International Conference on Complex, Intelligent and Software Intensive Systems

Hybrid Performance Modeling and Prediction of Large-Scale Computing Systems

Sabri Pllana and Siegfried Benkner
Institute of Scientific Computing
Faculty of Computer Science
University of Vienna
Nordbergstrasse 15/C/3
1090 Vienna, Austria
Email: {pllana,sigi}@par.univie.ac.at

Fatos Xhafa
Department of Languages
and Informatics Systems
Polytechnic University of Catalonia
C/Jordi Girona 1-3
08034 Barcelona, Spain
Email: fatos@lsi.upc.edu

Leonard Barolli
Department of Information
and Communication Engineering
Fukuoka Institute of Technology
3-30-1 Wajiro-Higashi, Higashi-ku
Fukuoka 811-0295, Japan
Email: barolli@fit.ac.jp

***Abstract*—Performance is a key feature of large-scale computing systems. However, the achieved performance when a certain program is executed is significantly lower than the maximal theoretical performance of the large-scale computing system. The model-based performance evaluation may be used to support the performance-oriented program development for large-scale computing systems. In this paper we present a hybrid approach for performance modeling and prediction of parallel and distributed computing systems, which combines mathematical modeling and discrete-event simulation. We use mathematical modeling to develop parameterized performance models for components of the system. Thereafter, we use discrete-event simulation to describe the structure of system and the interaction among its components. As a result, we obtain a high-level performance model, which combines the evaluation speed of mathematical models with the structure awareness and fidelity of the simulation model. We evaluate empirically our approach with a real-world material science program that comprises more than 15,000 lines of code.**

I. Introduction

The solution of resource-demanding scientific and engineering computational problems involves the execution of programs on large-scale computing systems, which commonly consist of multiple computational nodes, in order to solve large problems or to reduce the time to solution for a single problem. However, there is a widening gap between the maximal theoretical performance and the achieved performance when a certain program is executed on a large-scale parallel and distributed computing system. This gap may be reduced by tuning the performance of a program for a specific computing system. Commonly, the programmer develops multiple versions of the program following various parallelization strategies. Thereafter, the programmer assesses the performance of each program version, and selects the program version that achieves the highest performance. The code-based performance tuning of a program is a time-consuming and error-prone process that involves many cycles of code editing, compilation, execution, and performance analysis. This problem may be alleviated by using the model-based performance evaluation.

In this paper we present a methodology and the corresponding tool-support for performance modeling and prediction of parallel and distributed computing systems, which may be used in the process of performance-oriented program development for providing performance prediction results starting from the early program development stages. Based on the performance model, the performance can be predicted and design decisions can be influenced without time-consuming modifications of large parts of an implemented program.

We propose a hybrid approach for performance modeling and prediction of parallel and distributed computing systems, which combines mathematical modeling and discrete-event simulation. Our aim is to combine the evaluation speed of mathematical models with the structure awareness and fidelity of the simulation model. For the purpose of evaluation of our approach we have developed a performance modeling and prediction system called Performance Prophet. We demonstrate the usefulness of Performance Prophet by modeling and simulating a real-world material science program that comprises more than $15,000$ lines of code. In our case study, the model evaluation with Performance Prophet on a single processor workstation is several thousand times faster than the execution time of the real program on our cluster.

The rest of this paper is organized as follows. Our approach for hybrid performance modeling and prediction of parallel and distributed computing systems is described in Section II. We evaluate empirically our approach in Section III. The related work is discussed in Section IV. Finally, Section V concludes the paper and briefly describes the future work.

II. Hybrid performance modeling and prediction

Commonly for performance modeling of computing systems is used mathematical modeling (MathMod) or discrete event simulation (DES). When applied separately, each of these approaches has severe limitations.

Mathematical models commonly represent the whole computing system as a symbolic expression that lacks the structural information [1]. An example of a mathematical performance model that models the program execution time is expressed as follows,

$$T_{ProgExec} = C_{Op} T_{Av},$$

where C_{Op} is the number of operations and T_{Av} is the average execution time of an operation. We may observe that there is no identifiable structural information in this model. The information such as the execution order of operations, or the control flow is not contained in the model.

Detailed simulation models commonly are so slow that the assessment of real-world programs is impractical, or for the model evaluation are needed very large resources (processors and memory) that may not be available. For instance RSIM is a simulator of CC-NUMA shared-memory machines [2]. RSIM comprises a detailed (that is a cycle-level) machine model that allows the analysis of the performance effects of architectural parameters. Therefore, it is suitable to evaluate various designs of CC-NUMA shared-memory machines. However, because the simulation of the program execution with RSIM is very slow (several thousands times slower than the program execution on the real machine), it is not suitable for evaluation of various designs of real-world programs.

