# Support for User-Defined Metrics in the Online Performance Analysis Tool G-PM*

Roland Wismüller[1], Marian Bubak[2,3], Włodzimierz Funika[2,3], Tomasz Arodź[2,3], and Marcin Kurdziel[2,3]

[1] LRR-TUM, Institut für Informatik, Technische Universität München, D-85747 Garching, Germany
[2] Institute of Computer Science, AGH, al. Mickiewicza 30, 30-059 Kraków, Poland
[3] Academic Computer Centre – CYFRONET, Nawojki 11, 30-950 Kraków, Poland
{bubak,funika}@uci.agh.edu.pl, wismuell@in.tum.de
*phone:* (+48 12) 617 39 64, *fax:* (+48 12) 633 80 54, *phone:* (+49 89) 289 17676

**Abstract.** This paper presents the support for user-defined metrics in the G-PM performance analysis tool. G-PM addresses the demand for aggressive optimisation of Grid applications by using a new approach to performance monitoring. The tool provides developers, integrators, and end users with the ability to analyse the performance characteristics of an application at a high level of abstraction. In particular, it allows to relate an application's performance to the Grid's performance, and also supports application-specific metrics. This is achieved by introducing a language for the specification of performance metrics (PMSL) and the concept of probes for providing application specific events and data. PMSL enables an easy specification of performance metrics, yet allowing an efficient implementation, required for the reduction of monitoring overhead.
**Keywords:** Grid, performance monitoring, user-defined metrics

## 1 Introduction

With Grid computing, a new level of complexity has been introduced to the development process of distributed applications. Grid enabled applications are presumed to be run not only in geographically distributed locations, but also on a diverse set of hardware and software platforms. At the same time, the emphasis is on the computing being as cost effective as possible. This leads to the requirements for the applications to utilise the available resources in the optimal way. Usually, performance degradations emerge as a result of an imbalance in computation distribution, flaws in the schema of application synchronisation operations, or suboptimal allocation of resources. Such conditions can be fully observed only during an application run in an environment that closely resembles the one in which the application is to be deployed. The best approach would

* This work was partly funded by the European Commission, project IST-2001-32243, CrossGrid

be to observe the application directly on the Grid. Consequently, Grid oriented performance monitoring systems become necessary tools for development of high performance Grid applications. These tools should work in on-line mode – it is a necessary prerequisite not only for the monitoring of applications with a very long execution time, but also for non-batch programs, like interactive applications or services. Moreover, the monitoring should not introduce a noticeable overhead as this might disturb the application's execution, thus rendering the measured performance properties useless.

A performance monitoring tool for Grid applications designed to fulfil the above-stated demands is being developed in the CrossGrid project [6]. The G-PM tool differs form related approaches in the respect that it combines the support for on-line performance measurement and automatic instrumentation of applications for measuring standard performance properties with the support for user-defined metrics and application-specific instrumentation.

This paper focuses on the support for user-defined metrics in the G-PM tool. In Section 2 the main concepts of the tool are briefly summarised. More detailed information can be found in [3]. The main part of the article is Section 3, which presents the Performance Metrics Specification Language PMSL and two usage examples. Furthermore, an approach to the efficient implementation of the measurements based on user-defined metrics is described. Section 5 presents concluding remarks along with information on the current state of G-PM.

## 2 The Main Concepts of the G-PM Tool

G-PM provides a wide range of measured quantities related to the performance of the analysed application and the Grid infrastructure. Since the performance of a Grid program is strongly dependent on the Grid environment, the integration of those two types of performance quantities in a single tool is essential for a meaningful performance analysis.

The analysis of the Grid infrastructure considers various performance quantities related to the hosts and sites constituting the Grid, as well as the network connecting them. However, the tool's main focus is on Grid applications. These application can be examined at multiple levels of detail, allowing to measure performance characteristics for the program as a whole, for specific functions inside the code, or for other, user specified code regions. The analysis can be narrowed down to parts of the executed application, e.g. a single process, or a set of processes running on given hosts or on particular Grid sites.

