# A Performance Interface for Component-Based Applications*

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

Craig Rasmussen, and Matthew Sottile
Advanced Computing Laboratory,
Los Alamos National Laboratory†
{crasmussen,matt}@lanl.gov

## Abstract

*This work targets the emerging use of software component technology for high-performance scientific parallel and distributed computing. While component software engineering will benefit the construction of complex science applications, its use presents several challenges to performance optimization. A component application is composed of a set of components, thus, application performance depends on the interaction (possibly non-linear) of the component set. Furthermore, a component is a "binary unit of composition" and the only information users have is the interface the component provides to the outside world. An interface for component performance measurement and query is presented to address optimization issues. We describe the performance component design and an example demonstrating its use for runtime performance tuning.*

## 1. Introduction

Throughout the history of scientific computing, application development has sought to leverage the power of abstraction in new software technology while continuing to harness the computing potential of high-end machines. However, the desire to manage the growing complexity in scientific problem solving with more flexible programming languages and framework-based development environments is naturally in tension with the need to deliver high performance on parallel and distributed systems, themselves undergoing equally complex architectural and technological evolution. The commonly accepted dogma is that the further software is away from the raw machine, the harder performance is to achieve. An important strategy to deal with this tension has been the creation of layered software infrastructures that can, at once, provide a rich middleware of capabilities upon which to create new scientific programming paradigms, while being implemented to run efficiently on different high-end execution platforms. The unfortunate compromise of this strategy is to further distance the scientific application developer from the now broader range of sources of performance behavior and possible performance problems. Indeed, as a result, performance itself becomes more complex to observe and to understand. Thus, as both the power and the complexity of scientific software environments and computing systems mutually advance, it is imperative that technology for performance evaluation and engineering keep pace.

While there has been excellent progress in software technology to meet the increasing demands of scientific application development, the effective integration of performance evaluation support is rare. Integration implies some understanding of the performance problems that will need to be addressed. Unfortunately, potential problems are not generally known entirely at the beginning of a new software technology project, or change as the software evolves. Integration also implies the existence of standard performance methods, techniques, and tools that can be readily applied to the types of performance problems expected. How then should we address the apparent dilemma where the use of advanced software development technology for creating sophisticated scientific computing environments, may result in software systems whose performance behavior we cannot adequately measure, understand, or improve? We believe the key is to implement a performance engineering strategy consistent with modern software architectures and engineering methodologies to both provide relevant performance support and utilize the power of the software abstractions and infrastructure.

The software challenges of building large-scale, complex scientific applications are beginning to be addressed by the use of component software technologies. The software engineering of scientific applications from components that can be "plugged" together will greatly facili-

*This research is funded by the United States Department of Energy's Office of Science under contract DE FG03-01ER25501.

†Los Alamos National Laboratory is operated by the University of California for the National Nuclear Security Administration of the United States Department of Energy under contract W-7405-ENG-36, LA-UR No. 03-0097.

tate construction of coupled simulations and improve their cross-platform portability. Groups such as the Common Component Architecture (CCA) Forum [4] are extending component engineering methods and infrastructure to address problems of parallel component interfaces, scientific data exchange, and cross-language interoperability.

However, the success of scientific component software will depend greatly on the ability to deliver high-performance solutions, in comparison with the performance that can be achieved using standard application implementations. Scientific components are more complex and diverse than typical software components or libraries, in their scale, execution modes, programming styles and languages, and system targets. Performance technology that lacks robustness, portability, and flexibility will inevitably prove incapable of addressing the software and platform integration requirements required for performance observation and analysis. In addition, the construction of applications by component composition requires robust performance engineering technology that can support different scientific code coupling scenarios. Many performance tools lack mechanisms for performance abstraction that can capture the different composition modes envisioned. Most importantly, performance engineering technology should be compatible with the component engineering methodologies and frameworks used to develop applications, or it will be neither routinely nor effectively applied by component and application developers.

