End-user Tools for Application Performance Analysis Using Hardware Counters

K. London, J. Dongarra, S. Moore, P. Mucci, K. Seymour, T. Spencer

Abstract

One purpose of the end-user tools described in this paper is to give users a graphical representation of performance information that has been gathered by instrumenting an application with the PAPI library. PAPI is a project that specifies a standard API for accessing hardware performance counters available on most modern microprocessors. These counters exist as a small set of registers that count "events", which are occurrences of specific signals and states related to a processor's function. Monitoring these events facilitates correlation between the structure of source/object code and the efficiency of the mapping of that code to the underlying architecture. The perfometer tool developed by the PAPI project provides a graphical view of this information, allowing users to quickly see where performance bottlenecks are in their application. Only one function call has to be added by the user to their program to take advantage of perfometer. This makes it quick and simple to add and remove instrumentation from a program. Also, perfometer allows users to change the "event" they are monitoring. Add the ability to monitor parallel applications, set alarms and a Java front-end that can run anywhere, and this gives the user a powerful tool for quickly discovering where and why a bottleneck exists. A number of third-party tools for analyzing performance of message-passing and/or threaded programs have also incorporated support for PAPI so as to be able to display and analyze hardware counter data from their interfaces.

Introduction

For years collecting performance data on applications has been an imprecise art. Users had to rely on timers to determine where bottlenecks were in their program, and this didn't tell them why the bottleneck was there. Today hardware counters exist on every major platform. These counters can provide information as to where bottlenecks exist, but more importantly why they exist. The problem is that access to these counters is often poorly documented, unstable or unavailable at the user level program. PAPI provided a solution that is well documented, easy to use and has a standard interface across all major processor platforms. In addition, the PAPI project has defined a standard set of events considered most relevant to application performance tuning and has mapped as many of these events as possible to native events available on the different platforms. While PAPI provided access to the counters, it still requires users to instrument their code with calls to PAPI. Also, changing the metric(s) being monitored requires changing the code, recompiling and running the application again.

Perfometer provides a simpler instrumentation of the code, with one additional function that can be inserted at the beginning of the code. A user can then monitor a variety of events "on the fly" without having to change their code. The intent of perfometer is to provide a fast, coarse-grained, easy way for a developer to find out where and why a bottleneck exists in a program. Then the developer can instrument those particular sections of code by hand with PAPI to obtain fine grain information about where and why the bottleneck exists. Future work will target making such fine-grained instrumentation more automatic.

Obtaining data for parallel applications and long running applications can be cumbersome and inefficient with traditional tools. The data collected may be a summary of statistics and not provide much insight to the dynamic runtime behavior of an application. Most tools require that the data being collected be specified prior to runtime and do not allow for runtime selection or adjustment of which events to measure or the level of detail or granularity at which measurements should be made. Perfometer provides mechanisms to change the event being monitored at runtime, as well as set alerts that will pause program execution when certain conditions are met. This allows users to run their program and when the program hits an area of poor performance, the program will be paused until they resume it. This makes it pracitical to use visual representation on long running programs. Also, perfometer can monitor one or more processes in a heterogeneous parallel programming environment making it a valuable tool in clustered and grid computing environments

Perfometer Architecture and Design

Perfometer has three different portions the GUI, the server and the library. The server runs on any platform that supports pthreads and the GUI runs on any platform that supports Java 1.2 or higher, while the library runs on every platform that PAPI supports. The library is layered on top of PAPI and supports both Fortran and C programs. All three portions communicate using TCP sockets so they need not be on the same machine. If monitoring just one application the server is not needed and the GUI can connect directly to the application as seen in figure 1. However, if monitoring more than one application the GUI has to connect to the server, which is responsible for filtering any information and passing the performance data back to the GUI as seen in figure 2.

