A novel approach for hybrid performance modelling and prediction of large-scale computing systems

Sabri Pllana* and Siegfried Benkner

Department of Scientific Computing,
University of Vienna,
Nordbergstrasse 15/C/3,
1090 Vienna, Austria
Email: pllana@par.univie.ac.at
Email: sigi@par.univie.ac.at
*Corresponding author

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: We present a novel approach for hybrid performance modelling and prediction of large-scale parallel and distributed computing systems, which combines Mathematical Modelling (MathMod) and Discrete-Event Simulation (DES). We use MathMod to develop parameterised performance models for components of the system. Thereafter, we use DES to describe the structure of the 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.

Keywords: performance modelling; parallel and distributed programs; simulation; performance prediction; MathMod; mathematical modelling; large-scale computing systems.


Biographical notes: Sabri Pllana is a Senior Research Scientist at the Department of Scientific Computing, University of Vienna. He holds his PhD degree (Dr.techn.) in computer science from the Vienna University of Technology. His current research interests include performance-oriented software engineering; high-level specification of parallel and distributed programs for multi-core, cluster and Grid computing systems; and modelling and simulation techniques. He is member of the IEEE.

Siegfried Benkner is the Head of the Department of Scientific Computing at the University of Vienna, Austria. He received his PhD degree in computer science from the Vienna University of Technology in 1994. He contributed to several EU projects (including PPPE, PREPARE, HPF+ and GEMSS) and was the Technical Director of the LTR project HPF+. His research interests include languages, compilers and runtime systems for parallel and distributed computing, programming environments, object-oriented software, component systems, web services and Grid computing. He is a member of the ACM and the IEEE.

Fatos Xhafa joined the Department of Languages and Informatics Systems of the Polytechnic University of Catalonia as an Assistant Professor in 1996 and is currently Associate Professor and member of the ALBCOM Research Group of this department. He received his PhD degree in
1 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 parallelisation 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 modelling 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 modelling and prediction of parallel and distributed computing systems, which combines Mathematical Modelling (MathMod) and Discrete-Event Simulation (DES). 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 modelling and prediction system called Performance Prophet. We demonstrate the usefulness of Performance Prophet by modelling 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 organised as follows. Section 2 introduces our approach. Our methodology for hybrid performance modelling and prediction of parallel and distributed computing systems is described in Section 3. We evaluate empirically our approach in Section 4. The related work is discussed in Section 5. Finally, Section 6 concludes the paper and briefly describes the future work.

2 Our approach

Our aim is to build high-level performance models of parallel and distributed computing systems. These performance models should offer comparative accuracy, since we usually use them to compare various parallelisation strategies of large scientific programs. The accuracy of model should be such that we can determine whether the version A performs better than the version B of program. The time spent for evaluation of these models should be short, in order to explore a large set of possible parallelisation strategies within a reasonable time. The performance model should be structure aware. This means that in the model should be identifiable components of the system and the relationship among these components. Moreover, to facilitate the task of model-based performance analysis it should be easy to modify the structure of model. The model-based performance analysis involves the modification of structure of the model to reflect system structural changes in order to predict what would be the performance of system under study if its structure is changed. The real-world scientific programs typically comprise a large number of code lines and their execution time is very long. The modelling methodology should be scalable and permit various levels of abstractions. The level of abstraction is adapted to the size of program that is modelled. The modelling of large programs is manageable if we increase the level of abstraction. The aim is to select such a level of abstraction that the model development and evaluation effort is manageable. Furthermore, the model evaluation should be much faster than the execution of program under study. For instance, if the execution of program under study takes several weeks, we do not want to spend for evaluation of the corresponding performance model several weeks as well. Therefore, we have tailored our approach to meet the following requirements: (1) efficiency of model building and evaluation, (2) structure awareness, (3) accuracy and (4) scalability (that is applicability to real-world computing systems).
Figure 1 A high-level description of the structure of a parallel and distributed computing system

Figure 1 depicts the structure of a parallel and distributed computing system at a level of abstraction that is suitable for the development of high-level performance models. We can identify two main components of computing system: workload and machine. The workload indicates a parallel and distributed computing architecture. The workload component comprises a set of processes. A process may comprise various types of elements: sequential, shared memory or distributed memory. The machine component comprises a set of nodes. Each node comprises a set of processors.

We reduce the modelling and evaluation effort by simplifying the model. 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. 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 computer architecture as well. For instance, we may group the components of processor and model the processor as a single element of the machine model (see machine component in Figure 1). By utilising grouping and neglecting techniques, we reduce the number of model elements. Subsequently, by reducing the number of model elements, the model development and evaluation effort is reduced.

