Specification of Performance Problems in MPI Programs with ASL *

Thomas Fahringer
Institute for Software Technology and Parallel Systems
University of Vienna
tf@par.univie.ac.at

Michael Gerndt
Central Institute for Applied Mathematics
Research Centre Juelich
m.gerndt@fz-juelich.de

Graham Riley
Department of Computer Science
University of Manchester
griley@cs.man.ac.uk

Jesper Larsson Träff
C&C Research Laboratories
NEC Europe Ltd.
traff@ccrl-nece.technopark.gmd.de

Abstract

Performance analysis is an important step in tuning performance critical applications. It is a cyclic process of measuring and analyzing performance data which is driven by the programmers hypotheses on potential performance problems. Currently this process is controlled manually by the programmer. The implicit knowledge applied in this cyclic process must be formalized in order to be reused in the automation of performance analysis tools. This article describes the performance property specification language developed in the APART Esprit IV working group which allows the specification of performance data via an object model and performance properties via a especially designed notation. Performance bottlenecks can then be identified based on the specification since bottlenecks are viewed as performance properties with a huge negative impact. We present the language in the context of MPI applications.

1 Introduction

Performance-oriented program development can be a daunting task. In order to achieve high or at least respectable performance on today's multiprocessor systems, careful attention to a plethora of system and programming paradigm details is required. Commonly programmers go through many cycles of experimentation involving gathering performance data, performance data analysis (a-priori and postmortem), detection of performance problems, and code refinements in slow progression. Clearly, the programmer must be intimately familiar with many aspects related to this experimentation process. Although there exists a large number of tools assisting the programmer in performance experimentation, it is still the programmer's responsibility to take most strategic decisions. A particular distressing fact is that many performance tools are platform and language dependent, cannot correlate performance data gathered at a lower level with higher-level programming paradigms, focus only on specific program and machine behavior, and do not provide sufficient support to infer important performance properties.

In this article we describe a novel approach to formalize performance bottlenecks and the data required in detecting those bottlenecks with the aim to support automatic performance analysis for a large variety of programming paradigms and architectures. This research is done as part of APART Esprit IV *Working Group on Automatic Performance Analysis: Resources and Tools* (APART)[www.fz-juelich.de/apart].

We will introduce the APART Specification Language (ASL) which allows to describe performance-related data by incorporating an object-oriented specification model and to define performance properties by using a novel formal notation. Our object-oriented specification model is used to declare – without the need to compute – performance information. It is similar to Java but uses only single inheritance and does not require methods. A novel syntax has been introduced to specify performance properties.

The organization of this article is as follows. Section 2 outlines the overall design of an automatic performance analysis environment that incorporates specification of per-

* The ESPRIT IV *Working Group on Automatic Performance Analysis: Resources and Tools* is funded under Contract No. 29488

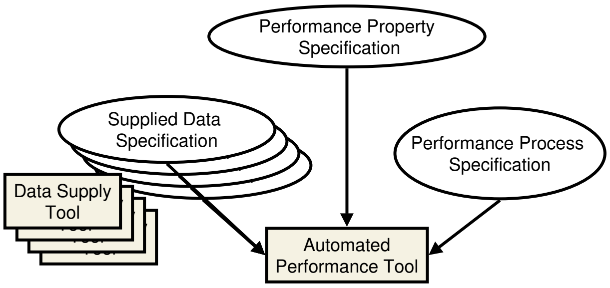

Figure 1. Design of an integrated automatic performance analysis environment.

formance properties, and related work is presented in Section 3. Section 4 presents the paradigm-independent base classes and Section 5 shows an example specification of performance-related data for MPI programs. The syntax for the specification of performance properties is described in Section 6 and examples are presented in Section 7. Conclusions and Future work are discussed in Section 8.

2 Overall Design

Performance property specification as described in this article can be considered as part of a possible design for an automatic performance analysis environment. This environment comprises three components (see Figure 1):

Performance Property Specification defines information about performance properties for the current programming paradigm and machine. It is based on the following terminology:

Performance-related Data: Performance-related data defines information that can be used to describe performance properties of a program. There are two classes of performance related data. First, static data specifies information that can be determined without executing a program on a target machine. Second, dynamic performance-related data describes the dynamic behavior of a program during execution on a target machine.

