ADVANCES IN THE TAU PERFORMANCE SYSTEM

Allen D. Malony, Sameer Shende, Robert Bell, Kai Li, Li Li, Nick Trebon
University of Oregon
{malony,sameer,bertie,likai,lili,ntrebon}@cs.uoregon.edu

Abstract To address the increasing complexity in parallel and distributed systems and software, advances in performance technology towards more robust tools and broader, more portable implementations are needed. In doing so, new challenges for performance instrumentation, measurement, analysis, and visualization arise to address evolving requirements for how performance phenomena is observed and how performance data is used. This paper presents recent advances in the TAU performance system in four areas where improvements in performance technology are important: instrumentation control, performance mapping, performance interaction and steering, and performance databases. In the area of instrumentation control, we are concerned with the removal of instrumentation in cases of high measurement overhead. Our approach applies rule-based analysis of performance data in an iterative instrumentation process. Work on performance mapping focuses on measuring performance with respect to dynamic calling paths when the static callgraph cannot be determined prior to execution. We describe an online performance data access, analysis, and visualization system that will form the basis of a large-scale performance interaction and steering system. Finally, we describe our approach to the management of performance data in a database framework that supports multi-experiment analysis.

Keywords: Performance, tools, parallel, distributed.

1. Introduction

There has long been a tension in the field of performance tools research between the need to invent new techniques to deal with performance complexity of next-generation parallel and distributive systems, and the need to develop tools that are robust, both in their function and in their scope of application. Perhaps this is just the nature of the field. Yet there are important issues concerning the advancement of performance tools "research" and the successful demonstration and use of performance "technology." A cynical perspective might argue against trying something new without first getting the existing technology to just work, and work reliably in real applications and environments. The all too commonly heard mantras "performance tools don't work" and "performance tools are too hard to use" might lead one to believe in this perspective, but research history does not necessarily justify such a strong cynical stance. There has been significant innovation in performance observation and analysis techniques in the last twenty year to address the new performance challenges parallel computing environments present [10]. Current attention is certainly being paid to easing the burden of tool use through automated analysis [1]. There have also been important technology developments that add considerable value to the performance tool repertoire, such as APIs for dynamic instrumentation [4] and hardware performance counters [3]. Why, then, is there an apparent disconnect between research results and the "reality" of tool usage in parallel application environments?

From our perspective as performance tool researchers, we take, perhaps, a controversial stance among our peers and argue that tool engineering is an important factor in this regard. The controversial part primarily concerns the notion of "research" and the rewards (or lack thereof) in a research career for tool development. Our counter position is that innovation in performance tools research is best advanced by "standing on the shoulders" of solid technology foundations. When that foundation does not exist, it must be developed. When a technology does exist, it should be integrated, if possible, and not reinvented. Indeed, many tools do not work reliably and, as a consequence, are hard to use. Many tools are not portable across parallel systems or reusable with different programming paradigms, and, as a consequence, have limited application. These results cannot be considered as positive results for the performance tool research community, that is, if reliability, portability, and robustness, in general, is considered worthy of research. We believe that they are, particularly in parallel computing. Furthermore, we contend that the future advances in performance tools research with the most direct potential effect in real application will be those that can best leverage and amplify existing robust performance technology.

In this paper, we consider four research problems being investigated in the TAU parallel performance system [9, 17] and describe the performance tools being developed to address them. These tools build on and leverage the capabilities in TAU (as well as the other technologies integrated in TAU) to provide robust, value-added solutions. While none of these solutions are necessarily "new," in the sense of a new research finding, the technology being developed is novel and will directly provide new capabilities to TAU users. After a brief description of the TAU performance system, we look at the problem of instrumentation control to reduce measurement overhead. Our work here builds on TAU's rich instrumentation framework. The second problem of callpath profiling requires a solution that maps performance measurements to dynamically occurring callpaths. Here, TAU's performance mapping API is utilized. Providing online performance analysis and visualization for large-scale parallel applications is the third problem we consider. Finally, we describe our early work to develop a performance database framework that can support multi-experiment performance analysis.

2. TAU Performance System

For the past twelve years, the TAU project has conducted research on performance tools for parallel and distributed systems. The goal of this work has mainly been the development of robust technology to meet evolving performance evaluation challenges of state-of-the-art parallel systems and applications. In particular, we have focused on problems of performance tool portability, extendability, and interoperation.

