View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Crossref

Scientific Programming 10 (2002) 35–44
IOS Press

Dynamic performance tuning supported by program specification

Eduardo César[a], Anna Morajko[a], Tomàs Margalef[a], Joan Sorribes[a], Antonio Espinosa[b] and Emilio Luque[a]
[a]*Computer Science Department, Universitat Autònoma de Barcelona, 08193 Bellaterra, Barcelona, Spain*
Tel.: +34 93 5812888; Fax: +34 93 5812478;
E-mail: ania@aows10.uab.es, {eduardo.cesar, tomas.margalef, joan.sorribes, emilio.luque}@uab.es
[b]*Isoco (Intelligent Software Components)*
E-mail: Tonie@isoco.com

Abstract: Performance analysis and tuning of parallel/distributed applications are very difficult tasks for non-expert programmers. It is necessary to provide tools that automatically carry out these tasks. These can be static tools that carry out the analysis on a post-mortem phase or can tune the application on the fly. Both kind of tools have their target applications. Static automatic analysis tools are suitable for stable application while dynamic tuning tools are more appropriate to applications with dynamic behaviour. In this paper, we describe KappaPi as an example of a static automatic performance analysis tool, and also a general environment based on parallel patterns for developing and dynamically tuning parallel/distributed applications.

1. Introduction

The main goal of parallel and distributed computing is to obtain the highest performance in a due environment. Designers of parallel applications are responsible for providing the best possible behaviour on the target system. To reach this goal it is necessary to carry out a tuning process of the application through a performance analysis and the modification of critical application/system parameters. This tuning process implies the monitoring of application execution in order to collect the relevant related information, then the analysis of this information to find the performance bottlenecks and determination of the actions to be taken to eliminate these bottlenecks.

The classical way of carrying out this process has been to use a monitoring tool that collects the information generated during the execution and use a visualisation tool to present users with the information in a more comprehensive way that tries to help in the performance analysis [1–3]. These tools help users in the collection of information and the presentation, but obliges them to carry out the performance analysis on their own. Therefore, this process requires a high degree of expertise detecting the performance bottlenecks and, moreover, in relating them to the source code of the application or to the system components. To complete the tuning cycle, it is necessary to modify the application code or the system parameters in order to improve application performance. Consequently, the participation of users in the whole process is very significant.

Many tools have been designed and developed to support this approach. However, the requirements asked of users with respect to the degree of expertise and the time consumed in this process, have not facilitated widespread use of such tools in real applications.

To overcome these difficulties, it is very important to offer users a new generation of tools that guide them in the tuning process, avoiding the degree of expertise required by the visualisation tools. This new generation of tools must introduce certain automatic features that help users and guide them in the tuning process or even carry out certain steps automatically in such a way that user participation can be reduced or even avoided. In this sense, two approaches can be distinguished: the static and the dynamic.

ISSN 1058-9244/02/$8.00 © 2002 – IOS Press. All rights reserved

In the static approach, the objective is to analyse application performance and then modify the source code, recompile it and re-run the application. Usually, this approach is based on a post-mortem analysis performed on a trace file obtained during the execution of the application. On the other hand, the dynamic approach tries to tune the application during the execution without stopping, recompiling or even re-running the application. To accomplish this objective, it is necessary to use dynamic instrumentation techniques that allow the modification of the application code on the fly.

These two approaches might appear to be opposed, but can actually be considered as complementary, since they cover different application ranges and there are several techniques and methodologies that are common to them. Both have their advantages and disadvantages, depending on the features of the application. The static approach has the advantage that, when the applications have a regular and stable behaviour, they can be tuned and once the tuning process has been completed, the application can be executed as many times as necessary without introducing any [monitoring] intrusion during the application execution.

However, there are many applications that do not have such a stable behaviour and change from run to run according to the input data, or even change their behaviour during one single run due to the data evolution. In this situation, the dynamic approach allows for following the application behaviour on the fly. This requires a continuous intrusion into the program that is not necessary when application behaviour is stable. Moreover, if the analysis is carried out on the fly during the execution of the application, the information available and time spent on analysis is considerably restricted, due to the need to modify the application in this particular run.

In the following sections of this paper, new tools covering both approaches are presented. In Section 2, we describe an automatic performance analysis tool based on a static approach. Section 3 introduces the principles of a dynamic tuning tool supported by pattern-based design environment. Section 4 describes the pattern-based design environment; Section 5 introduces the dynamic instrumentation techniques required to carry out the tuning on the fly, and finally, Section 6 presents certain conclusion to this work.