The G-PM tool uses the OCM-G monitoring service [2] for gathering of the raw performance data, both application- and Grid-related. The tool operates in an on-line manner, so the performance data are requested from the OCM-G and then analysed by G-PM during the application's execution. On top of the data returned by OCM-G, a set of standard, built-in metrics is defined. These include such quantities as communication volume and delay, or overhead of file I/O.

The metrics can then be displayed using a set of visualisation graphs, such as e.g. bar graphs and multi-curve plots. These visualisation types can present arbitrary numerical data, so any type of performance metrics which is quantitative in nature can be shown.

While the standard metrics are quite useful to the application developer, allowing him or her to inspect the details of various parts of the application, the measured quantities are often too low-level to provide a clear insight into the performance of a complex application executing in the Grid. For example:

- The developer usually wants to relate the application's performance to the performance of the Grid environment it is running on.
- When analysing interactive applications or services, the developer wants to inspect the performance of individual interactions or service requests.
- The application may use higher-level libraries. A tool should be able to measure performance data specific to these libraries.
- The developer may be interested in application specific metrics, like the number of iterations per second and/or the current residuum of an iterative numerical solver.

This shows that a means of adjusting the performance tool to the needs of its users is essential. In G-PM, this is achieved by supporting user-defined metrics, which allow to combine data from standard metrics with application/library specific events and/or data, thus enabling all the measurements suggested above. User-defined metrics typically will be at a higher, application specific level of abstraction. Once such a metrics, which is specified using a simple definition language, has been entered in the tool's user interface, it can be used in the same way as the built-in ones. In particular, it can be visualised using any of the available displays and can be used as a basis for other user-defined metrics. The essential difference to related approaches (see Sect. 4) is the fact that the user-defined metrics are evaluated *on-line* and in a *distributed, scalable way.*

## 3 User-defined Metrics

In the following, we present the technical details on how user-defined metrics are specified and how they are implemented in a distributed on-line fashion. Due to space limitation, we must refer the reader to [4, 14] for the detailed motivation, design goals, and the basic concepts behind this work.

### 3.1 Specification of Metrics

A user-defined, application specific metrics in G-PM is based on already existing metrics, plus some minimal, specific information from the user's application:

- Occurrences of important events in the application's execution,
- associations between related events,
- and (optionally) performance data computed by the application itself.

This information is provided via *probes*, i.e. special function calls, which receive a *virtual time* and the optional performance data as parameters. The virtual

```
IO_volume_for_interaction_vt(Process[] processes, File[] partners,
                             VirtualTime vt)
{
   PROBE end(Process p, VirtualTime vt);
   PROBE begin(Process p, VirtualTime vt);
   Process p;
   Value[] volume;
   volume[p] = IO_volume(p, partners, [START, NOW]) AT end(p, vt)
             - IO_volume(p, partners, [START, NOW]) AT begin(p, vt);
   return SUM(volume[p] WHERE p IN processes);
}
IO_volume_for_interaction(Process[] processes, File[] partners,
                          TimeInterval time)
{
   VirtualTime vt;
   Value[] volume;
   volume[vt] = IO_volume_for_interaction_vt(processes, partners, vt);
   return SUM(volume[vt] WHERE volume[vt].time IN time);
}
```

**Fig. 1.** PMSL specification for the example metrics

time is an arbitrary, but monotonically increasing integer value, which is used to specify associations between different events, i.e. probe executions.

Using this concept, it is possible to derive new metrics from existing ones (e.g. by relating two metrics to each other) or to measure a metrics only for specific program phases. For example, in an interactive, parallel application, the programmer may provide two probes (called `begin` and `end`), marking the beginning and end of the processing of a certain class of user interactions[1]. Probe calls belonging to the same interaction are identified by receiving the same value for the virtual time.

With this preparation, G-PM can e.g. be used to measure the amount of data transferred to/from disk for each interaction. Obviously, this amount is simply the total amount of data at the end of the interaction, minus the amount at its beginning. Rules like this are the means by which a G-PM user can specify new metrics at runtime.