The TAU performance system [9, 17] is our integrated toolkit for performance instrumentation, measurement, analysis, and visualization of large-scale parallel applications. It targets a general computation model consisting of shared-memory computing *nodes* where *contexts* reside, each providing a virtual address space shared by multiple *threads* of execution. The model is general enough to apply to many high-performance scalable parallel systems and programming paradigms. Because TAU enables performance information to be captured at the *node/context/thread* levels, this information can be mapped to the particular parallel software and system execution platform under consideration.

As shown in Figure 1, the TAU system supports a flexible instrumentation model that applies at different stages of program compilation and execution. The instrumentation targets multiple code points, provides for mapping of low-level execution events to higher-level performance abstractions, and works with multi-threaded, message passing, and mixed-mode parallel computation models. Different instrumentation techniques are supported, including dynamic instrumentation using the DyninstAPI [4]. All instrumentation code makes

calls to the TAU measurement API to provide a common measurement model. The TAU measurement library implements performance profiling and tracing support for performance events occurring at function, method, basic block, and statement levels. Performance experiments can be composed from different measurement modules (e.g., hardware performance monitors, such as PAPI [3]) and measurements can be collected with respect to user-defined performance groups. C, C++, Fortran 77/90, OpenMP, and Java languages are supported. The TAU data analysis and presentation utilities offer text-based and graphical tools to visualize the performance data as well as bridges to third-party software, such as Vampir [11] and Paraver [12] for sophisticated trace analysis and visualization.

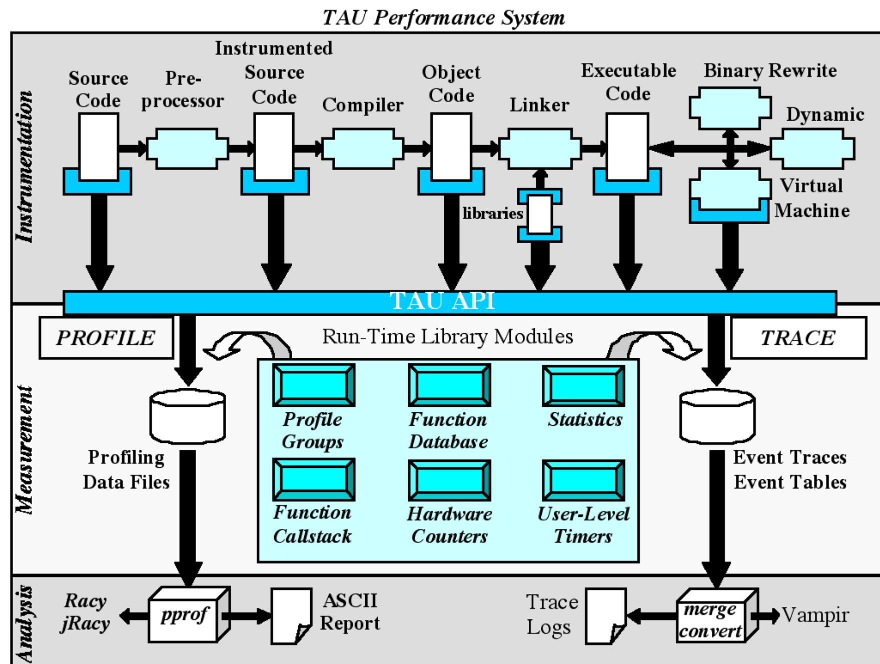

Figure 1. Architecture of the TAU performance system.

TAU has been ported to nearly all high-performance computing platforms and is being used extensively in the performance analysis of DOE applications. TAU is also being applied as the primary performance technology across a diverse set of code development projects, including Uintah [15], CCA [2], VTF [18], and SAMRAI [7]. Although the current set of features in the TAU performance system is quite substantial, it is important to note that users are always requesting new capabilities. The interesting research problems that arise concern how to develop new techniques to address these requests while

maintaining tight integration with the rest of the TAU system. The four problems below are all representative of such endeavors.

3. Measurement Overhead and Instrumentation Control

The selection of what "events" to observe when measuring the performance of a parallel application is an important consideration, as it is the basis for how performance data will be interpreted. The performance events of interest depend mainly on what aspect of the execution the user wants to see, so as to construct a meaningful performance view from the measurements made. Typical events include control flow events that identify points in the program that are executed, or operational events that occur when some operation or action has been performed. Events may be atomic or paired to mark begin and end points, for example, to mark the entry and exit of a routine. Choice of performance events also depends on the scope and resolution of the performance measurement desired. However, the greater the degree of performance instrumentation in a program, the higher the likelihood that the performance measurements will alter the way the program behaves, an outcome termed *performance perturbation*.