In this paper, we consider the design and development of a component interface for performance measurements within the CCA framework. The component developer can access a performance monitor (also a component) to create timers that gather data on component performance as it runs. The performance interface also allows users and other components to query a component, or the performance monitor, to gather statistics regarding component performance. This allows users to select the best performing component from a set of components supporting the same interface.

The benefit of the performance interface is demonstrated, in an example below, to automate the selection of an optimal set of working components. An optimizing component is used to evaluate components of the same functionality, but possibly different performance behavior. As the application runs, the optimizing component utilizes the performance API to gather statistics about the running application and decides which of the set of similar components to choose for optimal performance.

## 2. The Common Component Architecture

Component-based software architectures have grown popular in the general computing world in the past decade, but have yet to be seriously adopted in practice by scientific computing users. The primary reason for this is that scientific, or high-performance, computing users want just that – high performance. Systems such as CORBA[5], JavaBeans[8], COM[7], and others have either not run on the systems scientists use, or simply run far too slow for their applications. The Common Component Architecure (CCA) was started in 1997 as an effort to bring the component programming model to scientific users. Fundamentally, the CCA is a specification of the component programming pattern and the interface the components see to the underlying support substrate, or *framework*.

Component programming, much like object-oriented programming, provides a model for constructing software such that units of code (components and objects) expose a "public" interface to the outside while hiding their internal implementation features. Components extend the object model by allowing components to dynamically discover and expose interface information, something that is statically determined at compilation time in most object-oriented languages. The CCA requires components to describe their interfaces in the Scientific Interface Definition Language, or SIDL[9]. Like the IDL used by CORBA and Xerox ILU[6], the interfaces are defined in a language independent manner and are not bound to the source code or compiled binary of a component. The IDL simply describes the public interface so that external parties can discover what services are available and how they must be called.

In the CCA, a component is defined as a collection of *ports*, where each port represents a set of functions that are publicly available. A port is described using SIDL, and some form of wrapper exists in the implementation to map the SIDL interface to that of the implementation language. From the point of view of a component, there are two types of ports. Those that are implemented by a component are known as *provides* ports, and other components may connect to and use them. Other ports that a component will expect to be connected to and call are known as *uses* ports. Uses and provides ports are connected together as shown in Figure 1. The act of connecting components is referred to as component *composition*.

When a component is instantiated and allowed to execute, it registers the provides and uses ports with the underlying framework. This information allows external components to discover what ports or interfaces are available, and ensures that expected relationships between components are fulfilled, before allowing execution.

Port discovery is a service provided by the framework and is actually just another port that a component can connect to. For instance, a component can obtain a list from the framework of all components providing a specific interface or port. The component could then connect to each of the ports in the list in an iterative fashion and make calls on the

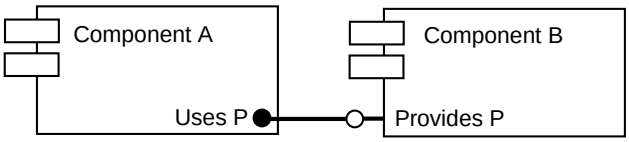


**Figure 1. Two CCA components: One *uses* a "P" port *provided* by the other.**

methods within the connected port. We exploit the concept of port discovery, in just this fashion, in the implementation of a basic optimization component in Section 5.

Iterative access to a common set of interfaces presupposes that such a set exists. One of the primary benefits of component-based programming is the adoption of common interfaces for domain-specific purposes. This allows different teams of individuals to develop components based on this "standard" component API. This in turn allows users of these components to pick and choose the particular component that best fits their needs.

In addition to providing a model for component creation and composition, the CCA Forum encourages communities to form and adopt standard component interfaces. Several different scientific computing communities have already begun this effort, including the attempt to adopt standard interfaces for partial differential equation (PDE) simulations, an equation solver interface, an "M-by-N" parallel coupling component, and interfaces for access to sophisticated data structures such as distributed and sparse matrices, and both regular and irregular meshes. One of the purposes of this paper is to continue this effort by proposing a component API for performance measurements.

## 3. TAU Performance System

The software engineering of CCA components and application development demands robust tools for performance measurement and analysis. Our approach to performance integration in CCA component software begins with the TAU performance system [11]. TAU provides technology for performance instrumentation, measurement, and analysis for complex parallel systems. 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.