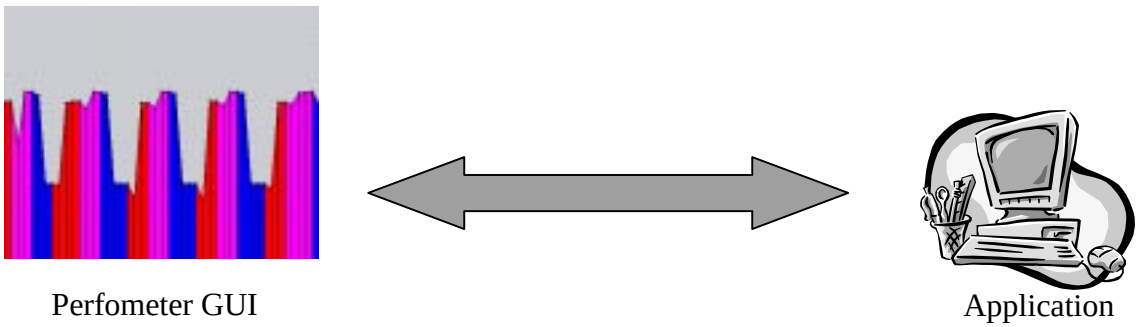

Figure 1: Perfometer being used to connect to a single process application

Library

The library serves as the backend, providing users three calls that they can put into their code. Perfometer() is the only necessary call and initializes all the data necessary and then prints out the port number that it is listening to, by default port 3545. Mark_perfometer(color, "label") is used to differentiate between different portions of the code in the GUI. The color will correspond to the color used by the GUI during that time

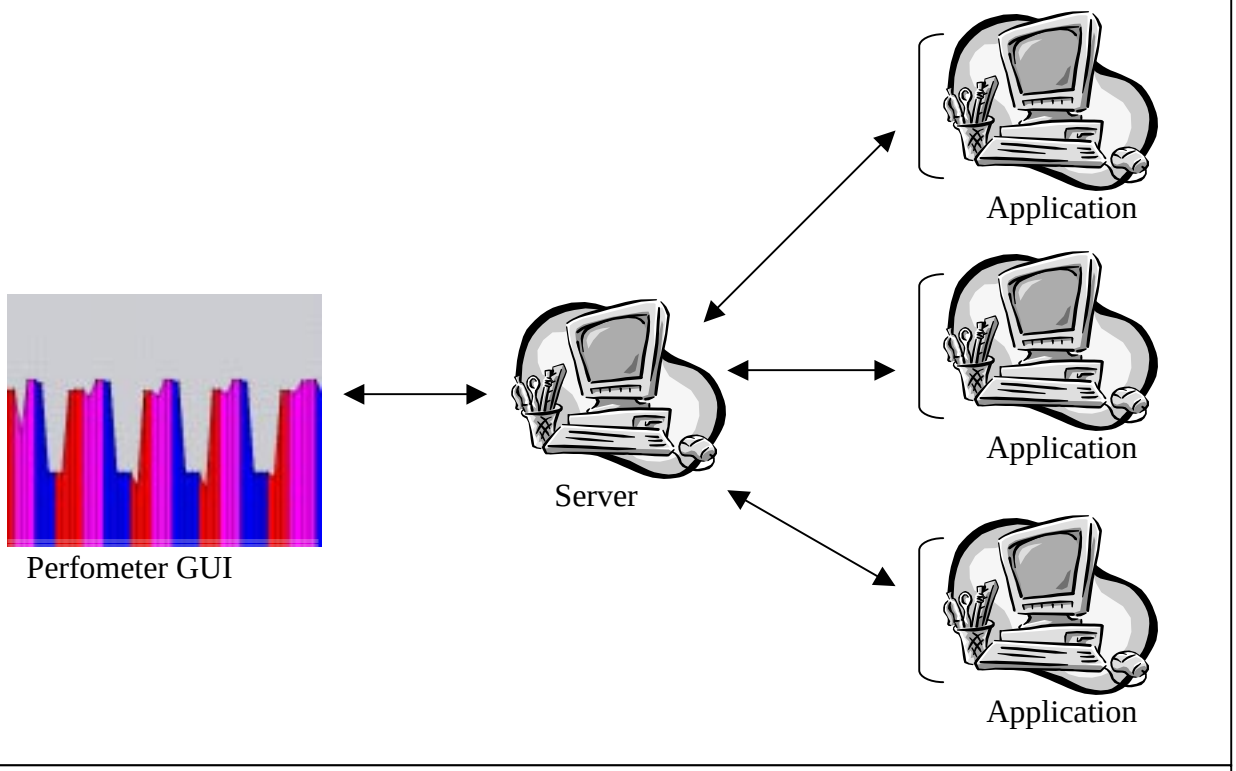

Figure 2: Perfometer being used to monitor multiple applications.

period, while clicking on the graph will display the label. And finally stop_perfometer() removes the perfometer instrumentation and continues the program execution. Perfometer sets up an interrupt timer (100ms by default), the handler for which calls a routine to collect information using PAPI. This information is then sent using non-blocking writes to the server/GUI. Any communication from the server/GUI is received by using asynchronous communication using signals to let the application know when something is waiting. The back-end does no manipulation of the data as it strives to be as lightweight as possible.

