FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik
D-52425 Jülich, Tel. (02461) 61-6402

Interner Bericht

Automatic Performance Analysis for CRAY T3E

Michael Gerndt, Bernd Mohr, Mario Pantano, Felix Wolf*

FZJ-ZAM-IB-9808

Mai 1998

(letzte Änderung: 26.05.98)

(*) University of Vienna
Institute for Software Technology and Parallel Systems
Liechtensteinstraße 22
A-1090 Vienna, Austria

Preprint: Proceedings of the Seventh Workshop on Compilers for Parallel Computers (CPC'98), University of Linköping, Sweden, June 29 - July 1 1998, pp. 69-78

Automatic Performance Analysis for CRAY T3E

M. Gerndt[1], B. Mohr[1], M. Pantano[2], F. Wolf[1]

[1]Research Centre Juelich (FZJ)
Central Institute for Applied Mathematics
52425 Jülich, Germany

{m.gerndt, b.mohr, f.wolf}@fz-juelich.de

[2]University of Vienna
Institute for Software Technology and Parallel Systems
Liechtensteinstrasse 22
A-1090 Vienna, Austria
pantano@par.univie.ac.at

Abstract

One of the reasons why parallel programming is considered to be a difficult task is that users frequently cannot predict the performance impact of implementation decisions prior to program execution. This results in a cycle of incremental performance improvements based on runtime performance data. While gathering and analyzing performance data is supported by a large number of tools, typically interactive, the task of performance analysis is still too complex for users. This article illustrates this fact based on the current analysis support on CRAY T3E. As a consequence, we are convinced that automatic analysis tools are required to identify frequently occuring and well-defined performance problems automatically. This article describes the novel design of a generic automatic performance analysis environment called KOJAK. Besides its structure we also outline the first component, EARL, a new meta-tool designed and implemented as a programmable interface to calculate more abstract metrics from existing trace files, and to locate complex patterns describing performance problems.

1 Introduction

The development of efficient parallel programs solving scientific and industrial challenges is a cyclic process of distinct phases: writing source code, executing it on the target platform, evaluating the performance of the program, and improving the program by manual transformation.

Quite elaborate tools are available assisting the application programmer in both steps of the performance analysis phase: program instrumentation and performance data inspection. Current techniques for program instrumentation cover: instrumented libraries, hardware performance monitors, source code instrumentation, object code instrumentation, and on-the-fly dynamic instrumentation. Performance data inspection is supported via interactive tools offering graphical and textual displays: tables of profiling information, time lines of event traces, animated diagrams, and statistical analyzes.

Current tools support application programmers via highly sophisticated user interfaces but lack knowledge-based user guidance. The application programmers have to be well trained. They do not only have to be experts for their applications and for potential performance bottlenecks but they have to understand the intricate details of the performance analysis environment also. This requires a time consuming learning effort for handling details such as reducing the amount of trace data and handling the powerful user interfaces. In addition, the application of these tools to real programs is a time-consuming task as well, and often, the revealed performance bottlenecks belong to a small number of well-defined performance problems, such as load balancing and excessive message passing overhead.

In summary, the current situation is dominated by an imbalance between the overhead in applying current performance analysis tools, learning overhead as well as case-based overhead, and the frequently revealed typical performance bottlenecks that seem to be easily detectable. This imbalance is one of the most important reasons why the user community does still not accept current analysis tools and, due to this reluctance, parallel computers do frequently not deliver high performance.

To improve the current situation, we designed KOJAK (Kit for Objective Judgement and Automatic Knowledge-based detection of bottlenecks), a generic environment for automatic performance analysis. The goal of KOJAK is to automatically reveal those well-defined typical bottlenecks; It is not intended to automatically detect all bottlenecks that might exist in parallel programs.

This article motivates the design based on the current performance analysis tool support for MPI programs on CRAY T3E. In this environment, application programmers can use three interactive performance analysis tools: Apprentice, PAT, and VAMPIR. All three tools provide partially overlapping performance information. In addition to these tools, the CRAY compilers provide static program information in the *Compiler Information File (CIF)*. The CIF contains mapping information to relate the optimized object code back to the source code. This information enables Apprentice to present performance data in relation to the original code.