Since no suitable specification mechanism existed for this purpose, we had to design a new language. It should be simple to use and should not force the user to think about the *implementation* of the metrics. The result, called PMSL therefore is a *declarative, functional language*. It only provides single assignment variables and does neither include control flow constructs nor alterable state. To support metrics based on events, PMSL includes a *special operator* (`AT`), which takes the value of an expression at the time of an event occurrence. Finally, the language provides a couple of *set operations* used for data aggregation.

Instead of formally introducing the syntax and semantics of PMSL, we will just discuss the specification for the example metrics introduced above, which

[1] For the sake of simplicity of demonstration we assume an SPMD-style program, thus, these probes will be executed in each process. See [14] for a more detailed discussion.

demonstrates most of the language's features. Fig. 1 shows the specification of our example metrics. In general, each metrics has parameters that define

1. the object(s) to be measured (e.g. `Process[] processes`: a list of processes),
2. restrictions to the measurement, like partner objects or code regions (e.g. `File[] partners`: a list of files),
3. a time specification. This can be a point in real time at which the measurement is done, a measurement interval in real time (`TimeInterval time`), or a point in virtual time (`VirtualTime vt`).

In Fig. 1 two metrics are defined: `IO_volume_for_interaction_vt` specifies a metrics for the amount of disk I/O for a point in virtual time, i.e. (in our example) for a single user interaction. Line 7 tells how to compute the contribution of one process: just subtract its total I/O volume at the `begin` event from the one at the `end` event. The term `[START,NOW]` defines the measurement time interval for the `IO_volume` metrics (which is a built-in metrics of G-PM): the lower bound is the start time of the whole measurement, the upper bound is the current time, i.e. the time of the event. Line 8 finally states that the return value is the sum of the contributions of all measured processes. `IO_volume_for_interaction` is a metrics that refers to a given interval of (real) time. Its result is the I/O volume caused by all user interactions in the specified measurement time interval. Since all measurement values in G-PM are time-stamped, this metrics can be computed by considering the results of `IO_volume_for_interaction_vt` for all virtual times and by summing up all results whose time-stamps lie in the measurement interval.

The examples show that the metrics are specified at a rather high level of abstraction, thus, users of G-PM should be able to define their own metrics according to their needs. Nevertheless, the specifications still can be converted to efficient, distributed on-line measurements, as we will show in the next section.

### 3.2 Implementation of Measurements

There are two reasons why the measurements must be implemented in a distributed fashion: First, the `AT` construct in PMSL requires a measurement value to be determined when an event occurs. In order to avoid communication delays which would adulterate the results, determining the measurement value must happen at the same place where the event occurs. Second, distributed processing of measurement data often significantly reduces the amount of data that must be sent to the G-PM tool via the network, thus reducing the perturbation of the measured system and increasing scalability.

The way from a PMSL specification to a distributed on-line measurement consists of five phases:

1. When the user defines the new metrics, it is parsed into an intermediate representation (IR).
2. Later, when the user defines a measurement of this metrics, the IR is partially evaluated, using the now known parameter values (i.e. objects to measure, measurement restrictions).

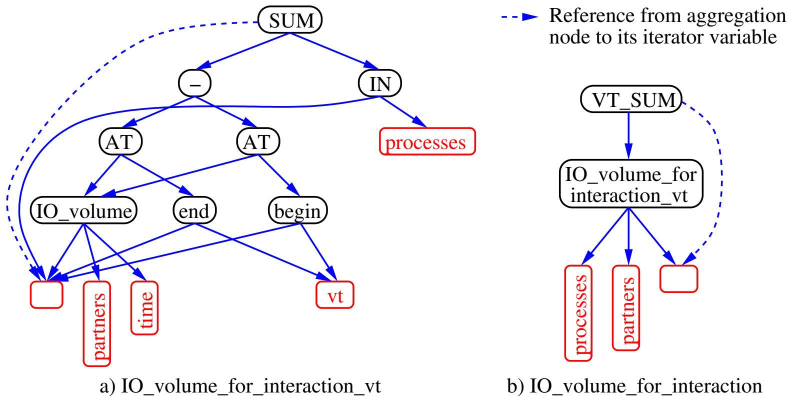



**Fig. 2.** IR of the example metrics

3. The partially evaluated IR is optimised in order to reduce the measurement's perturbation.
4. The necessary inferior measurements and the monitoring requests for all probe events are defined. At the same time, a dataflow graph is created, which controls the computation of the final measurement results.
5. The dataflow graph is distributed to the components of the OCM-G monitoring system and the measurements is started.

