

Integrated Tool Capabilities for Performance Instrumentation and Measurement

Sameer Shende, Allen Malony
Department of Computer and Information Science
University of Oregon
sameer@cs.uoregon.edu, malony@cs.uoregon.edu

Shirley Moore, Philip Mucci, Jack Dongarra
Department of Computer Science
University of Tennessee
shirley@cs.utk.edu, mucci@cs.utk.edu, dongarra@cs.utk.edu

Abstract:

As computer systems grow in size and complexity, tool support is needed to facilitate the efficient mapping of large-scale applications onto these systems. To help achieve this mapping, performance analysis tools must provide robust performance observation capabilities at all levels of the system, as well as map low-level behavior to high-level program constructs. This paper describes instrumentation and measurement strategies, together with a suite of performance analysis tools that implement these strategies.

Keywords: performance evaluation, hardware counters, dynamic instrumentation, profiling

1.0 Introduction

As computer systems grow in size and complexity, tool support is needed to facilitate the efficient mapping of large-scale applications onto these systems. To help achieve this mapping, performance analysis tools must provide robust performance observation capabilities at all levels of the system, as well as map low-level behavior to high-level program constructs. Performance observation requirements are determined by the performance problem being addressed and the performance evaluation methodology being applied.

Instrumentation of the application is necessary to capture performance data. Instrumentation may be inserted at various stages, from source code modifications to compile-time to link-time to modification of executable code either statically or dynamically during program execution. These instrumentation points have different mechanisms which vary in their ease of use and in the flexibility, level of detail, and user control of what data can be collected.

Performance data of various types can provide valuable insights into program behavior and point the way toward program transformations that will improve performance. Profiling data shows the distribution of a metric across source-level constructs, such as routines, loops, and basic blocks. In addition to timing facilities such as cycle counters, most modern microprocessors provide a rich set of hardware counters that capture functional unit, memory, and operating system events. Profiling can be based on either time or various hardware-based metrics, such as cache misses, for example. Correlations between profiles based on different events, as well as event-based ratios, provide derived information that can help to quickly identify and diagnose performance problems. In addition to profiling data, capturing event traces of program events, such as message communication events, helps portray the temporal dynamics of application performance.

A wide range of performance problems, performance evaluation methods, and programming environments need to be supported. A suite of tools, rather than one specific tool, can best provide the necessary flexibility in experiment design and in

selection and control of experiment mechanisms. Section 2 describes the instrumentation mechanisms it is desirable to support. Section 3 describes various types of measurements. Both these sections include examples of how the instrumentation and measurement strategies are implemented in the PAPI and TAU tool suites. Section 4 gives application examples to illustrate how the tools use these strategies to solve performance problems. Section 5 contains conclusions.

2.0 Instrumentation

To observe application performance, additional instructions or probes are typically inserted into a program. This process is called instrumentation. The execution of a program is regarded as a sequence of significant events. As events execute, they activate the probes which perform measurements. Thus, instrumentation exposes key characteristics of an execution.

2.1 Desired Features

For wide applicability of a performance tool, it is desirable for the tool to allow instrumentation of applications written in several languages such as Fortran90, C++, C and Java. It should also support multiple compilers (vendor, commercial, free) on each platform and be ported to several platforms. The tool should also support different threading models (such as OpenMP, pthread, sproc, Java) etc. Profiling tools that insert instrumentation should have an option for manual overrides to provide greater flexibility. The tool should allow selection of events to minimize perturbation and provide facilities for grouping events for relating low-level events to higher levels of abstraction. Runtime enabling and disabling of events is another desirable feature as is online viewing of performance data. The TAU profiling and tracing toolkit aims to provide these and other features. It has three distinct phases for instrumentation, measurement and analysis as shown in figure 1 below.

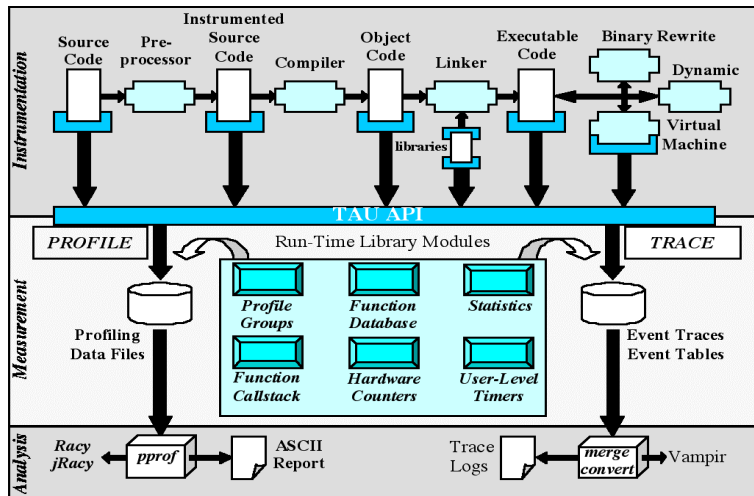


Figure 1: Architecture of TAU

The source code of a program undergoes a series of transformations before it executes as shown in Figure 1. Instrumentation can be added to the program at any of these levels, each of which imposes different constraints and opportunities for extracting program information. As information flows through these levels, different aspects of the program can be revealed. As we move from source code instrumentation techniques to binary instrumentation techniques, our focus shifts from a language specific to a more platform specific instrumentation approach.