2. A static automatic performance analysis tool: KappaPi

KappaPi (Knowledge based Automatic Parallel Program Analyser for Performance Improvement) [4] is a static automatic performance analysis tool that helps users in the performance improvement process by detecting the main performance bottlenecks, analysing the causes of those problems, relating the causes to the source code of the application and providing certain suggestions about the bottlenecks detected and the way of avoiding them. KappaPi was designed and developed at Computer Architecture and Operating System Group of the Universitat Autònoma de Barcelona.

This tool is based on a trace file post-mortem analysis and a knowledge base that includes the main bottlenecks found in message passing applications. The goal of KappaPi is to provide users with some certain hints that allow them to modify the application in order to improve performance.

2.1. KappaPi operation cycle

The first step is to execute the application with a monitoring tool in order to get the trace file that will be analysed by the KappaPi analyser. The trace file includes all the events occurred during the execution of the application, related to the communication actions undertaken by the different processes of the application. There are several tools that provide this kind of trace file, but for our purpose it is necessary that each event includes additional information that will be useful during the analysis phase. Besides the kind of event that has occurred (the source process, the destination process in a communication action and so on) it is very important that the monitoring tool inserts the time stamp of each event and the source code line that is responsible for that particular event during the execution of the application. TapePVM is a monitoring tool for PVM applications that includes these features, and VampirTrace for MPI applications also includes the required information.

Once the trace has been generated, KappaPi tool can be invoked. As a first step, KappaPi makes a general overview of application performance by measuring the efficiency of the different processors of the system. KappaPi considers as performance inefficiencies those intervals where processors are not doing any useful work; they are simply blocked, waiting for a message. So, the efficiency of a processor is considered as the percentage of time where it is doing useful work. When there are idle time intervals, these time intervals should be avoided in order to improve application performance. The best situation would be to have all the processors completely busy doing useful work during the execution of the application. In this first step, users get some information about the overall behaviour of the application, but have no idea about the bottlenecks and their causes.

After this initial classification, KappaPi starts the deep analysis by looking for performance bottlenecks. KappaPi takes chunks from the trace file and classifies the performance inefficiencies detected in that chunk. It must be pointed out that several inefficiencies can correspond to the same performance bottleneck, because, in many cases, the inefficiencies are repeated throughout the execution of the application. The detected bottlenecks are classified in a table according to the inefficiency time incurred. After analysing the first chunk, the second chunk is analysed and a new table is built and joint to the initial one in such a way that the new inefficiency time of the same bottleneck is added to the first one. The process is repeated for all the chunks and finally KappaPi provides a sorted table indicating the worst performance bottlenecks.

The next stage in the KappaPi analysis is the classification of the most important inefficiencies. For this purpose, it relates these inefficiencies with certain existing categories of behaviour using a rule-based knowledge system. From this point, inefficiencies are transformed into specific performance problems that must be studied in order to build up certain hints to for users.

To carry out this classification, KappaPi tool takes the trace file events as input and applies the set of rules deducing a list of facts. The deduced facts are kept in a list so that, in the next iteration of the algorithm, higher order rules apply to them. The process terminates when no more facts are deduced.

The query process finishes after the performance problem has been identified (when fitting in one of the categories of the rule-based system). The next step in the analysis is to take advantage of the problem-type information to carry out a deeper analysis that determines the causes of the performance bottleneck with the objective of building an explanation of this problem for users.

2.2. *KappaPi knowledge base*

Three main types of problems are differentiated [5]: communication related, synchronisation and program structure problems. This classification does not claim to be a complete taxonomy of the performance problems in message passing programs. It only reflects different types of scenarios that commonly appear when analysing the performance of message-passing applications. These reflect very different situations that require special care when trying to improve the performance of an application.

Given an execution interval of low efficiency, the description of a problem allows the possibility of engaging a searching process that looks for the existence of the problem in the interval of interest. Consequently, the ultimate objective of performance-problem descriptions is to automate the search process for problems in the performance data. As with any search process, it can be viewed as a two-step process of query formulation and the execution of this query in the performance data space.

The queries define the high level constructs of the application programming model. In this way, the system recognises a programming structure that is close to users with the subsequent objective of finding its performance limitations and suggesting possible improvements to users.

Therefore, we must build a language to express these queries and a system with which execute them. Additionally, the need for automation also requires the creation of the queries that will implement the search process. For this purpose, we have built a simple rule-based system that carries out a process of deduction using the trace file events and the deduced facts of the system.