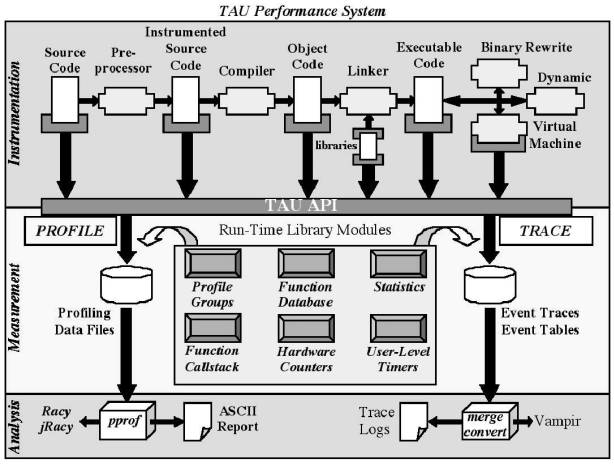


**Figure 2. TAU Performance System Architecture**

As shown in Figure 2, TAU 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 and message passing parallel computation models. Instrumentation code makes calls to the TAU measurement API. The TAU measurement library implements performance profiling and tracing support for performance events occurring at function, method, basic block, and statement levels during execution. Performance experiments can be composed from different measurement modules (e.g., hardware performance monitors) and measurements can be collected with respect to user-defined performance groups. 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 [12] for sophisticated trace analysis and visualization.

## 4. Performance Interface for Components

### 4.1. Component Measurement

Given the TAU performance measurement technology, the important question becomes what is the approach best suited for component performance measurement. There are two measurement types we envision based on how a component is instrumented: 1) with direct calls to a measurement library or 2) using an abstract measurement interface. The difference is depicted in Figure 3. TAU specializes in multi-level performance instrumentation target-

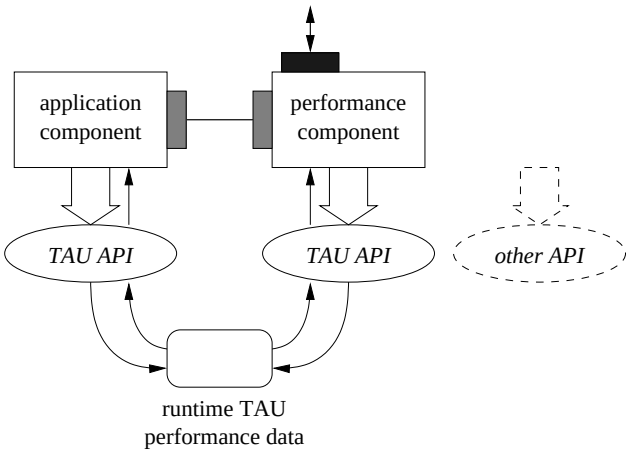


**Figure 3. Measurement Component Interface.**

ing a common performance measurement API. We can instrument the component code to call TAU measurement routines directly using the API, as shown in the left part of Figure 3. To facilitate the instrumentation, TAU provides automated source instrumentation tools (based on the Program Database Toolkit (PDT)[10]) and dynamic instrumentation support using DyninstAPI[2]. As shown, the TAU measurement system maintains runtime performance data that can be accessed via the API directly or stored in files at the end of component execution.

In contrast, the component could also be instrumented to call an abstract measurement component interface, as shown in the right part of Figure 3. This interface would be implemented by a *performance component* that targets a backend measurement system (in this case TAU ). There are several benefits to this approach. First, a component could be developed with "virtual instrumentation" in the sense that the abstract measurement interface is virtual (i.e., consists of virtual functions). The overhead of instrumentation is nullified until a performance component is instantiated. Second, it is possible to use any measurement system in the performance component that conforms to the interface. Lastly, the performance component can provide ports for other components to use, including ports to access performance data without touching the instrumented application component. This raises the possibility that the application component is instrumented directly, but the performance data is accessed via the performance component. The downside of this approach is that the measurement interface is possibly less efficient or that it does not allow certain types of detailed performance measurements to be made.