Most performance tools, including TAU, address the problem of performance perturbation indirectly using techniques to reduce the performance intrusion (i.e., overhead) associated with performance measurement. This overhead is a result of two factors: 1) the execution time to make the measurement relative to the "size" of the event, and 2) the frequency of event occurrence. The first factor concerns the influence of the measurement overhead on the observed performance of a *particular* event. If the overhead is large relative to the size of the event, the performance measurement is less likely to be accurate unless its overhead is compensated in some way. The overhead is typically measured in execution time, but can also include the impact on other metrics, such as hardware counts. The second factor relates to overheads as seen from the perspective of the entire program. That is, the higher the frequency of events, the larger percentage of the execution will be taken up by performance measurement.

Techniques to control performance intrusion are directed towards making performance measurement more efficient or controlling the performance instrumentation. The former is a product of engineering of the measurement system. That is, the lighter-weight the measurment system, the lower the overhead. Here, we are concerned with controlling performance instrumentation to remove or disable performance measurement for "small" events or events that occur with high frequency. Clearly this will eliminate the overhead otherwise generated, but how are these events determined before a measurment is made? It may be possible for sophisticated source code analysis to identify small code

segments, but this is not a complete solution since the execution time could depend on runtime parameters. Plus, we would like a solution to work across languages and few static analysis tools are available.

Instead, a direct measurement approach will likely be needed. The idea is that a series of instrumentation experiments would be conducted to observe the measurement overhead, weeding out those events resulting in unacceptable levels of intrusion. Whereas this performance data analysis and instrumentation control can be done manually, it is tedious and error-prone, especially when the number of performance events is large. Thus, the problem we addressed was how to develop a tool to help automate the process in TAU.

The TAU performance system instruments an application code using an optional *instrumentation control file* that identifies events for inclusion and exclusion. The TAU instrumentor's default behavior is to instrument every routine and method. Obviously, this instrumentation may result in high measurement overhead, and the user can manually modify the file to eliminate small events, or those that are not interesting to observe. As noted above, this is a cumbersome process. Instead, the *TAUreduce* tool allows the user to write instrumentation rules that will be applied to the parallel measurement data to identify which events to exclude. The output of the tool is a new instrumentation control file with those events de-selected for instrumentation, thereby reducing measurement overhead in the next program run.

Table 1 shows examples of the *TAUreduce* rule language. A *simple rule* is an arithmetic condition written as:

[EventName: | GroupName:] Field Operator Number

where *Field* is a TAU profile metric (e.g., numcalls, percent, usec, usec/call), *Operator* is one of $<$, $>$, or =, and *Number* is any number. A rule applies to all events unless specified explicitly, either by the *EventName* (e.g., routineA) or by the event *GroupName* (e.g., TAU_USER). In the latter case, all events that belong to the group are selected by the rule. A *compound rule* is a logical conjunction of simple rules. Multiple rules, appearing on separate lines, are applied disjunctively.

As a simple example of applying the instrumentation reduction analysis, consider two algorithms to find the kth largest element in a list of N unsorted elements. The first algorithm (*kth_largest_qs*) uses *quicksort* first to sort the list and then selects the kth element. The second algorithm (*select_kth_largest*) scans the list keeping a sorted set of the k largest elements seen thus far. At the end of the scan, it selects the least of the set. We ran the program on a list of 1,000,000 elements with k=2324, first with minimal instrumentation to determine the execution time of the two algorithms: *kth_largest_qs* (.188511

Description	**Rule**
Exclude all events that are members of the TAU_USER group and use less than 100 microseconds	$TAU_USER : usec < 100$
Exclude all events that have less than 100 microseconds and are called only once	$usec < 100\ \&\ numcalls = 1$
Exclude all events that have less than 100 microseconds per call or have a total inclusive percentage less than 5	$usecs/call < 100$ $percent < 5$

Table 1. Examples of TAUreduce rule language.

secs), *select_kth_largest* (.149594 secs). Total execution time was .36 secs on a 1.2 Ghz Pentium III machine.