Server

The server collects communication from the processes being monitored and passes it to the GUI. The main purpose of the server is to move any filtering from the

applications as not to impact the program that much, and to move it away from the GUI so the graphing of the data will not be slowed down. The server also provides a logging facility that can create a tracefile that the GUI can later load and play back. Currently, the server only filters information when an alarm is set. It will stop sending information back to the GUI and instead will check to see if the data is in the range of the expected values (this is set at runtime via the GUI). If that range is breached the server sends a command to the application to pause execution and informs the GUI. In the future, this filtering will be extended to make the task of monitoring parallel applications easier to understand and graph. Like the application library, when the server is executed it prints out the port that it is waiting on and the GUI connects to that port and then adds the application processes.

Graphical User Interface

The graphical user interface as seen in figure 3, is where all the performance information is displayed in a graph. As can be seen in figure 3, there are a few buttons.

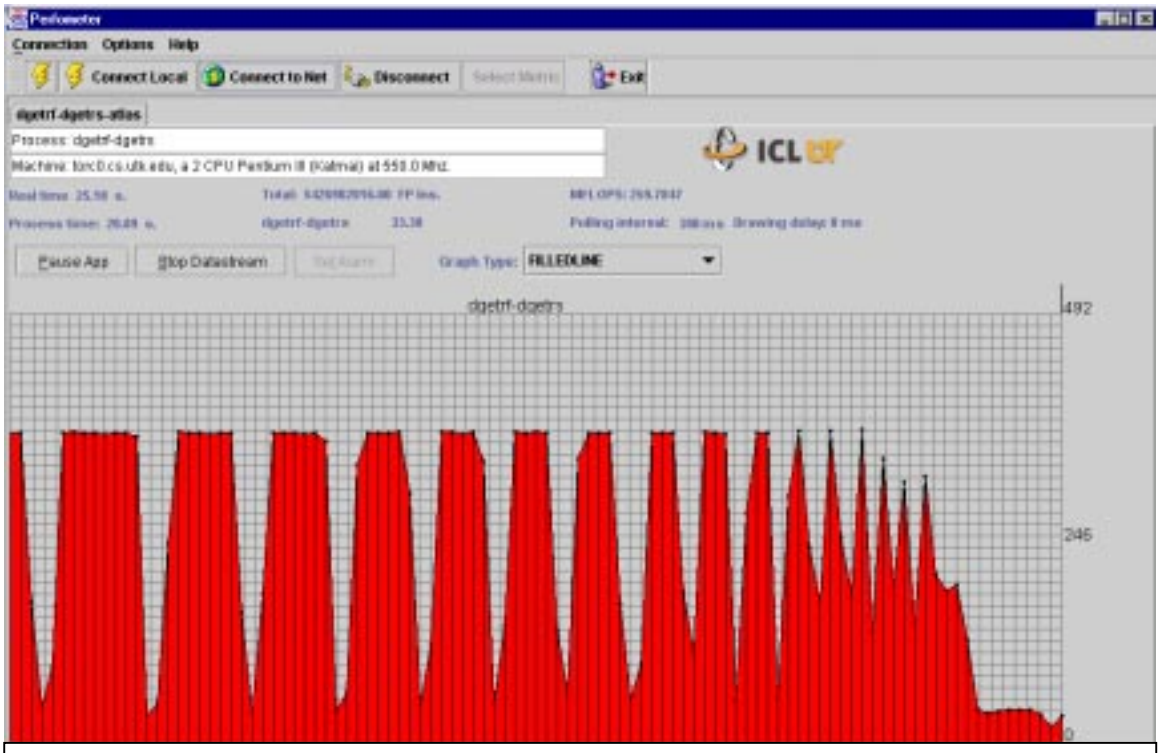

Figure 3: Example of perfometer monitoring a single application

The "Pause App" button will halt the application from executing and then will change to a "Resume App" button. The "Stop Datastream" will continue the program execution but will stop sending the data to the GUI to graph. The "Set Alarm" button which is grayed out in this example allows the user to set a range within which the data should fall. This range is set by setting the maximum and minimum values and a threshold of how many times those values can be violated before the alarm goes off. When the alarm goes off,

the application is paused until the user resumes the application. The other important button is the "Select Metric" button, this brings up a menu that allows a user to change to a multitude of different events including: Mflop/s, Level 1 cache miss rate, Level 2 cache miss rate, percent of mispredicted branches and any other PAPI standard event (currently well over 100 are defined). In the future, perfometer will also support the ability to change to native events.

Current work involves extending the graphical interface to display performance data for multithreaded and multiprocess programs in a scalable manner. Currently data for multiple processes are displayed as shown in Figure 4. Any process can be selected for the larger display by clicking on its representation in the multiprocess display.

