

Performance Prophet: A Performance Modeling and Prediction Tool for Parallel and Distributed Programs

Sabri Pllana
Institute of Scientific Computing
University of Vienna
Nordbergstrasse 15, 1090 Vienna, Austria
pllana@par.univie.ac.at

Thomas Fahringer
Institute of Computer Science
University of Innsbruck
Technikerstrasse 21a, 6020 Innsbruck, Austria
tf@dps.uibk.ac.at

Abstract

High-performance computing is essential for solving large problems and for reducing the time to solution for a single problem. Current top high-performance computing systems contain 1000's of processors. Therefore, new tools are needed to support the program development that will exploit high degrees of parallelism. The issue of model-based performance evaluation of real world programs on large scale systems is addressed in this paper. We present the PerformanceProphet, which is a performance modeling and prediction tool for parallel and distributed programs. One of the main contributions of this paper is our methodology for reducing the time needed to evaluate the model. In addition, we describe our method for automatic performance model generation. We have implemented PerformanceProphet in Java and C++. We illustrate our approach by modeling and simulating a real-world material science parallel program.

1 Introduction

High-performance computing is essential for solving complex problems in science and engineering. In order to solve large problems, or to reduce the time to solution for a single problem, scientific programs are executed on parallel and distributed computing systems, which may consist of large number of computing nodes each having multiple processors. For instance, the computer that is listed as number one in the current list of 500 hundred most powerful computers in the world (see [16]), contains 32768 processors. However, efficient program development for such large computing systems is a great challenge per se. Therefore, new tools are needed to support the program development that will exploit high degrees of parallelism. The issue of efficient model-based performance evaluation of real

world programs on large-scale systems will be addressed in this paper.

Since the time that the term Software Performance Engineering (SPE) was coined in [15], many researchers [11, 2] have investigated techniques for the performance improvement of software systems in general. We have investigated the possibility of using SPE for high-performance programs in *particular*, because for these kind of programs the performance is of a paramount importance. We have to build a tool that lets programmers develop and evaluate the performance model of a program at an early development stage. This early insight helps to improve design decisions and avoid time-consuming modifications of the code of an already implemented program. In order to support the graphical performance modeling of high-performance parallel and distributed programs, we have customized the Unified Modeling Language (UML) [7]. Our UML based *performance modeling* approach is presented in [9, 10]. In this paper we will focus on *performance evaluation*.

We present the PerformanceProphet, which is a performance modeling and evaluation tool for high-performance programs. One of the main contributions of this paper is our methodology for reducing the time needed to evaluate the model. Our methodology involves the model simplification and the combination of mathematical modeling with discrete event simulation. Furthermore, in this paper we describe our method for automatic performance model generation for scientific programs. The performance model is generated based on the UML model of the program.

The main results of this paper are, (i) a methodology for reducing the time needed to evaluate the model of a computing system, (ii) a taxonomy of techniques for simulation of computing systems, and (iii) a method for automatic performance model generation for scientific programs.

This paper is structured as follows. In Section 2 we will briefly describe a set of relevant techniques for performance modeling and evaluation of computing systems. Section 3

will present our methodology for reducing the time needed to evaluate models of large scale computing systems. This will be followed by an overview of PerformanceProphet. Some of the features of PerformanceProphet will be illustrated with a case study in Section 5. Finally, some concluding remarks are made and future work is outlined in Section 6

2 Background

In this section we briefly describe a set of relevant techniques that are used by PerformanceProphet for performance modeling and evaluation of computing systems.

2.1 Discrete Event Simulation (DES)

Discrete Event Simulation (DES) is the simulation of a system in which the *state* of corresponding model changes at only a discrete set of points in time. A collection of *system variables* describes the *state* of system. For instance, a system variable of a network system may be the number of messages that are waiting for transmission. An *event* is an occurrence at a point in time that may change the state of system. For instance, an event of a network system is the message arrival. An *action state* (or an *activity*) is a period of a specific length, whose duration is determined at the entry of state. For instance, an action state of a network system is the message transmission. The system may wait until a set of conditions becomes true in a *wait state* (or passive state). For instance, a component of the network system may wait until the condition *message arrived* is true. While the duration of an action state is possible to determine in advance, in general it is not possible to determine in advance the duration of a wait state [8].

