An Instrumentation Infrastructure for Grid Workflow Applications ⋆

Bartosz Balis[1], Hong-Linh Truong[3], Marian Bubak[1,2], Thomas Fahringer[3], Krzysztof Guzy[1], Kuba Rozkwitalski[2]

[1] Institute of Computer Science, AGH, Poland
{balis, bubak}@uci.agh.edu.pl,kubaroz@gmail.com
[2] Academic Computer Centre – CYFRONET, Poland
[3] Institute of Computer Science, University of Innsbruck, Austria
{truong,tf}@dps.uibk.ac.at

Abstract. Grid workflows are normally composed of multilingual applications. Monitoring such multilingual workflows in the Grid requires an instrumentation infrastructure that is capable of dealing with workflow components implemented in different programming languages. Moreover, Grid workflows introduce multiple levels of abstraction and all levels must be taken into account in order to understand the performance behaviour of the workflows. As a result, any instrumentation infrastructure for Grid workflows should assist the user/tool to conduct the monitoring and analysis at multiple levels of abstraction. However, the instrumentation of multilingual workflows at different levels of abstraction should be done in a unified way in an integrated environment, although it obviously employs various techniques. This paper presents an novel instrumentation infrastructure for Grid services that addresses the above-mentioned issues by supporting the instrumentation of multilingual Grid workflows at multiple levels of abstraction using a unified, highly interoperable interface.
Keywords: grid, monitoring, instrumentation, legacy applications

1 Introduction

Grid workflows based on modern service-oriented architecture (SOA) are normally multilingual, i.e. combine Java-based services with invocations of legacy code. Instrumentation of Grid workflows is a required step in order to collect monitoring data for debugging or analyzing performance of Grid workflows. However, the monitoring of multilingual applications in the Grid not only requires different instrumentation techniques but also interactions between various services involved in the instrumentation and monitoring have to use a well-defined, highly interoperable interface. Often these requirements are conflicting and cannot be unified into a single framework. However, while working with a Grid workflow, the user does not want to use different means to instrument and monitor different pieces of his/her workfow.

This paper presents a novel instrumentation infrastructure for Grid services that addresses the above-mentioned issues by supporting the dynamically enabled instrumen-

⋆ The work described in this paper is supported by the European Union through the IST-2002-511385 project K-WfGrid.

tation of multilingual Grid workflows at multiple levels of abstraction using a unified, highly interoperable interface.

The rest of this paper is organized as follows: Section 2 discusses concepts of instrumentation at multiple levels. Section 3 presents languages describing instrumented applications and instrumentation requests. Section 4 details instrumentation techniques. We present experiments in Section 5. Section 6 outlines the related work, followed by a summary of the paper and an outlook to the future work in Section 7.

2 Multiple Levels of Instrumentation

Grid workflows, in our view, introduce multiple levels of abstraction including *workflow, workflow region, activity, invoked applicaton* and *code region* [1]. To understand the performance behaviour of Grid workflows, it necessitates to analyze and correlate various performance metrics collected at these levels. Therefore, performance monitoring and analysis tools for Grid workflows have to operate at different levels and to correlate performance metrics between these levels. To provide performance metrics at workflow, workflow region and activity levels, the tools have to conduct the monitoring and measurement by instrumenting the enactment engine (EE) which is responsible for executing workflows. On the other hand, for analyzing metrics at invoked applications and code regions, the tools have to instrument invoked applications.