2.2 Source code instrumentation

Instrumentation at the source code level has several advantages. It allows the programmer to communicate higher-level domain-specific abstractions to the performance tool. This is especially useful if a tool cannot automatically infer such information. A programmer can communicate such events by annotating the source code at appropriate locations with instrumentation calls. This is easily done at the source level, but may be significantly more difficult elsewhere. Once the program undergoes a series of transformations to generate the executable code, specifying arbitrary points in the code for instrumentation and understanding program semantics at those points may not be possible.

Another advantage of source code instrumentation is that once an instrumentation library targets one language, it can provide portability across multiple compilers for that

language, as well as across multiple platforms. The API is independent of details below the compiler such as operating system dependencies and object file formats. For example, the Performance API (PAPI) project provides a portable library interface to the hardware performance counters available on most modern microprocessors [1]. PAPI provides both easy to use high level routines (start, stop, read, and flops calls), as well as a detailed programmable low-level interface that provides full access to the hardware, for use by application programmers to instrument their application source code.

Source code annotations can be inserted manually or automatically. Adding instrumentation calls in the source code manually can be a tedious task that introduces the possibility of instrumentation errors that can produce erroneous performance data. For example, a user may overlap timers in the source code. While syntactic errors can be caught during compilation, logical errors in instrumentation may be more difficult to detect; overlapping timers, for instance, can only be detected at runtime [2].

Some of these difficulties with source code instrumentation can be overcome by using a preprocessor. A preprocessor is implemented as a source-to-source translation that tool typically expands header files and performs macro substitutions during compilation. Such source-to-source transformations can be used to build an *instrumentor* that automatically introduces instrumentation, alleviating the burden on the programmer.

Preprocessor level instrumentation is commonly used to insert performance measurement calls at routine entry and exit points in the source code. To do this, a tool first needs to parse the application source code and locate the semantic constructs that are to be instrumented (such as routines, loops or individual statements). To insert code, the instrumentor also needs an interface to the parsed internal representation of the source code. Tools such as PDT [3] for C++, C and Fortran 90, Sage++ [4] for C++ and SUIF [5] for C and Fortran provide an object-oriented class library to access the data structures that represent the parsed intermediate form. In the TAU project, we have developed a source-to-source instrumentor (*tau_instrumentor*) using PDT. It supports instrumentation of C, C++ and F90 programs. The instrumented source code is then compiled and linked with the TAU performance measurement library.

Source code instrumentation requires access to the source code for instrumentation.

2.2.1 Instrumentation of system calls

While source code may not be available for some libraries, instrumentation at this level can still be used at call-sites, or points in the application source code where the library routines are called. Typically this is accomplished by a preprocessor that replaces the library call with a call to an instrumented version. For languages such as C and C++ which have a preprocessor within the compiler, a specialized preprocessor can be avoided. In TAU, a header file can be used to define macros that re-define the native library routines (e.g., `open`) with instrumented routines (e.g., `tau_open`). During compilation, the compiler preprocessor replaces calls to proprietary library routines with calls to the instrumented, wrapper libraries which perform measurements and call the appropriate library routines. This scheme does not require access to the library source code. However, it does require a minor modification (addition of a header file) to the application sources and requires the instrumented version of the library to be linked with the other object files and libraries. The main limitation of this approach is that it can only capture information about instances of library calls at specific call-sites that are redirected. If a pre-compiled library routine makes references to a wrapped routine, it is not possible to re-direct such a reference without access to its source code and re-compiling it with the appropriate header file.

Another tactic is to use the preload mechanism of a dynamic loader to start monitoring tools at process or thread creation time. A trap is created that calls a tool initialization routine in conjunction with executing the normal thread or process creation routine. This tactic is used in the *trapper* tool distributed with PAPI.

2.3 Library level instrumentation

Wrapper interposition libraries provide a convenient mechanism for adding instrumentation calls to libraries. A good example of this approach is found in the

Message Passing Interface (MPI) Profiling Interface [6]. MPI, a standard for inter-process message communication, is commonly used to implement parallel SPMD programs. As part of its definition, it includes alternative entry points for MPI routines. The MPI Profiling Interface allows a tool developer to interface with MPI calls in a portable manner without modifying the application source code and having access to the proprietary source code of the library implementation.

The MPI standard defines the native library routine with weak bindings and a name shifted interface. A weak binding allows two different routines with the same name to co-exist in a binary executable. If a tool re-defines the native call, it takes precedence. In this manner, a performance tool can provide an interposition library layer that intercepts calls to the native MPI library by defining routines with the same name (e.g., `MPI_Send`). These routines wrap performance instrumentation around a call to the name-shifted native library routine provided by the MPI profiling interface (e.g., `PMPI_Send`). The exposure of routine arguments allows a tool to track the sizes of messages and message tags as in TAU[7].