Performance Property: A performance property (e.g. load imbalance, communication, cache misses, redundant computations, etc.) characterizes a specific performance behavior of a program and can be checked by a set of *conditions*. Conditions are associated with a *confidence value* (between 0 and 1) indicating the degree of confidence about the existence of a performance property. In addition, for every performance property a *severity figure* is provided that specifies the importance of the property.

Performance Problem: A performance property is a performance problem, iff its severity is greater than a user- or tool-defined threshold.

Performance Bottleneck: A program has a unique performance bottleneck which is its most severe performance property. If this bottleneck is not a performance problem, then the program's performance is acceptable and does not need any further tuning.

Performance Process Specification reflects the knowledge applied in tuning the performance of programs which covers, for instance, how many hypotheses about performance problems are evaluated. This evaluation can be based on stepwise refinement, i.e. the process specification determines which hypotheses are evaluated first before more precise hypotheses are examined. It may be useful to prove that message passing is significant in a subroutine before examining individual MPI calls. More detailed analysis may require considerably more performance-related data.

Supplied Data Specification of a tool describes which performance-related data can be obtained from that tool. Moreover, query commands to access this data should be indicated. Based on the supplied data specification, an automated performance analysis environment can use existing tools to access data relevant in the search for performance problems and bottlenecks.

An integrated system combining all three specifications should substantially alleviate re-targeting of performance tools to a variety of architectures and programming paradigms, facilitate the development of new performance tools, as well as enhance existing tools by providing access to a wealth of performance information and analysis capabilities.

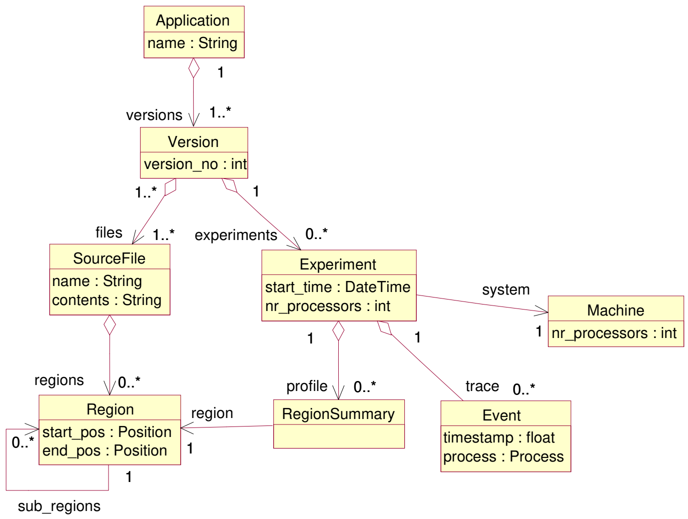

Figure 2. Base classes of performance-related data models.

3 Related work

The use of specification languages in the context of automatic performance analysis tools is a new approach. Paradyn [4] performs an automatic online analysis and is based on dynamic monitoring. While the underlying metrics can be defined via the *Metric Description Language* (MDL) [5], the set of searched bottlenecks is fixed. It includes *CPUbound*, *ExcessiveSyncWaitingTime*, *ExcessiveIOBlockingTime*, and *TooManySmallIOOps*.

A rule-based specification of performance bottlenecks and of the analysis process was developed for the performance analysis tool OPAL [3] in the SVM-Fortran project. The rule base consists of a set of parameterized hypothesis with proof rules and refinement rules. The proof rules determine whether a hypothesis is valid based on the measured performance data. The refinement rules specify which new hypotheses are generated from a proven hypothesis [2].

Another approach is to define a performance bottleneck as an event pattern in program traces. EDL [1] allows the definition of compound events based on extended regular expressions. EARL [7] describes event patterns in a more procedural fashion as scripts in a high-level event trace analysis language which is implemented as an extension of common scripting languages like Tcl, Perl or Python.

4 Performance-related Data Specification

Performance-related data are described by a set of classes following an object-oriented style with single-inheritance. Among others, class members can be of type FLOAT (e.g., for timing measurements), BOOLEAN (e.g., for flags), INT (e.g., for counting events), STRING (e.g., for naming applications or files), DATETIME (time at which some event occurs), *SETOF*, *ENUM*, and reference.

