Specification Techniques for Automatic Performance Analysis Tools

Michael Gerndt, Hans-Georg Eßer

Central Institute for Applied Mathematics
Research Centre Juelich
{m.gerndt, h.g.esser}@fz-juelich.de

Abstract. Performance analysis of parallel programs is a time-consuming task and requires a lot of experience. It is the goal of the KOJAK project at the Research Centre Juelich to develop an automatic performance analysis environment. A key requirement for the success of this new environment is its easy integration with already existing tools on the target platform. The design should lead to tools that can be easily retargeted to different parallel machines based on specification documents. This article outlines the features of the APART Specification Language designed for that purpose and demonstrates its applicability in the context of the KOJAK Cost Analyzer, a first prototype tool of KOJAK.

1 Introduction

Current performance analysis tools for parallel programs assist the application programmer in measuring and interpreting performance data. But, the application of these tools to real programs is a time-consuming task which requires a lot of experience, and frequently, the revealed performance bottlenecks belong to a small number of well-defined performance problems, such as load balancing and excessive message passing overhead. It is the goal of the KOJAK project (*Kit for Objective Judgement and Automatic Knowledge-based detection of bottlenecks*) at the Research Centre Juelich to develop an environment that automatically reveals well-defined typical bottlenecks [www.fz-juelich.de/zam/kojak].

We designed KOJAK [6] such that it is not implemented for a single target environment only, e.g. the Cray T3E currently installed at our center, but can easily be ported to other target platforms as well. KOJAK will use specification documents to interface to existing performance data supply tools and to specify potential performance problems of the target programming paradigm.

In parallel with the development of KOJAK automatic performance analysis techniques are investigated in the ESPRIT IV *Working Group on Automatic Performance Analysis: Resources and Tools* (APART) [www.fz-juelich.de/apart]. This article demonstrates the main features of the *APART Specification Language* (ASL) [3] within the context of the *KOJAK Cost Analyzer* (COSY) (Section 3). The performance data analyzed in COSY are specified as an ASL object model (Section 4.1) and represented at runtime via a relational database scheme.

The performance problems COSY is aiming at are specified as ASL performance properties (Section 4.2) based on the performance data model and are implemented via SQL queries (Section 5).

2 Related work

The use of specification languages in the context of automatic performance analysis tools is a new approach. Paradyn [8] performs an automatic online analysis and is based on dynamic monitoring. While the underlying metrics can be defined via the *Metric Description Language* (MDL) [9], 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 [5] 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 [4].

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 [10] 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.

3 Overall Design of the KOJAK Cost Analyzer

COSY [7] analyzes the performance of parallel programs based on performance data of multiple test runs. It identifies program regions, i.e. subprograms, loops, if-blocks, subroutine calls, and arbitrary basic blocks, with high parallelization overhead based on the region's speedup. It explains the parallelization overhead by identifying performance problems and ranking those problems according to their severity.

COSY is integrated into the CRAY T3E performance analysis environment. The performance data measured by Apprentice [2] are transferred into a relational database. The implementation of the interface between COSY and Apprentice via the database facilitates the integration with other performance data supply tools on CRAY T3E as well as the integration with other environments.

The database includes static program information, such as the region structure and the program source code, as well as dynamic information, such as execution time, number of floating point, integer and load/store operations, and instrumentation overhead. For each subroutine call the execution time as well as the pass count with the mean value and standard deviation, as well as the minimum and maximum values are stored.

After program execution Apprentice is started. Apprentice then computes summary data for program regions taking into account compiler optimizations. The resulting information is written to a file and transferred into the database. The database includes multiple applications with different versions and multiple test runs per program version. The data model is outlined in Section 4.1.

The user interface of COSY allows to select a program version and a specific test run. The tool analyzes the dynamic data and evaluates a set of performance properties (Section 4.2). The main property is the total cost of the test run, i.e. the cycles lost in comparison to optimal speedup, other properties explain these costs in more detail. The basis for this computation is the test run with the smallest number of processors. The performance properties are ranked according to their severity and presented to the application programmer.

4 Performance Property Specification

COSY is based on specifications of the performance data and performance properties. The specifications are presented in ASL in the next two subsections. ASL supports the following concepts:

Performance property: A performance property characterizes one aspect of the performance behavior of an application. A property has one or more conditions that can be applied to identify this property. It has a confidence expression that returns a value between zero and one depending on the strength of the indicating condition. Finally it has a severity expression that returns a numerical value. If the severity of a property is greater than zero, this property has some negative effect on the program's performance.

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

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

The entire specification consists of two sections. The first section models performance data while the second section specifies performance properties based on the data model.