The POMP interface for OpenMP provides a performance API target for source-to-source instrumentation tools (e.g., `Opari`) allowing for instrumented OpenMP codes that are portable across compilers and machine platforms [8]. Defined as a library API, the interface exposes OpenMP execution events of interest (e.g., sequential, parallel, and synchronization events) for performance observation, and passes OpenMP context descriptors to inform the performance interface library of region-specific information. The OPARI tool rewrites OpenMP directives in functionally equivalent, but source-instrumented forms, inserting POMP performance calls where appropriate. TAU uses the POMP interface in conjunction with OPARI for instrumentation of OpenMP programs.

2.4 Binary instrumentation

Executable images can be instrumented using binary code-rewriting techniques, often referred to as binary editing tools or executable editing tools. Systems such as `Pixie`, `EEL` [9] and `PAT` [10] include an object code instrumentor that parses an executable and

rewrites it with added instrumentation code. PAT can be used for call-site profiling and instrumentation at the routine level as well as gathering routine level hardware performance statistics. Pixie can count basic blocks within a routine and EEL can instrument a routine's control flow graph and instructions. In each case, the executable file is re-written with the appropriate instrumentation inserted.

The advantage of binary instrumentation is that there is no need to re-compile an application program and rewriting a binary file is mostly independent of the programming language. Also, it is possible to spawn the instrumented parallel program the same way as the original program, without any special modifications as are required for runtime instrumentation [11].

2.5 Dynamic Instrumentation

Dynamic instrumentation is a mechanism for runtime-code patching that modifies a program during execution. DyninstAPI [12] provides an efficient, low-overhead interface that is suitable for performance instrumentation. A tool that uses this API (also known as a mutator) can insert code snippets into a running program (also known as a mutatee) without re-compiling, re-linking or even re-starting the program. The mutator can either spawn an executable and instrument it prior to its execution, or attach to a running program. DyninstAPI inserts instrumentation code snippets in the address space of the mutatee. The code snippets can load dynamic shared objects in the running application and call routines, as well as read and write application data. The DyninstAPI translates code snippets into machine language instructions in the address space of the mutatee. It generates code to replace an instruction in the mutatee with a branch instruction to the instrumentation code. The replaced instruction calls a base trampoline which branches to a mini trampoline. A mini trampoline saves the registers and executes the code snippet(s) with appropriate arguments. Thereafter, it restores the original registers and calls the base trampoline. The base trampoline executes a relocated instruction and returns to the statement after the replaced instruction in the original code as described in [12].

Dynamic instrumentation overcomes some limitations of binary instrumentation by allowing instrumentation code to be added and removed at runtime. Also, the instrumentation can be done on a running program instead of requiring the user to re-

execute the application. The disadvantage of dynamic instrumentation is that the interface needs to be aware of multiple object file formats, binary interfaces (32/64 bit), operating system idiosyncrasies, as well as compiler specific information (e.g., to support template name de-mangling in C++ from multiple C++ compilers). To maintain cross language, cross platform, cross file format, cross binary interface portability is a challenging task and requires a continuous porting effort as new computing platforms and multi-threaded programming environments evolve.

TAU uses DyninstAPI to construct calls to the TAU measurement library and then insert these calls into the executable code. This is done by a mutator program (*tau_run*). The mutator loads the TAU dynamic shared object (the compiled TAU measurement library) in the addresses space of the mutatee. It parses the executable imkage for symbol table information and generates the list of modules and routines within the modules that are appropriate for instrumentation. The user can optionally provide a selective instrumentation list that specifies a list of routines for including or excluding from instrumentation. TAU can instrument sequential as well as parallel (MPI) programs [11].

The *dynaprof* tool distributed with PAPI uses dynamic instrumentation to allow the user to either load an executable or attach to a running executable and then dynamically insert instrumentation probes [13]. The user can list the internal structure of the application in order to select instrumentation points. Dynaprof inserts instrumentation in the form of *probes*. Dynaprof provides a PAPI probe for collecting hardware counter data and a wallclock probe for measuring elapsed time, both on a per-thread basis. Users may optionally write their own probes. A probe may use whatever output format is appropriate, for example a real time data feed to a visualization engine or a static data file dumped to disk at the end of the run.

3.0 Types of measurements

Post-mortem performance evaluation tools traditionally fall into two categories: profiling and tracing.

3.1 Profiling

Profiling characterizes the behavior of an application in terms of aggregate performance metrics. Profiles are typically represented as a list of various metrics (such as inclusive/exclusive wall-clock time) that are associated with program-level semantic entities (such as routines or statements in the program). Time is a common metric, but any monotonically increasing resource function can be used. Profiling can be implemented by sampling or instrumentation based approaches.

Sampling-based profiling periodically records the program state, and based on measurements made on those states, estimates the overall performance. Although sampling-based schemes suffer from incomplete coverage of the application and their accuracy depends on the sampling interval, they have a distinct advantage of fixed, low instrumentation overhead and consequently reduced measurement perturbation in the program.

In instrumentation-based profiling, measurements are triggered by the execution of instructions added to the code to track significant events in the program (such as the entry or exit of a routine, the execution of a basic block or statement, the send or receipt of a message communication operation). Typically, such profilers present the cost of executing different routines in a program.

