the performance analysis component. The performance analysis component retrieves profiling data through data subscription or query. Figure 11 presents the performance analysis GUI when analyzing a Montage workflow executed on two resources in UIBK. The left-pane shows the DAG of the WF. The middle-pane shows the dynamic code region call graph (DRG) of invoked applications of activities. We can examine the profiling data of instrumented code region on the fly. The user can examine the whole DRG of the application, or DRG of an activity instance (by choosing the activity in the DAG). By clicking on a code region, detailed performance metrics will be displayed in the right-pane. Depending on the invoked application, source code information may be available, thus code regions can be associated with their sources. We can examine historical profiling data of a code region, for example window *Historical Data* shows the execution time of code region `computeOverlap` executed on `hafner.dps.uibk.ac.at`. The user also can monitor resources on which activities are executed. For example, the window *Forecast CPU Usage* shows the forecasted CPU usage of `hafner.dps.uibk.ac.at`.

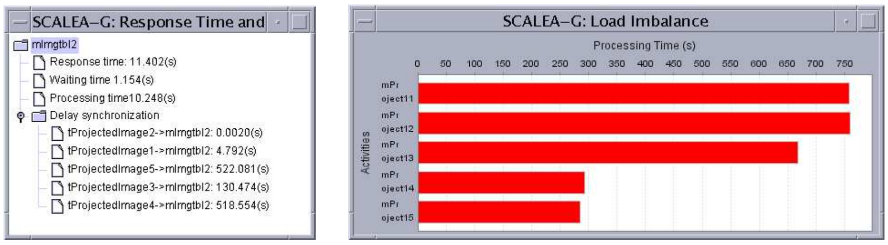
Figure 12(a) presents the response time and synchronization delay analysis for activity *mImgtbl2* when the Montage workflow, presented in Figure 7(c), is executed on 5 machines, 3 of AURORA6 and 2 of LINZ. The synchronization delays from *tProjectedImage3, 4, 5* to *tImgtbl2* are very high. This is caused by the high load imbalance between *mProject* instances, as



```
Online Application Data
  Experiment Montage
    mProject12
      hafner.dps.uibk.ac.at
        Process 7924
          Thread 0
            Region 41:pix2wcs[0:0:0]
            Region 44:computeOverlap[0:0
            Region 34:wcs2pix[0:0:0]
            Region 33:convertCoordinates
            Region 32:pix2wcs[0:0:0]
    mProject11
      olperer.dps.uibk.ac.at
        Process 10383
          Thread 0
            Region 32:pix2wcs[0:0:0]
            Region 33:convertCoordinates
            Region 34:wcs2pix[0:0:0]
            Region 41:pix2wcs[0:0:0]
            Region 44:computeOverlap[0:0
```

| Metric Name | Metric Value |
|---|---|
| nsubs | 0 |
| ncalls | 3,368,877 |
| wtime | 53,914,889 |

*Figure 11.* Performance analysis of workflow activities.



```
mImgtbl2
  Response time: 11.402(s)
  Waiting time 1.154(s)
  Processing time10.248(s)
  Delay synchronization
    tProjectedImage2->mImgtbl2: 0.0020(s)
    tProjectedImage1->mImgtbl2: 4.792(s)
    tProjectedImage5->mImgtbl2: 522.081(s)
    tProjectedImage3->mImgtbl2: 130.474(s)
    tProjectedImage4->mImgtbl2: 518.554(s)
```

(a) (b)

*Figure 12.* Analysis of Montage executed on 5 machines: (a) response time and synchronization delay of `mImgtbl`, and (b) load imbalance of `mProject`.

shown in Figure 12(b). The load imbalance is not due to the inequality of work distribution between `mProject` activities, but due to the differences in processing capability of resources in the Grid. The two machines in LINZ can process significantly faster than all machines in AURORA6. This detection indicates the workflow composition system and scheduling system do not take into account the processing capability of resources when constructing activities and distributing them on Grids.