Our aim is to combine the good features of both approaches. For instance, we would like to have the model evaluation efficiency of mathematical performance models and the structure awareness of simulation models. A model that combines mathematical modeling with discrete-event simulation is referred to as hybrid model [3].

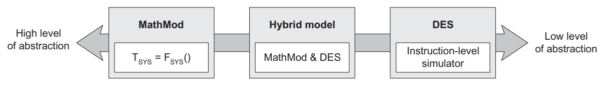


Fig. 1. Hybrid models combine the features of MathMod models and DES models. MathMod stands for Mathematical Modeling; DES stands for Discrete Event Simulation.

Figure 1 shows that, considering the level of abstraction, the hybrid performance models of computing systems reside somewhere between MathMod models and DES models. An important feature of hybrid models is that they permit the system modeling at various levels of abstraction. The MathMod dominated hybrid models are at a higher level of abstraction and more efficient than the DES dominated hybrid models. On the other hand, the structure of system under study is modeled in more detail with the DES dominated hybrid models.

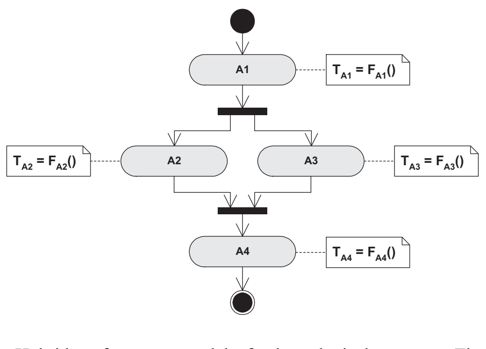
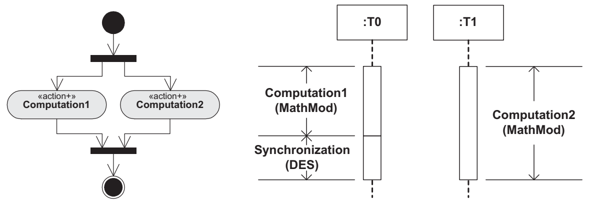


Fig. 2. Hybrid performance model of a hypothetical program. The performance of activities is modeled with MathMod. The control flow is modeled with DES.

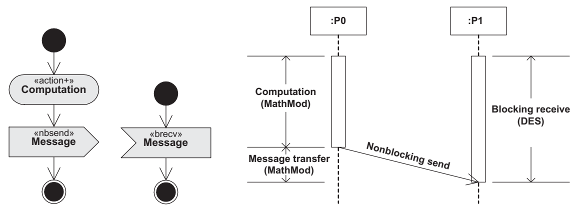
Figure 2 depicts the activity diagram of a hypothetical program. Activities $\{A_i | 1 \leq i \leq 4\}$ correspond to the code blocks of program. To each activity A_i is associated a parameterized *cost function* $F_{Ai}()$, which models the execution time of activity A_i. Functions $\{F_{Ai}() | 1 \leq i \leq 4\}$ are obtained using the MathMod techniques. The structure of program, which includes activities and their order of execution, is modeled with DES.



(a) (b)

Fig. 3. Hybrid performance modeling of a parallel region.

Figure 3 depicts our hybrid approach for performance modeling of a parallel region. The parallel region executes activities *Computation1* and *Computation2* in parallel (see Figure 3(a)). Thread T_0 executes activity *Computation1*, whereas activity *Computation2* is executed by thread T_1 (see Figure 3(b)). The execution of activities *Computation1* and *Computation2* is modeled with MathMod. The synchronization of threads is modeled with DES.



(a) P_0 (b) P_1 (c) Interprocess communication

Fig. 4. Hybrid performance modeling of point-to-point interprocess communication.

Figure 4 depicts our hybrid approach for performance modeling of point-to-point communication. 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 *activity* (see Figure 4(a)). The sending process P_0 uses a *nonblocking send* to send the message (see Figure 4(a)), whereas the receiving P_1 process uses a *blocking receive* to receive the message (see Figure 4(b)). *Computation* and the *message transfer* are modeled with MathMod, whereas waiting to receive the message is simulated with DES (see Figure 4(c)).

The model of parallel and distributed program (that is the workload model) is one of components of the computing system model. We apply the same methodology for building of the whole computing system model, which includes the machine model (that is the computer architecture) and the workload model. The behavior of the whole computing system is split-up into action states and wait states. 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. Wait states are used to model code blocks that involve multiple processing units such as parallel regions, or waiting for the availability of a machine resource such as a processor. *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* [4]. Therefore, we model the performance behavior of action states with MathMod techniques, whereas we simulate the behavior of wait states.