Then the code was instrumented fully and run again. The profile results are shown in the top half of Figure 2. Clearly, there is significant performance overhead and the execution times are not accurate, even though TAU's per event measurement overhead is very low. We defined the rule

$$usec > 1000\ \&\ numcalls > 400000\ \&\ usecs/call < 30\ \&\ percent > 25$$

and applied *TAUreduce*, eliminating the events marked with "(*)". Running the code again produced the results in the lower half of Figure 2. As seen, the execution times are closer to what we expect.

```
NODE 0;CONTEXT 0;THREAD 0:
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
              msec         msec                          usec/call
---------------------------------------------------------------------------------------
100.0           13        4,982           1           4    4982030 int main
 93.5        3,223        4,659 4.20241E+06 1.40268E+07          1 void quicksort (*)
 62.9      0.00481        3,134           5           5     626839 int kth_largest_qs
 36.4          137        1,813          28      450057      64769 int select_kth_largest
 33.6          150        1,675      449978      449978          4 void sort_5elements (*)
 28.8        1,435        1,435 1.02744E+07           0          0 void interchange (*)
  0.4           20           20           1           0      20668 void setup
  0.0       0.0118       0.0118          49           0          0 int ceil

NODE 0;CONTEXT 0;THREAD 0:
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
              msec   total msec                          usec/call
---------------------------------------------------------------------------------------
100.0           14          383           1           4     383333 int main
 50.9          195          195           5           0      39017 int kth_largest_qs
 40.0          153          153          28          79       5478 int select_kth_largest
  5.4           20           20           1           0      20611 void setup
  0.0         0.02         0.02          49           0          0 int ceil
```

Figure 2. Example application of *TAUreduce* tool.

While the above example is rather simple, the *TAUreduce* tool can be applied to large parallel applications. It is currently being used in Caltech's ASAP ASCI project to control instrumentation in the Virtual (Shock) Test Facility (VTF) [18]. *TAUreduce* is part of the TAU performance system distribution and, thus, is supported on all platforms where TAU is available. It is currently being upgraded to include analysis support for multiple performance counters.

One important comment about this work is that it deals with a fundamentally practical problem in parallel performance observation, that is, the tradeoff of measurement detail and accuracy. By eliminating events from instrumentation, we lose the ability to see those events at all. If the execution of small routines accounts for a large portion of the execution time, that may be hard to discern without measurement. On the other hand, accurate measurement is confounded by high relative overheads. We could attempt to track these overheads at runtime and subtract accummulated overhead when execution time measurements are made. This is something we are pursuing in TAU to increase timing accuracy, but it requires determining a minimum overhead value from measurement experiments on each target platform. Another avenue is to change performance instrumentation on-the-fly as the result of identifying high overhead events. We are also considering this approach in TAU.

4. Performance Mapping and Dynamic Callpath Profiling

To observe meaningful performance events requires placement of instrumentation in the program code. However, not all information needed to interpret an event of interest is available prior to execution. A good example of this occurs in callgraph profiling. Here the objective is to determine the distribution of execution time along the dynamic routine calling paths of an application. A callpath of length k is a sequence of the last $k-1$ routines called. To measure execution time spent on a callpath requires identifying the begin and end points during which a callpath is "active." These points are just the entry and exit of a called routine. If $k=1$, callpath profiling is the measurement of amount of time spent in a routine for each of its calling parents. The basic problem with callpath profiling is that the identities of all k-length calling paths ending at a routine may not, and generally are not, known until the application finishes its execution. How, then, do we identify the dynamic callpath events in order to make profile measurements?

One approach is not to try to not identify the callpaths at runtime, and instead instrument just basic routine entry and exit events and record the events in a trace. Trace analysis can then easily calculate callpath profiles. The problem, of course, with this approach is that the trace generated may be excessively large, particularly for large numbers of processors. Unfortunately, the instru-

mentation and measurement problem is significantly harder if callpath profiles are calculated online.

If the whole source is available, it is possible to determine the entire static callgraph and enumerate all possible callpaths, encoding this information in the program instrumentation. These callpaths are static, in the sense that they could occur; dynamic callpaths are the subset of static callpaths that actually do occur during execution. Once a callpath is encoded and stored in the program, the dynamic callpath can then be determined directly by indexing a table of possible next paths using the current routine id. Once the callpath is known, the performance information can be easily recorded in pre-reserved static memory. This technique was used in the CATCH tool [5].

Unfortunately, this is not a robust solution for several reasons. First, source-based callpath analysis is non-trivial and may be only available for particular source languages, if at all. Second, the application source code must be available if a source-based technique is used. Third, static callpath analysis is possible at the binary code level, but the routine calls must be explicit and not indirect. This complicates C++ callpath profiling, for instance.

To deliver a robust, general solution, we decided to pursue an approach where the callpath is calculated and queried at runtime. The TAU measurement system already maintains a callstack that is updated with each entry/exit performance event (e.g., routine entry and exit). Thus, to determine the k-length callpath when a routine is entered, all that is necessary is to traverse up the callstack to determine the last $k-1$ events that define the callpath. If this is a newly encountered callpath, a new measurement profile must be created at that time because it was not pre-allocated. The main problem is how to do all of this efficiently.