**Parsing PMSL into an IR.** As an intermediate representation for PMSL specifications we use simple expression-DAGs (directed acyclic graphs), where inner nodes are operations, while leaf nodes are variables or constants. Fig. 2 shows the IRs for the example metrics of Fig. 1. Note that using an expression-DAG as IR is only possible since PMSL is a functional language. Most of the translation process is relatively straightforward, with three notable issues:

– Assignments are handled by subsequently replacing all uses of the assigned variable with the DAG of the assignment's right hand side. For indexed variables, this replacement includes an index substitution. Thus, local variables like the `volume` arrays in Fig. 1 are completely eliminated.
– Variables used as iterators in aggregation functions (e.g. `p` in line 8 of Fig. 1) are substituted by anonymous, private copies. This is necessary to avoid conflicts when the same variable is used in different aggregations.
– After creating the IR, we perform a common subexpression elimination. This avoids multiple definitions of the same inferior measurements later. In our example, the two references to the `IO_volume` metrics are unified in Fig. 2a.

The main problem at this stage, however, is the handling of the time parameters. While they are explicit input parameters in the specification, this is not the case in the implementation. An on-line performance analysis tool just requests a measurement value from time to time. Thus, the actual time of a measurement is the time when the underlying monitoring system processes this request. Fortunately, the use of time is rather limited in PMSL:

- A virtual time parameter can not be modified, but just passed to inferior metrics and probes. Thus, in the implementation we can convert it from an input to an output parameter. However, we must then ensure that all "uses" of the parameter provide the same value (e.g. in line 7 of Fig. 1 both probes must provide the same value for vt). This is done by taking the virtual time into account in the firing rules of the dataflow nodes created in phase 4.
- Real time (and time interval) parameters can not be modified, too. Like virtual time, they can be passed to inferior metrics. This case is simply handled by requesting the values for these metrics at (approximately) the same time.
  But a time interval can also be used as an operand to an `IN` operator inside an aggregation function, like in line 15 of Fig. 1. This case is detected via pattern matching in the IR, which checks for constructs of the form
  *aggregation_function* `(` *expr* `WHERE` *expr*`.time IN` *time_interval* `)`
  We then introduce special aggregation nodes (e.g. `VT_SUM` in Fig. 2b) which later handle this case.

Of course, the time can also used to compute the value of the metrics. However, this is done using a special metrics `Time`, which returns the current time, instead of using the time parameters. Thus, there are no difficulties during the evaluation.

**Partial Evaluation of the IR.** Once the user defines a measurement based on the specified metrics, all the parameters of this metrics are fixed[2]. Thus, we can assign values to some of the leaf nodes in the DAG and evaluate its inner nodes. For each node, the evaluation either results in a constant or – if some of the node's operands still have unknown values – in a new DAG. The result of this phase is shown in Fig 3a, where we assumed that a measurement of `IO_volume_for_interaction` should be performed for two processes (*p1* and *p2*), restricted to consider only I/O to a file named *file1*.

The figure shows that besides the actual evaluation, a couple of other transformations is applied in this phase:

- Uses of metrics, which have been specified via PMSL, are "inlined", i.e. the node has been replaced with the partially evaluated DAG of this metrics. In our example, the node `IO_volume_for_interaction_vt` of Fig. 2b has been replaced in this way.
- Since the parameters of all used metrics must evaluate to constants in this phase, there is no need anymore to represent them as explicit child nodes. Instead, the metrics nodes for built-in metrics directly contain the full measurement specification.
- Similarly, probe nodes now directly contain the information needed to request the monitoring of this probe.