2.2 Mathematical Modeling (MathMod)

We use Mathematical Modeling (MathMod) to build *cost functions*. Cost functions model the time spent at a specific state of the system. For instance, the execution of a code block such as a sequence of computational operations, or service time of a machine resource such as network subsystem. Some of the commonly used mathematical modeling techniques for the cost function development are (i) *interpolation*, (ii) *regression*, (iii) *random variate generation*, and (iv) *number of operations*. The outcome of *number of operations* method is an expression that relates the number of operations (for instance computational and communication operations) and the execution time of a code block.

3 Methodology

The terms *workload*, *machine*, and *system* are used frequently in the literature, but not always with the same meaning. In this paper the term *workload* indicates a distributed and parallel program, whereas the term *machine* indicates a distributed and parallel computing architecture. The term *system* indicates a computing system. In our approach, the workload model and the machine model are considered as integral parts of the computing system model (see Figure 1(a)).

Our aim is to build high-level performance models of distributed and parallel computing systems. These performance models should offer *comparative accuracy*, since we usually use them to *compare* various parallelization strategies of large scientific programs. The time spent for evaluation of these models should be short, in order to explore a large set of possible parallelization strategies within a reasonable time. We aim to reduce the time needed to evaluate the model, by (i) simplifying the model, and by (ii) combining mathematical modeling (MathMod) and discrete event simulation (DES). We have briefly described DES and MathMod in Section 2.

The model simplification is achieved by using *grouping* and *neglecting* techniques. For instance, in the process of building a model for a scientific program, several program statements are *grouped* and considered as a single element of the program model. In this paper a group of one or more program statements is called *code block*. On the other hand, program statements that do not influence strongly the performance are *neglected*. Grouping and neglecting techniques are used to simplify the model of the computer architecture as well. For instance, components of the network subsystem of a computational node of a cluster architecture may be grouped and modeled as a single element of the model. By utilizing grouping and neglecting techniques we reduce the number of model elements. Subsequently, by reducing the number of model elements, the model evaluation time and memory requirements may be reduced.

Hybrid simulation models, which combine MathMod techniques and DES, present an efficient alternative to simulation-only models. Several approaches to combine these two techniques are described in [13, 4]. Here we first briefly describe our methodology of hybrid model building for a distributed and parallel program, and then we generalize our methodology for the whole computing system. A distributed and parallel program consists of two classes of code blocks:

1. Code blocks which involve one *processing unit* (a process or thread), for instance the execution of a sequence of computational operations.
2. Code blocks which involve multiple processing units,

such as collective communication operations and critical regions. This type of building blocks may impose synchronization among the processing units.

We use MathMod to model the execution time of the first type of building blocks as a parameterized *cost function* (see Section 2.2). On the other hand, for the behavioral modeling of the second type of building blocks we use DES.

The model of the distributed and parallel program (i.e. workload model) is one of components of the computing system model (see Figure 1(a)). We apply the same methodology for building of the whole computing system model, which includes the machine model and the workload model. The whole computing system model is split-up into *action states* and *wait states*. An *action state* is a period whose duration is calculated at the entry of state. Examples of action states include the execution of a code block such as a sequence of computational operations, or service time of a machine resource such as network subsystem. A *wait state* waits until a set of conditions becomes true. Wait states are used to model code blocks that involve multiple processing units such as critical regions, or waiting for the availability of a machine resource such as a network link.

The performance behavior of action states is modeled with MathMod techniques, whereas the behavior of wait states is simulated.

3.1 Techniques for Simulation of Computing Systems

In this section we briefly describe existing techniques for simulation of computing systems, and we compare and contrast them with our approach of *integrated simulation* (see Table 1).