III. Evaluation

For the purpose of evaluation of our approach we have developed a performance modeling and prediction system called Performance Prophet [10].

A. Performance Prophet

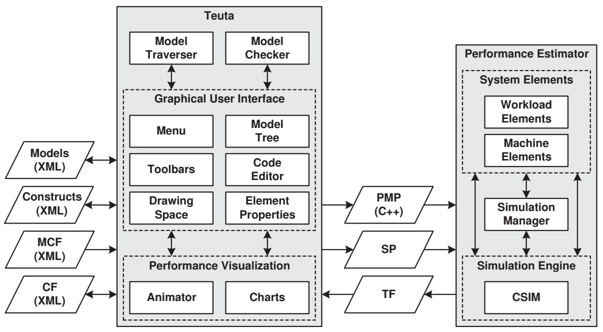


Fig. 5. The architecture of Performance Prophet. Abbreviations: Model Checking File (MCF), Configuration File (CF), Performance Model of Program (PMP), System Parameters (SP), Trace File (TF).

Figure 5 depicts the architecture of Performance Prophet. The main components of Performance Prophet are Teuta and Performance Estimator. Teuta is a platform independent tool for graphical modeling of parallel and distributed programs. The role of Performance Estimator, in the context of Performance Prophet, is to estimate the performance of a program on a computing machine.

Teuta comprises the following parts: Model Checker, Model Traverser, Graphical User Interface (GUI), and the components for Performance Visualization (see Figure 5). The GUI of Teuta is used for the development of performance model based on the Unified Modeling Language (UML) [5]. The Model Checker is used to verify whether the model conforms to the UML specification. The Model Traverser is used for generation of different model representations (XML and C++). The Performance Visualization components are used for visualization of the performance results.

Element MCF indicates the XML file, which is used for the model checking. The XML files that are used for the configuration of Teuta are indicated with the element CF.

The communication between Teuta and the Performance Estimator is done via elements PMP, SP and TF. Element PMP indicates the C++ representation of the program's performance model. PMP is generated by Teuta and serves as input information for the Performance Estimator. Element SP indicates a set of system parameters. The parameters of system include the number of computational nodes, the number of processors per node, the number of processes, and the number of threads. The Performance Estimator uses SP for building the model of system, whose performance is estimated. Element TF represents the trace file, which is generated by the Performance Estimator as a result of the performance evaluation. Teuta uses TF for the visualization of performance results.

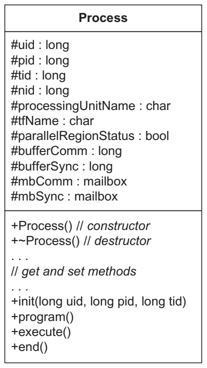
The Performance Estimator comprises the following components: Simulation Manager, CSIM, Workload elements, and Machine elements (see Figure 5). The Simulation Manager builds the system model based on the user specification, starts and ends the simulation run, and stores the performance results. A set of Workload and Machine elements are provided for building of the system model. In what follows in this section we describe components of the Performance Estimator in more detail.

CSIM (Mesquite Software [6], [7]) is a process-oriented general-purpose simulation library. CSIM supports the development of discrete-event simulation models, by using the standard programming languages C and C++. Because of the nature of compiled C and C++ programs and CSIM's dynamic memory allocation, the developed simulation models are compact and efficient. CSIM supports the process-oriented world view. The system is represented by a set of static components (that is CSIM facilities) and a set of dynamic components (that is CSIM processes) that use the static components. CSIM provides a set of abstractions (such as processes and facilities) for the model development, and many useful features (such as statistics collection or random variate generation) that are needed in a simulation study. CSIM processes operate in parallel in simulated time. Therefore, CSIM provides mechanisms for the synchronization of processes and for the interprocess communication. For the synchronization of CSIM processes are commonly used CSIM events. The communication among CSIM processes is accomplished via CSIM mailboxes.

Based on CSIM we have developed a set of C++ classes that model basic program and machine components. Examples of these components include *Process*, *Send*, *Receive*, *ParallelDo*, and *Node*.

Figure 6 depicts the class *Process*, which we have developed to model processing units (that is processes or threads) of a computing system. The design of class *Process* permits the modeling of a large group of parallel and distributed scientific programs.

