Improving Time to Solution with Automated Performance Analysis

Shirley Moore, Felix Wolf, and Jack Dongarra
Innovative Computing Laboratory
University of Tennessee
{shirley,fwolf,dongarra}@cs.utk.edu

Bernd Mohr
Zentralinstitut für Angewandte Mathematik
Forschungszentrum Jülich
b.mohr@fz-juelich.de

December 10, 2004

1.0 Introduction

High performance computing is playing an increasingly critical role in advanced scientific research as simulation and computation are becoming widely used to augment and/or replace physical experiments. However, the gap between peak and achieved performance for scientific applications running on high-end computing (HEC) systems has grown considerably in recent years. The complex architectures and deep memory hierarchies of HEC systems present difficult challenges for performance optimization of scientific applications. Tools are needed that collect and present relevant information on application performance in a scalable manner so as to enable developers to easily identify and determine the causes of performance bottlenecks. According to the Report of the High-End Computing Revitalization Task Force (HECRTF) [1], the single most important metric for high-end system performance is *time to solution* for the scientific applications of interest. Time to solution includes not only execution time, but also development time. Portable, easy-to-use, effective performance tools aim to reduce both development and execution time.

In order to collect performance data, the application must be instrumented in some manner. To be most useful for performance tuning, the data should be collected at routine or even basic block or loop granularity. For developers of large-scale applications to implement this level of instrumentation manually is too time-consuming and thus not feasible. Automated instrumentation techniques are needed that can collect the relevant data with a minimum of effort.

Developers of scientific applications for HEC systems are not necessarily experts in high performance computing architectures and performance analysis. For this reason, performance data at the level of un-interpreted hardware counter data or communication statistics or traces

may not be useful to these developers. Higher level abstractions that identify various types of performance problems, such as inefficient use of the memory hierarchy or excessive synchronization delay for example, and that map these problems to the relevant application source code, will be much more useful and allow performance tuning to be done with much less time and effort.

The amount of performance data collected for applications running on HEC systems can be overwhelming. While analysis of event tracing has proved to be a superior technique to identify performance problems at a high level of abstraction, it usually suffers from scalability problems associated with trace-file size. Even collecting detailed profiling data for large numbers of processes can be unwieldy. Current display tools are limited in their representation of large-scale performance data.

This paper describes our efforts at addressing the above problems as part of the KOJAK project [2,3].

2.0 Automated Instrumentation

Figure 1 gives an overview of KOJAK's architecture and its components. The KOJAK analysis process is composed of two parts: a semi-automatic multi-level instrumentation of the user application followed by an automatic analysis of the generated performance data.