Rules are divided into different levels. The deduction process applies all rules in the first level to the trace events until no further facts are deduced. Then, these recently deduced facts serve as input to the next level of rules and the deduction process applies again. This process will continue until the last level of rules is finished. In this way, higher order facts can be deduced from lower level events. For example, a couple of a send and a receive event can deduce a communication between two processes Building facts on others previously deduced allows the system to detect higher order execution situations. In principle, this system can allow the detection of any high level construction that decomposes afterwards in small, lower level operations.

Rules encapsulate special program execution configurations that commonly represent performance problems. These configurations range from the detailed low-level situations such as the behaviour of the communication receives of certain processes to certain global collaboration schemes of the application such as the master/worker. The actual classification used contains the following situations:

Communication problems

– Blocked Sender

Communication problem caused by two blocked linked receives. In this case, one process is blocked waiting for a message from a process, which is also blocked, waiting for a message from a third process. Rules defined for this problem are:

(blocked sender, process p1, process p2, process p3) is deduced when finding:
(receive at process p3 from process p2) &
(receive at process p2 from process p1) &
(send from process p1 to process p2)

– Multiple output
Communication problem caused by a serialisation of the output messages of a process. The rule needed to detect this problem is:

(multiple output, from process p1, to process p2, process p3, . . .) =
(receive at process p2 from process p1) &
(receive at process p3 from process p2) &
(send from process p1 to process p2) &
(send from process p1 to process p3)

Synchronisation problems

– Barrier synchronisation
Barrier waiting times create a delay in the execution of the application. The rule needed for detection is:

(barrier problem, process p1 blocked time xx, process p2 blocked time yy, . . .) =
(barrier call, process p1, blocked time)
(barrier call, process p2, blocked time)

Program structure problems

– Master/worker
Master/worker collaboration scheme generates idle intervals. The rules defined for this problem are:

(master/worker, p1 and p2) =
(dependence, p1, p2)&
(relationship, p1, p2)
(dependence, p1, p2) =
(communication, p1, p2) &
(blocked, p2, from p1)
(relationship, p1, p2) =
(communication, p1, p2) &
(communication, p2, p1)
(communication, p1, p2) =
(send, from p1, to p2) &
(receive, at p2, from p1)

– SPMD unbalance problems
Data partition between a group of processes derives in a loss of performance. Rules needed refer to the detection of a task link graph where all task are connected to each other (rel is equal to relationship):

(complete subgraph, p1, p2, p3, . . . , pn) =
(rel p1, p2) & (rel p1, p3) & . . . & (rel p1, pn)
& (complete subgraph, p2, p3, . . . , pn)
(rel p1, p2) =
(communication, p1, p2) &
(communication, p2, p1)

2.3. Building recommendations for users

The process of building a recommendation for users starts with the simpler objective of building an expression of the highest-level deduced fact that includes the situation found, the importance of such a problem and the program elements involved in the problem. In some cases, it is possible to evaluate the impact of a certain change in the program that created the problem. Only then will it be possible to calculate the impact of a different solution and build a suggestion for users.

The creation of this description strongly depends on the nature of the problem found, but in the majority of cases, there is a need to collect more specific information to complete the analysis. In these cases, it is necessary to access the source code of the application and to look for specific primitive sequence or data reference. Therefore, some specialised pieces of code (or “quick parsers”), which look for specific source information, must be called to complete the performance analysis description.

This last stage of the performance analysis can be thought of as an information gathering process. Its objective is to use the identification of the performance problems found in the analysis to build a description of these problems for users. This description represents the feedback that the tool is giving to users. Therefore, the given information includes a description of the performance problems found; the importance of the problem related to the global execution, and the program elements that are involved in the problem. Sometimes, this gathering can create a new, deeper analysis of the problem to describe the causes of its generation. In such cases, it is useful to look for specific details in the application or in the trace file under analysis.

3. Dynamic performance tuning supported by program specification

As was mentioned in the introduction, a different approach from static automatic performance analysis is that of dynamic performance tuning. This fits a set of applications that can behave in a different way for different executions. Such an approach would require neither developer intervention nor even access to the source code of the application. The running parallel application would be automatically monitored, analysed and tuned without the need to re-compile, re-link and restart.

Dynamic performance tuning of parallel applications is a task that must be carried out during application execution and, therefore, there are certain points to be considered:

- It is necessary to minimise the intrusion of the tool. Besides the classical monitoring intrusion, in dynamic performance tuning there are certain additional overheads due to monitor communication, performance analysis and program modifications.
- The analysis must be quite simple, because decisions must be taken in a short time to be effective in the execution of the program.
- The modifications must not involve a high degree of complexity, because it is not realistic to assume that any modification can be done on the fly.