On CRAY T3E, KOJAK will be based on interfaces to these information sources. It will take over all those tasks that currently have to be done by the programmer:

- guide program instrumentation
- execute the program
- manage performance data of multiple program runs
- evaluate performance data

The remainder of the paper motivates and introduces the design of KOJAK. It is presented in the context of CRAY T3E but KOJAK is designed to be a generic tool which can be specialized for multiple targt machines and programming environments. Section 2 summarizes related work. Section 3 overviews the available performance analysis tools on CRAY T3E. Section 4 describes a first component of KOJAK, a programmable trace analyzer, which will be used to determine performance metrics that are not directly included in the trace files. The last section introduces the design concepts of KOJAK in the context of the T3E environment.

2 Related work

Automatic performance analysis of parallel programs has been investigated by a small number of projects. The most well known one is Paradyn developed at the University of Wisconsin-Madison by Jeff Hollingsworth et.al. [10]. Paradyn is based on dynamic instrumentation, i.e., the executable image is modified at runtime according to instrumentation requests. Paradyn supports automatic analysis via its Performance Consultant for PVM-based message passing programs. It is based on the W3 search model, i.e., searching through the three-dimensional space along the Why, Where, and When axes. It introduced the approach of structuring the search into a cycle of selecting hypotheses, instrumenting the code, evaluating the performance data, and refining the hypotheses according to a predefined hierarchy of performance bottlenecks. Paradyn is based on a dynamic instrumentation which is not available on CRAY T3E. Paradyn cannot be used as a post-execution analyzer which is desirable if the parallel machine is a limited resource. In addition, there is no interface to already existing monitoring tools and thus Paradyn cannot take performance information of other sources into account.

KAPPA-PI is an automatic performance analyzer for PVM-programs developed at the Universitat Autonoma de Barcelona [5]. It is a post-execution tool implemented in PERL that evaluates traces generated by the Tape/PVM monitoring library. Based on a predefined list of performance bottlenecks it searches for performance problems and their causes. In addition to trace data, it analyzes the source code based on pattern matching. The early version seems to be limited and not easily adaptable to other environments.

One additional design, POIROT, was published by Robert Helm and Allen Malony [8]. The design is based on the concept of heuristic classification. The main properties of a program run are extracted from trace data by a process called abstraction. These properties are matched against a database of possible performance bottlenecks. The selected bottleneck is refined to fit additional properties of the program run. Performance data are gathered via an environment interface that makes Poirot independent of intricate details of the programming environment, e.g., how to instrument a program. Poirot has never been demonstrated and no further publications exist.

3 Information sources

The programming environment of the CRAY T3E supports performance analysis via interactive tools. CRAY itself provides two tools, Apprentice and PAT, which are both based on summary information. In addition, Apprentice accesses source code information to map the performance information to the statements of the original source code. Besides these two tools, programmers can use VAMPIR, a trace analysis tool developed at our institute. Within a collaboration with CRAY, instrumentation and trace generation for VAMPIR is being integrated into the next version of PAT.

3.1 Compiler Information File

The F90 and C compilers on CRAY T3E generate on request for each source file a *compiler information file (CIF)*. This file includes information about the compilation process (applied compiler options, target machine characteristics, and compiler messages) and source information of the compilation units (procedure information, symbol information, information about loops and statement types, cross-reference information for each symbol in the source file).

Apprentice requires this information to link the performance information back to the source code. CIFs are initially in ASCII format but can be converted to binary format. The information can be easily accessed in both formats via a library interface.

3.2 Apprentice

Apprentice is a post-execution performance analysis tool for message passing programs [4]. Originally it was designed to support the CRAFT programming model on the CRAY T3E predecessor system, the CRAY T3D. Apprentice analyzes summary information collected at runtime via an instrumentation of the source program.