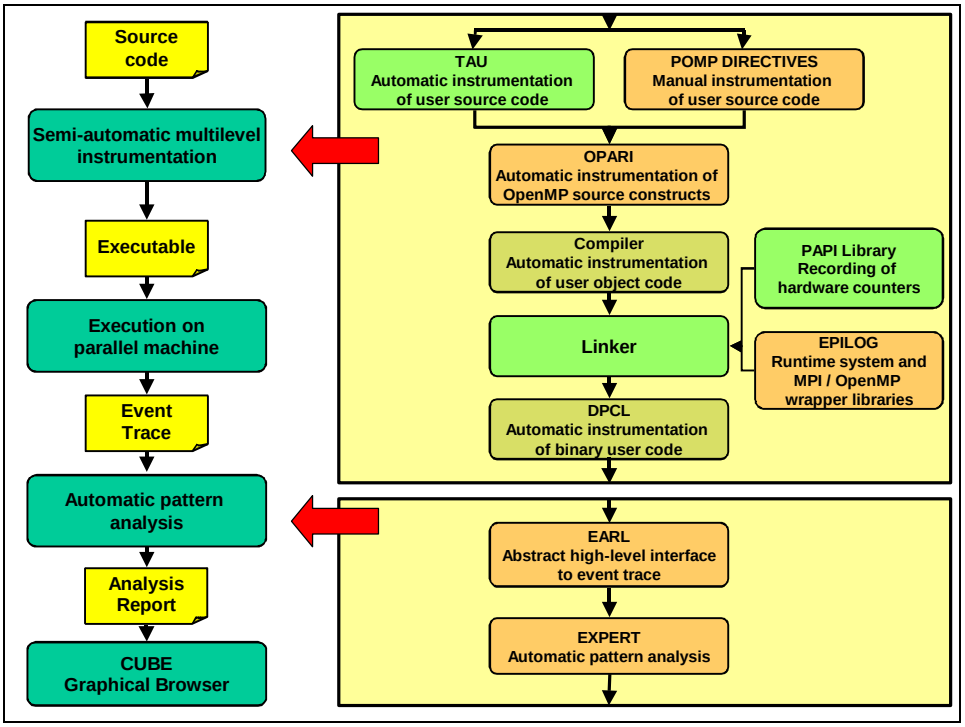

Figure 1. KOJAK Architecture

The event traces generated by EPILOG capture MPI point-to-point and collective communication as well as OpenMP parallelism change, parallel constructs, and synchronization. In addition, data from hardware counters accessed using the PAPI library [4,5] can be recorded in the event traces. To make measurements with the EPILOG system, the user's application must be instrumented at specific important points, or *events*, to activate EPILOG library calls. Events of interest include sending and receiving messages, user function entries and exits, entering and exiting OpenMP regions, and synchronization operations such as acquiring and releasing locks. Automated instrumentation is supported by compiler instrumentation on the following platforms:

- Linux clusters using the PGI compilers
- Hitachi SR-8000
- Sun Solaris (Sun Fortran90 compilers only)
- NEC SX

The instrumentation of user function entries and exits on the above systems is based on undocumented and unsupported compiler options. Discussions are underway with additional vendors to provide similar instrumentation hooks. Ideally these compiler instrumentation hooks will become fully supported in the future. On the above systems, all necessary instrumentation of user functions, MPI functions, and OpenMP constructs is handled by the "kinst" command. In the commands to build the application (e.g., in a makefile), the user need only precede all compile and link commands with "kinst". For example, instead of the command

```
% mpif90 myprog1.f90 myprog2.f90 –o myprog
```

the command

```
% kinst mpif90 myprog1.f90 myprog2.f90 –o myprog
```

would be executed.

For platforms on which compiler instrumentation using kinst is not supported, the users may manually instrument the desired functions and regions of their application by inserting POMP instrumentation directives and then using the "kinst-pomp" command in the same way as described above for "kinst". POMP instrumentation directives are supported for Fortran and C/C++ and are replaced by the necessary instrumentation calls by our source-to-source transformation tool OPARI [6]. In the case of OpenMP programs, OPARI also automatically instruments all OpenMP constructs and OpenMP run-time library calls by inserting calls to the POMP monitoring API [7]. An advantage of using POMP instrumentation directives is that the instrumentation is ignored during normal compilation. An INST BEGIN/INST END pair can be used to mark any user-defined sequence of statements, again with a single argument giving a

name for the code region. At least the main program function must be instrumented in this way, and in addition, an INST INIT directive must be inserted as the first executable statement of the main program.

While fairly straightforward, such manual instrumentation of a large program is time-consuming and has an adverse effect on time to solution. In addition, the manual instrumentation must be redone with every new version of the program Fortunately, the TAU performance analysis system [8] provides an automated source code instrumentation mechanism that can be used with the EPILOG library. TAU is a cross-platform tool that supports a wide variety of HEC platforms. To use TAU's automated source code instrumentation, the user should first configure and build TAU with the desired options. To use EPILOG, TAU should be configured with the –TRACE and –epilog options. Then only two changes need to be made to the application makefile. First, a makefile stub with the necessary TAU definitions, which was created when the appropriate library was built, should be included. Then the user need only precede all compile and link commands with $(TAU_COMPILER). All MPI functions, user functions, and OpenMP constructs will then be instrumented with EPILOG library calls. TAU also uses KOJAK's OPARI system [6] to automatically instrument OpenMP constructs. Although TAU currently supported automated source code instrumentation only down to the routine level for non-OpenMP codes, there are plans to extend the automated instrumentation capability to the basic block and loop level.