Instruction-driven simulation models the workload with a stream of instructions. An instruction-level simulator models the machine. During the simulation, each instruction is interpreted by the simulator. In the instruction interpretation phase, the simulator may usually execute several instructions for a simulated instruction. Because of the low-level machine model the simulation is usually very slow. Therefore, it is not considered as an appropriate technique for simulation of real-world programs. Furthermore, instruction level simulators are usually limited to the instruction set of a specific processor type.

Execution-driven simulation models the workload by extending the actual program with calls to the simulation library. Usually the calls to routines of the simulation library are inserted in the source code of the actual program. For instance, for a message passing program, calls to message passing library (for instance MPI) are replaced with calls to the simulation library. The extended actual program is compiled and linked with the simulation library. During

Table 1. Simulation techniques for computing systems.

Simulation Technique	Workload	Machine
Instruction Driven	Instruction Stream	Instruction-Level Simulator
Execution Driven	Extended Program	Simulation Library
Trace Driven	Event Trace	DES
Model Driven	MathMod or DES	MathMod or DES
Integrated Simulation	MathMod and DES	MathMod and DES

the execution of the extended program, the simulation library models the performance behavior of the machine. A major drawback of this simulation technique is considered the memory requirement for simulation. Execution-driven simulation requires at least as much memory as the actual program, which behavior is simulated. The high memory requirement presents a major obstacle for the simulation of large-scale computing systems. Execution-driven simulation technique is usually used when the code of the fully developed program exists.

Trace-driven simulation models the workload with a stream of communication and computation events. The event trace is generated by observing the execution of program and storing the events of interest (such as message exchange) into a *trace file*. The machine is modeled with DES. During the simulation, the simulator simulates the events that are recorded in the trace file. This simulation technique is usually used when the code of the fully developed program exists.

Model-driven simulation models the workload and the machine by using MathMod or DES. Examples of commonly used modeling techniques include Petri net, and Markov chains. These techniques are not appropriate for performance evaluation of real-world programs, because in case of the large number of states is difficult to derive an analytical solution. In this case DES is used.

Our simulation technique, *integrated simulation*, applies an hybrid approach of combining MathMod and DES for

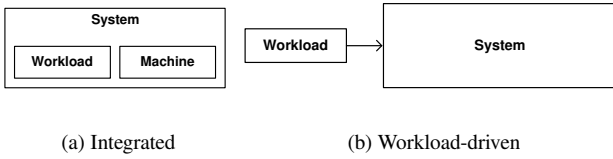


Figure 1. Types of simulation models for computing systems.

modeling the whole computing system. We use the same approach to build the model of workload and the model of machine. The workload model is integrated with the machine model in order to build the model of the computing system (see Figure 1(a)). Modeling elements, from which the workload model and machine model are composed, are implemented as classes. During the model building phase, instances (i.e. *objects*) of these classes are used to create an instance of the computing system. During the simulation run, an object simulates the behavior of a component of the system over time. Objects may interact with each other in order to simulate the behavior of the whole system.

Commonly used techniques for simulation of the computing systems (instruction-driven, execution-driven, trace-driven and model-driven) consider that the workload *logically* is not part of the system, and the machine is identified as the whole computing system (see Figure 1(b)). The workload is used as vehicle to *drive* the system during the simulation. Usually, these techniques do not use the same modeling approach for the workload and the machine (see Table 1). For instance, trace-driven simulation technique models the workload with a file which contains a sequence of events, and machine with DES.

3.2 Hybrid Performance Modeling

Our hybrid approach of performance modeling is illustrated with an example of point-to-point communication (see Figure 2). Process P_0 , after performing some computation, sends a message to process P_1 . Process P_1 receives the message. Computation is modeled with an *action state* (see Figure 2(a)). A *nonblocking send* is used to send the message (see Figure 2(a)), whereas a *blocking receive* is used to receive the message (see Figure 2(b)). *Computation* and *message transfer* are modeled with MathMod, whereas waiting to receive the message is simulated with DES.