3.1.1 Time based measurements

Profilers can make measurements of exclusive and inclusive time spent in a routine. For timing, wallclock time, process virtual time or CPU time are commonly used metrics. TAU uses PAPI to access the above timing information.

3.1.2 Hardware counter based measurements

Instead of timing information, profilers can use counts from hardware performance counters that PAPI provides. This allows profilers to present exclusive and inclusive counts of low-level counters for each instrumented routine.

3.2 Tracing

While profiling is used to get aggregate summaries of metrics in a compact form, it cannot highlight the time varying aspect of the execution. To study the post-mortem spatial and temporal aspect of performance data, event tracing, that is, the activity of capturing an event or an action that takes place in the program, is more appropriate. Event tracing usually results in a log of the events that characterize the execution. Each event in the log is an ordered tuple typically containing a time stamp, a location (e.g., node, thread), an identifier that specifies the type of event (e.g., routine transition, user-defined event, message communication, etc.) and event-specific information. In a parallel execution, trace information generated on different processors must be merged. This is usually based on the time-stamp which can reflect logical time or physical time. The logical time uses local counters for each process incremented when a local event takes place. The physical time uses a globally synchronized real time clock. TAU can produce event-traces that are merged and can be converted to the ALOG, SDDF, Paraver or Vampir trace formats.

3.3 Real-time performance monitoring

Post-mortem analysis of profiling data or trace files has the disadvantage that analysis cannot begin until after program execution has finished. Real-time performance monitoring allows users to evaluate program performance during execution.

Real-time performance monitoring is supported by the *perfometer* tool that is distributed with PAPI . By connecting the graphical display to the backend process (or processes) running the application code that has been linked with the *perfometer* and PAPI libraries, the tool provides a runtime trace of a chosen PAPI metric, as shown in figure 2 for floating point operations per second (PAPI_FLOPS). The user may change the performance event being measured by clicking on the appropriate button. The intent of *perfometer* is to provide a fast coarse-grained easy way for a developer to find out where a bottleneck exists in a program.

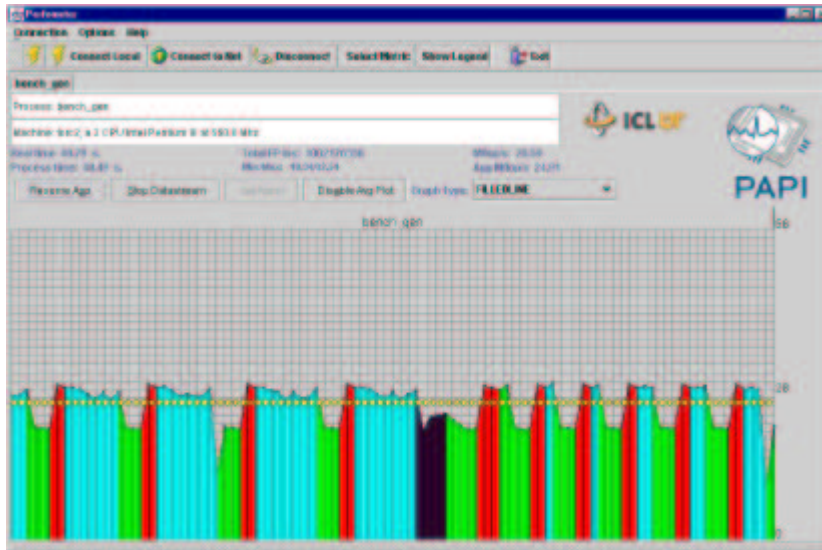


Figure 2. Perfomater real-time display

5.0 Examples

In this section we illustrate how PAPI and TAU can be used to understand the performance of an application.

5.1 Matrix Multiply

<pre> TAU_PROFILE_TIMER(t1, "multiply-regular", "void (void)"); for (n = 0; n < SIZE; n++) for (m = 0; m < SIZE; m++) { A[n][m] = B[n][m] = n + m; C[n][m] = 0; } TAU_PROFILE_START(t1); for (i = 0; i < SIZE; i++) for (j = 0; j < SIZE; j++) for (k = 0; k < SIZE; k++) C[i][j] += A[i][k] * B[k][j]; TAU_PROFILE_STOP(t1); /* FIG A: ORIGINAL */ </pre>	<pre> TAU_PROFILE_TIMER(t1, "multiply-regular", "void (void)"); for (n = 0; n < SIZE; n++) for (m = 0; m < SIZE; m++) { A[n][m] = B[n][m] = n + m; C[n][m] = 0; } TAU_PROFILE_START(t1); for (i = 0; i < SIZE; i++) for (k = 0; k < SIZE; k++) /* ijk->ikj */ for (j = 0; j < SIZE; j++) C[i][j] += A[i][k] * B[k][j]; TAU_PROFILE_STOP(t1); /* FIG B: LOOP INTERCHANGE */ </pre>
<pre> TAU_PROFILE_START(strip_timer); for (i = 0; i < SIZE; i++) for (sz = 0; sz < SIZE; sz += CACHE) { //vl = min(SIZE-sz, CACHE); vl = (SIZE - sz < CACHE ? SIZE - sz : CACHE); for (strip = sz; strip < sz+vl; strip++) for (k = 0; k < SIZE; k++) C[i][strip] += A[i][k] * B[k][strip]; } TAU_PROFILE_STOP(strip_timer); /* FIG C: STRIP MINING OPTIMIZATION */ </pre>	<pre> TAU_PROFILE_START(strip_timer); for (i = 0; i < SIZE; i++) for (sz = 0; sz < SIZE; sz += CACHE) { //vl = min(SIZE-sz, CACHE); vl = (SIZE - sz < CACHE ? SIZE - sz : CACHE); for (k = 0; k < SIZE; k++) for (strip = sz; strip < sz+vl; strip++) C[i][strip] += A[i][k] * B[k][strip]; } TAU_PROFILE_STOP(strip_timer); /* FIG D: LOOP INTERCHANGE WITH STRIP MINING */ </pre>