The instrumentation is performed by the compiler and is triggered via an appropriate compiler switch. To reduce the overhead of the instrumentation, the programmer can selectively compile the source files with and without instrumentation. The instrumentation is done in a late phase of the compilation after all optimizations already occured. This prevents that instrumentation affects the way code is compiled. During runtime, summary information is collected at each processor for each basic block. This information comprises:

- execution time
- number of floating point, integer, and load/store operations
- instrumentation overhead

For each subroutine call the execution time as well as the pass count is determined. At the end of a program run, the information of all processors is summed up and written to the *runtime information file (RIF)*. In addition to the summed up execution times and pass counts of subroutines calls, their mean value and standard deviation, as well as the minimum and maximum values are stored.

The size of the resulting RIF is typically less than one megabyte. But the overhead due to the instrumentation can easily be a factor of two which results from instrumenting every basic block. This severe drawback of the instrumentation is partly compensated in Apprentice by correcting the timings based on the measured overhead.

When the user starts Apprentice to analyze the collected information, the tool first reads the RIF as well as the CIFs of the individual source files. The performance data measured for the optimized code are related back to the original source code. Apprentice distinguishes between:

- parallel work: user-level subroutines
- I/O: system subroutines for performing I/O
- communication overhead: MPI and PVM routines, SHMEM routines
- uninstrumented code

The available barcharts allow the user to identify critical code regions that take most of the execution time or with a lot of I/O and communication overhead. Since all the values have been summed up, no specific behaviour of the processors can be identified. Load balance problems can be detected by inspecting the execution times of calls to synchronization subroutines, such as global sums or barriers. Based on the available information, the processors with the least and the highest execution time can be identified.

While Apprentice does not evaluate the hardware performance counters of the DEC Alpha, it estimates the loss due to cache misses and suboptimal use of the functional units. Based on the number of instructions and a very simple cost model (fixed cycles for each type of instruction) it determines the loss as the difference between the estimated optimal and the measured execution time.

3.3 VAMPIR

VAMPIR (Visualization and Analysis of MPI Resources) is an event trace analysis tool [11] which was developed by the Central Institute for Applied Mathematics of the Research Centre Jülich and now is commercially distributed by a German company named PALLAS. Its main application area is the analysis of parallel programs based on the message passing paradigm but it also has been successfully used for other areas (e.g., for SVM-Fortran traces to analyze shared virtual memory page transfer behaviour [7] or to analyze CRAY T3E usage based on accounting data). VAMPIR has three components:

- The VAMPIR tool itself is a graphical event trace browser implemented for the X11 Window system using the Motif toolkit. It is available for any major UNIX platform.
- The VAMPIR runtime library provides an API for collecting, buffering, and generating event traces as well as a set of wrapper routines for the most commonly used MPI and PVM communication routines which record message traffic in the event trace.
- In order to observe functions or subroutines in the user program, their entry and exit has to be instrumented by inserting calls to the VAMPIR runtime library. Observing message passing functions is handled by linking the program with the VAMPIR wrapper function library.

  VAMPIR comes with a source instrumenter for ANSI Fortran 77. Programs written in other programming languages (e.g., C or C++) have to be instrumented manually. To improve this situation, our institute in collaboration with CRAY Research is currently implementing an object code instrumenter for CRAY T3E. This is described in the next section.

During the execution of the instrumented user program, the VAMPIR runtime library records entry and exits to instrumented user and message passing functions and the sending and receiving of messages. For each message, its tag, communicator, and length is recorded. Through the use of a configuration file, it is possible to switch the runtime observation of specific functions on and off. This way, the program doesn't have to be re-instrumented and re-compiled for every change in the instrumentation.

Large parallel programs consist of several dozens or even hundreds of functions. To ease the analysis of such complex programs, VAMPIR arranges the functions into groups, e.g., user functions, MPI routines, I/O routines, and so on. The user can control/change the assignment of functions to groups and can also define new groups.

VAMPIR provides a wide variety of graphical displays to analyze the recorded event traces:

- The dynamic behaviour of the program can be analyzed by timeline diagrams for either the whole program or a selected set of nodes. By default, the displays show the whole event trace, but the user can zoom-in to any arbitrary region of the trace. Also, the user can change the display style of the lines representing messages based on their tag/communicator or the length. This way, message traffic of different modules or libraries can easily be visually separated.