Throughout the workflow development procedure, a subgraph named `mProjectedImage` which includes $tRawImage \rightarrow mImgtbl1 \rightarrow mProject1$ in single resource version is replaced by subgraphs of $tRawImage \rightarrow mImgtbl1 \rightarrow mProject1 \rightarrow tProjectedImage$ in a multi-resource version. These subgraphs basically provide projected images to the `mAdd` activity, therefore, we consider they are equivalent in terms of QoS (to the user point of view); they are replaced refinement graphs. We collect and store

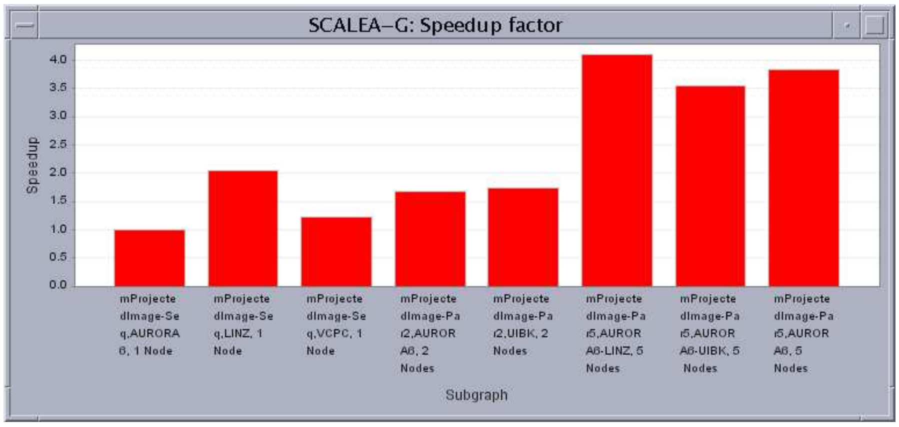


*Figure 13.* Speedup factor for subgraph `ProjectedImage` of Montage workflows.

performance of these subgraphs in different experiments. Figure 13 shows the speedup factor for the subgraph `mProjectedImage` of Montage workflows executed on several experiments. The execution of `mProjectedImage` of the workflow executed on single resource in LINZ is faster than that of its refinement graph executed on two resources (in AURORA6, or UIBK). However, the execution of `mProjectedImage` of workflow executed on 5 resources, 3 of AURORA6 and 2 of LINZ, is just very slightly faster than that executed on 5 resources of AURORA6. The reason is that the slower activities executed on AURORA6 resources have a significant impact on the overall execution of the whole `mProjectedImage` as presented on Figure 12(b).

## 7. Related Work

The Grid computing aims at addressing the interoperability and integration of diverse resources and services. To achieve that aims, it is necessary to provide well-defined interface to access these resources and services by hiding their specific features. The Grid Application Toolkit (GAT) [9] presents a high level application programming that shields low level details of underlying Grid sites from the user. The Grid dynamic instrumentation service and GAT follow the same general approach: using well-defined service interface to hide low level details of Grids.

Several tools support performance analysis for Grid applications such as GRM [12], OCM-G [13]. Our tool differs from these tools in many aspects. Firstly, our tool is OGSA-based service. Secondly, we support dynamic instrumentation of Grid workflow-based application. GRM, for example, supports only manual instrumentation while OCM-G combines source code instrumentation with a mechanism to dynamically enable instrumentation probes. OCM-G also supports profiling and and periodically updating monitoring data [14], but it limits to MPI programs. Existing tools supporting dynamic instrumentation, e.g., Paradyn [3] and DPCL [18] are not designed to work with the Grid. Nor do these tools provide enough accessible and interoperable interface that our Grid dynamic instrumentation service introduces. However, similar to Paradyn and DPCL, our Grid instrumentation service uses dynamic instrumentation techniques provided by Dyninst [15].

Monitoring of workflows has been discussed for many years. Many techniques have been introduced to study quality of service and performance models of workflows, e.g. [28, 16], and to support monitoring and analysis of the execution of the workflow on distributed systems, e.g., in [35, 8]. Our work and existing work share many general concepts of performance metrics and monitoring techniques for the workflow in distributed systems. However, most existing work concentrates on business workflows and Web services processes while our work targets to the performance of scientific workflows executed in Grids which are more diverse and dynamic, and inter-organizational. We support dynamic instrumentation of activity instances, monitoring and performance analysis of workflows based on not only execution status but also performance measurements obtained by instrumenting the invoked application, and resource monitoring data. The performance monitoring and analysis is not limited to activity level, but covers also code regions of invoked applications. Moreover, we support multi-workflow analysis.