Figure 3: Matrix multiply algorithm coded using four techniques

In the above figure 3, we see the original code (Figure A, top left) of a simple matrix multiply algorithm. The main matrix multiply loop that computes the dot product of two

matrices is instrumented with TAU. When it is executed on a Pentium III/500 MHz Xeon processor, we can see the wallclock time (152.85 secs), the total floating point operations (2.14 E9) and the data cache misses (1.11 E9) reported by TAU using PAPI, as shown in figure 4 below. We can verify that the number of floating point instructions executed matches our expected value (for problem size $n=1024$, $2*n^3 = 2.1E9$).

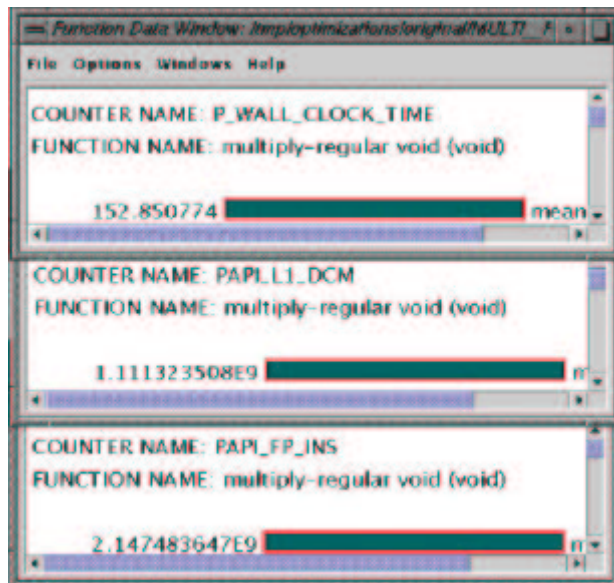


Figure 4: Wallclock time, level 1 data cache misses and floating point instructions executed by the original program.

To optimize this program, we perform the loop-interchange optimization in an effort to reduce the data cache misses. The loops j and k are reversed to produce the code in Figure (B). PAPI and TAU allow us to evaluate the effect of this change to the source

code. Below, we show the performance data associated with this change.

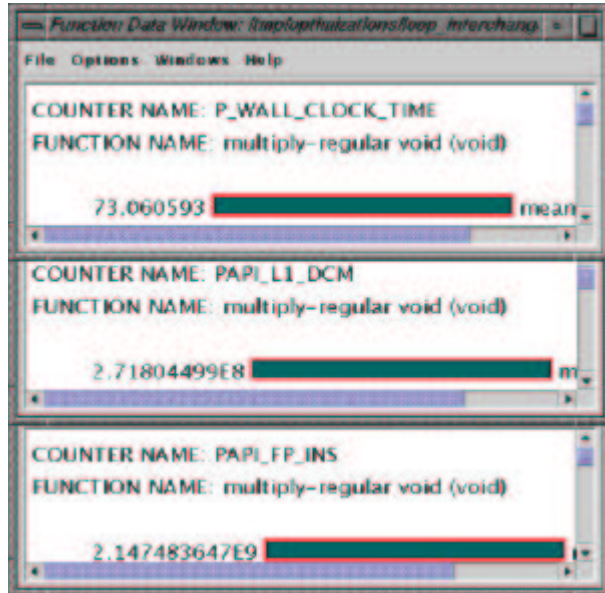


Figure 5: Metrics for loop interchange optimization

With this change, the wallclock time reduces to 73.06 secs, and we see an order of magnitude decrease in the number of data cache misses while retaining the number of floating point instructions.

To further reduce the data cache misses, the strip mining optimization is applied to the original code (figure3 (C)) by performing the computation on strips of size 128 (CACHE in the code). The performance data obtained for this change are shown below in Figure 6.