Our approach combines MathMod and simulation for the performance assessment of computing systems. If the system under study is available, we use measurement experiments as a source of data for model building and validation. An alternative source of data is the literature, such as scientific publications that describe the properties of system under study. Our modelling procedure considers only those components of the system that strongly influence the overall system performance. We use MathMod to develop parameterised performance models for components of the system. Thereafter, we use DES to describe the structure of system and the interaction among its components. As a result, we obtain a high-level performance model that combines the evaluation speed of mathematical models with the structure awareness and fidelity of the simulation model.

3 Hybrid performance modelling

Commonly for performance modelling of computing systems is used MathMod or 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 (Kerbyson et al., 2005). An example of a mathematical performance model that models the program execution time is expressed as follows:

\[ T_{\text{ProgExec}} = C_{\text{Op}} T_{\text{av}} \]

where \( C_{\text{Op}} \) is the number of operations and \( T_{\text{av}} \) 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 (Pai et al., 1997; RSIM, 2008). 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 MathMod with DES is referred to as hybrid model (Schwetman, 1978).

Figure 2 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 modelling 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 modelled in more detail with the DES dominated hybrid models. Table 1 shows a summary of features of MathMod, hybrid and DES performance models of computing systems.

Figure 3 depicts the activity diagram of a hypothetical program. Activities \( \{ A_i \mid 1 \leq i \leq 4 \} \) correspond to the code blocks of program. To each activity \( A_i \) is associated a parameterised cost function \( F_{A_i}(\cdot) \), which models the execution time of activity \( A_i \). Functions \( \{ F_{A_i}(\cdot) \mid 1 \leq i \leq 4 \} \) are obtained using the MathMod techniques. The structure of program, which includes activities and their order of execution, is modelled with DES.

Figure 4 depicts our hybrid approach for performance modelling of a parallel region. The parallel region executes activities \( \text{Computation}_1 \) and \( \text{Computation}_2 \) in parallel (Figure 4a). Thread \( T_0 \) executes activity \( \text{Computation}_1 \), whereas activity \( \text{Computation}_2 \) is executed by thread \( T_1 \) (Figure 4b). The execution of activities \( \text{Computation}_1 \) and \( \text{Computation}_2 \) is modelled with MathMod. The synchronisation of threads is modelled with DES.
Figure 2  Hybrid models combine the features of MathMod models and DES models (MathMod stands for Mathematical Modelling; DES stands for Discrete-Event Simulation).

Figure 3  Hybrid performance model of a hypothetical program. The performance of activities is modelled with MathMod. The control flow is modelled with DES.

Figure 4  Hybrid performance modelling of a parallel region.

Figure 5  Hybrid performance modelling of point-to-point interprocess communication.
Figure 5 depicts our hybrid approach for performance modelling 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 modelled with an activity (Figure 5a). The sending process $P_0$ uses a nonblocking send to send the message (Figure 5a), whereas the receiving $P_1$ process uses a blocking receive to receive the message (Figure 5b). Computation and the message transfer are modelled with MathMod, whereas waiting to receive the message is simulated with DES (Figure 5c).

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 behaviour 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 (Paul, 1993). Therefore, we model the performance behaviour of action states with MathMod techniques, whereas we simulate the behaviour of wait states.

4 Evaluation
For the purpose of evaluation of our approach, we have developed a performance modelling and prediction system called Performance Prophet.

<table>
<thead>
<tr>
<th>Technique</th>
<th>Efficiency</th>
<th>Structure</th>
<th>Accuracy</th>
<th>Scalability</th>
</tr>
</thead>
<tbody>
<tr>
<td>MathMod</td>
<td>High</td>
<td>Not aware</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>Hybrid</td>
<td>Medium</td>
<td>Aware</td>
<td>Medium</td>
<td>Medium</td>
</tr>
<tr>
<td>DES</td>
<td>Low</td>
<td>Aware</td>
<td>High</td>
<td>Low</td>
</tr>
</tbody>
</table>

Figure 6 The architecture of Performance Prophet [abbreviations: Model Checking File (MCF), Configuration File (CF), Performance Model of Program (PMP), System Parameters (SP), Trace File (TF)]

4.1 Performance Prophet
Figure 6 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 modelling 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 Visualisation (Figure 6). The GUI of Teuta is used for the development of performance model based on the Unified Modelling Language (UML) (UML 2.0, 2005). 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 Visualisation components are used for visualisation of the performance results.

Element Model Checking File (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 Configuration File (CF).