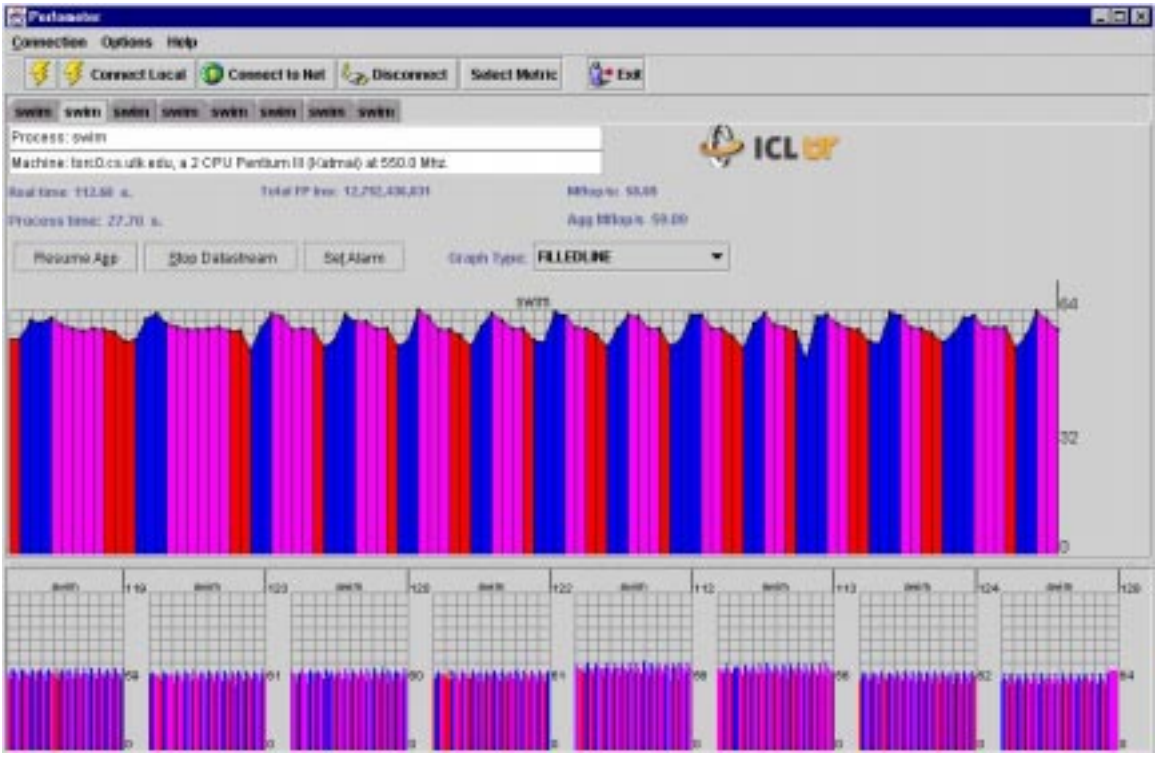

Figure 4. Example of perfometer monitoring a parallel application

Other End-user Tools

Visual Profiler, or vprof, is a tool developed at Sandia National Laboratory for collecting statistical program counter data and graphically viewing the results on Linux Intel machines[]. vprof uses statistical event sampling to provide line-by-line execution profiling of source code. vprof can sample clock ticks using the profil system call. The vprof developer has added support for PAPI so that vprof can also sample the wide range

of system events supported by PAPI. A screenshot showing vprof examination of both profil and PAPI_TOT_CYC data is shown in Figure 5.