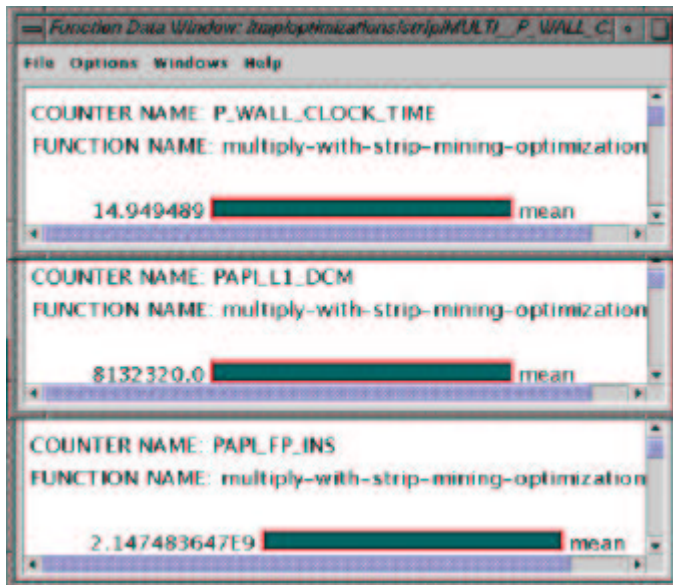


Figure 6: strip mining optimization data

We observe that this has a dramatic effect and reduces the number of data cache misses from $1E9$ to $8.1E6$. This reduces the wallclock time from 152 secs to 14.94 secs while keeping the same floating point instructions.

Finally, we combine the loop interchange and strip mining optimizations to produce code in Figure 3 (D). The performance data for the main loop is shown below in figure 7.

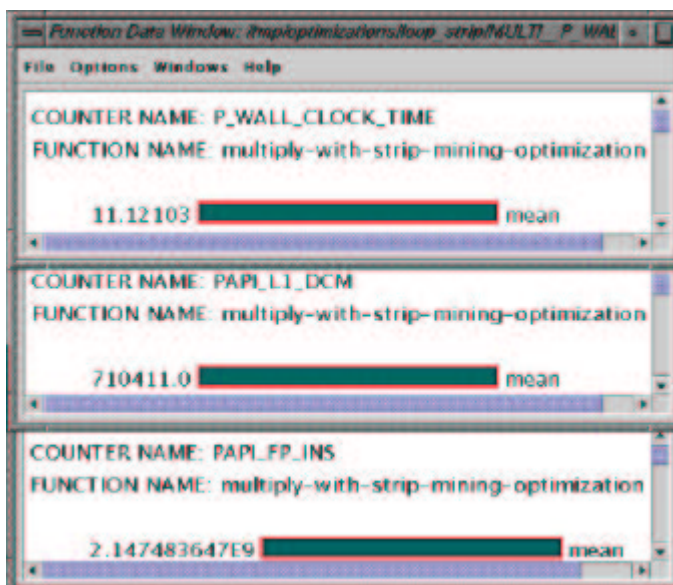


Figure 7: Combining loop interchange with strip mining

We notice that the time taken reduces to 11.12 secs and the number of data cache misses to 7.1E5.

This example illustrates how we can observe the effect of changes in the source code on the performance of the application. To conduct this experiment TAU is configured with PAPI to measure hardware performance counters, wallclock time and process virtual time.

5.2 PETSc

PETSc (The Portable, Extensible Toolkit for Scientific Computation) is a suite of data structures and routines for the scalable (parallel) solution of scientific applications modeled by partial differential equations. It employs the MPI standard for all message-passing communication. To evaluate the performance of a 2-d driven cavity code that uses a velocity-vorticity formulation and a finite difference discretization on a structured grid, we use PAPI and TAU and PDT. The source code of PETSc is written in C. TAU's automatic source code instrumentation technique based on PDT is used to instrument PETSc source code. Also, TAU's MPI wrapper interposition library instruments all MPI calls. This allows us to deploy instrumentation at two different levels that share the same API for measurement. The instrumented sources are compiled and linked with the TAU measurement library. In this case, we configure TAU to use PAPI for accessing wallclock time and hardware performance counters. TAU provides selective instrumentation capabilities that allow us to identify lightweight routines that have a high instrumentation overhead. These are removed by re-instrumenting the source code and automatically generating an exclude list that allows TAU to ignore this set of routines while instrumenting others. This helps produce accurate performance data for the remaining routines.