For all these reasons, the analysis and modifications cannot be very complex. Since monitoring, evaluation and modification must be done in execution time, it is very difficult to carry this out without previous knowledge of the structure and functionality of the application. The programmer can develop any kind of program; hence, the generated bottlenecks can be extremely complicated. In such a situation, the analysis and the modifications might be extremely difficult. If knowledge about the application is not available, the applicability and effectiveness of our approach is significantly reduced. Therefore, an effective solution is to extract as much information from the application development framework as possible.

We therefore propose an environment that covers all of the aspects mentioned above. The environment consists of two main parts: an application development framework and a dynamic performance tuning tool. The first part provides the programmers with the design and development of their application. Users are constrained to use a set of programming patterns, but by using them, they skip the details related to the low level parallel programming. The main goal of the second part – the dynamic performance tuning tool – is to improve performance by modifying the program during its execution without recompiling and rerunning it. This task is achieved by monitoring the application, analysing the performance behaviour and finally tuning selected parts of the running program. The whole environment (the application design tool together with the dynamic performance-tuning tool) allows the programmer to concentrate on the application design without taking into account low level details and without having to worry about program performance.

When developers builds the application in our environment, they use the patterns provided by the framework. Hence, the kind of structures and paradigms that a developer has chosen is a known entity. On the other hand, patterns provided by the framework are well-known structures that may present certain well-known performance bottlenecks. We can therefore define information about the application that is capable of being used by the tuning environment. This information allows the tuning tool to know what must be monitored (measure points), what the performance model is and what can be changed to obtain better performance (tuning points). Using this knowledge, the dynamic performance tuning tool is simplified, because the set of performance bottlenecks to be analysed and tuned are only those related to the programming patterns offered to users.

3.1. Environment modules

Our pattern-based programming and dynamic performance tuning environment consists of several modules:

1. *Application framework* – this tool is based on common parallel patterns and offers support to users in developing their parallel application. The framework provides specific information to the dynamic performance tuning environment that can simplify the tuning of the pattern-based application on the fly.
2. *Monitor* – this tool collects events produced during the execution of the parallel application. To collect them, the monitor dynamically inserts instrumentation into the original program execution, taking into account all the running processes of the application. Generally, the instrumentation is specified by the framework (measure points), but it can also be specified interactively by a user

before program execution. If the analyser requires more or less information, it can notify the monitor to change the instrumentation dynamically during run-time.

3. *Performance Analyser* – this module is responsible for the automatic performance analysis of a parallel application "on the fly". During the execution, the analysis tool receives selected events that occur in the application's processes. Using received events and the knowledge given by the framework (performance model), the analyser detects the performance bottlenecks, determines the causes and decides what should be tuned in the application to improve performance. Detected problems and recommended solutions are also reported to users.
4. *Tuner* – this module automatically modifies a parallel application. It utilises solutions given by the analyser as well as information provided by the framework (tuning points). Tuner manipulates the running process and improves the program performance by inserting appropriate modifications. It has no need to access a source code or program restart.

Figure 1 shows how the described modules dynamically interact among themselves and with the application (in the run-time phase), also indicating the information that they obtain from the framework used in building the application (development phase).

On the one hand, our approach requires an application framework, which includes knowledge about the patterns and the behaviour of their implementation in the parallel application. On the other hand, to accomplish the goals of our dynamic tuning approach, we need to use a dynamic instrumentation technique. Only this technique allows the inclusion of certain new code in a running program without accessing the source code. The following sections describe two main parts of our environment in further detail. These are: the application framework and dynamic performance tuning supported by dynamic instrumentation.

4. Application framework

The solution to most concurrent problems could be obtained from the application of a finite set of design patterns [6,7]. Moreover it is possible to offer a finite set of pattern implementations (frameworks), which depend on the design pattern used and also on the implementation paradigm (message passing, shared memory).

We have focused our work on the frameworks devoted to message passing systems, with two main objectives in mind:

– Allow programmers to concentrate on codifying application-related issues, concealing low-level details of the communication library from them.
– Facilitate the dynamic performance tuning of the application, defining a performance model for each framework.

To obtain the first objective, we provide a library to offer users the possibility of developing an application based on parallel programming paradigms such as Master-Worker, Pipeline, SPMD, and Divide & Conquer.