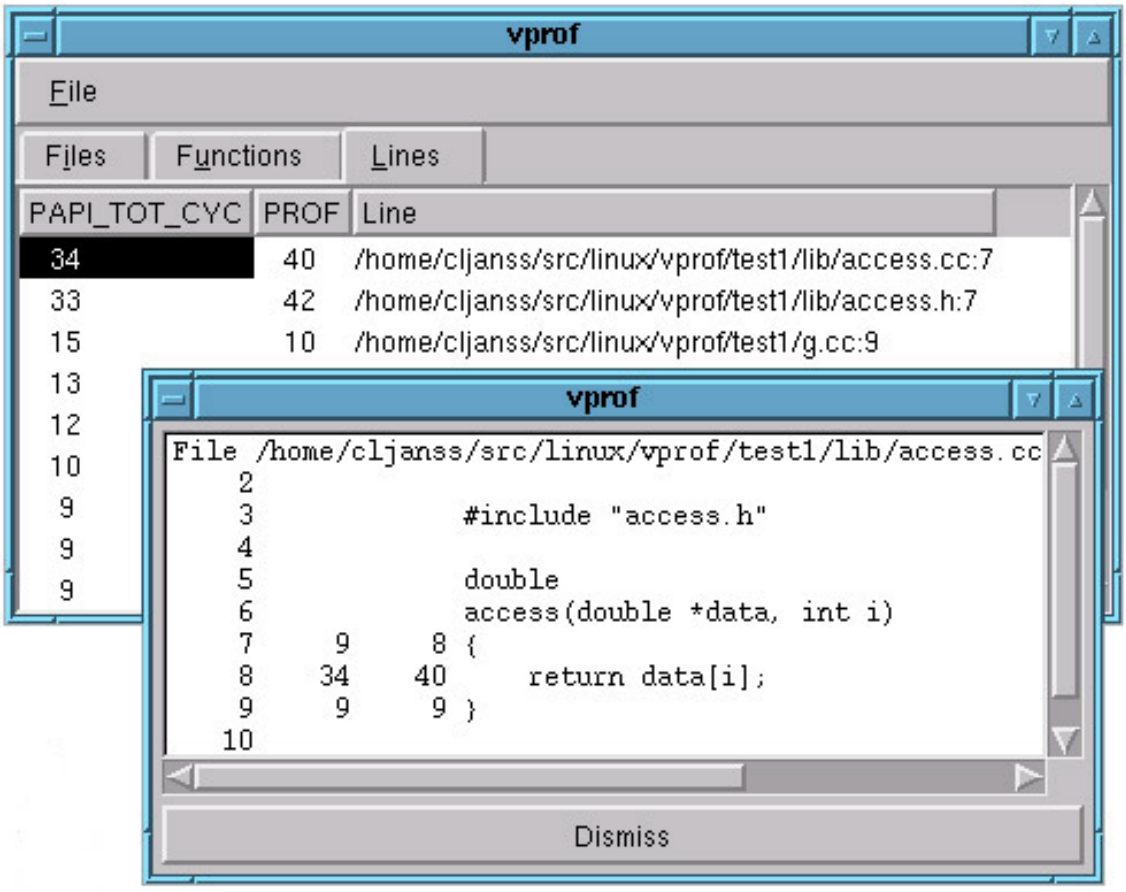

Figure 5. vprof displaying profil and PAPI_TOT_CYC data

SvPablo is a graphical source code browser and performance visualizer that has been developed as part of the University of Illinois' Pablo project[5,6]. SvPablo supports automatic instrumentation of HPF codes with Portland Group's HPF compiler and interactive instrumentation of C and Fortran programs. During execution of an instrumented code the SvPablo library maintains statistics on the execution of each instrumented event on each processor and maps these statistics to constructs in the original source code. The current version of SvPablo includes support for the MIPS R10000 hardware performance counters. The next version of SvPablo, being developed at the IBM Advanced Computing Technology Center, has integrated support for PAPI. Screenshots of SvPablo displays of PAPI metrics are shown in Figures 6 and 7.