- The parallelism display shows the number of nodes in each function group over time. This allows to easily locate specific parts of the program, e.g., parts with heavy message traffic or I/O.

- VAMPIR also provides a large number of statistical displays. It calculates how often each function or group of functions got called and the time spent in there. Message statistics show the number of messages sent, and the minimum, maximum, sum, and average length or transfer rate between any two nodes. The statistics can be displayed as bar-charts, histograms, or textual tables.

  A very useful feature of VAMPIR is that the statistic displays can be linked to the timeline diagrams. By this, statistics can be calculated for any arbitrary, user selectable part of the program execution.

- If the instrumenter/runtime library provides the necessary information in the event trace header, the information provided by VAMPIR can be related back to source code. VAMPIR provides a source code and a call graph display to show selected functions or the location of the send and the receive of a selected message.

In summary, VAMPIR is a very powerful and highly configurable event trace browser. It displays trace files in a variety of graphical views, and provides flexible filter and statistical operations that condense the displayed information to a manageable amount. Rapid zooming and instantaneous redraw allow to identify and focus on the time interval of interest.

However, because of its power and complexity, VAMPIR is not easy to use, especially for application programmers. Also, with very large traces, a user looking for problems/bottlenecks would have to look through the displays and zoom in and out for a long time. The programmer would never know whether he missed something because he didn't look carefully enough or zoomed in at the wrong places. Clearly, a more "automatic" way of analyzing large traces is needed.

3.4 PAT

PAT (Performance Analysis Tool) is the second performance tool available from CRAY Research for CRAY T3E. The two main differences to Apprentice are that no source code instrumentation or special compiler support is necessary. The user only needs to re-link his/her application against the PAT runtime library (because CRAY Unicos doesn't support dynamic linking). Second, PAT aims at keeping the additional overhead to measure/observe program behaviour as low as possible. PAT is actually three performance tools in one:

1. PAT allows the user to get an rough overview about the performance of the parallel program through a method called sampling, i.e., interrupting the program at regular intervals and evaluating the program counter. PAT can calculate then the percentage of time spent in each function. The sampling rate can be changed by the user to adapt it to the execution time of the program and to keep overhead low. Because the sampling method provides only a statistical estimate of the actual time spent in a function, the tool also provides a measure of confidence in the sampling estimate.

   In addition PAT determines the total, user, and system time of the execution run and the number of cache misses and the number of either flointing point, integer, store, or load operations. These are measured through the DEC Alpha hardware counters. The user can select the hardware counter by setting an environment variable. All this statistical information is stored after the execution in a so-called *Performance Information File* (PIF).

2. If a more detailed analysis is necessary, PAT can be used to instrument and analyze a specific function or set of functions in a second phase. PAT can instrument object code (however only on the function level). This is a big advantage especially for large complex programs because they do not have to be re-compiled for instrumentation. In addition, it is possible to analyze functions contained in system or 3rd-party libraries. A third advantage is that programs written in more than one language can be handled. The big disadvantage is that it is more difficult to relate the results back to the source code.

   This detailed investigation of function behaviour is called *Call Site Report* by PAT. It records for each call site of the instrumented functions how often it got called and time spent in this instantiation of the function. Execution times are measured with a high-resolution timer. The results are available for each CPU used in the parallel program. The next version of PAT will allow to gather hardware counter statistics for instrumented functions as well.

3. Last, if a very detailed analysis of the program behaviour is necessary, PAT also supports event tracing. The object instrumenter of PAT can also be used to insert calls to entry and exit trace routines around calls to user or library routines. Entry and exit trace routines can be provided in two ways:

   - The user can supply function-specific wrapper functions. The routines must be written in C, they must have the same number and same types of arguments as the routine they are tracing, and finally, the wrapper function name for a function *func* must be *func*_trace_entry for entry trace routines and *func*_trace_exit for exit trace routines.
   - If specific wrapper routines for the requested function are not available, PAT uses generic wrapper code which just records the entry and exit of the function in the event trace.