The structure of *Process* class is depicted in Figure 6(a). The unit ID (*uid*), process ID (*pid*) and thread ID (*tid*) are used to uniquely identify the processing unit during the simulation. The node ID (*nid*) indicates the computational node on which the processing unit is mapped. The attribute *processingUnitName* is mainly used to identify the processing



(a) Structure

```
1: void Process::execute()
2: { // CSIM process
3:   create(processingUnitName);
4:   set_priority(2);
5:   node[nid].getCPUs()->reserve();
6:   program();
7:   node[nid].getCPUs()->release();
8:   end();
9: };
```

(b) Method execute()

Fig. 6. Class Process.

unit in simulation reports. The performance evaluation results of processing unit are stored in the file that is specified in the attribute *tfName*. The attribute *parallelRegionStatus* indicates whether the processing unit is executing a parallel region (for instance, a code region enclosed within OpenMP directives PARALLEL and END PARALLEL [8]). The attributes *bufferSync* and *mbSync* serve for the synchronization among processing units. The communication among processing units is performed via attributes *bufferComm* and *mbComm*. We may observe that the CSIM type *mailbox* is used to define *mbComm* and *mbSync*. The methods of class *Process* for getting or setting values of attributes are straightforward, and therefore, they are not depicted in Figure 6(a). The methods *init()* and *end()* are invoked when the operation of process is initialized and completed respectively. The method *program()* models the performance behavior of the program under study. Teuta generates automatically the code for the method *program()* based on the UML model that is specified by the user.

Figure 6(b) depicts the implementation of method *execute()* of class *Process*. In the line 3 is used the CSIM statement `create()` to define the method *execute()* as a CSIM process. The CSIM statement `set_priority()` sets the priority of the process (see line 4). Higher values of the priority mean higher priority of process execution. For instance, the process with priority 2 will execute before the process with priority 1 if the priority determines the order of execution. In the line 5 the process obtains the processor from the node. The statement in the line 6 invokes the method *program()*, which models the performance behavior of the program under study. In the line 7 the process releases the processor. Line 8 is used to notify the end of process execution.

Basically, the method *program()* of class Process specifies the execution flow of a collection of performance modeling elements. Each performance modeling element corresponds to a code block of the program, whose performance is modeled (see Figure 7). The execution of a performance modeling element models the performance behavior of a code block during the program execution.

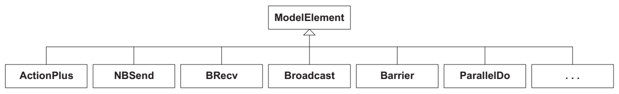
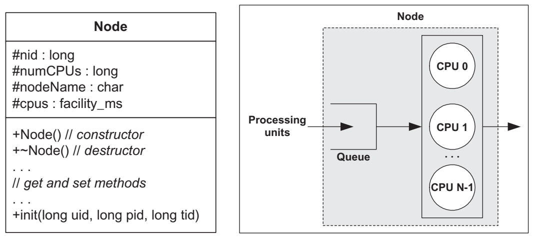


Fig. 7. Performance modeling elements of program.

Figure 7 depicts the hierarchy of classes of Performance Estimator that are used for construction of the method *program()*. On the top of hierarchy is the class *ModelElement*. The subclasses of class *ModelElement* correspond to various code blocks of parallel and distributed programs. A group of one or more program statements is referred to as a *code block*. Examples of subclasses of class *ModelElement* include: *ActionPlus*, *NBSend*, *BRecv*, *Broadcast*, *Barrier*, and *ParallelDo*. Instances of these subclasses are used to represent the performance modeling elements in the method *program()* of class *Process* (see Figure 6).



(a) (b)

Fig. 8. Class *Node*.

Figure 8 depicts the class *Node*, whose instances we use for modeling the computational nodes of computer architectures. The structure of class *Node* is depicted in Figure 8(a). Attribute *nid* is used to uniquely identify instances of the class *Node*. The number of processors per node is specified with attribute *numCPUs*. Attribute *nodeName* is used to specify the name of the node. The CSIM type *facility_ms* is used to define processors of the node (that is *cpus*). The method *init()* of class *Node* is invoked when operation of the node is initialized.

Figure 8(b) depicts the structure of a node. A node comprises a set of processors $\{CPU_0, CPU_1, .., CPU_{N-1}\}$ and a queue. Processing units (processes or threads) may use any of the available processors. If there is no processor available, then processing units wait in the queue. The default queue discipline is *first come first served*. Other queue disciplines, such as *round robin*, may be specified. If the *round robin* queue discipline is specified, then a processing unit uses a processor for the specified amount of time. Thereafter, the processing unit is preempted and the next processing unit that is waiting in the queue obtains the processor. Commonly, high performance programs are mapped on machines with sufficient hardware resources in the manner that processing units do not have to compete for processors. Nevertheless, the capability of simulation of situations when multiple processing units share one processor may be useful to reveal the performance drawbacks of such mappings.