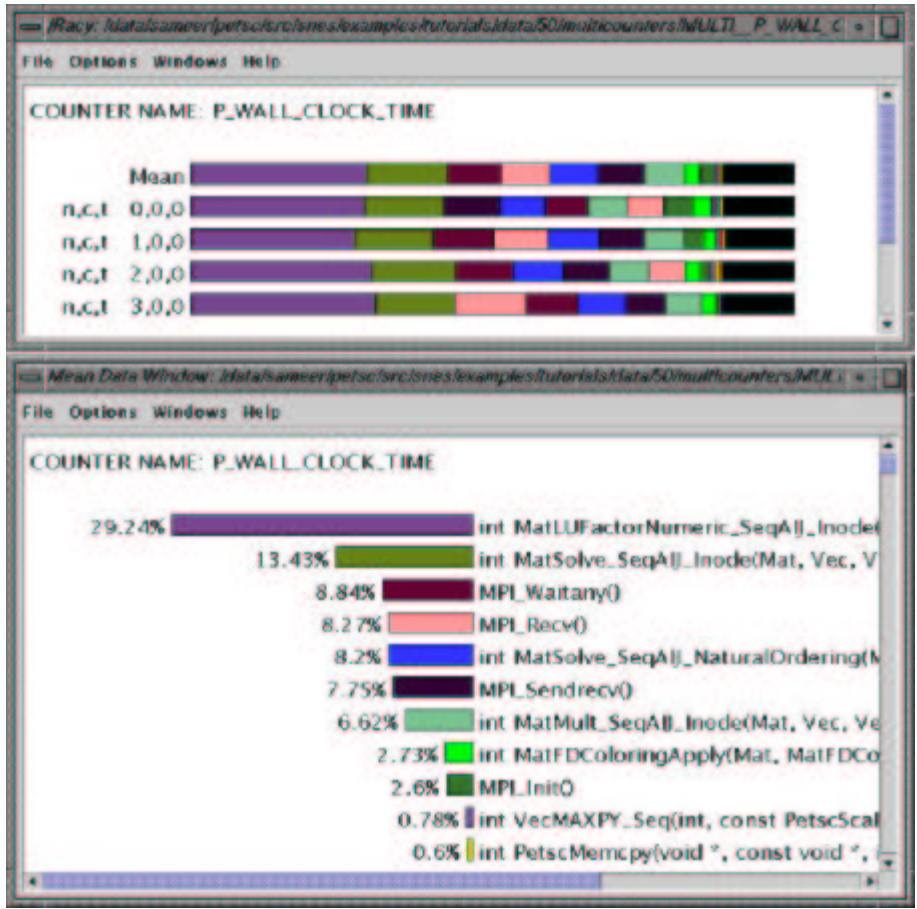


Figure 8: Wallclock time experiment with PETSc

Figure 8 shows the exclusive wallclock time spent in all nodes. The application was run with a grid size of 50x50 on four Pentium III/550 MHz processors.

The floating point instructions executed by the code are shown below in Figure 9.

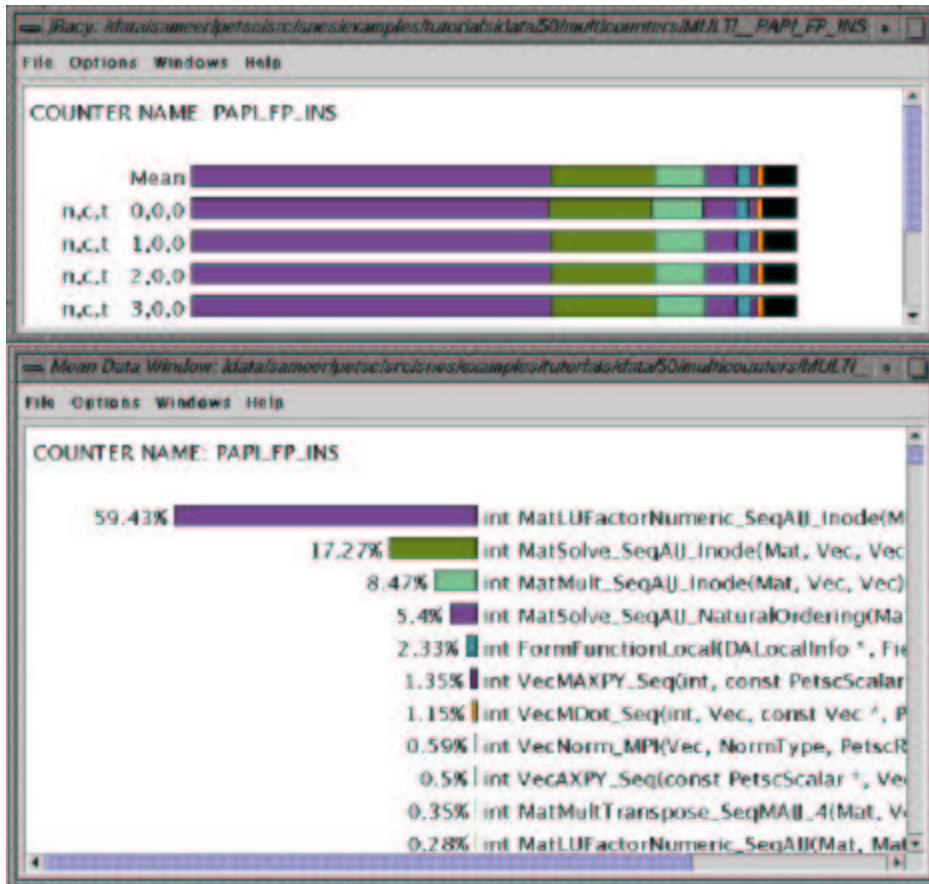


Figure 9: Floating point instructions executed by the example

Here, we see that the routine `MatLUFactorNumeric_SeqAIJ_Inode` executes 59% of floating point instructions. To examine the cache characteristics of the code we look at the level 1 data cache misses data produced by PAPI and TAU. These data are shown in figure 10 below.

Tracing shows us global timeline views where interprocess message communication is represented by line segments as shown in Figure 12.