   In addition, PAT provides extra tracing runtime system calls, which can be inserted in the source code and allow to switch tracing on and off, and to insert additional information into the trace (e.g., information unrelated to functions).

   The tracing features of PAT were developed in a collaboration of Research Centre Jülich with CRAY Research. Our institute implemented all the necessary special wrapper functions for all message passing functions available on the T3E (MPI, PVM, and SHMEM and for both the C and Fortran interfaces) which record the message traffic in the event trace. In addition, we implemented a tool for converting the event traces contained in PIF files into VAMPIR trace format.

The major drawback of PAT's object instrumentation is the very low-level interface for specifying the functions to be instrumented. The user has to specify the function names as they appear in the object code, i.e., C++ functions or F90 functions which are local or contained in modules have to be specified in the mangled form (e.g., "__0FDfooPd" instead of the C++ function name "int foo(double *)"). Clearly, a more user friendly or automatic way for the instrumenter interface needs to be added to PAT.

In addition, the combination of three different instrumentation/analysis techniques into a single tool is very confusing for users. This confusion is further increased since the supported techniques overlap with the techniques applied in the other tools.

3.5 Summary

The previous subsections pointed out that the CRAY T3E has a programming environment that includes the most advanced performance analysis tools. On the other hand, each of these tools comes with its own instrumentation, provides partially overlapping information, and has a totally different user interface. The programmer has to understand the advantages and disadvantages of all the tools to be able to select and apply the right ones. Table 1 summarizes the main features of these three tools.

4 EARL: Postprocessing trace data

As a first step to automate trace analysis, we designed and implemented a new meta-tool named EARL (Event Analysis and Recognition Language). EARL is actually a new high-level trace analysis language which allows to easily construct new trace analysis tools by writing scripts in the EARL language. These are then executed by the EARL interpreter. It is intented as a programmable interface to calculate more abstract metrics, and to locate complex patterns describing performance problems.

Although EARL is designed to be a generic event trace analysis tool, the current prototype concentrates on the analysis of event traces generated from message passing programs. This is not really a restriction as most uses of event tracing are in the field of parallel programming on distributed memory machines (which almost all use a one or two sided message passing scheme for communication). In addition, analysis of message passing traces is well understood and

	Apprentice	VAMPIR	PAT		
data collection	instrumentation via compiler	source instrumentation by preprocessor	sampling	object code instrumentation	
intrusion	high / corrected	high	low	high	
selective instr.	not required, optional	required	–	required	
selection interface	compiler switch	GUI or ASCII file	–	ASCII interface or file	
level of detail	summary information	event trace	statistics	summary information	event trace
information	total time and oper. counts for basic blocks and call sites	subroutine start/stop and send/recv events	statistical distr. of time (subroutines)	total time and pass counts for call sites	subroutine start/stop and send/recv events
strength	source-level, analysis of loops	many displays for message passing history and for statistics of arbitrary execution phases	low overhead profiling	total time for call sites, no recompilation	object code instr. for VAMPIR, no recompilation

Table 1: User interface and properties of performance analysis tools on CRAY T3E

therefore allowed us to provide high-level, well known abstractions as the programming interface to an EARL user.

Much of the power of EARL comes through its very high-level abstraction of an event trace which allows a programmer to concentrate on the trace analysis and let EARL take care of the different trace formats and their encoding of functions and event types, of input handling and buffering, and of keeping track of message queues and call stacks.

An EARL programmer can view an event trace as a sequence of events. EARL defines four predefined event types: entering and leaving a region, and sending and receiving a message. There may be more event types defined depending on the underlying trace format. A region is a named section of the traced program (e.g., it could be a loop or basic block, but mostly it is a function or subroutine). If supported by the trace format, regions may be organized in groups (e.g., user or system functions).

For all event types the following information is provided: the number of the event, the location, the timestamp, the event type, and the number of the enter event which determines the region in which the event happened. The enter and exit event types have an additional region attribute specifying the name of the region entered or left, and send and recv have attributes describing the destination, source, tag, length, and communicator of the message. In addition, the recv event type has a sendptr attribute pointing to the corresponding send event.