Object-oriented programming techniques are the natural way for implementing patterns, not only due to their capacity for encapsulating behaviour, but also offering a well-defined interface to users. For these reasons, we have designed a class hierarchy in C++, which encapsulates the pattern behaviour, and also a class to encapsulate the communication library. In this sense, using our API the programmer simply has to fill in those methods related to the particular application being implemented, indicating the computation that each process has to perform, and the data that must be communicated with other processes.

A configuration tool complements this library, where users indicate the general structure of the application and the data structures that will be communicated by each process. The tool uses this configuration information to generate the adequate object structure and the communication classes.

To fulfil the second objective, it is necessary to develop a performance model for each framework that allows knowing what the ideal behaviour of the pattern should be, how far is the real behaviour from this ideal and how this ideal behaviour could be reached. Consequently, we have to analyse possible performance bottlenecks for each framework, which measures have to be taken to detect these bottlenecks, and what actions might be taken to overcome them.

The objective is to detect from outside the application that there is some (possibly undetermined) performance problem using very little information, and then try to isolate the specific problem by gathering new data from the application. The measures that have to be obtained to detect and isolate the performance problems are defined by the performance model of the

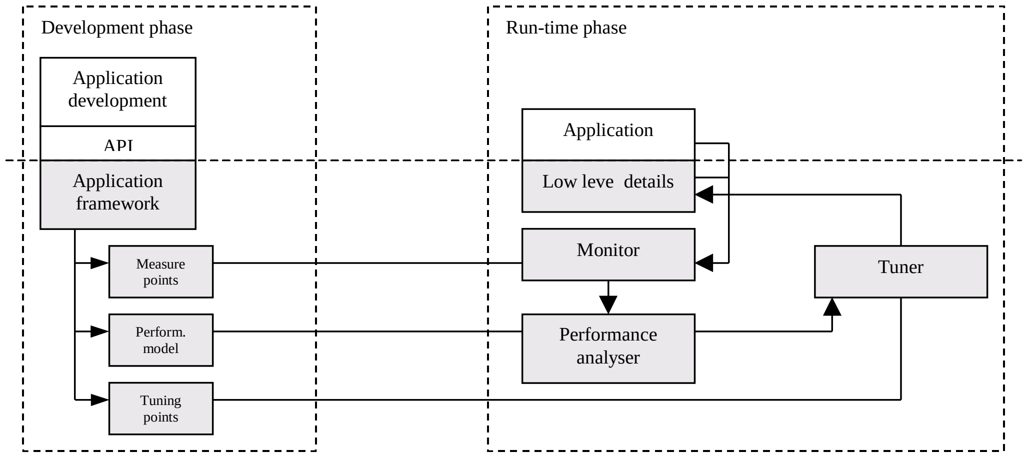


Fig. 1. Design of the dynamic tuning environment supported by program specification.

used framework, which in turn is used by the tuning tool to decide where, and when, to introduce corrective actions.

We have adopted a methodology that allows a unified approach to the definition, analysis and implementation of each framework, but which also defines a way to define new frameworks in the future (flexibility). The methodology includes:

– A general description of the framework.
– Establishing the elements to specify user-application in terms of the selected framework (interface). This includes initialisation and ending, functional description, and communication management.
– Characterising the associated framework bottlenecks.
– Determining the parameters needed to detect these bottlenecks (measure points).
– Determining the parameters that could be changed to overcome these bottlenecks and the actions that could be taken on them (tuning points).

The frameworks that have been included up to this point are the following:

– *Master-Worker:*

  ∗ *Description:* this framework consists of a master process that generates requests to other processes called workers. These workers make a computation on the requests and then send the results back to the master.
  ∗ *Interface:* how the master generates tasks, the actual computation that must be undertaken by each worker and the processing that the master must carry out on the received results.
  ∗ *Bottlenecks:* performance differences among workers, too few workers, too many workers, computational differences in requests processing (due, for example, to the task granularity).
  ∗ *Measure points:* communications times, workers computation times.
  ∗ *Tuning points:* task distribution (which includes message size, i.e., the number of tasks sent at one time to a worker), and worker numbers.

– *Pipeline:*

  ∗ *Description:* this pattern represents those algorithms that can be divided in an ordered chain of processes, where the output of a process is forwarded to the input of the next process in the chain.
  ∗ *Interface:* work that must be carried out at each stage of the pipe, input and the output data and connection among stages.
  ∗ *Bottlenecks:* significant performance differences among stages, bad communication/ computation ratio.
  ∗ *Measure points:* computing the time and data load for each stage, stage waiting time.
  ∗ *Tuning points:* the number of consecutive stages per node, the number of parallel instances of a stage.