Our approach is that we apply static instrumentation techniques to the enactment engine. For example, we can instrument the enactment engine by statically inserting sensors into source code of EE in order to monitor execution behaviour of workflows and workflow activities. For applications invoked within workflow activities, we apply dynamically enabled instrumentation. That is the instrumentation is conducted before the runtime and the measurement is dynamically enabled at runtime. Even though dynamically enabled instrumentation mostly supports measuring performance metrics of program units and function calls, but not of arbitrary code regions, for Grid workflow applications, we believe that measuring the performance at a level of program unit and function call should be enough. Due to the complex and distributed nature of Grid workflows, we believe that when analyzing the performance of workflow applications, most users are interested in observing the performance at the level of workflow, program unit and function call, rather than at loop or statement levels. The invoked applications, e.g., Grid service operations or legacy scientific programs, are also diverse and can be multilingual. Therefore, we must have a common infrastructure that allows us to instrument such applications. While we could reuse existing instrumentation techniques, we have to provide a common interface for performing the instrumentation of different types of applications.

One of important goals of our work is to support various forms of legacy code (libraries, forked jobs, parallel applications). Supporting legacy code is important, since today most mature applications are written in Fortran, C or C++ and rather adapted to the Grid than re-engineered to Java-based services.

3 Program Representations and Instrumentation Requests

As discussed in the previous section, the instrumentation system has to support multiple levels of abstraction and to work with diverse and multilingual applications. Therefore, we need various, quite different, instrumentation techniques, each suitable for a specific type of applications and abstraction level. Besides the detailed instrumentation techniques, we must address interface between the instrumentation requestor and the instrumentation system. The interface between them answers two basic questions: (1) how the structure of multilingual applications are represented?, and (2) how the instrumentation requests are specified? The main desire in answering these questions is that the requestor should treat different applications implemented in different languages using a unified means. To this end, we have to provide a neutral way to represent objects to be instrumented and requests used to control the instrumentation. Our approach is to use SIR [2] for representing applications and to develop XML-based instrumentation requests based on that representation.

Standardized Intermediate Representation: We have developed an XML-based representation named SIRWF (Standardized Intermediate Representation for Workflows) to describe the structure of invoked applications of workflow activities; SIRWF is developed based on SIR [2]. The main objective is to express information of different types of applications, which is required by instrumentation systems, using a single intermediate representation. The SIR represents most interesting information for instrumentation, such as program units and functions, while shields low level details of the application from the user. SIRWF, a simplified version of SIR, currently can represent invoked applications and code regions (at program unit and function call levels) of workflows.

Workflow Instrumentation Request language (WIRL): Given application structure represented in SIRWF, we develop an XML-based language for specifying instrumentation requests named WIRL (Workflow Instrumentation Request Language). WIRL, based on IRL [3], is an XML-based language. A WIRL request consists of experiment context and instrumentation tasks. Experiment context (e.g., activity identifier, application name, computational node, etc.) identifies applications to be instrumented and includes information used to correlate monitoring data to the monitoring workflow. Instrumentation tasks specify instrumentation operations. Examples of instrumentation tasks can be a request for all instrumented functions within an application, to enable or disable an instrumented code, etc. The current requests include (1) `ATTACH` to attach/prepare the instrumentation of a given application, (2) `GETSIR` to get SIR of the application, (3) `ENABLE` to enable/instrument a code region, (4) `DISABLE` to disable/deinstrument a code region, and (5) `FINALIZE` to finish the instrumentation.

Code region identifiers are obtained from the list of functions provided by the instrumentation system. In an instrumentation request, performance metrics as well as user-defined events can be specified. Performance metrics are defined by the metric ontology [1] while user-defined events include event names with associated event attributes (names and values). For the instrumentation of user-defined events, the request also allows the client to specify the location to which the event probes should be inserted, e.g., before or after a function call.

4 Instrumentation Infrastructure

Fig. 1 depicts a basic architecture in which entities involved in instrumentation and monitoring of a sample workflow application are shown. In this figure, a workflow enactment engine *GWES* (Grid Workflow Execution Service) [4] creates a workflow activity (Activity 1) which in turn invokes an MPI application. *GEMINI* is the monitoring and instrumentation infrastructure which implements and adapts various instrumentation techniques and systems in a single and unified framework. GEMINI executes instrumentation and data subscription requests and notifies subscribers about new data. *Portal* is the user interface from which the user submits workflows, as well as controls the instrumentation, performance monitoring and analysis. The figure puts emphasis on how the events related to the execution of the workflow flow from various distributed locations: GWES, an invoked application running in a workflow activity, and a legacy MPI application. The framework needs to collect those pieces of data, correlate the collected data, and present them to clients, e.g. in the portal.