An alternative to source code instrumentation is to use automatic binary instrumentation. KOJAK support binary instrumentation on IBM systems where the optional DPCL (Dynamic Probe Class Library) package [9] has been installed. The user need only precede compile and link commands with "kinst-dpcl" and launch the resulting program using the "elg-dpcl" command. TAU supports binary instrumentation using the Dyninst[10,11] library. The *tau_run* tool dynamically loads the specified TAU instrumentation library and instruments the application at runtime. All user and MPI functions are instrumented.

If EPILOG has been built with hardware counter support enabled, then hardware counter data can be recorded as part of the event records. To request the measurement of certain counters, the user must set the environment variable ELG_METRICS to a colon-separated list of counter names. EPILOG uses the PAPI library [4,5] to access the hardware counters. All of the PAPI standard metrics are supported for data collection although not all are currently supported for automated analysis.

Any of the instrumentation methods described above will cause an EPILOG trace file to be produced when the application is run. The per-process trace files generated during the

execution will be automatically merged into a single trace file when execution ends. The resulting trace file can be analyzed using KOJAK's automated performance analysis as explained below.

3.0 Automated Performance Analysis

Large-scale applications running on HEC systems can produce extremely large trace files. Visualization tools such as Vampir and Intel Trace Analyzer [12], Jumpshot [13], and Paraver [14] can provide a graphical view of the state changes and message passing activity represented in the trace file, as well as provide statistical summaries of communication behavior. However, it is difficult and time-consuming for even expert users to identify performance problems from such a view or from large amounts of statistical data. Spending large amounts of time analyzing performance data manually has a negative effect on time to solution. KOJAK's EXPERT tool is an automatic trace analyzer that attempts to identify specific performance problems. Internally, EXPERT represents performance problems in the form of execution patterns that model inefficient behavior. These patterns are used during the analysis process to recognize and quantify inefficient behavior in the application.

The performance problems addressed by EXPERT include inefficient use of the parallel programming model and low CPU and memory performance. Internally patterns are specificed as C++ classes that provide callback methods to be called upon occurrence of specific event types in the event stream. The pattern classes are organized in a specialization hierarchy, as shown in Figure 2. There are two types of patterns: 1) simple profiling patterns based on how much time or some other metric (e.g., cache misses) is spent in certain MPI calls or code regions, and 2) patterns describing complex inefficiency situations usually described by multiple events – e.g., late sender in point-to-point communication or synchronization delay before all-to-all operations. Recent work has taken advantage of the specialization relationships to obtain a significant speed improvement for EXPERT and to allow more compact pattern specifications [15]. Each pattern calculates a (call path, location) matrix containing the time spent on a specific behavior in a particular (call path, location) pair, where a location is a process or thread. Thus, EXPERT maps the (performance problem, call path, location) space onto the time spent on a particular performance problem while the program was executing in a particular call path at a particular location. After the analysis has been finished, the mapping is written to a file and can be viewed using the CUBE display tool.