– *SPMD* (*Single Program Multiple Data*)*:*

  ∗ *Description:* this represents those algorithms where the same processing is applied on different data portions, with some communication patterns among processing elements.
  ∗ *Interface:* this specifies the task that must be carried out for all the processes, including the

data communication (send-receive) pattern and protocol (all-to-all, 2D mesh, 2D torus, 3Dcube, and so on).
  - *Bottlenecks:* performance differences among processes.
  - *Measure points:* computing time and data load of each process, waiting time for other processes.
  - *Tuning points:* number of intercommunicating processes per node, number of instances of a process, and, in certain cases, data distribution.

– *Divide and Conquer:*

  - *Description:* each node receives some data and decides to process it or to create certain new processes with the same code and distribute the received data among them. The results generated at each level are gathered to the upper level. Each process receives partial results, carries out some computation based on them and passes the result to the upper level.
  - *Interface:* processing of each node, the amount of data to be distributed and the initial configuration of nodes.
  - *Measure points:* completion time for each branch, computation time for each process, branch depth.
  - *Tuning points:* number of branches generated in each division, data distribution among branches, and branch depth.

5. Dynamic performance tuning by dynamic instrumentation

The main goal of our work is to provide an environment that automatically improves the performance of parallel programs during run-time. To be able to achieve this objective, we make use of a special dynamic instrumentation technique. The implementation of the technique is provided by a library called DynInst, which is presented in the first subsection. The second subsection describes further details on our dynamic tuning tool that is based on DynInst and the application framework.

5.1. Dynamic instrumentation: DynInst

The principle of dynamic instrumentation is to defer program instrumentation until it is in execution and insert, alter and delete this instrumentation dynamically during program execution. This approach was first used in the Paradyn tool developed at the University of Wisconsin and University of Maryland. In order to build an efficient automatic analysis tool, the Paradyn group developed a special API that supports dynamic instrumentation. The result of their work was called DynInst API [8].

DynInst is an API for runtime code patching. It provides a C++ class library for machine independent program instrumentation during application execution. DynInst API allows attaching to an already-running process or starting a new process, creating a new piece of code and finally inserting created code into the running process. The next time the instrumented program executes the block of code that has been modified, the new code is executed. Moreover, the program being modified is able to continue its execution and does not need to be re-compiled, re-linked, or restarted. DynInst manipulates the address-space image of the running program and, thus, this library only needs to access a running program, not its source code. However, DynInst requires an instrumented program to contain debug information.

The process to be instrumented is simply called application or mutatee. A separate process that modifies an application process via DynInst is called mutator. The DynInst API is based on the following abstractions:

– *point* – a location in a program where new code can be inserted, i.e. function entry, function exit.
– *snippet* – a representation of a piece of executable code to be inserted into a program at a given point; a snippet must be built as an AST (Abstract Syntax Tree). It can include conditionals, function calls, loops, etc.
– *thread* – a thread of execution (this means process or a lightweight-thread).
– *image* – refers to the static representation of a program on disk. Each thread is associated with exactly one image.

Taking into account the possibilities offered by the DynInst library, it is possible to insert code into the running application. Our dynamic tuning tool uses this library for two main objectives:

– Insert code for monitoring purposes to collect information on the behaviour of the application. The module supporting this function will be called the "monitor".

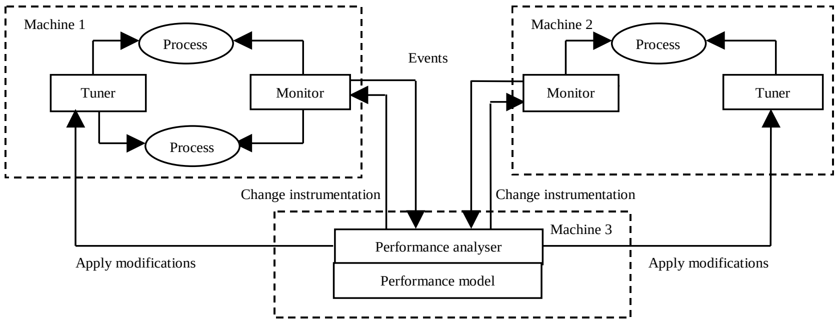


Fig. 2. Dynamic tuning system design.

– Insert code for performance tuning. The main goal of dynamic tuning is to improve the performance on the fly. Therefore, it is necessary to change the code of the application. The module supporting this function will be called the “tuner”.

5.2. *Dynamic performance tuning architecture*

The current version of our dynamic tuning tool is implemented in C++ language and is dedicated to PVM-based applications. However, the system architecture is open and could be easily extended to support applications that use other message passing communication libraries. In fact, the communication library details are hidden in our approach, since the low-level code is generated automatically by the application framework.