In this article we present a library of classes that represent static and dynamic performance-related information for MPI programs. We distinguish two sets of classes. First, the set of base classes which is independent of any programming paradigm, and second, programming paradigm dependent classes.

Note that we expect most data models described with ASL will have a similar overall structure. This similarity was captured in the base classes. Future data models can build specialized classes in form of subclasses.

Figure 2 shows the UML representation of the base classes. The translation of the UML diagrams into the specified syntax is straightforward. Initially, there is an application for which performance analysis has to be done. Every application has a name and may possibly have a number of implementations, each with a unique version number. Versions may differ with respect to their source files and experiments. Every source file (the contents of which is stored in a generic string) has one or several static code regions each of which is uniquely specified by *start_pos* (position where region begins in the source file) and *end_pos* (position where region ends in the source file). A position in a region is defined by a line and column number with respect to the given source file.

Experiments – denoting the second attribute of a version – are described by the time (*start_time*) when the experiment started and the number of processors (*nr_processors*) that were available to execute the version. Furthermore, an

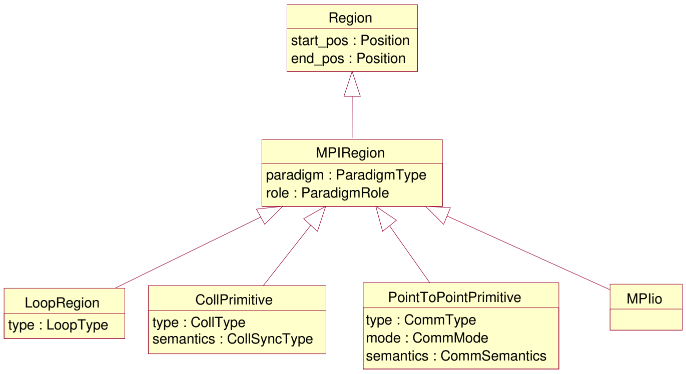

Figure 3. Regions in the MPI data model.

experiment is also associated with a static description of the machine (e.g. number of processors available) that is used for the experiment. Every experiment includes also dynamic data, i.e. a set of region summaries (*profile*) and a set of events (*trace*). The class *RegionSummary* describes performance information across all processes employed for the experiment. Region summaries are associated with the appropriate region. The class *Event* represents information about individual events occuring at runtime, such as sending a message to another process. Each event has a *time_stamp* attribute determining when the event occured and a *process* attribute determining in which process the event occured.

5 MPI Class Hierarchy

In this section we describe static and dynamic information for MPI. Figure 3 outlines the classes for static MPI regions. Class *MPIRegion* is a subclass of Region (see Figure 2) and contains two attributes: *paradigm* and *role* which, respectively, relate to the paradigm implemented (e.g. master-slave, divide-and-conquer, and bulk-synchronous) and to the role (e.g. master/slave send/receive operation) of a given region in a paradigm. *MPIRegion* is further refined to:

- *LoopRegion* specifies different loop types such as DO, WHILE, or FORALL loop.
- *CollPrimitive* refers to various collective operations. This class comprises an attribute *type* for the type of collective operation (e.g. reduction or broadcast), and an attribute *sync* for a specific synchronization mode (e.g. barrier or nobarrier).
- *PointToPointPrimitive* provides more specific information about the point-to-point communication. An attribute *type* determines whether the underlying communication is based on a send or receive operation. The communication mode (e.g. buffered, synchronous, ready) is denoted by attribute *mode*. Blocking or nonblocking communication can be defined by attribute *semantics*.
- *MPIio* provide information about MPI Input/Output routines.

The precise semantics of the above mentioned MPI communication modes and types can be found in [6].