B. Case study

In this section we demonstrate the usefulness of Performance Prophet by modeling and simulating a real-world material science program. For our case study we use LAPW0, which is a part of WIEN2k package [9]. WIEN2k is a program package for calculation of the electronic structure of solids based on the density-functional theory. It is worth to mention that the 1998 Nobel Prize in Chemistry was awarded to Walter Kohn for his development of the density-functional theory [10]. LAPW0 calculates the effective potential within a unit cell of a crystal. The code of LAPW0 program is written in Fortran 90 and MPI [11]. LAPW0 comprises about 15,000 lines of code.

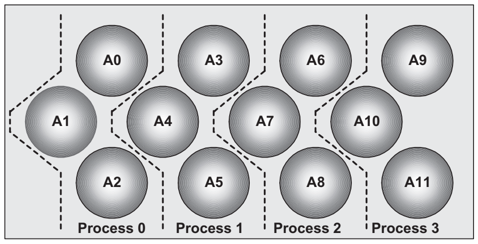


Fig. 9. An instance of LAPW0 domain decomposition. Number of atoms (NAT) is 12; number of atoms per process (PNAT) is 3.

LAPW0 is executed in SPMD fashion (all processors execute the same program) on a multiprocessor computing system. A domain decomposition approach is used for parallelization of LAPW0 (see Figure 9). The unit of material, for which LAPW0 calculates the effective potential, comprises a certain number of atoms (NAT). Atoms are evenly distributed to the available processes. This means that each process is responsible for calculation of the effective potential for a subset of atoms. For NP available processes, each process obtains $PNAT = NAT/NP$ atoms. LAPW0 uses an algorithm that aims to distribute a similar (if not the same) number of atoms to each process for any given positive integer values of the number of atoms and the number of processes.

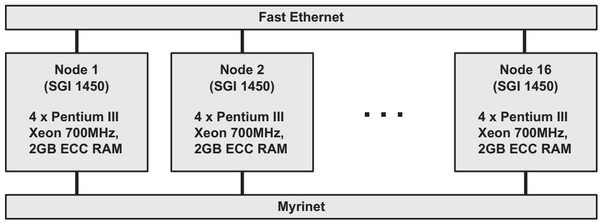
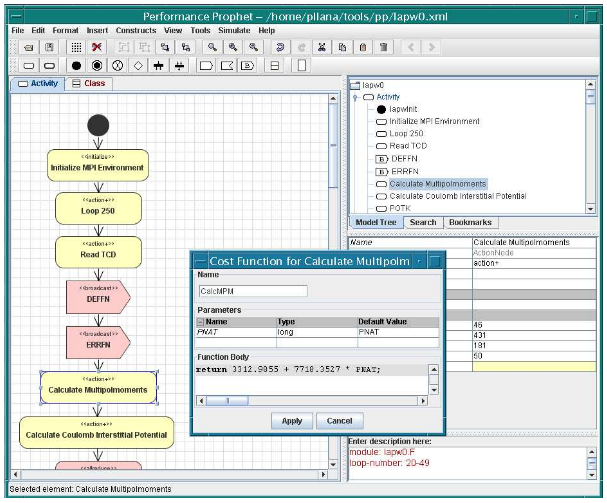


Fig. 10. Experimentation platform. Gescher cluster has 16 SMP nodes. Each node has four processors.

Figure 10 depicts the architecture of Gescher cluster, which is located at Institute of Scientific Computing, University of Vienna [12]. Gescher is a 16 node SMP cluster. All nodes of Gescher are of type SGI 1450. Each node of the cluster has four Pentium III Xeon 700MHz processors , and 2GB ECC RAM. The nodes of Gescher are interconnected via a 100Mbit/s Fast Ethernet network and a Myrinet network. For our experiments we have used the Fast Ethernet network. Gescher serves as our platform for performance measurement experiments of LAPW0.

In what follows in this section we develop and evaluate the model of LAPW0 with Performance Prophet. We validate the model of LAPW0 by comparing simulation results with measurement results.


Performance Prophet – /home/pllana/tools/pp/lapw0.xml
File Edit Format Insert Constructs View Tools Simulate Help
Activity
Class
<<initialize>>
Initialize MPI Environment
<<action+>>
Loop 250
<<action+>>
Read TCD
<<broadcast>>
DEFFN
<<broadcast>>
ERRFN
<<action+>>
Calculate Multipolmoments
<<action+>>
Calculate Coulomb Interstitial Potential
<<allreduce>>
lapw0
Activity
lapwInit
Initialize MPI Environment
Loop 250
Read TCD
DEFFN
ERRFN
Calculate Multipolmoments
Calculate Coulomb Interstitial Potential
POTK
Model Tree
Search
Bookmarks
Cost Function for Calculate Multipolm
Apply
Cancel
Enter description here:

Name: CalcMPM

Parameters

Name	Type	Default Value
PNAT	long	PNAT

Function Body

```
return 3312.9855 + 7718.3527 * PNAT;
```

Name	Calculate Multipolmoments
	ActionNode
	action+
	46
	431
	181
	50

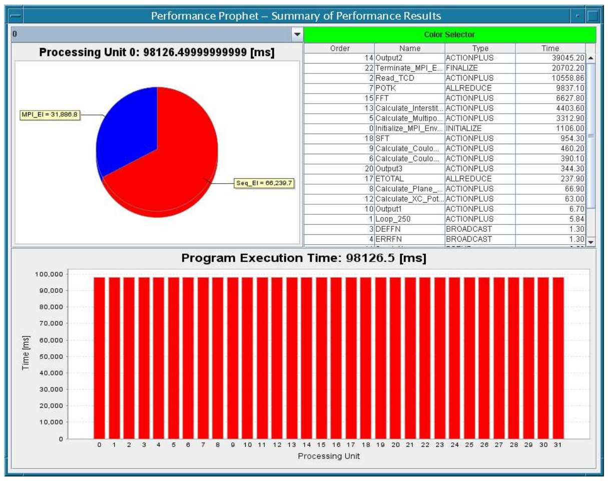
```
module: lapw0.F
loop-number: 20-49
```

Selected element: Calculate Multipolmoments

Fig. 11. Performance modeling of LAPW0.

Figure 11 illustrates the procedure for the development of performance model of LAPW0 with Performance Prophet. Due to space limitations, in Figure 11 it is depicted just a fragment of the UML model of LAPW0. We developed the model of LAPW0 by using the modeling elements that are available in the toolbar of Performance Prophet. Basically, Performance Prophet permits to associate to each modeling element a cost function. A cost function models the execution time of the code block that is represented by the performance modeling element. Figure 11 depicts the association of cost function *CalcMPM* to action *Calculate Multipolmoments*. This cost function was generated based on measurement data by using regression. Regression is a technique for fitting a curve through a set of data values using some goodness-of-fit criterion.

Figure 12 depicts the visualization of performance prediction results of LAPW0 with Performance Prophet. 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 overall program execution time, we sort the table by execution time.


Performance Prophet – Summary of Performance Results

Processing Unit 0: 98126.49999999999 [ms]

Order	Name	Type	Time
14	Output2	ACTIONPLUS	39045.20
22	Terminate_MPI_E...	FINALIZE	20702.20
2	Read_TCD	ACTIONPLUS	10558.86
7	POTK	ALLREDUCE	9837.10
15	FFT	ACTIONPLUS	6627.80
13	Calculate_Interstit...	ACTIONPLUS	4403.60
5	Calculate_Multipo...	ACTIONPLUS	3312.90
0	Initialize_MPI_Env...	INITIALIZE	1106.00
18	SFT	ACTIONPLUS	954.30
9	Calculate_Coulo...	ACTIONPLUS	460.20
6	Calculate_Coulo...	ACTIONPLUS	390.10
20	Output3	ACTIONPLUS	344.30
17	ETOTAL	ALLREDUCE	237.90
8	Calculate_Plane_...	ACTIONPLUS	66.90
12	Calculate_XC_Pot...	ACTIONPLUS	63.00
10	Output1	ACTIONPLUS	6.70
1	Loop_250	ACTIONPLUS	5.84
3	DEFFN	BROADCAST	1.30
4	ERRFN	BROADCAST	1.30

Program Execution Time: 98126.5 [ms]

Fig. 12. Visualization of performance prediction results for LAPW0. Results are obtained from the simulation of execution of LAPW0 on eight four-processor nodes for problem size 32 atoms. On each processor is mapped one process (total 32 processes).

TABLE I
SIMULATION AND MEASUREMENT RESULTS FOR LAPW0. IN $NxPy$, x DENOTES THE NUMBER OF NODES N, AND y DENOTES THE TOTAL NUMBER OF PROCESSES P. T_s IS SIMULATED TIME, T_m IS MEASURED TIME, AND T_e EVALUATION TIME. ALL TIMES ARE EXPRESSED IN SECONDS [S].