In general, the parallel application environment usually collects several computers. A parallel application consists of several intercommunicating processes that solve a common problem. Processes are mapped on a set of computers and hence each process may be physically executed on a different machine. This situation means that it is not enough to improve processes separately without considering the global application view. To improve the performance of the entire application, we need to access global information about all processes on all machines. And to obtain this objective, we need to distribute the modules of our dynamic tuning tool (monitors and tuners) to all machines where application processes are running.

In Fig. 2, we present a scheme for the dynamic tuning tool indicating the distribution of the modules. The figure also illustrates the interactions between all the modules of the dynamic tuning system that we describe in the following paragraphs in more detail.

To collect events that happen during the execution of each process, the monitor makes use of the DynInst library and dynamically inserts the instrumentation into the original process execution. Using information given by the framework, this module can instrument each process at points that are highly specific for the monitored application (it knows the points where a bottleneck can occur) thus minimising intrusiveness. From the implementation point of view, we insert a piece of monitoring code (snippet) into the running program at all points that are needed to discover performance problems. Such a snippet logs events that happen during program execution. The logging snippet may be inserted at arbitrary points defined by the used patterns, for instance at the entry and/or exit of pvm_send and pvm_recv functions if communication is a potential bottleneck. When the function is executed, the snippet code logs timestamp, all function parameters and execution time, and sends them as events to the analyser.

The monitor distribution indicates that events from different tasks are collected on different machines. However, our approach to dynamic analysis requires the global and ordered set of events, and thus we have to send all events to a central location. The analyser module can reside on a dedicated machine collecting events from all distributed monitors. For faster and easier analysis, this module also uses information given by the framework – especially the performance model that is the consequence of the pattern or patterns chosen to develop an application. Although this module has such knowledge from the framework, the analysis is still assumed to be time-consuming. It can significantly increase application execution time if both – the analyser and the application – are running on the same machine. In order to reduce intrusion, the analysis should be executed on the dedicated and distinct machine (the performance “optimiser” machine).

The analysis must be carried out globally by taking the behaviour of the entire application into consideration. The collected events are used to detect potential problems. Obviously, during the analyser computation, monitor modules can still trace the application. In certain situations, the analyser may need more information about program execution to detect a problem or determine the action to be taken. Therefore, it can request the monitor to change the instrumentation dynamically in order to provide more detailed information about specific program behaviour. Consequently, the monitor must be able to modify program instrumentation – add more or remove whatever is redundant – depending on the needs of the performance analysis. To detect problems, find their causes and provide a solution, we take advantage of the knowledge and experience gained from work undertaken on the KAPPA-PI tool.

The last module – tuner – receives the decision from analyser and automatically modifies the application during run-time using DynInst library. It is based on knowledge of mapping problem solutions to code changes (tuning points). Therefore, when a problem has been detected and the solution has been given, tuner must find appropriate modifications and apply them dynamically into the running process. Here our framework is also very useful, because it provides information about parameters that can be changed and actions that can be taken to overcome the bottleneck. Therefore, tuner knows what it must modify in order to improve performance. The inclusion of some new code into a process must be done during run time without recompiling and re-running it. Applying modifications requires access to the appropriate process; hence tuner must be distributed on different machines.

For example, when an application is based on master/worker pattern, the parameter, which is important for a good performance, is the number of workers. Therefore, if there are insufficient workers doing the work, an application might need far more time to finish. The analyser discovers this problem and recommends increasing the number of workers. The tuner receives this information, and by using the knowledge provided by the application framework, finds out which parameter in the application represents this number. To change the code, it finds the variable in running process via DynInst, and modifies the value. The rest of the work is carried out by the framework runtime. The framework detects the change of the variable and adjusts the number of workers accordingly. In our example, the next time that the application distributes the data, there will be more workers to do the work.

6. Conclusions

We have presented two kinds of tools for automatic performance analysis. KappaPi is a knowledge-based static automatic performance analysis tool that analyses trace file looking for bottlenecks and provides certain hints to users. Users can take advantage of these hints in order to modify the application to improve performance. The second approach to parallel/distributed performance analysis and tuning includes a pattern-based application design tool and a dynamic performance tuning tool. The sets of patterns included in the pattern-based application design tool have been selected to cover a wide range of applications. They offer well-defined behaviour, and the bottlenecks that can occur are also very well determined. In this sense, the both analysis of the application and performance tuning on the fly can be carried out successfully. Using this environment, the programmers can design its application in a fairly simple way, and then have no need to concern themselves about any performance analysis or tuning, as dynamic performance tuning automatically takes care of these tasks.