Figure 4 describes summary information which reflects the dynamic behavior of MPI programs. Class *MPIRegionSummary* in Figure 4 extends *RegionSummary* (see Figure 2) and reflects dynamic performance information across all processes that execute a region. *MPIRegionSummary* has two attributes: *sums* and *process_sums* which, respectively, describe summary information for a specific region across all processes and individually for the processes. The attributes of class *MPISummary* are given as sums across all processes with respect to all instances of a specific region:

- *comm_time*: communication time
- *sync_time*: barrier synchronization time
- *io_time*: input/output time
- *idle_time*: idle time
- *message_length*: sum of the length of all messages sent
- *duration*: execution time
- *nr_executions*: number of executions of a given region

6 Performance Property Specification

The property specification (Figure 5) defines the name of the property, its context via a list of parameters, and the

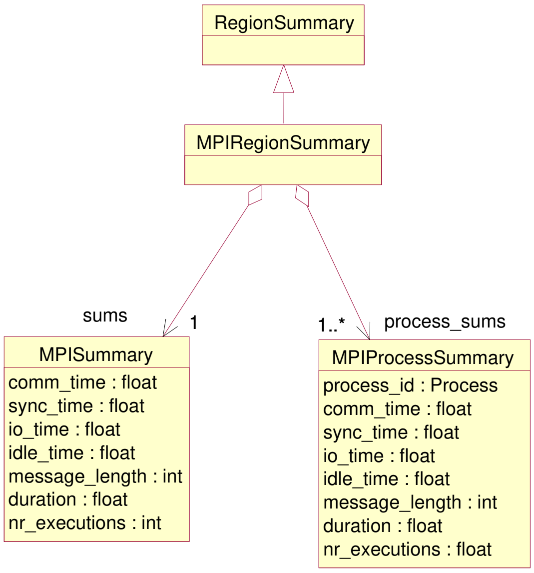

Figure 4. Summaries in the MPI data model

condition, confidence, and severity expressions. The property specification is based on a set of parameters. These parameters specify the property's context and parameterize the expressions. The context specifies the environment in which the property is evaluated, e.g. the program region and the test run.

The condition specification consists of a list of conditions. A condition is a predicate that can be prefixed by a condition identifier. The identifiers have to be unique in respect to the property since the confidence and severity specifications can refer to the conditions via those condition identifiers.

The confidence specification is an expression that computes the maximum of a list of confidence values. Each confidence value is computed via an arithmetic expression resulting in a value in the interval of zero and one. The value can be guarded by a condition identifier introduced in the condition specification. The condition identifier represents the value of the condition.

The severity specification has the same structure as the confidence specification. It computes the maximum of the individual severity values of the conditions. The severity specification will typically be based on a parameter specifying the ranking basis. If, for example, a representative test run of the application has been monitored, the time spent in message passing should be compared to the total execution time. If, instead, a short test run is the basis for performance evaluation since the application has a cyclic behaviour, the message passing overhead should be compared to the execution time of the shortened loop.

7 MPI Performance Properties

This section demonstrates the ASL constructs for specifying performance properties in the context of the message passing paradigm. Although most of the properties are independent of the specific message passing library, the terminology used is based on MPI.

```
MPIRegionSummary summary(MPIRegion r,
  Experiment e)= UNIQUE({sumr IN e.profile
                    WITH sumr.region==r});
```

```
float duration(MPIRegion r,
  Experiment e)=summary(r,e).sums.duration;
```

In most property specifications it is necessary to access the summary data of a given region for a given experiment. Therefore, we defined the summary function that returns the appropriate *MPIRegionSummary* object. It is based on the set operation *UNIQUE* that selects arbitrarily one element from the set argument which has cardinality one due to the design of the data model.

The second function determines the execution time of the region in the given experiment. The return value is the sum of the individual execution times of all MPI processes.

property	**is**	`PROPERTY` *pp-name* '(' *arg-list* ')' '{' [`LET` *def* ∗ `IN`] *pp-condition* *pp-confidence* *pp-severity* '};'
arg	**is**	*type ident*
pp-condition	**is**	`CONDITION` ':' *conditions* ';'
conditions	**is**	*condition*
	or	*condition* `OR` *conditions*
condition	**is**	['(' *cond-id* ')']*bool-expr*
pp-confidence	**is**	`CONFIDENCE` ':' `MAX` '(' *confidence-list* ')' ';'
	or	`CONFIDENCE` ':' *confidence* ';'
confidence	**is**	['(' *cond-id* ')' '->'] *arith-expr*
pp-severity	**is**	`SEVERITY` ':' `MAX` '(' *severity-list* ')' ';'
	or	`SEVERITY` ':' *severity* ';'
severity	**is**	['(' *cond-id* ')' '->'] *arith-expr*