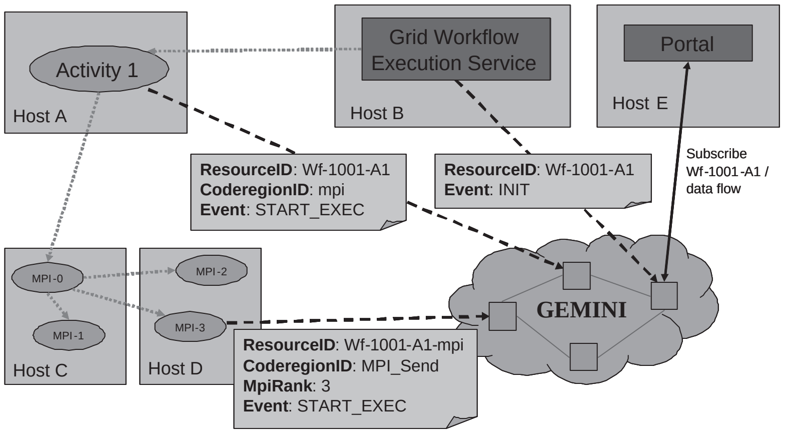

Fig. 1. Instrumentation framework architectuire

Fig. 2 presents the course of action in a basic instrumentation and monitoring scenario (MPI part of the application being omitted for simplification). In this scenario, after the user submits a workflow to GWES, GWES initializes the workflow, enacts the consecutive activities and invokes *invoked applications* performing the task of activities. GWES also sends some events to GEMINI concerning workflow status (not all shown in the figure). Within GWES, every single workflow or activity instance is associated with a unique ID. When an invoked application of an activity instance is started, GWES sends the unique ID of the activity instance to the invoked application (see section 4.2). All IDs associated with activity instances can be obtained from GWES through the Portal. When the invoked application starts its execution, it registers itself to GEMINI by passing its unique ID. Meanwhile, by using the ID, the user can select invoked applications of interest and then request SIRWF of the selected invoked application in order

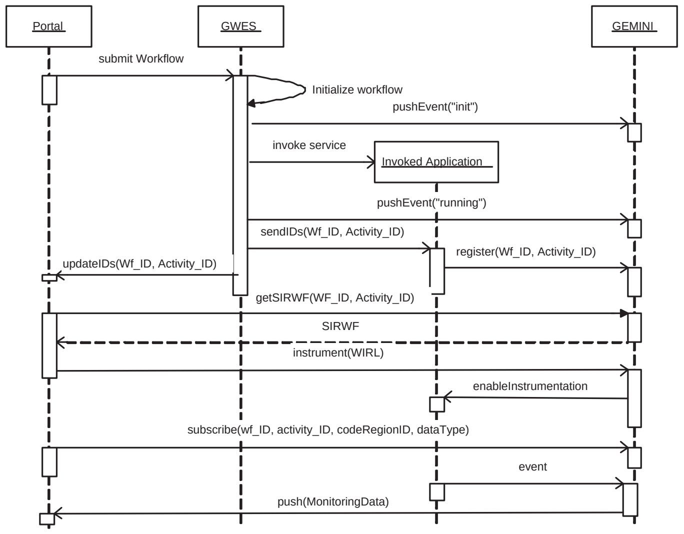

Fig. 2. Multiple levels of instrumentation scenario

to perform the monitoring and measurement. Based on the SIRWF, the user can select code regions and specify interesting metrics to be evaluated. A WIRL request will be created and sent to GEMINI which activates the measurement. In order to receive the data, the user specifies a PDQS (Performance Data Query and Subscription) request and sends it to GEMINI. When the execution of the invoked application reaches the instrumentation code, events containing monitoring data are generated and monitoring data is sent to GEMINI which in turn pushes the data to the subscriber (user).