In addition to the basic event trace model, EARL provides the concepts of regions and messages. These are defined as pairs of matching events: enter/exit or send/recv respectively. For each position in the event trace, EARL defines a region stack per node and a message queue implemented as lists of enter and send events which define the regions entered and messages not yet received at that time. All these facilities together allow to easily process complex event patterns made out of regions and messages.

The EARL interpreter reads and decodes the underlying trace format and maps it automatically to the EARL event types and attributes. This allows the programmers to write their trace analysis scripts independent from the format of the event trace and of the encoding of event types and function/region names. Currently, EARL supports the VAMPIR [1] and ALOG [9] trace formats. Instead of re-inventing the wheel when implementing the EARL language, we started with the well known scripting language TCL and extended it with commands for event trace and event record handling. The extensions are implemented in C++.

EARL supports the following functionality:

- Trace handling: opening and closing event trace files

- Event record access: both sequential access and random access is supported.

- State access: EARL automatically keeps track of the state of the region stacks and the message queue for the current event.

- General information access: allows to get a list of all defined event types, regions, attributes, and to get the number of nodes used in the application.

We just completed our first prototype of EARL. For a complete and more detailed description see [14]. Early experiments show that although simple in design, EARL is a powerful and easy to use meta-tool for experts to implement generic or custom-made program or application domain-specific event trace analysis tools. Because of its programmability and flexibility, EARL can be used for a wide range of event trace analysis tasks:

1. calculation of performance indices and trace statistics of all kinds
2. finding all locations of possible bottlenecks (which then can be analyzed with traditional graphical trace analysis tools if necessary)
3. performance visualization and animation (as far as TK or other TCL graphics extensions are suitable for this task)
4. experiment management where within an experiment the instrumentation of the parallel program and generation of traces is based on results calculated from earlier runs. This allows to implement automatic program optimization tools.
5. application or domain-specific versions of these tasks

In the next months, we want to implement a library of useful generic EARL scripts and subroutines which then can be used by programmers to analyze their parallel applications. We also hope to implement additional decoder modules for other trace formats (e.g., PICL [6] or SDDF [13]) and to add more direct support for traces generated by programs in other programming paradigms than message passing.

Also, we will use EARL in our KOJAK project as a trace post-processing tool. In this project, we plan to explore different ways of representing and locating bottlenecks. Here, we plan to use EARL in order to easily implement and evaluate the different methods.

5 KOJAK design

The complexity of the current performance analysis environment on the CRAY T3E as well as similar experience in other environments, e.g., the SVM-Fortran programming environment and the native message passing environment on the Intel Paragon and the HPF+ environment [3], motivate the design of the automatic performance analysis environment KOJAK (Kit for Objective Judgement and Automatic Knowledged-based detection of bottlenecks).

The goals of the KOJAK project are to:

- automatically detect frequently occuring performance problems
- develop a generic tool that can be specialized for multiple programming paradigms and target machines
- integrate the new tool into existing environments and thus reuse already available performance information

It is important to note that we are aiming at detecting frequently occuring performance problems only. There will still remain the area of very program-specific bottlenecks that can only be detected by manually applying existing performance analysis tools.

The structure of KOJAK is shown in Figure 1. The entire performance analysis enviroment for the target system will consist of three main components: information supply and transformation tools, the performance database, and KOJAK.

The information supply tools on CRAY T3E are the compilers, Apprentice, VAMPIR and PAT. In addition, we already have the first transformation tool available, EARL, which is partly an incarnation of the generic summarizer tool. Further transformation tools will be a scalability analyzer computing speed up numbers from different program runs and a prediction tool deducing performance characteristics from already existing performance data [12].

The performance database will be the repository for all performance data. For each supply tool, a specialized input filter will be developed which transforms the tool-specific format into the database representation. KOJAK will trigger the information supply tools to generate required data and to input these into the database and will use the database query interface to retrieve information required for the analysis process.