4 PerformanceProphet Overview

In this section we give an overview of the PerformanceProphet.

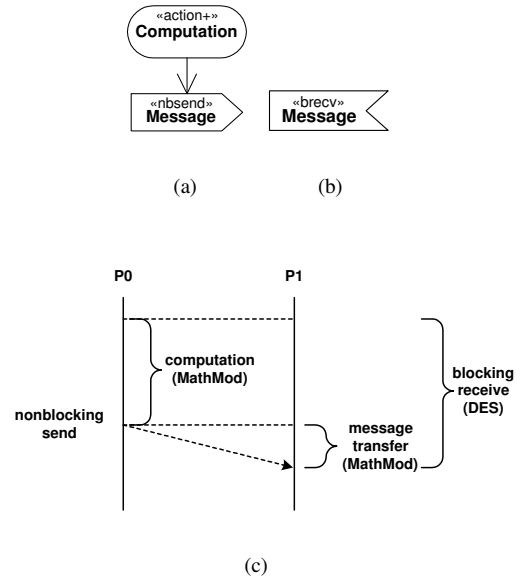


Figure 2. Hybrid modeling of point to point communication.

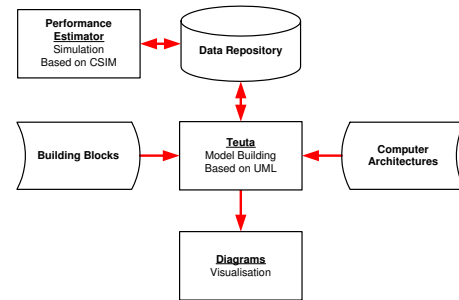


Figure 3. PerformanceProphet architecture.

Figure 3 depicts the architecture of PerformanceProphet. The main components of PerformanceProphet are *Teuta* and *Performance Estimator*.

Teuta is a platform independent tool for UML-based modeling of parallel and distributed programs. The model of the distributed and parallel program (i.e. workload model) is composed from the predefined *building blocks* (i.e. workload elements). An overview of the types of workload elements is depicted in Figure 4.

We have used the Java programming language for the implementation of *Teuta*, based on the Model-View-Controller (MVC) paradigm [6]. MVC is a paradigm that enforces the separation between the user interface and the rest of the application. The main components of *Teuta* are *model checking* component and *model traversing* component.

The model checking component of *Teuta* is responsible

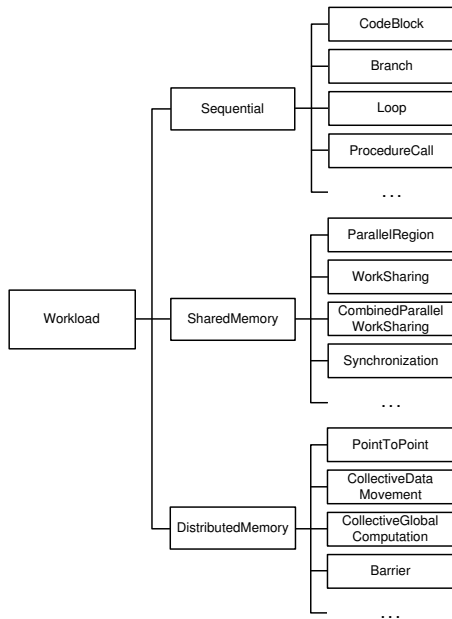


Figure 4. An overview of the types of workload elements.

for the correctness of the model. The rules for model checking are specified by using our XML based Model Checking Language (MCL). The model checker gets the model description from an MCL file. This MCL file contains a list of available diagrams, modeling elements and the set of rules that defines how the elements may be interconnected. Based on the *UML Rule Set* the model checker verifies whether the application model conforms to the UML specification. In addition, Teuta supports *semantic model checking* for the domain of High Performance Computing (HPC). The *HPC Domain Rule Set* specifies whether two modeling elements can be interconnected with each other, or nested one within another, based on their semantics. For instance, it is not allowed to place the modeling element *BARRIER* within the modeling element *CRITICAL*, because this would lead to the deadlock.