Mapping callpath identity to its profile measurement is an example of what we call *performance mapping*. TAU implements a performance mapping API based on the *Semantic Event Association and Attribute* (SEAA) model [14]. Here an association is built between the identity of a performance event (e.g., a callpath) and a performance measurement object (e.g., a profile) for that event. Thus, when the event is encountered, the measurement object linked to that event can be found, via a lookup in a mapping table, and the measurement made.

In the case of callpath performance mapping, new callpaths occur dynamically, requiring new profile objects to be created at runtime. This can be done efficiently using the TAU mapping API. The callpath name is then hashed to serve as the index for future reference. Because routine identifiers can be long strings, TAU optimizes this process by computing the hash based on addresses of the profile objects of its $k-1$ parents. While the extra overhead to perform this operation is fixed, the accumulated overhead will depend on the number

of unique k-length callpaths encountered in the computation, as each of these will need a separate profile object created.

We have implemented 2-level callpath profiling in TAU as part of the current TAU performance system distribution. The important result is that this capability is available in all cases where TAU profiling is available. It is not restricted by programming language, nor source code access required, as dynamic instrumentation (via DyninstAPI [4]) can be used when source is not available. Also, all types of performance measurements are allowed, including measuring hardware counts for each callpath. Finally, in the future, we can benefit from the overhead reduction mechanisms to eliminate particular callpaths from measurement consideration.

Unfortunately, unlike a static approach, the measurement overhead of this dynamic approach increases as k increases because we must walk the callstack to determine the callpath. We have discussed several methods to do this more efficiently, but none lead to a fixed overhead for any k, and adopting a general k solution for the 2-level case would result in greater cost. Most user requests were for 2-level callpaths to determine routine performance distribution across calling parents, and this is what has been implemented in TAU. It should be noted that there are no inherent limitations to implementing solutions with $k > 2$. Also, if it is possible to determine callpaths statically, TAU could certainly use that information to implement a fixed-cost solution.

5. Large-Scale Performance Monitoring and Steering

Parallel performance tools offer the program developer insights into the execution behavior of an application. However, most tools do not work well with large-scale parallel applications where the performance data generated comes from thousands of processes. Not only can the data be difficult to manage and the analysis complex, but existing performance display tools are mostly restricted to two dimensions and lack the customization and interaction to support full data investigation. In addition, it is increasingly important that performance tools be able to function online, making it possible to control and adapt long-running applications based on performance feedback. Again, large-scale parallelism complicates the online access and management of performance data. It may be desirable to use existing computational steering systems for this purpose, but this will require performance analysis and visualization to be integrated with these tools.

As a result of our work with the University of Utah [16], we found ourselves in a position to design and prototype a system architecture for coupling advanced three-dimensional visualization with online performance data access, analysis, and visualization in a large-scale parallel environment. The architecture, shown in Figure 3, consists of four components. The "performance

data integrator" component is responsible for interfacing with a performance monitoring system to merge parallel performance samples into a synchronous data stream for analysis. The "performance data reader" component reads the external performance data into internal data structures of the analysis and visualization system. The "performance analyzer" component provides the analysis developer a programmable framework for constructing analysis modules that can be linked together for different functionality. The "performance visualizer" component can also be programmed to create different displays modules.

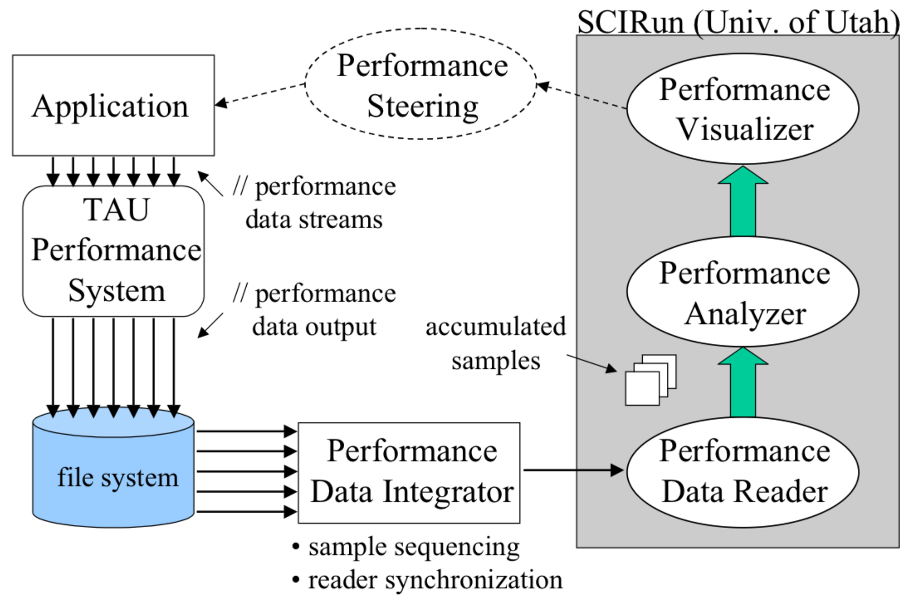

Figure 3. Online performance analysis and visualization architecture.