[2] With the exception of the time parameters, which are handled as described before.

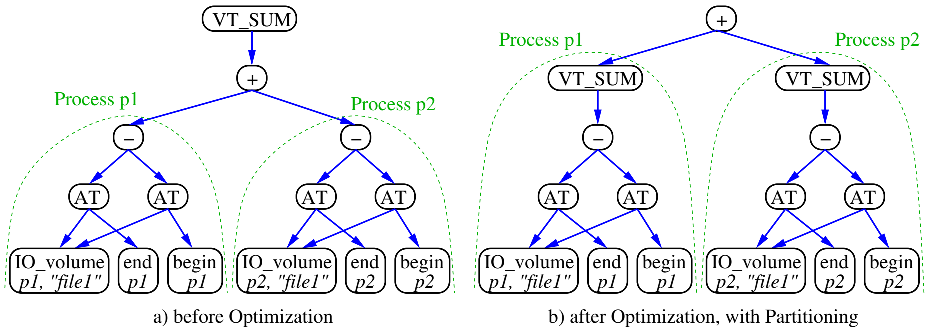



**Fig. 3.** Partially evaluated IR

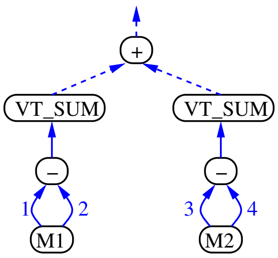
– Aggregation nodes are replaced by simple operations when the iteration set is known. E.g. the "SUM" node of Fig. 2a is replaced by a "+" node with two operands, one for each process considered.

**Optimisation of the Partially Evaluated IR (PEIR).** The PEIR can be optimised in several ways to improve the quality of the resulting measurement. The main goal is to reduce the frequency and volume of data which needs to be sent over the network. This can be achieved by moving nodes aggregating over virtual time down towards the leaves. E.g. in Fig. 3a, the labelled sub-DAGs can be evaluated locally in the context of the monitored processes. However, the "+" node combines data from different processes. This means that whenever an `end` event occurs, some communication is necessary. By interchanging the "+" and "VT_SUM" nodes, as shown in Fig. 3b, the aggregation over time can now also be done locally. Communication is only required when the aggregated values are needed because the result of the measurement has been requested.

**Definition of Measurements and Monitoring of Probes.** In this phase, a measurement is defined for each metrics node in the PEIR. Likewise, the monitoring of all probes used in the PEIR is requested. The latter results in a callback function being invoked when a probe is executed. This function usually will read some of the inferior measurements. Their results are again delivered via a callback. Since we will get these results at different times, because they are triggered by different probes, the best way to combine them into the final result is by using a dataflow approach. Thus, we also create a proper dataflow graph in this phase.

The result for our example is shown in Fig. 4. It shows both the dataflow graph and the activities performed by the different callbacks. The tokens flowing through the dataflow graph consist of a measurement value, its time stamp, and optionally a virtual time stamp and a requester identification. The latter is needed when the same measurement is read by different probe callbacks. In these cases, the requester identification is used to determine the dataflow arc to which the result is sent. E.g. in Fig. 4, node "M1" will forward its result token

Dataflow graph:




| Callback | Actions performed |
|---|---|
| $p_i$ : begin<br>(i = 1,2) | get vt from event notification<br>request a result of measurement i<br>with callback data = (vt, 2+2*(i–1)) |
| $p_i$ : end<br>(i = 1,2) | get vt from event notification<br>request a result of measurement i<br>with callback data = (vt, 1+2*(i–1)) |
| measurement $_i$<br>(i = 1,2) | get (value, time) from measurement result<br>get (virtualTime, requesterID) from callback data<br>pass (value,time,virtualTime,requesterID) to node $M_i$ |

→ data flow is triggered by execution of probes
⇢ data flow is triggered by requesting the measurement's result

measurement $_i$ : measurement of IO_volume for process $p_i$

**Fig. 4.** Implementation of measurement

to the arc labelled 2 when the result was requested by the `begin` callback, and to arc labelled 1 when requested by the `end` callback.