The model traversing component of Teuta provides the possibility to *walk* through the model, visit each modeling element, and access its properties (for instance element name). We use the model traversing for the generation of various model representations (such as XML and C++).

The implementation of Performance Estimator is based on CSIM [14]. CSIM is a general purpose simulation engine, which offers a set of generic classes that can be used to compose discrete event simulation models. In addition, based on these classes it is possible to define new classes that model components of a specific type of system by using C++ programming language.

The Performance Estimator evaluates the performance of a parallel and distributed program on a target cluster of SMP's. In order to accomplish this, we have developed a set of C++ classes that model basic program and machine components. Examples of these components include *Sequential Code Region*, *Send*, *Receive*, *Barrier*, and *SMP node*. The machine model for clusters of SMP's is composed by specifying parameters such as number of nodes and number of processors per node in the graphical user interface.

The communication between Teuta and the Performance Estimator is done via *Data Repository*. The repository implementation is based on the PostgreSQL [5] open-source relational database system.

The design of PerformanceProphet enables the model-based performance analyzes of parallel and distributed programs. The user develops graphically the performance model of a program based on the UML [10] by using Teuta; the program model is composed of existing building blocks. Thereafter, the program model is automatically transformed from UML into a C++ representation. Finally, the Performance Estimator evaluates the performance of the program on the computer architecture selected by user. In this way, it is possible to experiment with the model rather than with the real program.

4.1 Automatic Performance Model Generation

In this section we describe the main algorithm for performance model generation. As input serves the UML model of the program. The algorithm generates the C++ representation of the program performance model.

Usually, scientific programs are written in imperative languages such as *Fortran* or *C*. This type of programs is executed on parallel and distributed computing systems, which may consist of multiple nodes each having multiple processors, in order to solve large problems, or to reduce the time to solution for a single problem [3]. Message passing library (MPI) [1] is usually used to express internode parallelism, whereas OpenMP is used to express intranode parallelism.

We have identified that UML activity diagrams are suitable for modeling scientific imperative programs [10]. Therefore, we usually model a scientific program with one or more activity diagrams. Cost functions may be associated with modeling elements of an activity diagram. Usually, they are associated with *action states*.

Figure 5 depicts an instance of the process of workload characterization. The idea is to identify *code blocks* of a program that determine the performance. We may identify, for an existing program, code blocks that determine the performance by using a tool for profiling, such as SCALEA [17].

Figure 5(a) shows a code block of a program. This is

a nested loop, therefore it may strongly influence the performance. Figure 5(b) shows, for the code block, its corresponding UML model, which is a fragment of an activity diagram. However, this detailed UML representation of the code block is not necessary, since we are interested on the rough performance estimation. Therefore, we model the performance of the code block depicted in Figure 5(a) with the UML modeling element *action+* as it is shown in Figure 5(c). The associated cost function $F(M,N)$ models the execution time of the *action+* instance *CodeBlockX*.

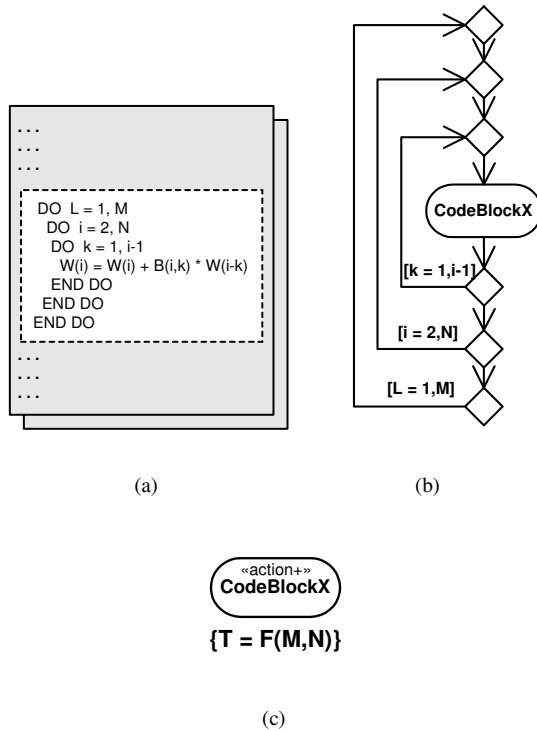


Figure 5. Program modeling.