Our prototype is based on the TAU performance system, the Uintah computational framework [15], and the SCIRun [13] computational steering and visualization system. Parallel profile data from a Uintah simulation are sampled and written to profile files during execution. The performance data integrator reads the performance profile files, generated for each profile sample for each thread, and merges the files into a single, synchronized profile sample dataset. Each profile sample file is assigned a sequence number and the whole dataset is sequenced and timestamped. A socket- based protocol is maintained with the performance data reader to inform it of the availability of new profile samples and to coordinate dataset transfer.

The performance profile reader, implemented as a SCIRun module, inputs the merged profile sample dataset sent by the data integrator and stores the dataset in an internal C++ object structure. A profile sample dataset is orga-

nized in a tree-like manner according to TAU profile hierarchy:

$$node \rightarrow context \rightarrow thread \rightarrow profile\ data$$

Each object in the profile tree has a set of attribute access methods and a set of offspring access methods.

Using the access methods on the profile tree object, all performance profile data, including cross-sample data, is available for analysis. Utah's SCIRun [13] provides a programmable system for building and linking the analysis and visualization components. A library of performance analysis modules can be developed, some simple and others more sophisticated. We have implemented two generic profile analysis modules: *Gen2DField* and *Gen3DField*. The modules provide user control that allows them to be customized with respect to events, data values, number of samples, and filter options. Ultimately, the output of the analysis modules must be in a form that can be visualized. The *Gen2DField* and *Gen3DField* modules are so named because they produce 2D and 3D *Field* data, respectively. SCIRun has different geometric meshes available for Fields. We use an *ImageMesh* for 2D fields and a *PointCloudMesh* for 3D fields.

The role of the performance visualizer component is to read the Field objects generated from performance analysis and show graphical representations of performance results. We have built three visualization modules to demonstrate the display of 2D and 3D data fields. The *Terrain* visualizer shows ImageMesh data as a surface. The user can select the resolution of the X and Y dimensions in the Terrain control panel. A *TerrainDenotator* module was developed to mark interesting points in the visualization. A different display of 2D field data is produced by the *KiviatTube* visualizer. Here a "tube" surface is created where the distance of points from the tube center axis is determined by metric values and the tube length correlates with the sample. The visualization of PointCloudMesh data is accomplished by the *PointCloud* visualizer module.

The SCIRun program graph in Figure 4 shows how the data reader, analyzer, and visualizer modules are connected to process parallel profile samples from a Uintah application. The visualization is for a 500 processor run and shows the entire parallel profile measurement. The performance events are along the left-right axis, the processors along the in-out axis, and the performance metric (in this case, the exclusive execution time) along the up-down axis. Denotators are used to identify the performance events in the legend with the largest metric values. This full performance view enables the user to quickly identify major performance contributors.

Although this work is in the early stages, it demonstrates the significant tool advances possible through technology integration. As the Utah C-SAFE ASCI project moves towards Uintah computations with adaptive-mesh refinement

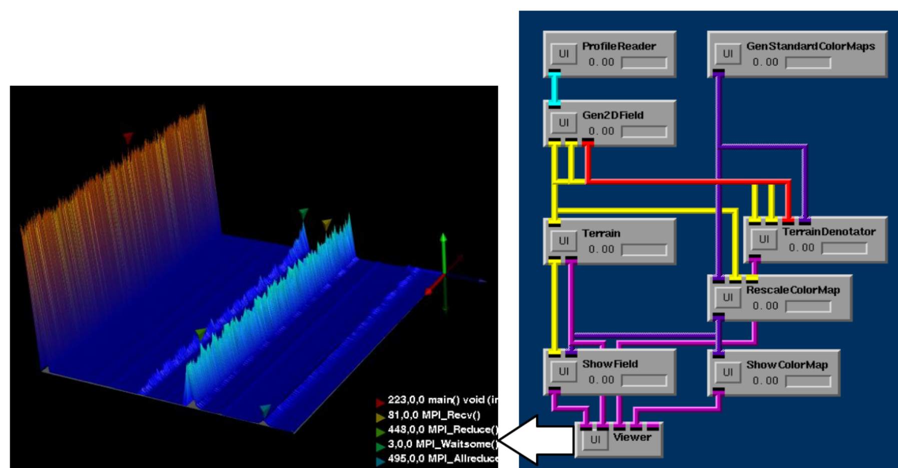

Figure 4. Performance profile visualization of 500 Uintah processes.

capabilities, we expect the relevance of online performance analysis to increase in importance. We are developing new performance visualization modules and extending the performance profile data to accommodate hardware counter statistics. Since SCIRun is being positioned as a computational steering system for Uintah, the implementation of the online performance tool in SCIRun well positions it for use as a customizable performance steering tool.