The communication between Teuta and the Performance Estimator is done via elements Performance Model of Program (PMP), System Parameters (SP) and Trace File (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 SP. 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 visualisation of performance results.
The Performance Estimator comprises the following components: Simulation Manager, CSIM, Workload elements and Machine elements (Figure 6). 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 (Schwertman, 2001; Mesquite, 2008)] is a process-oriented general-purpose simulation library. CSIM supports the development of DES 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 synchronisation of processes and for the interprocess communication. For the synchronisation 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 7 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 7(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 processing UnitName is mainly used to identify the processing 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 [e.g. a code region enclosed within OpenMP directives PARALLEL and END PARALLEL (OpenMP, 2008)]. The attributes bufferSync and mbSync serve for the synchronisation 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 7(a). The methods init() and end() are invoked when the operation of process is initialised and completed, respectively. The method program() models the performance behaviour 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 (Pllana et al., 2008b).

![Figure 7](image-url)
Figure 7(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 behaviour 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 modelling elements. Each performance modelling element corresponds to a code block of the program, whose performance is modelled (Figure 8). The execution of a performance modelling element models the performance behaviour of a code block during the program execution.

Figure 8 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 modelling elements in the method program() of class Process (Figure 7).

![Figure 8](image) Performance modelling elements of program

![Figure 9](image) The structure of program modelling elements

Figure 9 illustrates the structure of modelling elements. All classes of Performance Estimator that model various code blocks of parallel and distributed programs are defined as subclasses of the class ModelElement. The structure of class ModelElement is depicted in Figure 9(a). The attribute id is used to uniquely identify instances of the class ModelElement. The relationship of instances of the class ModelElement with processing units is established with attributes myUID, myPID, myTID. Attributes elType and elName determine the type (e.g. ACTIONPLUS) and the name of instance (such as A1). The attribute tfName specifies the file, in which the performance evaluation result of modelling element is stored. The method checkPermission() verifies whether the element should be executed by the processing unit. The result of performance evaluation of modelling element is stored using the method output().

Figure 9(b) depicts the structure of class ActionPlus. Class ActionPlus is defined as a subclass of class ModelElement. Commonly, we use instances of class Action-Plus to model various types of single-entry single-exit code blocks. In Figure 9(b), we may observe that class Action-Plus defines no attributes (the corresponding compartment is empty), but it inherits all attributes of the class Mod-element. The method execute() of class ActionPlus models the performance behaviour of a code block. Other subclasses of class ModelElement (such as NBSend or BRecv) have similar structure like class ActionPlus. Each subclass of class ModelElement has one or more execute() methods. Methods execute() are distinguished by the number and the type of parameters. Each method execute() models the performance of a code block at specific level of abstraction.
Figure 10 depicts the class \textit{Node}, whose instances we use for modelling the computational nodes of computer architectures. The structure of class \textit{Node} is depicted in Figure 10(a). Attribute \textit{nid} is used to uniquely identify instances of the class \textit{Node}. The number of processors per node is specified with attribute \textit{numCPUs}. Attribute \textit{nodeName} is used to specify the name of the node. The CSIM type \textit{facilityms} is used to define processors of the node (that is \textit{cpus}). The method \textit{init()} of class \textit{Node} is invoked when operation of the node is initialised.

Figure 10(b) depicts the structure of a node. A node comprises a set of processors \{CPU\textsubscript{0}, CPU\textsubscript{1}, ..., CPU\textsubscript{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 \textit{first come first served}. Other queue disciplines, such as \textit{round robin}, may be specified. If the \textit{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.

4.2 Case study

In this section, we demonstrate the usefulness of Performance Prophet by modelling and simulating a real-world material science program. For our case study we use LAPW0, which is a part of WIEN2k package (Schwarz et al., 2002). 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. LAPW0 calculates the effective potential within a unit cell of a crystal. The code of LAPW0 program is written in Fortran 90 and MPI (Gropp et al., 1999). LAPW0 comprises about 15,000 lines of code.

LAPW0 is executed in SPMD fashion (all processors execute the same program) on a multiprocessor computing system. A domain decomposition approach is used for parallelisation of LAPW0 (Figure 11). 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 any given positive integer values of the number of atoms and the number of processes, 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.