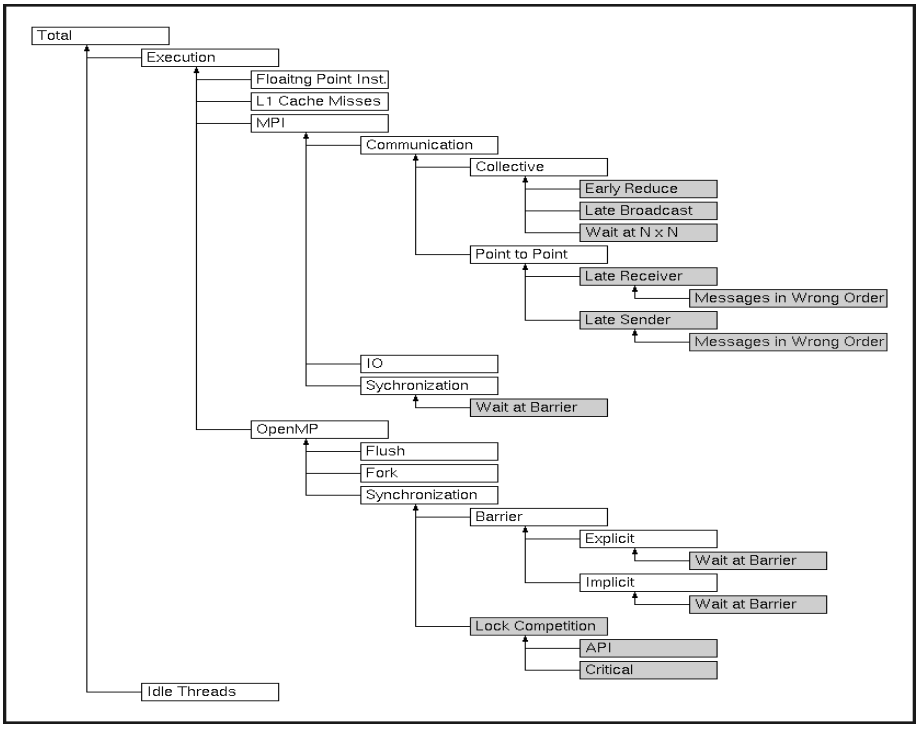

Figure 2. EXPERT pattern specialization hierarchy

The CUBE display for a crystal growth simulation [16] run on eight processors of a Linux cluster is shown in Figure 3. The display consists of three coupled tree browsers, representing the metric, the program, and the location dimensions from left to right. The user can switch between a call tree and a flat profile view of the program dimension, with the default being the call-tree view. The nodes in the metric tree represent performance metrics, the nodes in the call tree represent call paths, and the nodes in the system tree represent machines, nodes, processes, and threads. A user can perform two types of actions: selecting a node or expanding/collapsing a node. At any given time, there are two nodes selected, one in the metric tree and one in the call tree. Each node is labeled with a severity value. A value shown in the metric tree represents the sum of a particular metric for the entire program, that is, across all call paths and all locations. A value shown in the call tree represents the sum of the selected metric across all locations for a particular call path. A value shown in the location tree represents the selected metric for the selected call path and a particular location. All numbers may be displayed either as absolute values or as percentages. To help identify metric/resource combinations with a high severity, values are ranked using colors. The color legend shows a numeric scale mapping values to colors. Note that all hierarchies in CUBE are inclusion hierarchies, meaning that a child

node represents a subset of a parent node. The severity value in CUBE follows the principle of *single representation* – that is, within a tree each fraction of the severity is displayed only once. The purpose of this strategy is to have a particular problem appear only once in the tree and thus help identify it more quickly.