4.1 Instrumentation at Multiple Levels of Abstraction

Instrumentation of Workflow Execution Service: Monitoring data at the level of workflow, workflow region and activity can be obtained from the GWES. GWES controls and executes the workflow and its activities so it can provide a number of events relevant to the execution of workflows such as events indicating the workflow and activity state changes (*initiated, running, complete*, etc.) Currently GWES is manually instrumented by inserting sensors, which collect monitoring data, into the source code. Applying only static instrumentation to GWES should be sufficient because only a few places in the GWES code have to be instrumented and as the overhead of invoking the probe functions is minimal, the instrumentation can be permanently installed and active.

Instrumentation of Invoked Applications and Code Regions of Java-based Services: In doing this, we currently employ byte-code instrumentation techniques and dynamically control the measurement process at runtime. Sensors are inserted into the byte-code using the BCEL tool [5]. At the same time SIRWF is created and saved with the class. Instrumented versions of classes are placed in different locations than the original ones. Instrumentation of an individual code region is conditional. Dynamic activation and deactivation of the instrumentation amounts to changing the value of a condition variable at runtime.

Instrumentation of Invoked Applications and Code Regions of Legacy Code: The possibility to use legacy code in services is twofold: (1) as legacy libraries invoked by means of JNI (Java Native Interface) calls or (2) as legacy jobs submitted from within a service. The first case can be handled in the same way as instrumentation of code regions described in Section 4.1, only tools and libraries for legacy code have to be developed. In the latter case, the legacy jobs invoked from services are often parallel applications, for example computationally intensive simulations. In our framework, monitoring of such applications will be handled by an external monitoring system, the OCM-G [6]. To be integrated seamlessly with the GEMINI, OCM-G has to provide SIRWF for legacy jobs, be able to handle WIRL-based instrumentation requests and represent monitoring in GEMINI data representation. To this end, we have substantially extended OCM-G, for example by introducing a SIRWF generation, and a GEMINI OCM-G-sensor for integrating OCM-G with GEMINI in terms of interfaces and data representation. The detailed process of OCM-G adaptation, instrumentation, and SIRWF generation for legacy programs is presented in [7].

Selection of Instrumentation Scope: A general problem in low-level monitoring and instrumentation of applications is how to select the proper parts of the applications to be instrumented and taken into account in monitoring. The application usually consists of hundreds of code units (classes, functions, files and libraries) of which many are irrelevant for the user, since they are not part of application's logic but belong to external libraries, etc. This problem is valid both for Java applications and e.g. C applications. The method, which we currently employ, is to explicitly specify the subset of code units (e.g. jars, classes or C-files) that are of interest and only those will be instrumented and included in SIRWF.

4.2 Passing Correlation Identifier to Invoked Applications

Monitoring data concerning a single workflow is generated in distinct places, mainly within GWES and invoked applications. In order to correlate those different pieces of data together, and to be able to correlate data requests with the incoming monitoring data, we need to associate the monitoring data with a unique ID, a 'signature'. In our case, this ID is the workflow and activity IDs, both generated by GWES. Obviously those IDs must also be passed to invoked applications which are web service operations.

This problem is a case of a more general issue of sending *execution context* to web services. However, Web services currently do not support any context information. For pure, stateless web services, the only information passed to an invoked service is via operation's parameters. There are a few possibilities to pass context information, as discussed in [8], for example: (a) by extending service's interface with additional parameter for context information, (b) by extending the data model of the data passed to a service, (c) by inserting additional information in SOAP header, and reading it from within the service. An additional option is to use the state provided by a service, but this only works for services that support state, e.g. WSRF-based ones.