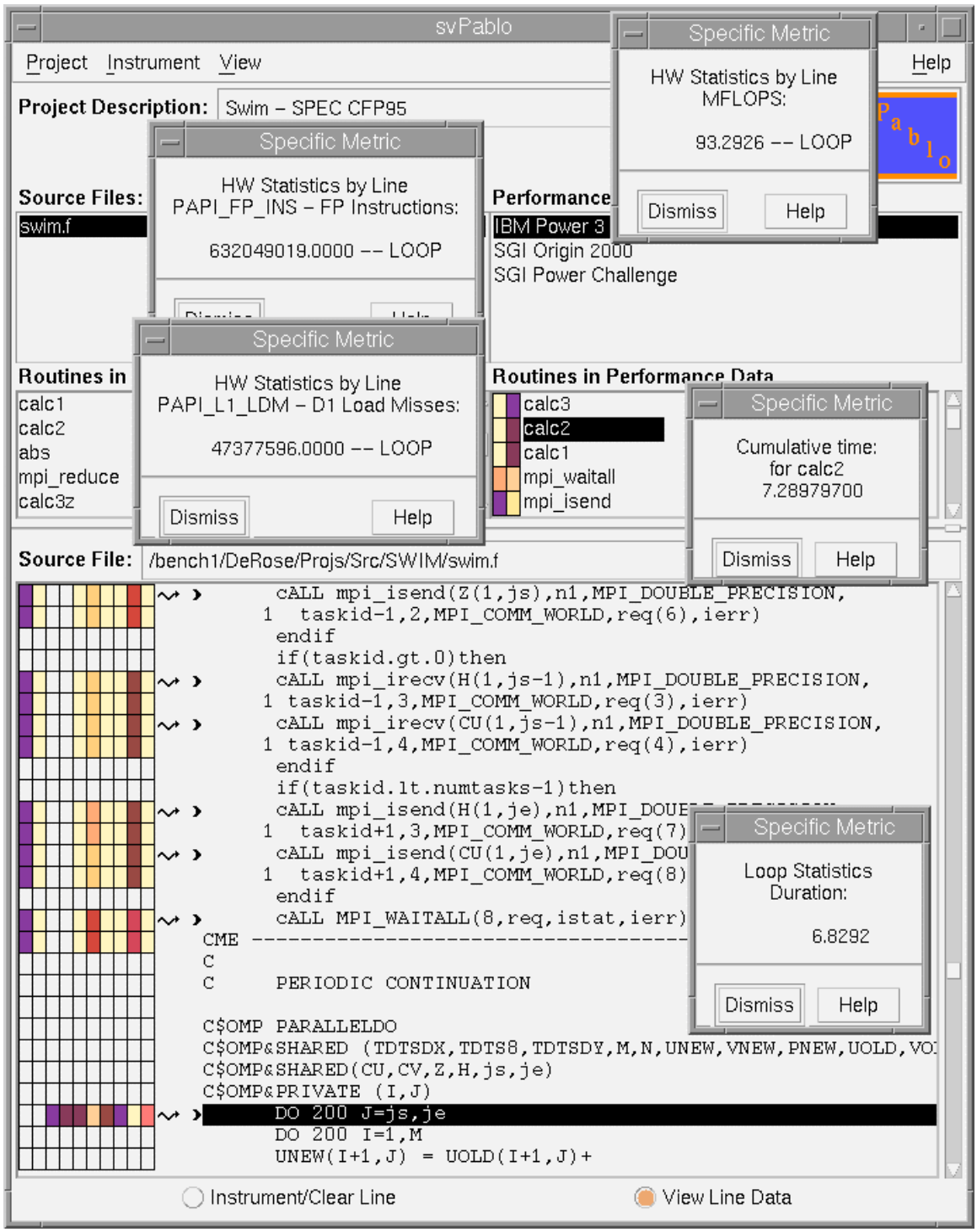

Figure 6. SvPablo source code browser displaying PAPI metrics

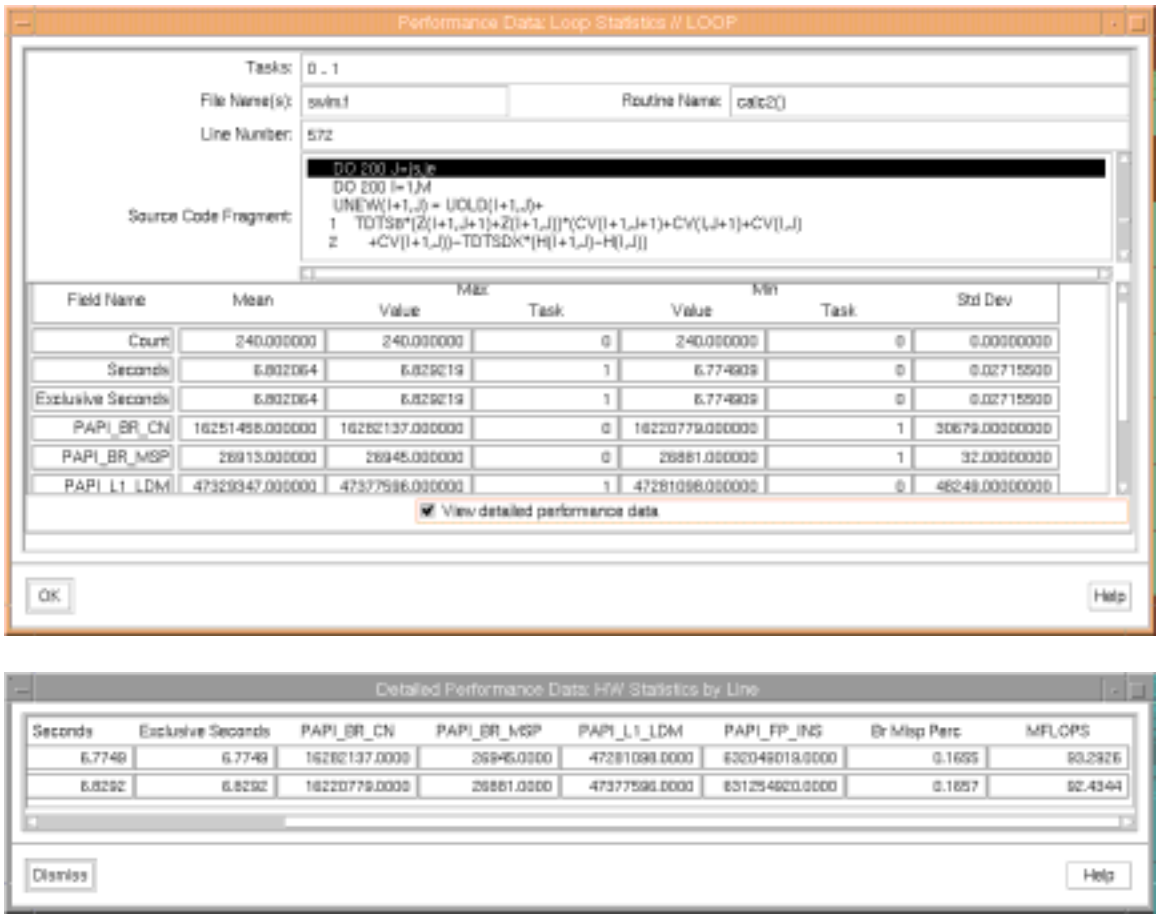

Figure 7. SvPablo statistics displays showing PAPI data

DEEP from Pacific-Sierra Research stands for Development Environment for Parallel Programs. DEEP provides an integrated interactive GUI interface that binds performance, analysis, and debugging tools back to the original parallel source code. DEEP supports Fortran 77/90/95, C, and mixed Fortran and C in Unix and Windows 95/98/NT environments. DEEP supports both shared memory (automatic parallelization, OpenMP) and distributed memory (MPI, HPF, Data Parallel C) parallel program development. A special version of DEEP called DEEP/MPI is aimed at support of MPI programs. DEEP provides a graphical user interface for program structure browsing, profiling analysis, and relating profiling results to source code. DEEP developers are incorporating support for PAPI so that statistics for the standard PAPI metrics can be viewed and analyzed from the DEEP interface. A screenshot of the DEEP/MPI interface displaying PAPI data for L2 cache misses is shown in Figure 8.