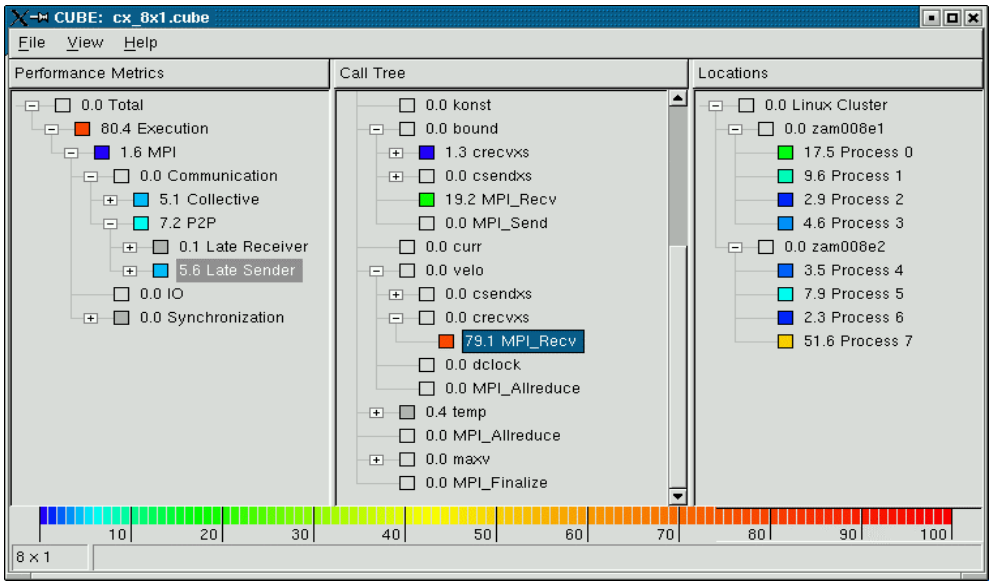

Figure 3. CUBE display showing metric, call tree, and location dimensions

Performance optimization of parallel applications usually involves multiple experiments to compare the effects of different code versions, different execution configurations, or different input data. In addition, hardware characteristics may limit the availability of certain performance data, such as hardware counter data, in a single run, requiring multiple experiments to obtain a full set of data. A user may also wish to combine the results obtained using different monitoring tools that cannot be applied simultaneously. Finally, results of analytical modeling or simulation may need to be compared with experimental data. The traditional method of comparing different experiments is to put multiple single-experiment views side by side or to plot overlay diagrams. Previous research on multi-experiment analysis described in [17] uses an operator to calculate a list of resources showing significant discrepancies between different experiments. However, this difference operator maps from its input space containing entire experiments into a smaller representation consisting of a list of resources. A repeated application is not possible, and further processing would require a logic or display different from the one suitable for the original input data. With our approach the output of multi-experiment analysis can be represented just like its input, allowing us to use the same set of tools to process and display it. The CUBE performance

algebra can be used to compare, integrate, and summarize performance data of message-passing and/or multithreaded applications from multiple experiments including results obtained from simulations and analytical modeling. The algebra consists of a data model to represent the data in a platform-independent fashion plus arithmetic operations to subtract, merge, and average the data from multiple experiments., All operations are closed in that their results are mapped into the same space, yielding an entire "derived" experiment including data and metadata. Figure 4 shows the differences between two versions of a nano-particle simulation, with raised reliefs indicating performance improvements and sunken reliefs indicating performance degradations. The differences are broken down along the various dimensions. The CUBE performance algebra is described in further detail in [18].