We have used the SOAP-header approach, since it is transparent and does not require the involvement of services' developers. Thus, when invoking services, GWES inserts additional information to SOAP headers which is accessed in the services by instrumentation initialization functions to obtain workflow and activity identifiers.

4.3 Organizing and Managing Monitoring Data

The instrumentation supports gathering monitoring data at multiple levels; data collected within an experiment (e.g., of a workflow) is determined through IDs. Currently the framework does neither support merging data from different levels nor provide a storage system managing that data, although part of monitoring data is temporarily stored in distributed GEMINI monitoring services. The main goal of the framework is to support online performance monitoring and analysis so that the monitoring data is just returned to clients (e.g., performance analysis services) via query or subscription and the clients will analyze the monitoring data.

5 Experiment

We have implemented a prototype of our instrumentation infrastructure. The current implementation supports, among others, generation of SIRWF, static insertion of probes for Java and C applications, and dynamic control (activation/deactivation) of instrumentation based on SIRWF representation. In this section we present an experiment to illustrate our concepts.

In our experiment, we use GWES to invoke a service for numerical integration. The actual computation is done by a parallel MPI job invoked from the service which performs a parallel adaptive integration algorithm. The monitoring and instrumentation of the MPI application is realized by the OCM-G monitoring system which is integrated with GEMINI, as described in [7].

In our experiment, we wanted to obtain an event trace showing the consecutive stages of execution of the workflow – from the GWES events to the execution of code regions in the MPI application; we were interested in parts in which the actual computation was performed. After the application has been started, it was suspended so that required measurements could be set up from the beginning. The SIRWF obtained for the invoked application is shown in Fig. 3 (a). The `experiment` section identifies a particular runtime part of the workflow (e.g. an invoked application, a legacy job). Within invoked application `numeric.pa_integrate`, a subroutine named `mpi_integrate`. This subroutine actually invokes a legacy application written in MPI. The legacy code consists of a function named `main` and other functions such as `eval` and `MPI_Send`. In the monitoring view, we are interested in monitoring the following calling chain: `mpi_integrate` → `main` → `eval`. Using information in the SIRWF, we decide to instrument the `numeric.pa_integrate` invoked application, the `main` functions and the calls to `eval` function in all MPI processes. Part of the corresponding WIRL submitted to GEMINI is shown in Fig. 3. As GWES is statically instrumented, events from GWES are always generated and reported to GEMINI.

When the instrumentation has been set up, we send a subscription request to GEMINI to obtain all events related to the monitored workflow. We show the results in a form

```
<sirApp>
 <experiment> ... </experiment>
  <sir><unit type="subroutine"
   name="numeric.pa_integrate">
   <coderegion type="call" id="1">
    <calee name="mpi_integrate"/>
   </coderegion>
  </unit>
  <unit type="subroutine"
        name="mpi_integrate">
   <coderegion type="call" id="2">
     <calee name="main"/>
   </coderegion>
  </unit>
  <unit type="function" name="main">
   <coderegion type="call" id="3">
     <calee name="MPI_Send"/>
   </coderegion>
     ...
   <coderegion type="call" id="14">
     <calee name="eval"/>
   </coderegion>
     ...
 </unit></sir>
</sirApp>
```

(a)

```
<wirl>
 <request name="ENABLE">
   <experiment>
      <wfInstanceID>wf1</wfInstanceID>
      <pu><activityID>wf1_a1</activityID>
          <invokedAppId>wf1_a1_ia1
          </invokedAppId>
      </pu>
   </experiment>
   <task>
     <coderegion name="main" id="2"/>
     <coderegion name="eval" id="14"/>
     <events location="BEFORE">
        <event>
          <eventname>BEGIN_EXEC</eventname>
        </event>
     </events>
     <events location="AFTER">
        <event>
          <eventname>END_EXEC</eventname>
        </event>
     </events>
   </task>
 </request>
</wirl>
```

(b)

Fig. 3. Instrumentation: (a) SIRWF (simplified) and (b) WIRL (simplified)