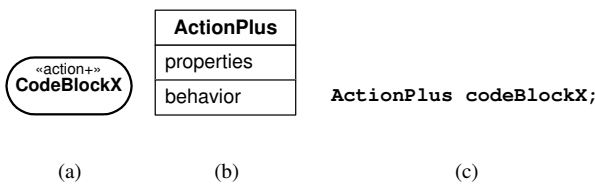


Figure 6. From the UML model to the performance model.

Figure 6 depicts an instance of transition from the model of a program, represented by UML activity diagram, to the workload model which is represented in C++ . The element

action+ is used to represent a code block of a program (see Figure 6(a)). For this modeling element we have defined the class *ActionPlus* ((see Figure 6(b)). The properties of the UML modeling element *action+* (for instance the element ID) are mapped to the *properties* of the class *ActionPlus*. The performance behavior of the element is defined in the method *behavior* of the class. This method is responsible for advancing the time in the simulation clock. This time is estimated either by a parameterized cost function or by simulating the behavior of the element. The name of the UML modeling element, in our example *codeBlockX*, is mapped to the name of the instance of the class *ActionPlus* (see Figure 6(c)).

Figure 7 describes our algorithm for the automatic generation of the performance model from the UML model of the program. The UML model with its diagrams and modeling elements forms a tree structure. During the generation process the tree is programmatically traversed, which makes possible to visit each modeling element and read its properties. Performance relevant modeling elements of the UML model are identified based on the element property *stereotype name*, which defines the type of the element.

5 Case Study: Program LAPW0

In this section we illustrate some of the features of PerformanceProphet by modeling and simulating a distributed scientific program. The program for our study LAPW0, which is a part of the WIEN2k package [12], was developed at Vienna University of Technology. The code of LAPW0 program is written by using FORTRAN90 and MPI. The LAPW0 program consists of 100 file modules; a module is a file that contains the source code.

We carried out a model-based performance analyzes of LAPW0 program, by following these steps: (i) we developed a high-level, graphical performance model with UML; (ii) PerformanceProphet transformed the program model from UML into a C++ representation; (iii) PerformanceProphet evaluated program performance. In this way, we were able to experiment with the model rather than with the real program.

Figure 8 depicts the UML model of the program LAPW0. In addition, it shows an instance of the association of a cost function to an activity.

The visualization of simulation results is depicted in Figure 9. The *bar chart* shows execution times for all 32 processes. The *pie chart* shows details for the process which is selected by the user in the drop down list (in this case process 0). The table shows simulation results for each performance modeling element of the selected process. Results, that are shown in the table, can be sorted in ascending or descending order by any column. For instance, if we want to identify elements that significantly contribute to the over-

Input: uml_model, UML representation

Output: perf_model, C++ representation

Method: A tree structure, which contains the model with its diagrams and modeling elements, is traversed during the generation process. Performance relevant modeling elements of the UML model are identified based on the stereotype name, which defines the type of the element.

```
// Identify and select performance modeling elements
FORALL(is diagram of uml_model) DO
  FORALL(is element of diagram) DO
    IF(element is performance element)
      add element to perf_elements;
    ENDIF
  ENDFOR
ENDFOR
// Globals
FORALL(variable of uml_model is global) DO
  add variable to perf_model;
ENDFOR
// Cost functions
FORALL(is element of perf_elements) DO
  IF(element has function)
    add function to perf_model;
  ENDIF
ENDFOR
// Program
// Locals
FORALL(variable of uml_model is local) DO
  add variable to perf_model;
ENDFOR
// Declare performance modeling elements
FORALL(is element of perf_elements) DO
  identify the type of element;
  add element representation to perf_model;
ENDFOR
// Define modeling elements and their control flow
FORALL(is diagram of uml_model) DO
  FORALL(is element of diagram) DO
    identify the type of element;
    add element representation to perf_model;
  ENDFOR
ENDFOR
```

Figure 7. The algorithm for performance model generation.

all program execution time, we sort the table by execution time.