6. Peformance Database Framework

Empirical performance evaluation of parallel and distributed systems often generates significant amounts of performance data and analysis results from multiple experiments as performance is being investigated and problems diagnosed. Yet, despite the broad utility of cross-experiment performance analysis, most current performance tools support performance analysis for only a single application execution. We believe this is due primarily to a lack of tools for performance data management. Hence, there is a strong motivation to develop performance database technology that can provide a common foundation for performance data storage and access. Such technology could offer standard solutions for how to represent the performance data, how to store them in a manageable way, how to interface with the database in a portable manner, and how to provide performance information services to a broad set of analysis tools and users. A performance database system built on this technology could serve both as a core module in a performance measurement and analysis system, as well as a central repository of performance information contributed to and shared by several groups.

To address the performance data management problem, we designed the Performance DataBase Framework (PerfDBF) architecture shown in Figure 5. The PerfDBF architecture separates the framework into three components: performance data input, database storage, database query and analysis. The performance data is organized in a hierarchy of *applications*, *experiments*, and *trials*. Application performance studies are seen as constituting a set of experiments, each representing a set of associated performance measurements. A trial is a measurement instance of an experiment. We designed a Performance Data Meta Language (PerfDML) and PerfDML translators to make it possible to convert raw performance data into the PerfDB internal storage. The Performance DataBase (PerfDB) is structured with respect to the *application/experiment/trial* hierarchy. An object-relational DBMS is specified to provide a standard SQL interface for performance information query. A Performance DataBase Toolkit (PerfDBT) provides commonly used query and analysis utilities for interfacing performance analysis tools.

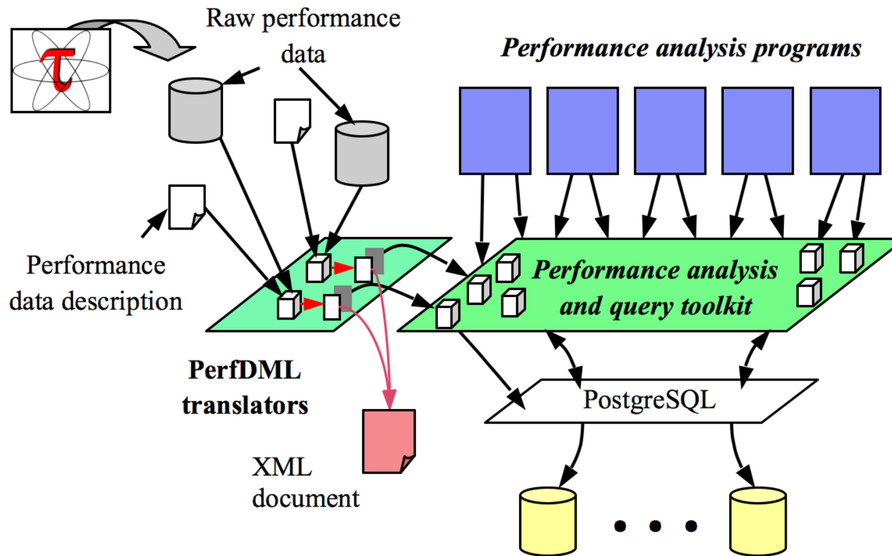

Figure 5. TAU performance database framework

To evaluate the PerfDBF architecture, we developed a prototype for the TAU performance system for parallel performance profiles. The prototype PerfDBF converts the raw profiles to PerfDML form, which is realized using XML technology. Database input tools read the PerfDML documents and store the performance profile information in the database. Analysis tools then utilize the PerfDB interface to perform intra-trial, inter-trial, and cross-experiment query and analysis. To demonstrate the usefulness of the PerfDBF, we have developed a scalability analysis tool. Given a set of experiment trials, representing

execution of a program across varying numbers of processors, the tool can compute scalability statistics for every routine for every performance metric. As an extension of this work, we are applying the PerfDBF prototype in a performance regression testing system to track performance changes during software development.