A big advantage of this dataflow scheme is that it immediately supports distributed processing. An arc between dataflow nodes processed on different hosts just translates to a communication link.

**Measurement.** During the measurement, tokens are created in the dataflow graph either when a probe is executed or when the measurement result is requested. Usually, a dataflow node fires, i.e. performs its operation and produces a result token, when a token is available on each of its input arcs. In our implementation, there are two extensions to this general rule:

– If an input token contains a virtual time stamp, the operation waits until it has a complete set of operands with identical virtual time stamps (e.g. the "–" nodes in Fig. 4).
– When a dataflow node aggregating over virtual time (e.g. the "VT_SUM" node) triggers, it does not produce a result token but rather updates an internal summary value. The node produces a result token only upon a special request. In our example, this request is issued when the result of the complete measurement is requested.

## 4 Related Work

There already exist several tools which offer a user-definable data evaluation. One example are visualization systems like AVS[3], which allow the user to customize the data processing by connecting modules to a dataflow pipeline. A similar approach, specifically targeted to performance analysis, has been taken by Pablo [12]. Paraver [7] offers a menu-driven specification of user-defined metrics. However, all of these approaches only support a *centralized* and *off-line*

[3] `http://www.avs.org`

data analysis, where large amounts of raw data (event traces in case of Pablo and Paraver) have to be sent to a single computer and have to be analysed there. In contrast, G-PM targets a *distributed on-line* analysis, where the raw data is processed at its origin, which greatly reduces the load of the network and the computer executing the tool's user interface. Only this kind of analysis allows to provide an immediate feedback on the behaviour of an application.

Some other performance analysis tools also use languages to define metrics. Examples are Paradyn [11] with its measurement definition language MDL [10], EXPERT [16] with the trace analysis language EARL [15], and the JavaPSL property specification language [9]. In contract to PMSL, these languages are used only *internally* in the tools; they are not suited for being exposed to the user. MDL and EARL are *imperative*, rather than *declarative* languages, i.e. a metrics is defined by exactly specifying the implementation of its measurement. JavaPSL is declarative, but aims at different goals. In fact, JavaPSL and PMSL have a common root in the APART Specification Language ASL [8]. A major contribution of PMSL is that it allows to combine *events* with existing metrics.

Finally, our approach of using expression DAGs has some similarity to dynamic instrumentation techniques as implemented by Dyninst [5] and DPCL [13]. The main difference is that the DAG created from a PMSL specification *combines information from distributed sources*, i.e. its evaluation must be spread across several processes. In contrast, in Dyninst and DPCL the DAGs are local, i.e. they are always executed by a single process. However, the concepts fit together rather well: In an advanced implementation, we could use Dyninst or DPCL to efficiently implement the local sub-DAGs marked in Fig. 3.

## 5 Conclusions

The paper presented a method to support user-defined metrics for monitoring of Grid enabled, distributed applications, which relies on the PMSL performance metrics specification language. The language allows to combine information from event occurrences, application specific data, and existing metrics in order to define new, higher-level metrics. These metrics may summarize information from different processes of the distributed application; the necessary data processing is performed on-line, using an efficient distributed dataflow scheme. Consequently, applications can be monitored with minimal additional overhead.

The PMSL language and its translation have been implemented as a first prototype, where the optimizations of the PEIR and the distribution of the dataflow graphs are not yet included. This implementation is integrated into the current version of the G-PM tool, which will be released inside the CrossGrid project in Spring 2004. The implementation of optimizations and the fully distributed evaluation based on the OCM-G monitoring system is work in progress.

## References


1. Balaton, Z., Kacsuk, P., Podhorszki, N., and Vajda, F.: From Cluster Monitoring to Grid Monitoring Based on GRM. In: *Euro-Par 2001 Parallel Processing, 7th*

*International Euro-Par Conference*, August 2001, Manchester, UK, Lecture Notes in Computer Science 2150, pp. 874-881, Springer-Verlag, 2001.