NAT = 32					
System	T_s [s]	T_m [s]	T_e [s]	T_m/T_e	Error [%]
N1P4	280	264	0.01	26,400	6
N2P8	170	166	0.02	8,300	2
N4P16	126	131	0.04	3,275	3
N8P32	98	113	0.08	1,413	13
NAT = 64					
System	T_s [s]	T_m [s]	T_e [s]	T_m/T_e	Error [%]
N1P4	543	501	0.01	50,100	8
N2P8	314	264	0.02	14,700	7
N4P16	211	197	0.04	4,925	7
N8P32	184	164	0.09	1,822	12

We validated the performance model of LAPW0 by comparing simulation results with measurement results for two problem sizes and four system configurations. The problem size is determined by the parameter NAT, which indicates the number of atoms in a unit of the material. We have validated the performance model of LAPW0 for $NAT = 32$ and $NAT = 64$. The system configuration is determined by the number of nodes and the number of processing units. We have validated the performance model of LAPW0 for the following system configurations: one node and four processes (N1P4), two nodes and eight processes (N2P8), four nodes and 16 processes (N4P16), eight nodes and 32 processes (N8P32). Each node comprises four processors. On each processor is mapped one process.

Table I depicts simulation and measurement results for LAPW0. The second column of table, which is indicated with T_s, shows the performance prediction results for LAPW0 that we have obtained by simulation. Measurement results of LAPW0 are presented in the third column, which is indicated with T_m. The column that is indicated with T_e presents the CPU time needed for evaluation of the performance model of LAPW0 by simulation. All simulations were executed on a Sun Blade 150 (UltraSPARC-IIe 650MHz) workstation. We compare the time needed to execute the real LAPW0 program on our SMP cluster with the time needed to evaluate the performance model on a Sun Blade 150 workstation in the column that is indicated with T_m/T_s. We may observe that model-based performance evaluation of LAPW0 with Performance Prophet was several thousand times faster than the corresponding measurement-based evaluation. The rightmost column of the table shows the percentage error, which serves to quantify the prediction accuracy of Performance Prophet. We have calculated the percentage error using the following expression,

$$Error[\%] = \frac{|T_s - T_m|}{T_m}100,$$

where T_s is the simulated time and T_m is the measured time. We may observe that the prediction accuracy of Performance Prophet for LAPW0 was between 2% and 13%. The average percentage error was 7%. Simulation and measurement results for LAPW0 are graphically presented in Figure 13.

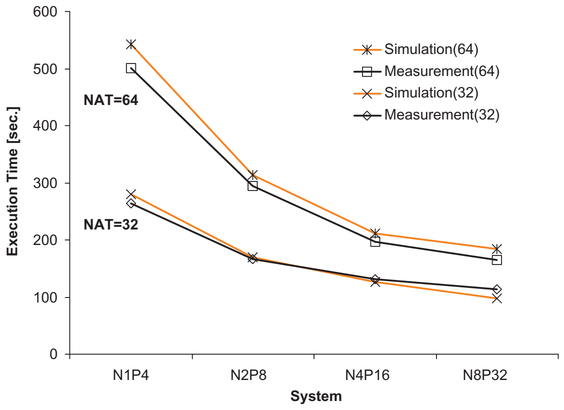


Fig. 13. Simulation and measurement results for LAPW0.

C. A Caveat to our Approach

Performance Prophet uses high-level models of machine (that is computer architecture) in order to decrease the effort for model evaluation. Since the target user of Performance Prophet is the program developer, implementation details of machine model are hidden from the user. The user may simply modify parameters of the machine model via the GUI of Performance Prophet, but some C++ programming is required for the structural modification of machine model. Since the machine model is at a high-level of abstraction and its structure may be changed only programmatically, we consider that Performance Prophet is not well suited for computer architecture developers.

IV. Related Work

Most of approaches for performance evaluation of computing systems [13] are able to cope only with small programs such as matrix-vector multiplication. There are several reasons for the lack of scalability: (1) a very complex code analysis is used during the workload modeling that does not scale up to the size and complexity of the real-world programs [1], (2) a detailed machine model is used that is so slow that makes impractical the simulation of real-world programs [2], [14], or (3) for the model evaluation are required very large resources (processors and memory) that may not be available [15], [16], [17]. Our approach has addressed this issue by using model simplification techniques, combination of mathematical modeling with discrete event simulation, and by using a simple machine simulation model.

Performance models that represent the whole program and machine as a symbolic expression lack the structural information [1]. Consequently, it is difficult to identify the part of system that is responsible for the suboptimal performance. Our approach supports the development of performance models at various levels of abstraction. For instance, for workload modeling are used UML activity diagrams [18]. An activity may represent a single instruction, or larger blocks of the program (for instance a *loop*), or the whole program. Furthermore, our approach uses the discrete-event simulation to describe the structure of system and the interaction among its components.