Numerous performance monitoring and analysis tools have been developed for the Grid, as studied in [21, 45], but most of them do not support the monitoring and analysis of Grid scientific WFs. Most effort on supporting the scientist to develop Grid workflow-based applications is focused on workflow languages, and workflow construction and execution systems, but not concentrated on monitoring and performance analysis of the Grid WFs. P-GRADE [27] is one of few tools that supports tracing of workflow applications. Instrumentation probes are automatically generated from the graphical representation of the application. It however limits to MPI and PVM applications. Our Grid workflow monitoring and performance analysis service combines online monitoring execution of activities with online profiling analysis. The support of dynamic instrumentation does not limit to MPI or PVM applications.

## 8. Conclusion and Future Work

The dynamics and diversity of the Grid requires a dynamic and flexible mechanism in conducting the performance analysis of Grid applications. This paper presents a dynamic approach to the performance instrumentation, monitoring, and analysis of Grid workflows. We have introduced a novel Grid service to support dynamic instrumentation of workflow-based applications. We have presented a Grid performance analysis service that can be used to monitor and analyze the performance of scientific workflows in the Grid on the fly. The Grid performance analysis service which combines dynamic instrumentation, activity execution monitoring, and performance analysis of WFs in a single system presents a dynamic and flexible way to conduct the performance monitoring and analysis of scientific WFs. Workflows and their relevant performance metrics are stored and utilized for for comparing the performance of subgraphs of workflows and supporting multi-workflow analysis. We are currently working towards the full implementation of our prototype, and are in the process to integrate the prototype into the ASKALON toolset [19].

We should stress that the dynamic approach for conducting the performance monitoring and analysis of scientific WFs we present does not require us to perform the monitoring and analysis at all levels such as whole-workflow, activity and code region level. We should conduct the performance analysis from a high level to a low level and the performance analysis should be based on specific WFs. For example, if we consider invoked applications as black-boxes, we can collect only activity states and events from the workflow invocation and control service. However, depending on workflows (e.g., workflows based on Web services, activities implemented in Java invoke legacy C/Fortran code), through the instrumentation of invoked applications of activities, performance measurements of interactions among activities (e.g., an invoked application of an activity calls a function of the invoked application of another activity) or within an activity (e.g., an Java method calls C/Fortran functions) may be collected and analyzed.

In the current implementation, we manually instrument WIC in order to get the execution status of activities. To avoid that, we can extend workflow specification language with directives specifying monitoring conditions. These directives will be translated into code used to publish events containing execution status of activities into the monitoring middleware. WIC can also provide well-defined interfaces for the monitoring service to access execution status of activities. Our performance monitoring and analysis limits to DAG workflows. Recently, scientific workflows which have structured loops (e.g., do while structure) are proliferated. Currently, we are investigating to extend our techniques to cover workflows with structured loops. Another aspect is that while we focus on invoked applications as executable programs (each activity instance invokes an executable program), there exist workflows that each activity instance invokes a Web Service operation (likely written in Java). This type of workflows will require different instrumentation mechanism, e.g. dynamic instrumentation of Java services. Meanwhile, the process of analysis, monitoring and instrumentation is controlled by the end-user, but it should be automated. The issues mentioned above will be addressed in the $6^{th}$FP EU K-Wf Grid project [2] in order to support the performance monitoring and analysis of Grid applications in a knowledge-based workflow system.