We plan to base the implementation of the performance database on a standard database system because of the following database features:

- high-level specification of the available performance data
- low-level storage management
- data persistence
- client/server support for tool development

With respect to performance considerations, the granularity of the performance information has to be defined carefully [2]. Summary information, e.g. total execution time of program regions, and source code information can easily be stored in a database and efficiently accessed. Individual records of large trace files

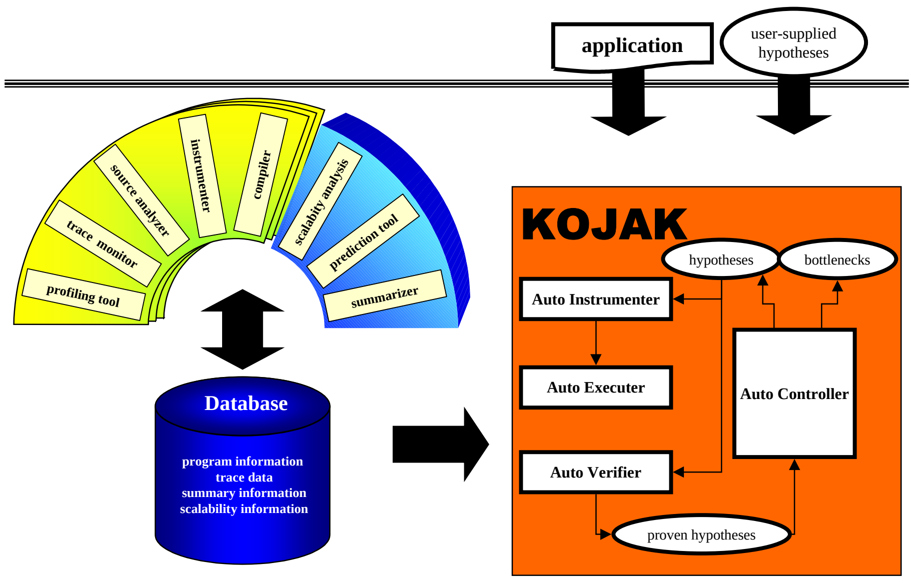

Figure 1: Structure of the automatic analysis environment KOJAK

will not be stored in the database, but the entire trace files will be stored as individual entries.

Information, required to prove performance problems, not included in the database but available in the trace files will be extracted from the trace files with the help of EARL. The advantage of storing trace files in the database is that the database will also serve as a vehicle for organizing multiple program versions and program executions.

Similar to Paradyn, KOJAK is based on the concept of hypothetical assumptions of existing bottlenecks (hypotheses). If a hypothesis is correct, it is called a bottleneck. For example, the hypothesis *load balancing problem in subroutine foo* can be checked by analyzing the execution time measured by Apprentice, VAMPIR, or PAT. If this hypothesis can be proven, a refinement is necessary to determine the location more precisely. Thus, the hypothesis should be refined into a set of hypotheses, one for each barrier synchronization in the subroutine.

KOJAK consists of four components which together control the whole environment:

1. Auto Controller

   The auto controller is the central component which guides the whole process by refining proven hypotheses. Based on the current set of hypotheses the other three components will be applied.

2. Auto Instrumenter

   The auto instrumenter analyzes the set of hypotheses and determines the best way to gather the required information. For the hypothesis mentioned above it decides whether to access available data or whether to run the program with any of the instrumentations, e.g., source code instrumentation for Apprentice or VAMPIR, or object code instrumentation with PAT. Its decision has to take all hypotheses into account to gather as much information as possible in the next program run. The instrumenter has to know the instrumentation interfaces to generate the correct commands.

3. Auto Executer

   Once the program is instrumented, it has to be executed on the target machine and the trace files have to be inserted into the database. This is the task of the executer which is based on the information how to run a program and how to handle the generated information.

4. Auto Verifier

   For each hypothesis a predicate has to be checked against the database to prove that the hypothesis is valid and thus is a bottleneck. The verifier retrieves the required information from the database to check all the hypothesis. In addition to information in the database complex patterns can be directly checked on the trace files based on EARL.

During the whole process a lot of information is required which should not be hardcoded. According to our goal to develop a generic tool that can be specialized for each target environment, the intensive use of specification languages as well as the representation of knowledge in a flexible form, such as rules, is required.