The simulation and measurement results for two problem sizes and four machine sizes are presented numerically in Table 2 and graphically in Figure 10. The problem size is determined by the parameter NAT, which represents the number of atoms in a unit of the material. The number of nodes of the cluster architecture determines the machine size. Each node of the cluster has four CPUs. One process of the LAPW0 program is mapped onto one CPU of the cluster architecture. We validated the simulation model by comparing the simulation results with the measurement results, and found that our simulation model provides performance prediction results with sufficient accuracy to compare various designs of the program LAPW0.

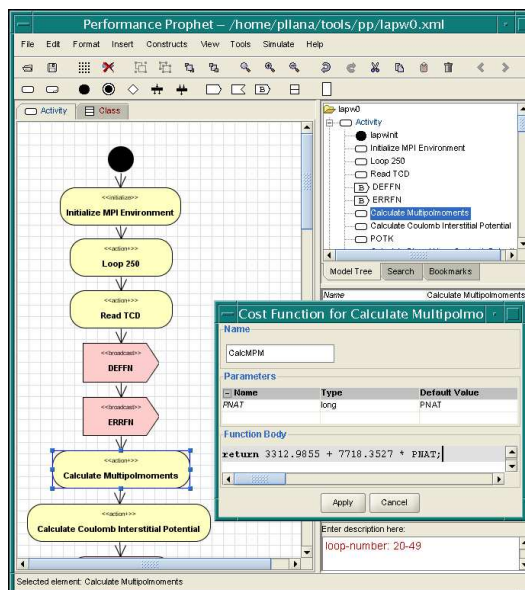


Figure 8. Performance modeling of the LAPW0.

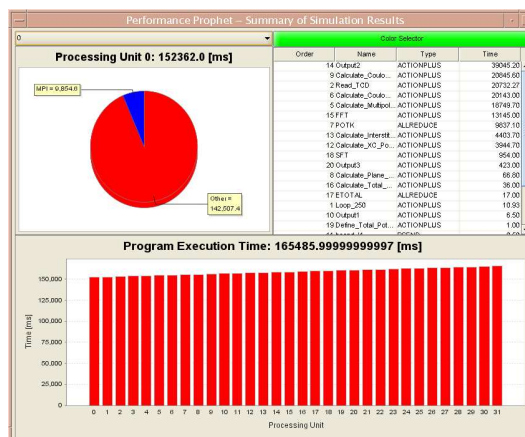


Figure 9. Visualization of simulation results for LAPW0.

6 Conclusions

In this paper we have presented PerformanceProphet, which provides the tool support for performance evaluation of large-scale computing systems. Furthermore, we have described our methodology for reducing the time needed to evaluate the model. Some of the features of PerformanceProphet have been illustrated with a case study. We have validated the simulation model by comparing the simulation results with the measurement results, and found that our simulation model provides performance prediction re-

Table 2. Simulation and measurement results for LAPW0. The number of atoms determines the problem size. The machine size is determined by the number nodes N . In the machine configuration notation $NxPy$, x denotes the number of nodes N , and y denotes the total number of processes P .