The main purpose of the PerfDBF work is to fill a gap in parallel performance technology that will make it possible for performance tools to interoperate. The PPerfDB [8] and Aksum [6] projects have demonstrated the benefit of providing such technology support and we have hopes to merge our efforts. We already see benefits within the TAU toolset. Our parallel performance profile, ParaProf, is able to read profiles that are stored in PerfDBF. In general, we believe the key will be to find common representations of performance data and database interfaces that can be adopted as the lingua franca among performance information producers and consumers. Its implementation will be an important enabling step forward in performance tool research.

7. Conclusions

The research work we presented in this paper reflects our view that advances in performance technology will be a product of both innovative ideas and strong engineering. More importantly, as the performance complexity of parallel and distributed systems increases, it will be important to develop performance tools on a robust technology foundation, leveraging existing capabilities to realize more sophisticated functionality. We believe the tools described above demonstrate this result. Each is or will be implemented in a form that can be distributed with the TAU performance system. While this may go beyond what is necessary to “prove” a research result, it is in the application of a performance tool on real performance problems where its merit will be truly determined.

References

[1] APART, IST Working Group on Automatic Performance Analysis: Real Tools. See http://www.fz-juelich.de.

[2] R. Armstrong, et al., “Toward a Common Component Architecture for High-Performance Scientific Computing,” High Performance Distributed Computing Conference, 1999. See http://www.cca-forum.org.

[3] S. Browne, 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), pp. 189–204, Fall 2000.

[4] B. Buck and J. Hollingsworth, “An API for Runtime Code Patching,” *International Journal of High Performance Computing Applications*, **14**(4), pp. 317–329, Winter 2000.

[5] Luiz DeRose and Felix Wolf, “CATCH - A Call-Graph Based Automatic Tool for Capture of Hardware Performance Metrics for MPI and OpenMP Applications,” Euro-Par 2002, pp. 167–176.

[6] T. Fahringer and C. Seragiotto, "Experience with Aksum: A Semi-Automatic Multi-Experiment Performance Analysis Tool for Parallel and Distributed Applications," Workshop on Performance Analysis and Distributed Computing, 2002.

[7] R. Hornung and S. Kohn, "Managing Application Complexity in the SAMRAI Object-Oriented Framework, *Concurrency and Computation: Practice and Experience*, special issue on Software Architectures for Scientific Applications, 2001.

[8] K. Karavanic, PPerfDB. See http://www.cs.pdx.edu/ karavan/research.htm.

[9] A. Malony and S. Shende, "Performance Technology for Complex Parallel and Distributed Systems," in *Distributed and Parallel Systems From Instruction Parallelism to Cluster Computing*, G. Kotsis and P. Kacsuk (Eds.), Kluwer, pp. 37–46, 2000.

[10] A. Malony, "Tools for Parallel Computing: A Performance Evaluation Perspective," in Handbook on Parallel and Distributed Processing, J. Blazewicz, K. Ecker, B. Plateau, an D. Trystram (Eds.), 2000, Springer-Verlag, pp. 342–363.

[11] W.Nagel, A.Arnold, M.Weber, H.Hoppe, and K.Solchenbach, "Vampir: Visualization and Analysis of MPI Resources," *Supercomputing*, **12**(1):69–80, 1996.

[12] Paraver, European Center for Parallelism of Barcelona, Technical University of Catalonia. See http://www.cepba.upc.es/paraver/index.html.

[13] S. Parker, D. Weinstein, and C. Johnson, "The SCIRun Computational Steering Software System," in *Modern Software Tools in Scientific Computing*, E. Arge, A. Bruaset, and H. Langtangen (Eds.), Birkhauser Press, pp. 1–44, 1997.

[14] Sameer S. Shende, "The Role of Instrumentation and Mapping in Performance Measurement," PhD Thesis, University of Oregon, 2001.

[15] J. St. Germain, J. McCorquodale, S. Parker, and C. Johnson, "Uintah: A Massively Parallel Problem Solving Environment,"High Performance Distributed Computing Conference, pp. 33–41, 2000.

[16] J. St. Germain, A. Morris, S. G. Parker, A. D. Malony, and S. Shende, "Integrating Performance Analysis in the Uintah Software Development Cycle," International Symposium on High Performance Computing, pp. 190–206, 2002.

[17] TAU (Tuning and Analysis Utilities). See http://www.acl.lanl.gov/tau.

[18] VTF, Virtual Test Shock Facility, Center for Simulation of Dynamic Response of Materials. See http://www.cacr.caltech.edu/ASAP.