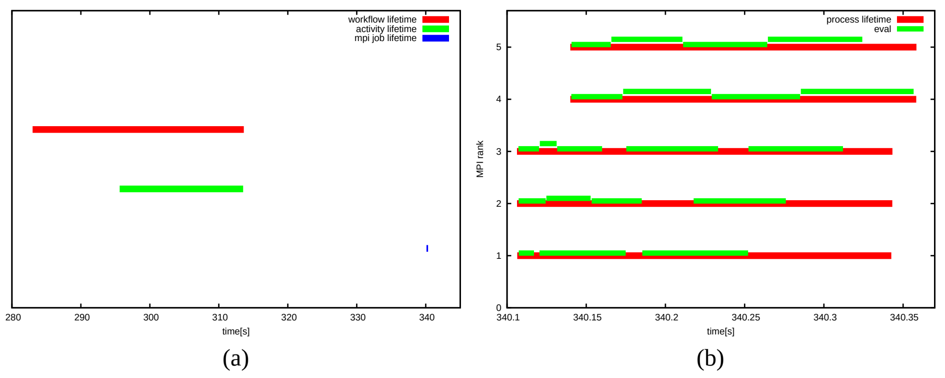

Fig. 4. Monitoring results: (a) Workflow perspective (b) Legacy perspective

of bars representing the time span of different execution units. Fig. 4 (a) shows a comparison between the execution of the entire workflow, the single activity that has been executed, and the MPI job executed from the activity. As we can see, the delay related to workflow submission is enormous in comparison to the actual computation. Fig. 4 (b) uses a different scale to show the time breakdown for individual code regions inside the MPI processes. Multiple calls to the `eval` function are due to the iterative process of adaptive integration.

6 Related Work

Techniques like source code, binary or dynamic instrumentation are well-known and widely used such as in TAU [9], Paradyn [10], Dyninst [11]. Many efforts have been spent on Java byte-code and dynamic instrumentation, e.g., [12–15]. The main aspect in which our work differs from these tools is that we focus on developing extensible and interoperable instrumentation interfaces, e.g. WIRL and SIRWF, for performance measurement of Grid applications and on integrating various instrumentation mechanisms for Grid applications. While the above-mentioned tools also target to large scale, long-run applications, they support and are developed on high performance parallel systems where issues of heterogeneity, loosely integration, multilingual applications, and interoperability are not centered points. PPerfGrid [16] geographically collects and exchanges performance data of parallel programs by using Grid services but it does not address the instrumentation and monitoring of Grid workflow-based applications. To our best knowledge, there is no instrumentation infrastructure that supports multiple levels of instrumentation for multilingual Grid workflows.

The APART working group proposes the SIR as a high level means to describe structure of applications to be instrumented by using XML [2]. However, APART SIR represents only single applications, not workflows. We adopt SIR, simplifying and using it for our instrumentation. There are works on proposing XML-based requests for instrumentation such as MIR[17] and MRI [18]. However, those requests are used for instrumentation of individual invoked applications, rather than for workflows.

AKSUM tool supports dynamic instrumentation of distributed Java applications [13]. The instrumentation also uses SIR and MIR. However, it neither supports Grid- nor workflow-based applications. Moreover, the instrumentation is limited to Java code only while our instrumentation supports legacy code (e.g., C/Fortran) as well.

7 Conclusion and Future Work

The contribution of this paper is a novel framework for instrumentation of multilingual and Grid-based workflows. The instrumentation and monitoring of complex, multilingual, workflow of Grid services are handled seamlessly in an integrated system. By employing GEMINI, the concepts of SIR and WIRL, and various instrumentation techniques, we are able to provide a unified and comprehensive view on the execution of workflow applications at various levels.

At the moment we have a working prototype of our infrastructure. We are working on full integration of service-level and legacy-level instrumentation, full implementation of SIRWF, transparency of application monitoring, and a fully reliable passing of workflow and activity IDs to all parts of the workflow application. Moreover, the scalability and robustness of the system need to be evaluated.