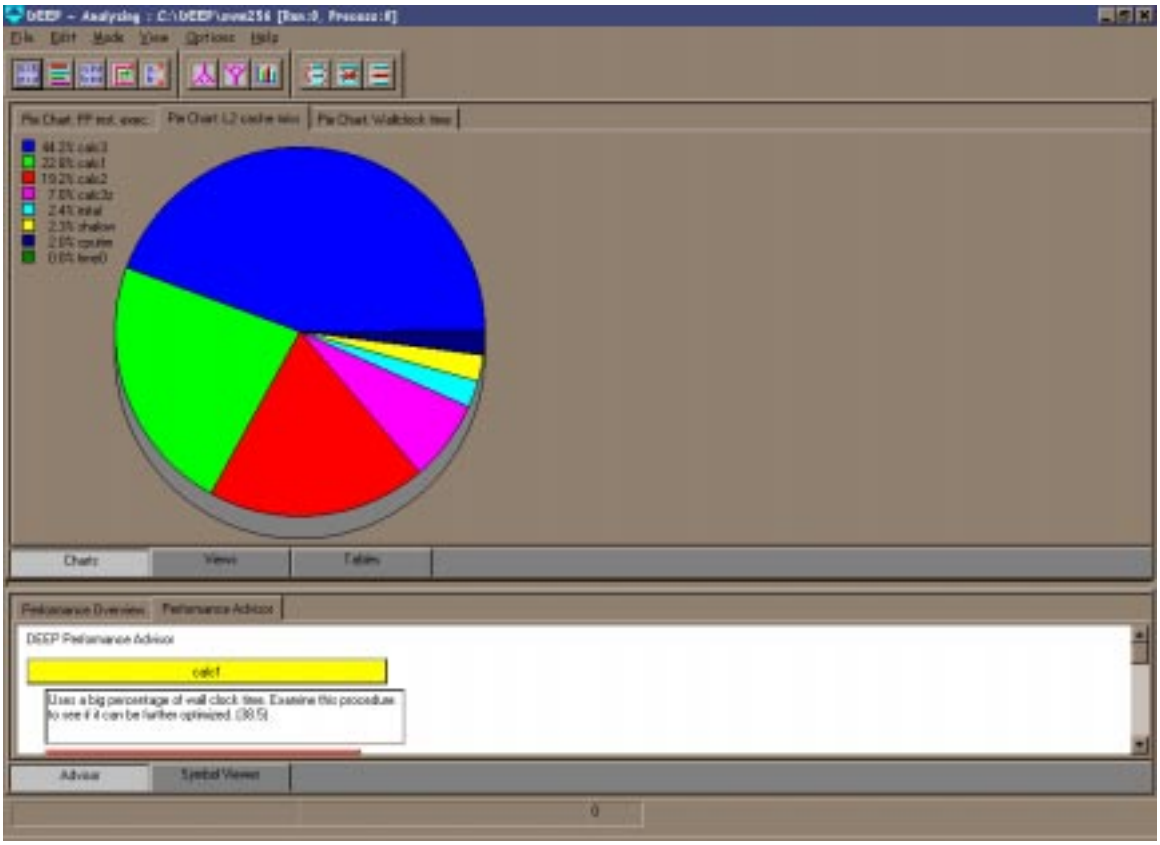

Figure 8. DEEP/MPI displaying PAPI data for L2 cache misses

TAU (Tuning and Analysis Utilities) is a performance analysis environment for C++, Java, C, Fortran 90, HPF, and HPC++ being developed at the University of Oregon. TAU uses PAPI for reading the hardware performance counters and provides portable profiling for OpenMP and MPI parallel programs. Use of PAPI maintains TAU portability while enabling access to platform-specific performance metrics. Figure 9 illustrates use of PAPI by TAU to profile a mixed mode MPI/OpenMP program using floating point instructions as the profiling metric.

Figure 9. Use of TAU with PAPI on the NAS Parallel Benchmark LU

VAMPIR is a performance analysis tool for MPI parallel programs developed by Pallas in Germany. The next version of VAMPIR will support OpenMP in addition to MPI and will use PAPI to access hardware performance counters. With PAPI, VAMPIR will be able to use hardware counter data to guide detailed event based analysis which will allow users to quickly locate performance bottlenecks in their applications. VAMPIR users previously were able only to scan the visualization of a trace file and attempt to correlate events in that view with cumulative statistics displays. Displaying various PAPI metrics alongside the VAMPIR timeline view will allow users to quickly pinpoint the location of performance bottlenecks.

The **KAPPro** toolkit, developed by Intel/KAI for OpenMP programming, consists of the Assure debugger, the Guide OpenMP compiler, and the GuideView OpenMP performance analysis tool. Pallas and Intel/KAI are developing a new performance analysis tool set for combined MPI and OpenMP programming which uses PAPI to access the hardware performance counters. PAPI's standard performance metrics, which include metrics for shared memory processors (SMPs), will provide accurate and relevant performance data for the clustered SMP environments targeted by the new tool set.

Conclusions

PAPI aims to provide the tool designer and application engineer with a consistent interface and methodology for use of the performance counter hardware found in most major microprocessor lines. The main motivation for this interface is the increasing divergence of application performance from near peak performance of most machines in the HPC marketplace. This performance gap is largely due to the disparity in memory and communication bandwidth at different levels of the memory hierarchy. With no viable hardware solution in sight, users requiring the optimal performance must expend significant effort on single processor and shared memory optimization techniques. To address this problem users need a compact set of robust and useful tools to quickly diagnose and analyze processor specific performance metrics. To that end, many design efforts have wastefully reinvented the software infrastructure necessary for a suite of program analysis tools. PAPI directly challenges this model by focusing on a reusable, portable and functionality-oriented infrastructure for performance tool design. It is hoped that through additional collaborative efforts, PAPI will become one of a number of modular components for advanced tool design and program analysis.

The PAPI specification and software, as well as documentation and additional supporting information, are available from the PAPI web site at http://icl.cs.utk.edu/projects/papi.

References

[1] Browne, S., J. Dongarra, N. Garner, G. Ho, and P. Mucci. "A Portable Programming Interface for Performance Evaluation on Modern Processors", *International Journal of High-Performance Computing Applications* 14:3, Fall 2000, pp. 189-204.

[2] Browne, S., J. Dongarra, N. Garner, K. London, and P. Mucci, "A Scalable Cross-Platform Infrastructure for Application Performance Optimization Using Hardware Counters", *SC'2000*, Dallas, Texas, November, 2000.

[3] Luiz DeRose and Daniel A. Reed. "SvPablo: A Multi-Language Perfomrance Analysis System", Proceedings of the 1999 International Conference on Parallel Processing, September 1999, pp. 311-318.

[4] Luiz DeRose, Ying Zhang, and Daniel A. Reed. "SvPablo: A Multi-Language Performance Analysis System", 10th International Conference on Computer Performance Evaluation – Modeling Techniques and Tools – Performance Tools'98, pp. 352-355. Palma de Mallorca, Spain, September 1998. http://vibes.cs.uiuc.edu/

[5] Curtis L. Janssen. The Visual Profiler, Version 0.4, October 1999. http://aros.ca.sandia.gov/~cljanss/perf/vprof/doc/README.html