Figure 5. ASL property specification syntax.

```
property costs(MPIRegion r,
 Experiment e, Region rank_basis){
LET
  float CostSum =
     summary(r,e).sums.comm_time +
     summary(r,e).sums.sync_time +
     summary(r,e).sums.io_time;
IN
  CONDITION:   CostSum>0;
  CONFIDENCE:  1;
  SEVERITY:    CostSum /
               duration(rank_basis,e);
}
```

The most general performance property characterizes the region as having some performance overheads or costs. The costs of a region can be subdivided into time for communication, time for synchronization, i.e. barrier synchronization, and time for I/O. The region has this property if *CostSum* is greater than 0. Clearly the confidence in that condition is one.

The severity of this property is the fraction of the time spent for costs compared to the duration of ranking basis, typically the duration of the main program. Note, that *comm_time*, *sync_time*, *io_time*, and *duration* are sums of the time spent in each process.

The severity of this property is larger than the severity of the individual properties for each of the categories. This may lead to the selection of the cost property as a performance problem according to the predefined severity threshold while the individual properties, i.e. *communication_costs*, *synchronization_costs*, and *io_costs* may not be marked as performance problems.

```
property communication_costs (
     MPIRegion r,
     Experiment e,
     Region rank_basis){
LET
  float cost =
    summary(r,e).sums.comm_time;
IN
  CONDITION:   cost>0;
  CONFIDENCE:  1;
  SEVERITY:    cost /
               duration(rank_basis,e);
}
```

This property determines whether a region includes communication. Its condition and severity is based on the appropriate global sums in the performance-related data model. The severity is the fraction of the communication costs in relation to the execution time of *rank_basis*.

A communication statement has the property *frequent_communication* if small messages are communicated. The condition compares the execution time per execution with the maximum communication time for small messages.

Whether messages are called big depends on the opinion of the tool designer or the application programmer. Therefore, this threshold should be a parameter of the performance tool. The severity specification is equal to the severity specification of the previous communication properties.

```
property frequent_communication (
     MPIRegion r,
     Experiment e,
     Region rank_basis){
LET
  float cost = summary(r,e).sums.comm_time;
IN
  CONDITION:
    (typeof(r)==PointToPointPrimitive
     OR
    (typeof(r)==CollPrimitive AND
         x.region.type != Barrier))
    AND
      cost>0 AND cost /
        summary(r,e).sums.nr_executions
        <small_messages_threshold;
  CONFIDENCE:  1;
  SEVERITY:    cost /
               duration(rank_basis,e);
}
```

```
property load_imbalance_at_barrier(
     MPIRegion r, Experiment e,
     Region rank_basis){
LET
  float max_time=max( x.duration
      WHERE x IN
      summary(r,e).process_sums );
  float min_time=min( x.duration
      WHERE x IN
      summary(r,e).process_sums );
  float max_wait=max_time - min_time;
IN
  CONDITION:
  (C1) typeof(r)==CollPrimitive AND
       r.type==Barrier AND max_wait>0
  ||
  (C2) typeof(r)==CollPrimitive AND
       r.type==Barrier AND
       summary(r,e).sums.idle_time>0;
  CONFIDENCE: 1;
  SEVERITY:
   MAX((C1)->max_wait/(duration(
     rank_basis,e)/e.nr_processors),
     (C2)->summary(r,e).sums.idle_time/
     duration(rank_basis,e));
}
```

The *load_imbalance_at_barrier* property has two conditions. The first condition can be evaluated if the idle times cannot be measured, while the second condition is based on the idle times. While the confidence value is equal for both conditions, the severity is specified by different formulas. If the first condition is satisfied, the severity is determined by dividing *max_wait* time by the mean duration of each process. If the second condition is fulfilled, the sum of the idle times in all processes is compared to the sum of the individual execution times.

```
proprety overloaded_master(
    MPIRegion r,
    Experiment e,
    Region rank_basis){
LET
  float idle_time =
     summary(r,e).sums.idle_time /
     (e.nr_processors-1);
IN
  CONDIION:
    (r.role == ReceiveSlave
     OR r.role == SendSlave)
    AND idle_time>0;
  CONFIDENCE: 1;
  SEVERITY:
    idle_time/duration(rank_basis,e);
}
```

The *overloaded_master* property is related to the master-slave paradigm. In this paradigm, four communication statements are special statements. In the master, a send operation distributes the task to the slaves and a receive operation collects the results. Those statements play the *SendMaster* and *ReceiveMaster* role. In the slaves, a receive operation (*ReceiveSlave* role) accepts tasks and a send operation (*SendSlave* role) returns the results.