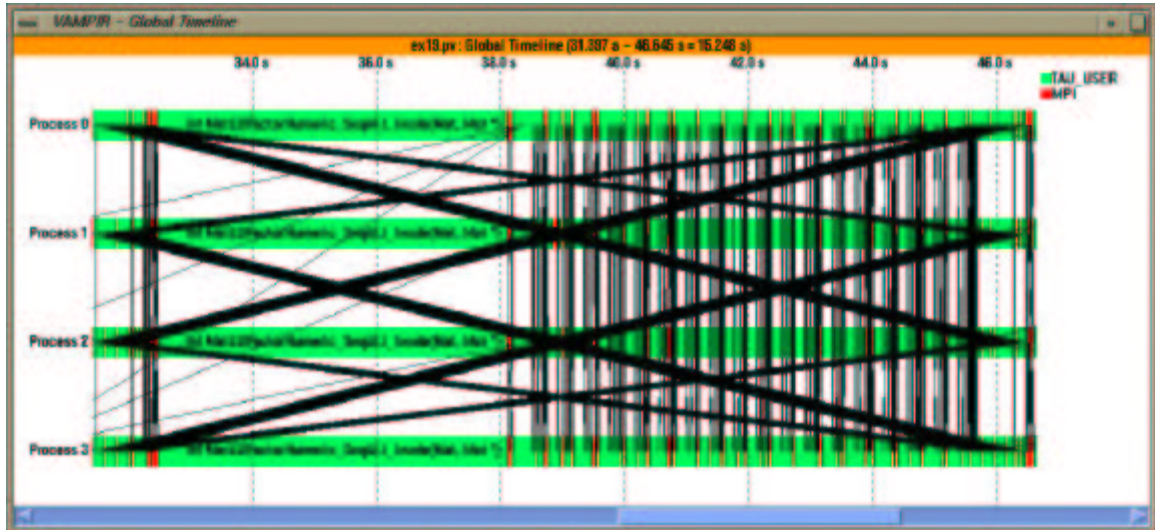


Figure 12: Global timeline view of PETSc

The event-traces generated by TAU are visualized in Vampir [Vampir], a commercial trace visualization tool from Pallas GmbH.

With the help of these two examples, we show how PAPI and TAU can be used together to generate a wealth of performance data that can guide optimization decisions.

5.0 Conclusions

Performance analysis on today's complex computer systems requires robust tool capability for flexible and portable performance instrumentation and measurement. We have described a framework for instrumentation and measurement of applications, as well as some examples from a suite of tools that implement parts of this framework. PAPI and TAU have been integrated to be able to produce a range of profiling data based on both time and hardware counter metrics. We have illustrated with two examples how such data can be used to analyze and tune application performance.

References

1. Browne, S., et al., *A Portable Programming Interface for Performance Evaluation on Modern Processors*. International Journal of High-Performance Computing Applications, 2000. **14**(3): p. 189-204.
2. Shende, S., et al. *Portable Profiling and Tracing for Parallel Scientific Applications using C++*. in *ACM SIGMETRICS Symposium on Parallel and Distributed Tools*. 1998.
3. Lindlan, K., et al. *A Tool Framework for Static and Dynamic Analysis of Object-Oriented Software with Templates*. in *SC 2000*. 2000. Dallas, Texas.
4. Bodin, F., et al. *Sage++: an Object-oriented Toolkit and Class Library for Building Fortran and C++ Restructuring Tools*. in *Second Annual Object-Oriented Numerics Conference*. Second Annual Object-Oriented Numerics Conference.
5. Wilson, R., et al., *SUIF: An Infrastructure for Research on Parallelising and Optimizing Compilers*. ACM SIGPLAN Notices, 1994. **29**(12): p. 31-37.
6. *MPI: A Message Passing Interface Standard*. International Journal of Supercomputing Applications, 1994. **8**(3/4).
7. Malony, A.D. and S. Shende, *Performance Technology for Complex Parallel and Distributed Systems*, in *Distributed and Parallel Systems: From Instruction Parallelism to Cluster Computing*, P. Kacsuk and G. Kotsis, Editors. 2000, Kluwer: Norwell, MA. p. 37-46.
8. Mohr, B., et al., *Design and Prototype of a Performance Tool Interface for OpenMP*. Journal of Supercomputing, 2002. **23**: p. 105-128.
9. Larus, J. and T. Ball, *Rewriting Executable Files to Measure Program Behavior*. Software Practice and Experience, 1994. **24**(2): p. 197-218.
10. Galarowics, J. and B. Mohr, *Analyzing Message Passing Programs on the Cray T3E with PAT and VAMPIR*. 1998, ZAM Forschungszentrum: Juelich, Germany.
11. Shende, S., A.D. Malony, and R. Ansell-Bell. *Instrumentation and Measurement Strategies for Flexible and Portable Empirical Performance Evaluation*. in *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2001)*. 2001: CSREA.

12. Buck, B. and J. Hollingsworth, *An API for Runtime Code Patching*. Journal of High Performance Computing Applications, 2000. **14**(4): p. 317-329.
13. Mucci, P., *Dynaprof*. <http://www.cs.utk.edu/~mucci/dynaprof>, July 2002.