## References

1. 'http://www-unix.globus.org/toolkit/docs/3.2/core/developer/message_security.html'.
2. 'K-WF Grid Project. http://www.kwfgrid.net'.
3. 'Paradyn Parallel Performance Tools, http://www.cs.wisc.edu/paradyn/'.
4. 'The Grid Resource Allocation and Management (GRAM), http://www-unix.globus.org/toolkit/docs/3.2/gram/ws/index.html'.
5. 'Worldflow Management Coalition: Terminology and glossary. Technical Report WFMC-TC-1011, Feb 1999'.
6. A. Cooke et al.: 2003, 'R-GMA An Information Integration System for Grid Monitoring'. In: *Proceedings of 11th International Conference on Cooperative Information Systems (CoopIS 2003).* Sicily,Italy.
7. Aalst, W. M. P. V. D., A. H. M. T. Hofstede, B. Kiepuszewski, and A. P. Barros: 2003, 'Workflow Patterns'. *Distrib. Parallel Databases* **14**(1), 5–51.
8. Abate, A. F., A. Esposito, N. Grieco, and G. Nota: 2002, 'Workflow performance evaluation through WPQL'. In: *Proceedings of the 14th international conference on Software engineering and knowledge engineering*. pp. 489–495.
9. Allen, G., K. Davis, T. Goodale, A. Hutanu, H. Kaiser, T. Kielmann, A. Merzky, R. van Nieuwpoort, A. Reinefeld, F. Schintke, T. Schütt, E. Seidel, and B. Ullmer: 2005, 'The Grid Application Toolkit: Towards Generic and Easy Application Programming Interfaces for the Grid'. **93**(3), 534–550.
10. AustrianGrid, 'http://www.austriangrid.at/'.
11. Balaton, Z. and G. Gombas: 2003, 'Resource and Job Monitoring in the Grid'. In: *Proceedings. Euro-Par 2003 Parallel Processings*. Klagenfurt, Austria.
12. Balaton, Z., P. Kacsuk, N. Podhorszki, and F. Vajda: 2001, 'From Cluster Monitoring to Grid Monitoring Based on GRM'. In: *Proceedings. 7th EuroPar'2001 Parallel Processings*. Manchester, UK, pp. 874–881.
13. Balis, B., M. Bubak, W. Funika, T. Szepieniec, and R. Wismüller: 2002, 'An Infrastructure for Grid Application Monitoring'. *LNCS* **2474**, 41–49.
14. Balis, B., M. Bubak, W. Funika, R. Wismüller, M. Radecki, T. Szepieniec, T. Arodz, and M. Kurdziel: 2004, 'Grid environment for on-line application monitoring and performance analysis.'. *Scientific Programming* **12**(4), 239–251.
15. Buck, B. and J. K. Hollingsworth: 2000, 'An API for Runtime Code Patching'. *The International Journal of High Performance Computing Applications* **14**(4), 317–329.
16. Cardoso, J., A. P. Sheth, and J. Miller: 2003, 'Workflow Quality of Service'. In: *Proceedings of the IFIP TC5/WG5.12 International Conference on Enterprise Integration and Modeling Technique*. pp. 303–311.
17. Deelman, E., J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, K. Blackburn, A. Lazzarini, A. Arbree, and S. Koranda: 2003, 'Mapping Abstract Complex Workflows onto Grid Environments'. *Journal of Grid Computing* **1**, 25–39.
18. DeRose, L., T. Hoover Jr., and J. Hollingsworth: 2001, 'The Dynamic Probe Class Library: An Infrastucture for Developing Instrumentation for Performance Tools'. In:

*Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS-01)*. Los Alamitos, CA, pp. 66–66.