32 Atoms			
Machine	Sim.[sec.]	Measure.[sec.]	Error[%]
N1P4	271	264	2.48
N2P8	160	166	3.57
N4P16	116	131	11.13
N8P32	86	113	23.82
64 Atoms			
Machine	Sim.[sec.]	Measure.[sec.]	Error[%]
N1P4	527	501	5.16
N2P8	297	264	1.04
N4P16	194	197	1.62
N8P32	165	164	0.38

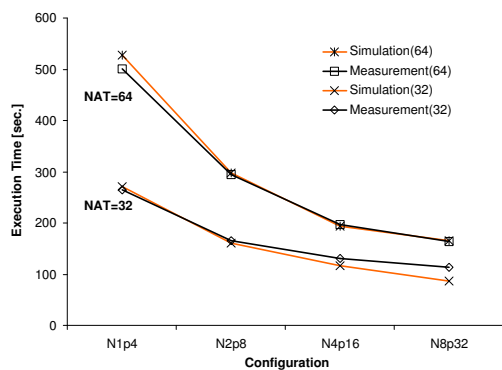


Figure 10. Simulation and measurement results for LAPW0. The number of atoms NAT determines the problem size. The machine size is determined by the number nodes N . In the machine configuration notation $NxPy$, x denotes the number of nodes N , and y denotes the total number of processes P .

sults with sufficient accuracy to compare various designs of the program LAPW0.

In future we plan to extend PerformanceProphet for performance modeling and prediction of Grid workflow applications.

References

- [1] L. Dagum and R. Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science and Engineering*, 5(1):46–55, Jan. /Mar. 1998.
- [2] H. E. El-Sayed, D. Cameron, and C. M. Woodside. Automation Support for Software Performance Engineering. In *Joint Int. Conf. on Measurement and Modeling of Computer Systems (Sigmetrics 2001/Performance 2001)*, ACM order no. 488010, Cambridge, MA, June 2001.
- [3] S. Graham, M. Snir, and C. Patterson. *Getting Up to Speed: The Future of Supercomputing*. The National Academies Press, 2004.
- [4] A. Hein and M. D. Cin. Performance and Dependability Evaluation of Scalable Massively Parallel Computer Systems with Conjoint Simulation. *ACM Transaction on Modeling and Computer Simulation*, 8:333–373, 1999.
- [5] R. Herzog. PostgreSQL - the Linux of Databases. *Linux Journal*, 46:14–24, February 1998.
- [6] G. Krasner and S. Pope. A cookbook for using the Model-View-Controller interface paradigm. *Journal of Object-Oriented Programming*, 1(3):26–49, 1988.
- [7] Object Management Group (OMG). Unified Modeling Language Specification. <http://www.omg.org>, March 2003.
- [8] R. Paul. Activity Cycle Diagrams and the Three Phase Approach. In *Proceedings of the 1993 Winter Simulation Conference*, pages 123–131, Los Angeles, California, United States, 1993. IEEE.
- [9] S. Pillana and T. Fahringer. On Customizing the UML for Modeling Performance-Oriented Applications. In *UML 2002, "Model Engineering, Concepts and Tools"*, LNCS 2460, Dresden, Germany. Springer-Verlag, October 2002.
- [10] S. Pillana and T. Fahringer. UML Based Modeling of Performance Oriented Parallel and Distributed Applications. In *Proceedings of the 2002 Winter Simulation Conference*, San Diego, California, USA, December 2002. IEEE.
- [11] R. Pooley and P. King. The Unified Modeling Language and Performance Engineering. *IEE Proceedings - Software*, 146(1):2–10, February 1999.
- [12] K. Schwarz, P. Blaha, and G. Madsen. Electronic structure calculations of solids using the WIEN2k package for material sciences. *Computer Physics Communications*, 147:71–76, 2002.
- [13] H. Schwetman. Hybrid Simulation Models of Computer Systems. *Communications of the ACM*, 21(9):718–723, 1978.
- [14] H. Schwetman. Model-based systems analysis using CSIM18. In *Winter Simulation Conference*. IEEE Computer Society Press, 1998.
- [15] C. Smith. *Performance Engineering of Software Systems*. Addison-Wesley, 1990.
- [16] TOP500 Supercomputer Sites, 2004. <http://www.top500.org/>.
- [17] H.-L. Truong and T. Fahringer. SCALEA: A Performance Analysis Tool for Parallel Programs. *Concurrency and Computation: Practice and Experience*, 15(11-12):1001–1025, 2003.