2. Baliś, B., Bubak, M., Funika, W., Szepieniec, T., and Wismüller, R.: An Infrastructure for Grid Application Monitoring. In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 9th European PVM/MPI Users' Group Meeting*, Sept. - Oct. 2002, Linz, Austria, Lecture Notes in Computer Science 2474, pp. 41-49, Springer-Verlag, 2002.
3. Baliś, B., Bubak, M., Funika, W., Szepieniec, T., and Wismüller, R.: Monitoring and Performance Analysis of Grid Application. In: *Computational Science - ICCS 2003*, June 2003, St. Petersburg, Russia, Lecture Notes in Computer Science 2657, pp. 214-224, Springer-Verlag, 2003.
4. Bubak, M., Funika, W., Wismüller, R., Arodz, T., and Kurdziel, M.: The G-PM Tool for Grid-oriented Performance Analysis. In: *1st European Across Grids Conference*, Santiago de Compostela, Spain, Feb. 2003. `http://wwwbode.in.tum.de/~wismuell/pub/santiago03a.ps.gz`
5. Buck, B. and Hollingsworth, J.K.: An API for Runtime Code Patching, *The International Journal of High Performance Computing Applications*, **14(4)**, Winter 2000, pp. 317–329.
6. *CrossGrid - Development of Grid Environment for interactive Applications*, EU Project, IST-2001-32243, Technical Annex. `http://www.eu-crossgrid.org`
7. European Center for Parallelism of Barcelona. *Paraver*. `http://www.cepba.upc.es/paraver/`
8. T. Fahringer, M. Gerndt, G. Riley, and J. L. Träff: Knowledge Specification for Automatic Performance Analysis. APART Technical Report, ESPRIT IV Working Group on Automatic Performance Analysis, Nov. 1999. `http://www.fz-juelich.de/apart-1/reports/wp2-asl.ps.gz`
9. T. Fahringer and C. Seragiotto: Modeling and Detecting Performance Problems for Distributed and Parallel Programs with JavaPSL. In *9th IEEE High-Performance Networking and Computing Conference, SC'2001*, Denver, CO, Nov. 2001.
10. J. R. Hollingsworth, B. P. Miller, M. J. R. Gonçalves, Z. Xu, O. Naim, and L. Zheng: MDL: A Language and Compiler for Dynamic Program Instrumentation. In *Proc. International Conference on Parallel Architectures and Compilation Techniques*, San Francisco, CA, USA, Nov. 1997. `ftp://grilled.cs.wisc.edu/technical_papers/mdl.ps.gz`.
11. Miller, B.P., et al.: *The Paradyn Parallel Performance Measurement Tools*. IEEE Computer, 28(11): 37-46, Nov. 1995.
12. University of Illinois. *Pablo Performance Analysis Environment: Data Analysis*. `http://www-pablo.cs.uiuc.edu/Project/Pablo/PabloDataAnalysis.htm`
13. University of Madison, Wisconsin. *Dynamic Probe Class Library (DPCL)*. `http://www.cs.wisc.edu/paradyn/DPCL/`
14. Wismüller, R., Bubak, M., Funika, W., and Baliś, B.: A Performance Analysis Tool for Interactive Applications on the Grid. *International Journal of High Performance Computer Applications*, Fall 2004, SAGE Publications. In print.
15. F. Wolf and B. Mohr: EARL - A Programmable and Extensible Toolkit for Analyzing Event Traces of Message Passing Programs. In *Proc. of the 7th International Conference on High- Performance Computing and Networking (HPCN 99)*, Lecture Notes in Computer Science, pp. 503–512, Amsterdam, 1999. Springer-Verlag.
16. F. Wolf and B. Mohr: Automatic Performance Analysis of MPI Applications Based on Event Traces. In *Euro-Par 2000 Parallel Processing, 6th International Euro-Par Conference*, Lecture Notes in Computer Science 1900, pp. 123–132, Munich, Germany, Aug. 2000. Springer-Verlag.