4.1 Data Model

The performance data can be easily modeled via an object-oriented approach. ASL provides constructs to specify classes similar to Java with single-inheritance only. Classes in the data model have attributes but no methods, since the specification will not be executed. The ASL syntax is not formally introduced in this article due to space limitations, instead, we present the performance data model used in COSY.

```
class Program {
  String Name;
  setof ProgVersion Versions;
}
```

```
class ProgVersion {
  DateTime Compilation;
  setof Function Functions;
  setof TestRun Runs;
  SourceCode Code;
}
```

The *Program* class represents a single application which is identified by its name. COSY can store multiple programs in its database. An object of that class contains a set of *ProgVersion* objects, each with the compilation timestamp, the source code, the set of functions (static information) and the executed test runs (dynamic information).

```
class TestRun {
    DateTime Start;
    int NoPe;
    int Clockspeed;
}
```

```
class Function {
    String Name;
    setof FunctionCall Calls;
    setof Region Regions;
}
```

A *TestRun* object determines the start time and the processor configuration. A *Function* object specifies the function name, the call sites, and the program regions in this function. All this information is static.

```
class Region {
    Region ParentRegion;
    setof TotalTiming TotTimes;
    setof TypedTiming TypTimes;
}
```

```
class TotalTiming {
    TestRun Run;
    float Excl;
    float Incl;
    float Ovhd;
}
```

The *Region* class models a program region with its parent region and its performance data gathered during execution. Performance data are modeled by two classes, according to the internal structure of Apprentice. The *TotalTiming* class contains the summed up exclusive and inclusive computing time as well as the overhead time. As there may be several test runs, there are also possibly several *TotalTiming* objects for a region.

The *TypedTiming* class determines the execution time for special types of overhead such as I/O, message passing and barrier synchronization – Apprentice knows 25 such types. As with the *TotalTiming* objects, there is a set of *TypedTiming* objects for every test run, but for each region there is at most one object per timing type and per test run.

```
class TypedTiming {
    TestRun Run;
    TimingType Type;
    float Time;
}
```

```
class FunctionCall {
    Function Caller;
    Region CallingReg;
    setof CallTiming Sums;
}
```

TypedTiming objects have three attributes: The *TestRun* attribute *Run* codes the specific test run of the program, *Type* (an enumeration type) is the work type

that is being considered in this object and *Time* is the time spent doing work of this type.

Call sites of functions are modeled by the *FunctionCall* class. A function call has a set of *CallTiming* objects which store the differences of the individual processes. A *CallTiming* object is composed of the *TestRun* it belongs to, the minimum, maximum, mean value, and standard deviation over a) the number of calls and b) the time spent in the function. For the four extremal values the processor that was first or last in the respective category is memorized.

Due to the design of Apprentice, the data model does not make use of inheritence. More complex data models can be found in [3].

4.2 Performance Properties

```
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
```

Fig. 1. ASL property specification syntax.

The property specification (Figure 1) defines the name of the property, its context via a list of parameters, and the 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 following example properties are checked by COSY. They demonstrate the ASL language features. Most of the property specifications make use of the following two functions:

```
TotalTiming Summary(Region r, TestRun t) = UNIQUE({s IN r.TotTimes
                                                   WITH s.Run==t});
float Duration(Region r, TestRun t) = Summary(r,t).Incl;
```

The first function *Summary* takes a *Region* r and a *TestRun* object and returns the unique *TotalTiming* object which is a member of $r.TotTimes$ belonging to that test run. The second function *Duration* uses *Summary* to extract the total execution time of the specified region in the specified test run. Note that all timings in the database are summed up values of all processes.

The first property *SublinearSpeedup* determines the lost cycles in relation to the test run with the minimal number of processors.

```
Property SublinearSpeedup(Region r, TestRun t, Region Basis) {
LET TotTimes MinPeSum = UNIQUE({sum IN r.TotTimes WITH sum.Run.NoPe ==
                                MIN(s.Run.NoPe WHERE s IN r.TotTimes)});
    float TotalCost = Duration(r,t) - Duration(r,MinPeSum.Run)
IN
    CONDITION:  TotalCost>0;    CONFIDENCE: 1;
    SEVERITY:   TotalCost/Duration(Basis,t);
}
```