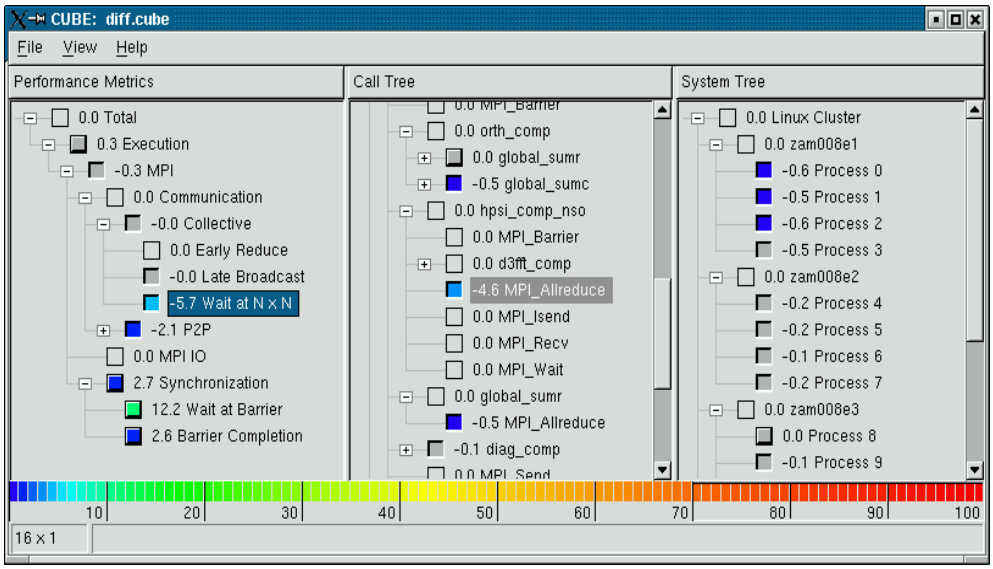

Figure 4. Intuitive display of differences between two code versions

4.0 Scalability Issues

While analysis of event tracing has proved to be a superior technique to identify performance problems at a high level of abstraction, it usually suffers from scalability problems associated with trace-file size. If the automated instrumentation techniques described in section 2 are applied non-discriminately, they can instrument all user and MPI functions and OpenMP constructs with trace library calls, resulting in a large amount of trace data being collected. Although KOJAK's approach of reducing the trace data to a higher-level, more compact representation using EXPERT results in a much smaller data file, the initial very large raw trace file can be problematic. Fortunately TAU provides a filtering mechanism available to reduce the instrumentation. An initial profiling run can be conducted to identify routines that are called a

very large number of times and for which trace data do not contribute much useful information. These routines can then be excluded from the automated instrumentation by specifying them in an exclude list. The *tau_reduce* tool can be used to generate the exclusion list automatically, and work is underway to add this capability to KOJAK's module that handles automatic user function instrumentation via compiler switches.

Even with careful filtering, large-scale applications can still produce very large trace files. In view of present and future architectures consisting of thousands of processors and in view of applications running on all or at least a major fraction of the available CPUs, KOJAK's current approach will become increasingly constrained by the potentially enormous size of the resulting event traces. The current approach of collecting a large trace file for an entire parallel program execution in a centralized location and then processing and reducing this single trace file, such as the approach used by the EXPERT trace analysis tool in KOJAK, will not scale to thousands of processors. Although recent improvements have made an order of magnitude improvement in EXPERT's efficiency [15], our future research in this area will focus on applying parallel and distributed processing approaches to the processing, reduction, and filtering of large-scale trace data.

KOJAK's current CUBE display will be unwieldy for representing HEC systems with thousands of processors. We plan to develop a highly optimized version of the CUBE display that replaces the current tree representation of processes and threads with a much more scalable multi-dimensional topology display reflecting the virtual topology of the application and/or the physical topology of the machine. As an integral part of parallel programming deals with choosing the right virtual topology, that is, the mapping of processes and threads onto the problem domain, a topology display will not only be much more scalable but can also provide more intuitive guidance in analyzing the influence of physical or logical communication structures.

5.0 Conclusions and Future Work

Automated approaches to performance instrumentation and analysis promise to increase programmer productivity and reduce time to solution by reducing both development time and execution time. Performance tuning is often a neglected part of application development because the amount of effort invested does not yield an adequate return in reduction of execution time. By automatically and accurately pinpointing the most severe performance problems, the amount of effort can be reduced while achieving greater performance gains.

Although some significant results have been obtained already, the pattern analysis used by EXPERT could be considerably improved. We have only begun to scratch the surface on the specification of patterns based on hardware counter data and on correlating these data with other events and with program data structures. The pattern search could also be made more accurate by applying it at the loop level. Scientific applications frequently contain computationally intensive nested loop structures, the tuning of which is critical to achieving good performance. We expect the combination of automated instrumentation at the loop level and the specification of patterns for analyzing nested loop performance to provide a powerful mechanism for achieving substantial performance gains with a minimum of effort.

To be most useful to a developer in tuning an application, information about cache and memory behavior should be presented in a way that relates it to program data structures at the source code level. KOJAK's EXPERT analyzer and CUBE display tool support post-mortem analysis of trace and/or profile data with a display that allows the user to interactively explore a similar three-dimensional performance space with the metric, call tree, and location dimensions displayed by coupled tree browsers. In order to enable performance analysis to focus specifically on memory hierarchy performance as it relates to data structures used by an application, we plan to extend the search space to include an explicit data structure dimension. This dimension will include various levels of data structures that may be distributed across multiple memories in a parallel system. Explicit representation of this dimension will better specification of patterns that represent inefficient memory system performance, as well as hyper-linking detected memory performance problems to entities in the other dimensions of the performance search space, such as the specific call path that is generating the particular memory performance problem.