19. Fahringer, T., A. Jugravu, S. Pllana, R. Prodan, C. S. Junior, and H.-L. Truong: 2005, 'ASKALON: A Tool Set for Cluster and Grid Computing'. *Concurrency and Computation: Practice and Experience* **17**(2-4), 143–169.
20. Foster, I. and C. Kesselman (eds.): 1999, *The Grid: Blueprint for a New Computing Infrastructure*. San Francisco, CA: Morgan Kaufmann.
21. Gerndt, M., R. Wismueller, Z. Balaton, G. Gombas, P. Kacsuk, Z. Nemeth, N. Podhorszki, H.-L. Truong, T. Fahringer, M. Bubak, E. Laure, and T. Margalef: 2004, *Performance Tools for the Grid: State of the Art and Future*, Vol. 30 of *Research Report Series, Lehrstuhl fuer Rechnertechnik und Rechnerorganisation (LRR-TUM) Technische Universitaet Muenchen*. Shaker Verlag. ISBN 3-8322-2413-0.
22. Globus Project, 'http://www.globus.org'.
23. gSOAP: C/C++ Web Services and Clients, 'http://www.cs.fsu.edu/˜engelen/soap.html'.
24. Gunter, D., B. Tierney, B. Crowley, M. Holding, and J. Lee: 2000, 'NetLogger: A Toolkit for Distributed System Performance Analysis'. In: *Proceedings of the IEEE Mascots 2000 Conference*.
25. JFreeChart, 'http://www.jfree.org/jfreechart/'.
26. JGraph, 'http://www.jgraph.com/'.
27. Kacsuk, P., G. Dozsa, J. Kovacs, R. Lovas, N. Podhorszki, Z. Balaton, and G. Gombas: 2003, 'P-GRADE: a Grid Programming Environment'. *Journal of Grid Computing* **1**(2), 171–197.
28. Kim, K.-H. and C. A. Ellis: 2001, 'Performance Analytic Models and Analyses for Workflow Architectures'. *Information Systems Frontiers* **3**(3), 339–355.
29. Krishnan, S., P. Wagstrom, and G. von Laszewski: 2002, 'GSFL : A Workflow Framework for Grid Services'. Technical report, Argonne National Laboratory, 9700 S. Cass Avenue, Argonne, IL 60439, U.S.A.
30. Laszewski, G., I. Foster, J. Gawor, and P. Lane: 2001, 'A Java Commodity Grid Kit'. *Concurrency and Computation: Practice and Experience* **13**(643-662).
31. Ludaescher, B., I. Altintas, C. Berkley, D. Higgins, E. Jaeger-Frank, M. Jones, E. Lee, J. Tao, and Y. Zhao: 2005, 'Scientific Workflow Management and the Kepler System'. *Concurrency and Computation: Practice & Experience, Special Issue on Scientific Workflows*.
32. Miller, B., M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall: 1995, 'The Paradyn Parallel Performance Measurement Tool'. *IEEE Computer* **28**(11), 37–46.
33. Montage, 'http://montage.ipac.caltech.edu'.
34. Podhorszki, N. and P. Kacsuk: 2003, 'Monitoring Message Passing Applications in the Grid with GRM and R-GMA'. In: *Proceedings of EuroPVM/MPI'2003*. Venice, Italy.
35. Savarimuthu, B. T. R., M. Purvis, and M. Fleurke: 2004, 'Monitoring and controlling of a multi-agent based workflow system'. In: *Proceedings of the second workshop on Australasian information security, Data Mining and Web Intelligence, and Software Internationalisation*. pp. 127–132.
36. Seragiotto, C., T. Li, T. Fahringer, B. Mohr, M. Gerndt, and H.-L. Truong: 2005, 'Standardized interfaces for representing, instrumenting, and monitoring Fortran, Java, C, and C++ Programs'. *Concurrency and Computation: Practice and Experience*. On submission.
37. Singh, M. P. and M. A. Vouk: 1996, 'Scientific Workflows'. In: *Position paper in Reference Papers of the NSF Workshop on Workflow and Process Automation in Information Systems: State-of-the-art and Future Directions*.

38. Sowa, J. F.: 2000, *Knowledge Representation: logical, philosophical, and compuational foundations*. Pacific Grove, CA: Brooks/Cole.
39. The Condor Team, 'DAGMan (Directed Acyclic Graph Manager)'. http://www.cs.wisc.edu/condor/dagman/.
40. Truong, H.-L. and T. Fahringer: 2003, 'SCALEA: A Performance Analysis Tool for Parallel Programs'. *Concurrency and Computation: Practice and Experience* **15**(11-12), 1001–1025.
41. Truong, H.-L. and T. Fahringer: 2004, 'SCALEA-G: a Unified Monitoring and Performance Analysis System for the Grid'. *Scientific Programming* **12**(4), 225–237. IOS Press.
42. Vouk, M. A. and M. P. Singh: 1996, 'Quality of Service and Scientific Workflows'. Technical Report TR-96-19, Department of Computer Science, North Carolina State University. Thu, 19 Sep 96 22:59:36 GMT.
43. Wainer, J., M. Weske, G. Vossen, and C. B. Medeiros: 1996, 'Scientific Workflow Systems'. In: *Proc. NSF Workshop on Workflow and Process Automation in Information Systems: State-of-the-Art and Future Directions*. Athens, Georgia.
44. Welch, V., F. Siebenlist, I. Foster, J. Bresnahan, K. Czajkowski, J. Gawor, C. Kesselman, S. Meder, L. Pearlman, and S. Tuecke: 2003, 'Security for Grid Services'. In: *12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03)*. Seattle, Washington, pp. 48–57.
45. Zanikolas, S. and R. Sakellariou: 2005, 'A Taxonomy of Grid Monitoring Systems'. *Future Generation Computing Systems* **21**(1), 163–188.