References

1. Truong, H.L., Fahringer, T., Nerieri, F., Dustdar, S.: Performance Metrics and Ontology for Describing Performance Data of Grid Workflows. In: Proceedings of IEEE International Symposium on Cluster Computing and Grid 2005, 1st International Workshop on Grid Performability, Cardiff, UK, IEEE Computer Society Press (2005)

2. Seragiotto, C., Truong, H.L., Fahringer, T., Mohr, B., Gerndt, M., Li, T.: Standardized Intermediate Representation for Fortran, Java, C and C++ Programs. Technical report, Institute for Software Science, University of Vienna (2004)
3. Truong, H.L., Fahringer, T.: SCALEA-G: a Unified Monitoring and Performance Analysis System for the Grid. Scientific Programming **12** (2004) 225–237 IOS Press.
4. Neubauer, F., Hoheisel, A., Geiler, J.: Workflow-based Grid Applications. **22** (2006) 6–15
5. BCEL: Byte Code Engineering Library (2006) http://bcel.sourceforge.net.
6. Balis, B., Bubak, M., Radecki, M., Szepieniec, T., Wismüller, R.: Application Monitoring in CrossGrid and Other Grid Project s. In: Grid Computing. Proc. Second European Across Grids Conference, Nicosia, Cyprus, Springer (2004) 212–219
7. Baliś, B., Bubak, M., Guzy, K.: Fine-grained Instrumentation and Monitoring of Legacy Applications in a Service-Oriented Environment. In: ICCS 2006, Reading, UK (2006)
8. Brydon, S., Kangath, S.: Web Service Context Information. (2005) https://bpcatalog.dev.java.net/nonav/soa/ws-context/index.html.
9. Sheehan, T., Malony, A., Shende, S.: A runtime monitoring framework for tau profiling system. In: Proceedings of the Third International Symposium on Computing in Object-Oriented Parallel Environments(ISCOPE's 99), San Franciso (1999)
10. Miller, B., Callaghan, M., Cargille, J., Hollingsworth, J., Irvin, R., Karavanic, K., Kunchithapadam, K., Newhall, T.: The Paradyn Parallel Performance Measurement Tool. IEEE Computer **28** (1995) 37–46
11. Buck, B., Hollingsworth, J.K.: An API for Runtime Code Patching. The International Journal of High Performance Computing Applications **14** (2000) 317–329
12. Guitart, J., Torres, J., Ayguad, E., Labarta, J.: Java instrumentation suite: Accurate analysis of java threaded applications (2000)
13. Seragiotto, C., Fahringer, T.: Performance Analysis for Distributed and Parallel Java Programs. In: Proceedings of IEEE International Symposium on Cluster Computing and Grid 2005, Cardiff, UK, IEEE Computer Society Press (2005)
14. Factor, M., Schuster, A., Shagin, K.: Instrumentation of standard libraries in object-oriented languages: the twin class hierarchy approach. In: OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, New York, NY, USA, ACM Press (2004) 288–300
15. Arnold, M., Ryder, B.G.: A framework for reducing the cost of instrumented code. In: PLDI. (2001) 168–179
16. Hoffman, J.J., Byrd, A., Mohror, K., Karavanic, K.L.: Pperfgrid: A grid services-based tool for the exchange of heterogeneous parallel performance data. In: IPDPS, IEEE Computer Society (2005)
17. Seragiotto, C., Truong, H.L., Fahringer, T., Mohr, B., Gerndt, M., Li, T.: Monitoring and Instrumentation Requests for Fortran, Java, C and C++ Programs. Technical Report AURORATR2004-17, Institute for Software Science, University of Vienna (2004)
18. Kereku, E., Gerndt, M.: The Monitoring Request Interface (MRI). In: 20th International Parallel and Distributed Processing Symposium (IPDPS 2006). (2006)