Future HEC systems may require the use of new parallel programming paradigms. We have prototyped an extension of the KOJAK toolset that is able to instrument, record, and analyze MPI-2 one-sided communication and synchronization features. This work can be extended easily to handle vendor-specific one-sided communication such as SHMEM or LAPI. Work is also underway to analyze Co-Array Fortran [19] applications using KOJAK.

In Section 4, we have already mentioned planned future work on improving the scalability of trace-based automated performance analysis.

Our claim that automated performance analysis can improve time to solution of scientific applications on HEC systems needs to be verified with experimental evidence. The amount of effort actually involved in using the tools needs to be measured and the performance gains obtained quantified under controlled conditions. We plan to investigate programmer productivity metrics and apply them to measure the effectiveness of the KOJAK approach.

References

1. Federal Plan for High-End Computing, Report of the High-End Computing Revitalization Task Force (HECRTF), Executive Office of the President, Office of Science and Technology Policy, May 2004, http://www.itrd.gov/pubs/2004_hecrtf/20040702_hecrtf.pdf.

2. KOJAK web site, http://icl.cs.utk.edu/kojak/

3. Wolf, F. and B. Mohr, Automatic performance analysis of hybrid MPI/OpenMP applications. *Journal of Systems Architecture*, November 2003, pp. 421-439.

4. PAPI web site, http://icl.cs.utk.edu/papi/

5. Browne, S., et al., A Portable Programming Interface for Performance Evaluation on Modern Processors. *International Journal of High-Performance Computing Applications* 14(3), 2000, pp. 189-204.

6. B. Mohr, A. Malony, S. Shende, F. Wolf, Towards a Performance Tool Interface for OpenMP: An Approach Based on Directive Rewriting, *Proceedings of the Third European Workshop on OpenMP - EWOMP'01*, Barcelona, Spain, September 2001.

7. B. Mohr, A. Malony, H.-Ch. Hoppe, F. Schlimbach, G. Haab, S. Shah, A Performance Monitoring Interface for OpenMP, *Proceedings of the fourth European Workshop on OpenMP - EWOMP'02*, Rome, Italy, September 2002.

8. TAU web site, http://www.cs.uoregon.edu/research/paracomp/tau/

9. DPCL web site, http://www-124.com/developerworks/opensource/dpcl/

10. Buck, B.R. and J.K. Hollingsworth, An API for Runtime Code Patching. *International Journal of High Performance Computing Applications* 14(4), 2000, pp. 317-329.

11. Dyninst web site, http://www.dyninst.org/

12. Intel Cluster Tools web site, http://www.intel.com/software/products/cluster/index.htm

13. Jumpshot web siute, http://www-unix.mcs.anl.gov/perfvis/software/viewers/index.htm

14. Paraver web site, http://www.cepba.upc.es/paraver/

15. Wolf, F., et al. Efficient Pattern Search in Large Traces through Successive Refinement, in *European Conference on Parallel Computing (Euro-Par)*. Pisa, Italy. August-September, 2004.

16. Mihelcic, M., H. Wenzl, and H. Wingerath, Flow in Czochralski Crystal Growth Melts. Forschungszentrum Jülich Technical Report Jül-2697, 1992.

17. Karavanic, K. L. and B. P. Miller, Experiment Management Support for Performance Tuning, in *SC'97*, San Jose, California, November 1997.

18. Song, F., et al. An Algebra for Cross-Experiment Performance Analysis, in *International Conference on Parallel Processing (ICPP)*. Montreal, Canada. August, 2004.

19. Numrich, R. W. and J.K. Reid, Co-Array Fortran for Parallel Programming, *ACM Fortran Forum* 17(2), pp 1-31, 1998.