6 Conclusions

This article described the very advanced performance analysis support on CRAY T3E, consisting of three tools Apprentice, VAMPIR, and PAT, but demonstrated that this environment is complex to handle for application programmers (subsection 3.5). Frequently, application programmers are satisfied with their program's performance if the most important performance problems have been detected and fixed. Many of the users of our two CRAY T3E systems installed at the Research Centre Jülich either insert their own measurements into their codes because this seems to be easier to them than learning the available tools, or they invest a lot of time to learn the tools and are disappointed because the detected problems belong to a small class of frequently occuring performance bugs.

The design of KOJAK, described in this article, takes into account the concepts and ideas already developed in other projects but combines these with the flexibility of a generic environment which can be specialized to cooperate with already existing performance analysis tools of the target parallel machine. This enables KOJAK to reuse already available performance data. This article also described the first step towards an implementation of KOJAK, the meta-tool EARL, which is a programmable interface to calculate abstract metrics and to locate complex patterns in trace files.

References

[1] A. Arnold, U. Detert, W.E. Nagel: *Performance Optimization of Parallel Programs: Tracing, Zooming, Understanding*, CRAY User Group Meeting, Denver, Col., Eds: R. Winget and K. Winget, pages 252-258, 1995

[2] R. Borgeest, Ch. Rödel: *Trace Analysis with a Relational Database System*, Fourth Euromicro Workshop on Parallel and Distribued Processing, pp. 243-250, 1996

[3] M. Calzarossa, L. Massari, A. Merlo, M. Pantano, D. Tessera: *Integration of a compilation system and a performance tool: the HPF+ approach*, HPCN Europe 1998, LNCS 1401, pp. 809-815, 1998

[4] CRAY Research, *Introducing the MPP Apprentice Tool*, CRAY Manual IN-2511, 1994

[5] A. Espinosa, T. Margalef, E. Luque: *Automatic Performance Evaluation of Parallel Programs*, Sixth Euromicro Workshop on Parallel and Distribued Processing, 1998

[6] G.A. Geist, M.T. Heath, B.W. Peyton, P.H. Worley: *PICL: A Portable Instrumented Communication Library*, Technical Report ORNL/TM-11130, Oak Ridge National Laboratory, Tennessee, 1990

[7] M. Gerndt, A. Krumme, S. Özmen: *Performance Analysis for SVM-Fortran with OPAL*, Proceedings Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'95), Athens, Georgia, pp. 561-570, 1995

[8] B.R. Helm, A.D. Malony: *Automating Performance Diagnosis: a Theory and Architecture*, International Workshop on Computer Performance Measurement and Analysis (PERMEAN '95), 1995

[9] V. Herrarte, E. Lusk: *Studying Parallel Program Behavior with Upshot*, Technical Report ANL-91/15, Mathematics and Computer Science Division, Argonne National Laboratory, 1991

[10] B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam, T. Newhall: *The Paradyn Parallel Performance Measurement Tools*, IEEE Computer, Vol. 28, No. 11, pp. 37-46, 1995

[11] W.E. Nagel, A. Arnold, M. Weber, H-C. Hoppe, K. Solchenbach, *VAMPIR: Visualization and Analysis of MPI Resources*, Supercomputer 63, Vol. 12, No. 1, pp. 69-80, 1996

[12] M. Noelle, M. Pantano, X-H. Sun: *Communication Overhead: Prediction and Its Influence on Scalability*, To Appear PDPTA'98

[13] D.A. Reed, R.D. Olson, R.A. Aydt, T.M. Madhyasta, T. Birkett, D.W. Jensen, A.A. Nazief, B.K. Totty: *Scalable Performance Environments for Parallel Systems*, 6th Distributed Memory Computing Conference, pages 562-569, IEEE Computer Society Press, 1991

[14] F. Wolf, B. Mohr: *EARL - A Programmable and Extensible Toolkit for Analyzing Event Traces of Message Passing Programs* -, Technical Report FZJ-ZAM-IB-9803, Research Centre Jülich, 1998