Figure 12 depicts the architecture of Gescher cluster, which is located at Institute of Scientific Computing, University of Vienna. 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 700 MHz processors, and 2 GB ECC RAM. The nodes of Gescher are interconnected via a 100 Mbit/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.
Figure 12 Experimentation platform. Gescher cluster has 16 SMP nodes. Each node has four processors

![Figure 12](image1)

Figure 13 Performance modelling of LAPW0 (see online version for colours)

![Figure 13](image2)

Figure 13 illustrates the procedure for the development of performance model of LAPW0 with Performance Prophet. Due to space limitations, in Figure 13 it is depicted just a fragment of the UML model of LAPW0. We developed the model of LAPW0 by using the modelling elements that are available in the toolbar of Performance Prophet. Basically, Performance Prophet permits to associate to each modelling element a cost function. A cost function models the execution time of the code block that is represented by the performance modelling element. Figure 13 depicts the association of cost function $\text{CalcMPM}$ to action $\text{Calculate Multipolmoments}$. 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 14 depicts the visualisation 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 modelling element of the selected process. Results, which 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.

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 2  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)

<table>
<thead>
<tr>
<th>System</th>
<th>$T_s$ (s)</th>
<th>$T_m$ (s)</th>
<th>$T_e$ (s)</th>
<th>$T_m/T_e$</th>
<th>Error (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>N1P4</td>
<td>280</td>
<td>264</td>
<td>0.01</td>
<td>26,400</td>
<td>6</td>
</tr>
<tr>
<td>N2P8</td>
<td>170</td>
<td>166</td>
<td>0.02</td>
<td>8300</td>
<td>2</td>
</tr>
<tr>
<td>N4P16</td>
<td>126</td>
<td>131</td>
<td>0.04</td>
<td>3275</td>
<td>3</td>
</tr>
<tr>
<td>N8P32</td>
<td>98</td>
<td>113</td>
<td>0.08</td>
<td>1413</td>
<td>13</td>
</tr>
</tbody>
</table>

$NAT = 64$

<table>
<thead>
<tr>
<th>System</th>
<th>$T_s$ (s)</th>
<th>$T_m$ (s)</th>
<th>$T_e$ (s)</th>
<th>$T_m/T_e$</th>
<th>Error (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>N1P4</td>
<td>543</td>
<td>501</td>
<td>0.01</td>
<td>50,100</td>
<td>8</td>
</tr>
<tr>
<td>N2P8</td>
<td>314</td>
<td>264</td>
<td>0.02</td>
<td>14,700</td>
<td>7</td>
</tr>
<tr>
<td>N4P16</td>
<td>211</td>
<td>197</td>
<td>0.04</td>
<td>4925</td>
<td>7</td>
</tr>
<tr>
<td>N8P32</td>
<td>184</td>
<td>164</td>
<td>0.09</td>
<td>1822</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 2 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 650 MHz) 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 $T_m/T_e$. 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:

$$\text{Error}[^\%] = \left| \frac{T_s - T_m}{T_m} \right| \times 100$$

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 15.
5 Related work

Most of approaches for performance evaluation of computing systems (Pllana et al., 2007) 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 modelling that does not scale up to the size and complexity of the real-world programs (Kerbyson et al., 2005), (2) a detailed machine model is used that is so slow that makes impractical the simulation of real-world programs (Hughes et al., 2002; RSIM, 2008) or (3) for the model evaluation are required very large resources (processors and memory) that may not be available (Kvasnicka et al., 2001; Zheng et al., 2004; Wilmarth et al., 2005). Our approach has addressed this issue by using model simplification techniques, combination of MathMod with DES, 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 (Kerbyson et al., 2005). 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 modelling are used UML activity diagrams (Pllana and Fahringer, 2002). An activity may represent a single instruction, or larger blocks of the program (e.g. a loop), or the whole program. Furthermore, our approach uses the DES to describe the structure of system and the interaction among its components.

An approach for hybrid performance modelling of computer systems is described in Schwetman (1978). The aim of this approach is the improvement of the efficiency of model evaluation, by avoiding the simulation of events that occur frequently. Two kinds of components of the computing system are distinguished: long-term components (such as the main memory) and short-term components (such as the processor). It is assumed that during the program execution the short-term components are used much more frequently than the long-term components. Therefore, the use of long-term components is modelled with DES, whereas the use of short-term components is modelled with MathMod. Please note that this hybrid approach is used to model only the machine component of computing system, and is not used for modelling the workload component.

Our hybrid approach is used for modelling the whole computing system (that is the workload and the machine). We integrate the workload model with the machine model in order to build the model of computing system. Therefore, our approach permits the development of compact and efficient performance models. In our approach, the behaviour of whole computing system is split up into action states (e.g. system is performing an activity) and wait states (e.g. system is waiting until a set of conditions become true). The performance behaviour of action states is modelled with MathMod techniques, whereas the behaviour of wait states is simulated. Furthermore, we use DES to model the structure of computing system.

6 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 MathMod and DES. 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 modelling 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 modelling 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 multi-core computing systems (Pllana et al., 2008a).

References


Large-scale computing systems