The property is based on the total costs, i.e. the lost cycles compared to a reference run with the smallest number of processors. If *TotalCost* is greater than zero, the region has the *SublinearSpeedup* property. The confidence value, which is one in all examples here, might be lower than one if the condition is only an indication for that property. The severity of the *SublinearSpeedup* property is determined as the fraction of the total costs compared to the duration of *Basis* in that test run.

```
Property MeasuredCost (Region r, TestRun t, Region Basis) {
   LET float Cost = Summary(r,t).Ovhd;
   IN  CONDITION:  Cost > 0;    CONFIDENCE: 1;
       SEVERITY:   Cost / Duration(Basis,t);
}
```

The total costs can be split up into measured and unmeasured costs. The *MeasuredCost* property determines that more detailed information might be

available (*Summary(r,t).Ovhd* is the overhead measured by Apprentice). If the severity of its counterpart, the *UnmeasuredCost*, is much higher, the reason cannot be found with the available data.

```
Property SyncCost(Region r, TestRun t, Region Basis) {
   LET float Barrier = SUM(tt.Time WHERE tt IN r.TypTimes AND tt.Run==t
                                      AND tt.Type == Barrier);
   IN CONDITION:  Barrier > 0;   CONFIDENCE: 1;
      SEVERITY:   Barrier / Duration(Basis,t);
}
```

The *SyncCost* property determines that barrier synchronization is a reason for overhead in that region. Its severity depends on the time spent for barrier synchronization in relation to the execution time of the ranking basis.

```
Property LoadImbalance(FunctionCall Call, TestRun t, Region Basis) {
  LET CallTiming ct = UNIQUE ({c IN Call.Sums WITH c.Run == t});
      float Dev  = ct.StdevTime;
      float Mean = ct.MeanTime;
  IN  CONDITION:  Dev > ImbalanceThreshold * Mean;  CONFIDENCE: 1;
      SEVERITY:   Mean / Duration(Basis,t);
}
```

The *LoadImbalance* property is a refinement of the *SyncCost* property. It is evaluated only for calls to the barrier routine. If the deviation is significant, the barrier costs result from load imbalance.

5 Implementation

The design and implementation of COSY ensures portability and extensibility. The design requires that the performance data supply tools are extended such that the information can be inserted into the database. This extension was implemented for Apprentice with the help of Cray Research. The database interface is based on standard SQL and therefore, any relational database can be utilized. We ran experiments with four different databases: Oracle 7, MS Access, MS SQL server, and Postgres. For all those databases, except MS Access, the setup was in a distributed fashion. The data were transferred over the network to the database server. While Oracle was a factor of 2 slower than MS SQL server and Postgres, MS Access outperformed all those systems. Insertion of performance information was a factor of 20 faster than with the Oracle server.

COSY is implemented in Java and is thus portable to any Java environment. It uses the standard JDBC interface to access the database. Although accessing the database via JDBC is a factor of two to four slower than C-based implementations, fetching a record from the Oracle server takes about 1 ms, the portability of the implementation outweighs the performance drawbacks. The overall performance depends very much on the work distribution between the client and the database. It is a significant advantage to translate the conditions of performance properties entirely into SQL queries instead of first accessing the data components and evaluating the expressions in the analysis tool.

6 Conclusion and Future Work

This article presented a novel design for performance analysis tools. As an example, COSY, a prototype component of the KOJAK environment, was presented. The design enables excellent portability and integration into existing performance environments. The performance data and the performance properties are described in ASL and can therefore easily be adapted to other environments. For this prototype, the specification is manually translated into a relational database scheme and the evaluation of the conditions and the severity expressions of the performance properties is transformed into appropriate SQL queries and ranking code by the tool developer. In the future, we will investigate techiques for the automatic generation of the database design from the performance property specification and the automatic translation of the property description into executable code.

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. CRAY Research: *Introducing the MPP Apprentice Tool*, Cray Manual IN-2511, 1994, 1994
3. Th. Fahringer, M. Gerndt, G. Riley, J.L. Träff: *Knowledge Specification for Automatic Performance Analysis*, to appear: APART Technical Report, Forschungszentrum Jülich, FZJ-ZAM-IB-9918, 1999
4. 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
5. 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
6. M. Gerndt, B. Mohr, F. Wolf, M. Pantano: *Performance Analysis on CRAY T3E*, Euromicro Workshop on Parallel and Distributed Processing (PDP '99), IEEE Computer Society, pp. 241-248, 1999
7. A. Lucas: *Basiswerkzeuge zur automatischen Auswertung von Apprentice-Leistungsdaten*, Diploma Thesis, RWTH Aachen, Internal Report Forschungszentrums Jülich Jül-3652, 1999
8. 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
9. Paradyn Project: *Paradyn Parallel Performance Tools: User's Guide*, Paradyn Project, University of Wisconsin Madison, Computer Sciences Department, 1998
10. 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