Acknowledgement

This work was supported by the Comision Interministerial de Ciencia y Tecnología (CICYT) under contract number TIC 98-0433.

References

[1] D.A. Reed, P.C. Roth, R.A. Aydt, K.A. Shields, L.F. Tavera, R.J. Noe and B.W. Schwartz, *Scalable Performance Analysis: The Pablo Performance Analysis Environment,* Proceeding of Scalable Parallel Libraries Conference, IEEE Computer Society, 1993, pp. 104–113.

[2] W. Nagel, A. Arnold, M. Weber and H. Hoppe: VAMPIR: Visualization and Analysis of MPI Resources, *Supercomputer* **1** (1996), 69–80.

[3] Y.C. Yan and S.R. Sarukhai, Analyzing parallel program performance using normalized performance indices and trace transformation techniques, *Parallel Computing* **22** (1996), 1215–1237.

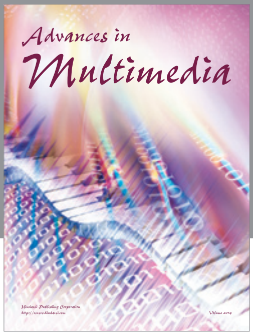
[4] A. Espinosa, T. Margalef and E. Luque, Integrating Automatic Techniques in a Performance Analysis Session, *Lecture Notes in Computer Science*, (Vol. 1900), (EuroPar 2000), Springer-Verlag, 2000, pp. 173–177.

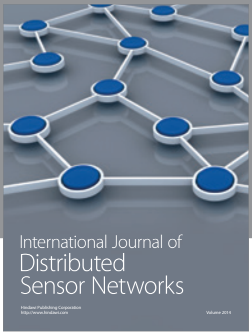
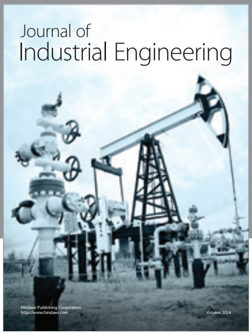
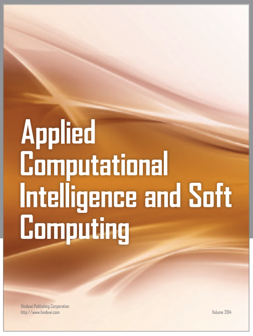
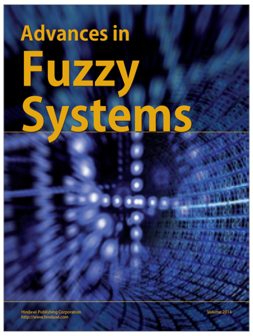
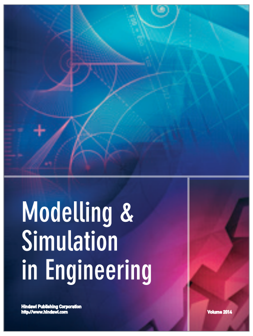
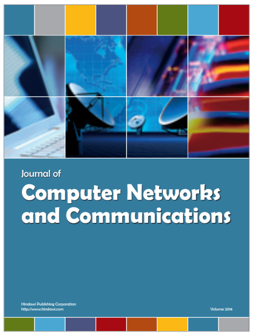
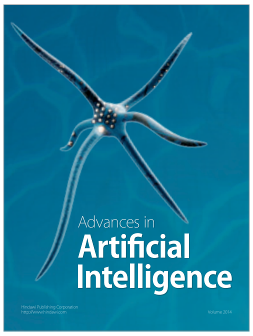
[5] A. Espinosa, *Automatic performance analysis of parallel programs,* PhD thesis. Universitat Autònoma de Barcelona, September 2000.

[6] J. Schaffer, D. Szafron, G. Lobe and I. Parsons, The Interprise model for developing distributed applications, *IEEE Parallel and Distributed Technology* **1**(3) (1993), 85–96.

[7] J.C. Browne, S. Hyder, J. Dongarra, K. Moore and P. Newton, Visual Programming and Debugging for parallel computing, *IEEE Parallel and Distributed Technology* **3**(1) (1995), 75–83.

[8] J.K. Hollingsworth and B. Buck, Paradyn Parallel Performance Tools, DynInstAPI Programmer's Guide, Release 2.0, University of Maryland, Computer Science Department, April 2000.



Hindawi

Submit your manuscripts at
http://www.hindawi.com