The *overloaded_master* property can be proven for the *ReceiveSlave* and the *SendSlave* operations. If the slaves have to wait for new tasks or for the delivery of the results of finished tasks, the master is too slow.

8 Conclusions and Future Work

This article describes the specification language that will be used in the APART working group to specify performance problems of parallel programs. It provides constructs to specify performance-related data as an object model and constructs to describe performance properties including conditions to prove their existence, confidence expressions to support fuzzy information, and a severity specification that allows to rank performance problems.

Three extensions to the current language design will be investigated in the future:

1. The current language has no support to find patterns in traces. Some performance problems cannot be proven based on summary information only. A good example is the message order problem from the grindstone suite [www.cs.umd.edu/h̃ollings]. The messages are sent in the reverse order than they are expected to arrive at the receiver. To check this problem, this specific pattern has to be found in the event trace. Either such a pattern can explicity be described in the language, similar to EDL or EARL introduced in Section 3, or the pattern

can be identified by an external tool and is checked in the specification via a specific external predicate.

2. The language might need to be extended by some sort of templates, facilitating the specification of similar performance properties. In the example specification in this article some of the properties result directly from the summary information, e.g. *communication_costs* is directly related to the measured time spent in IO operations. The specifications of those properties are indeed very similar and need not be specified individually.

3. Some sort of meta properties might be useful. For example, synchronization can be proven based on summary information, i.e. synchronization exists if the sum of the synchronization time in a region over all processes is greater than 0. A more specific property is to check, whether individual instances of the region or classes of instances are responsible for the synchronization due to some dynamic changes in the load distribution. Similar, more specific properties can be deduced for other properties as well. Therefore, some sort of meta property would be useful which evaluates another property in the context of instances instead of the entire program run.

Since the data models for the three paradigms do have a common structure and this common structure will very likely show up in real performance analysis environments, it is covered by a set of base classes that can be reused in new designs. The list of base classes will be extended in the future to cover also other common aspects, such as classes representing typical regions and classes for a standard set of trace events.

References

[1] P. Bates, J.C. Wileden: *High-Level Debugging of Distributed Systems: The Behavioral Abstraction Approach*, The Journal of Systems and Software, Vol. 3, pp. 255-264, 1983

[2] M. Gerndt, A. Krumme: *A Rule-based Approach for Automatic Bottleneck Detection in Programs on Shared Virtual Memory Systems*, Second Workshop on High-Level Programming Models and Supportive Environments (HIPS '97), in combination with IPPS '97, IEEE, 1997

[3] M. Gerndt, A. Krumme, S. Özmen: *Performance Analysis for SVM-Fortran with OPAL*, Proceedings Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'95), Athens, Georgia, pp. 561-570, 1995

[4] B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam, T. Newhall: *The Paradyn Parallel Performance Measurement Tool*, IEEE Computer, Vol. 28, No. 11, pp. 37-46, 1995

[5] Paradyn Project: *Paradyn Parallel Performance Tools: User's Guide*, Paradyn Project, University of Wisconsin Madison, Computer Sciences Department, 1998

[6] M. Snir, St. Otto, St. Huss-Lederman, D. Walker, J. Dongarra: *MPI - The Complete Reference*, MIT Press, ISBN 0-262-69216-3, 1998

[7] F. Wolf, B. Mohr: *EARL - A Programmable and Extensible Toolkit for Analyzing Event Traces of Message Passing Programs*, 7th International Conference on High-Performance Computing and Networking (HPCN'99), A. Hoekstra, B. Hertzberger (Eds.), Lecture Notes in Computer Science, Vol. 1593, pp. 503-512, 1999