V. Conclusions and Future Work

The performance-oriented program development for large-scale computing systems is a time-consuming, error-prone, and expensive process that involves many cycles of code editing, compiling, executing, and performance analysis. This problem is aggravated when the program developer has access to only a part of the computing system resources and for only a limited time. The limited access to large-scale computing systems is a common practice, because the resources of this kind of systems are shared among many users. The model-based performance analysis may be used to overcome these obstacles.

In this paper we have presented a hybrid approach for the development of high-level performance models of large-scale computing systems, which combines mathematical modeling and discrete-event simulation. Our aim was to combine the model evaluation efficiency of mathematical performance models with the structure awareness of simulation models.

For the purpose of evaluation of our approach we have developed Performance Prophet, which is a performance modeling and prediction system. Performance Prophet provides a UML-based GUI, which alleviates the problem of specification and modification of the performance model. Based on the user-specified UML model of a program, Performance Prophet automatically generates the corresponding performance model and evaluates it by simulation. We have demonstrated the usefulness of Performance Prophet by modeling and simulating LAPW0, which is a real-world material science program that comprises about $15,000$ lines of code. In our case study, the model evaluation with Performance Prophet on a single processor workstation was several thousand times faster than the execution time of the real program on our SMP cluster. We validated the model of LAPW0 by comparing the simulation results with measurement results for two problem sizes and four system configurations. The average prediction accuracy was 7%.

In future we plan to investigate the applicability of our approach for performance prediction of Grid workflows.

References

[1] D. Kerbyson, A. Hoisie, and H. Wasserman, "Use of Predictive Performance Modeling During Large-Scale System Installation," *Parallel Processing Letters*, vol. 15, no. 4, December 2005.

[2] "Rice Simulator for ILP Multiprocessors (RSIM)," http://rsim.cs.uiuc.edu/rsim/.

[3] H. Schwetman, "Hybrid Simulation Models of Computer Systems," *Communications of the ACM*, vol. 21, no. 9, pp. 718–723, 1978.

[4] R. Paul, "Activity Cycle Diagrams and the Three Phase Approach," in *Proceedings of the 1993 Winter Simulation Conference*. Los Angeles, California, United States: IEEE, 1993, pp. 123–131.

[5] Object Management Group (OMG), "UML 2.0 Superstructure Specification," http://www.omg.org, August 2005.

[6] "Mesquite Software," http://www.mesquite.com/.

[7] H. Schwetman, "CSIM19: A Powerful Tool for Building System Models," in *Winter Simulation Conference (WSC 2001)*. Arlington, VA, USA: ACM, December 2001, pp. 250–255.

[8] "Open specifications for Multi Processing (OpenMP)," http://www.openmp.org/.

[9] K. Schwarz, P. Blaha, and G. Madsen, "Electronic structure calculations of solids using the WIEN2k package for material sciences," *Computer Physics Communications*, vol. 147, pp. 71–76, 2002.

[10] "The 1998 Nobel Prize in Chemistry," http://nobelprize.org/nobel_prizes/chemistry/laureates/1998/.

[11] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI - 2nd Edition: Portable Parallel Programming with the Message Passing Interface (Scientific and Engineering Computation)*. MIT Press, 1999.

[12] "Gescher Cluster. University of Vienna, Institute of Scientific Computing," http://gescher.vcpc.univie.ac.at/.

[13] S. Pllana, I. Brandic, and S. Benkner, "Performance Modeling and Prediction of Parallel and Distributed Computing Systems: A Survey of the State of the Art," in *The Fifth International Conference on Complex, Intelligent and Software Intensive Systems - 3PGIC Workshop*. Vienna, Austria: IEEE Computer Society, April 2007.

[14] C. Hughes, V. Pai, P. Ranganathan, and S. Adve, "RSIM: Simulating Shared-Memory Multiprocessors with ILP Processors," *IEEE Computer*, vol. 35, no. 2, pp. 40–49, February 2002.

[15] T. Wilmarth, G. Zheng, E. Bohm, Y. Mehta, N. Choudhury, P. Jagadishprasad, and L. Kale, "Performance Prediction using Simulation of Large-scale Interconnection Networks in POSE," in *2005 Workshop on Principles of Advanced and Discributed Simulation (PADS)*. Monterey, California: IEEE Computer Society, June 2005.

[16] G. Zheng, G. Kakulapati, and L. Kale, "BigSim: A Parallel Simulator for Performance Prediction of Extremely Large Parallel Machines," in *18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*. Santa Fe, New Mexico, USA: IEEE Computer Society, April 2004.

[17] D. Kvasnicka, H. Hlavacs, and C. Ueberhuber, "Simulating Parallel Program Performance with CLUE," in *International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*. Orlando, Florida, USA: The Society for Modeling and Simulation International, July 2001, pp. 140–149.

[18] S. Pllana 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: IEEE, December 2002.