### 4.2. Performance Interface and Performance Component

Our approach offers both types of measurements discussed above. In particular, we have designed a performance instrumentation interface for component software and a performance component that implements this interface through a measurement port. This interface allows a user to create objects for timing, track application events, control the instrumentation at runtime, and query the performance data. TAU provides an implementation for each of these entities. Appendix 1 shows the SIDL definition for the performance interface we have developed.

**Timer Interface.** A timer interface allows the user to bracket parts of his/her code to specify a region of interest. The `Timer class` interface supports the `start` and `stop` methods. A timer object has a unique name and a signature associated with it. There are several ways to identify timers and performance tools have used different techniques. To identify a timer, one approach advocates the use of numeric identifiers and an associated table mapping the identifiers to names. While it is easy to specify and pass the timer identifier among routines, it has its drawbacks. Maintaining a table statically might work for languages such as Fortran90 and C, but it extends poorly to C++, where a template may be instantiated with different parameters. This aspect of compile time polymorphism makes it difficult to disambiguate between different instantiations of the same code. Also, it can introduce instrumentation errors in maintaining the table that maps the identifiers to names. This is true for large projects that involve several application modules and developers.

Our interface uses a dynamic naming scheme where timer names are associated with the timer object at runtime. A timer can have a unique name and a signature that can be obtained using runtime type information of objects in C++. Several logically related timers can be grouped together using an optional profile group. A profile group is specified using a name when a timer is created. TAU implements the generic Timer interface shown in the Appendix and introduces an optimization that allows it to keep track of only those timers that are invoked at least once. It maintains both exclusive and inclusive measurement values for each timer. Timers can be nested, but may not overlap (i.e., start and stop calls from one timer should not overlap those from another). When timers overlap, TAU detects this overlap at runtime and warns the user about this error in instrumentation.

It is important to note that this interface is independent of the nature of measurements that can be performed by the performance tool. For instance, TAU may be configured to record exclusive and inclusive wallclock time for each timer

for each thread of execution. Other measurement options that are currently supported include profiling with process virtual time or counts obtained from hardware performance counters. TAU also provides the option of making multiple measurements using a combination of wallclock time and/or hardware performance metrics in the same performance evaluation experiment. Thus, the timer interface is independent of the underlying measurements and is a vehicle for the user to specify interesting code regions that merit observation.

**Control Interface.** The control interface provided by the performance component allows us to enable and disable a group of timers at a coarse level. The user can disable all the groups and selectively enable a set of groups for refining the focus of instrumentation. Similarly, the user can start with all groups in an enabled state and selectively disable a set of groups.

**Query Interface.** The query interface allows the program to interact with the measurement substrate by querying for a variety of performance metrics. This interface allows the program to query the set of measurements that are being performed. These are represented in the interface as a list of counters. The counter names specify what is being measured. These are tool specific names such as `PAPI_FP_INS` or `PAPI_L1_DCM` that stand for the number of floating point instructions and level 1 data cache misses executed (as reported by PAPI[1]), respectively. The query interface reports the list of timers that are active at any given point in time. For each timer, it provides a set of exclusive and inclusive values for each counter. It provides the number of start/stop pairs (referred here as the number of calls) for each timer and also the number of timers that each timer called in turn. Instead of examining this data at runtime, an application may choose to store this information in files. This data may be read by an online monitor external to the application and analyzed as the application executes.

**Event Interface.** The event interface provided by the performance component allows a user to track application level events that take place at a specific location in the source code (as opposed to bracketing the code with start/stop calls). The generic event interface provides a single trigger method with a data parameter. This permits the user to associate the application data with the event. For example, to track the memory utilization in an application, a user may create a named event called "Memory used by arrays" and each time an array is allocated, this event might be triggered with the size of the chunk of memory allocated as its parameter. TAU implements the event class by keeping track of maxima, minima, mean, standard deviation, and number of samples as statistics. Another tool might for instance maintain quantiles for the same data.

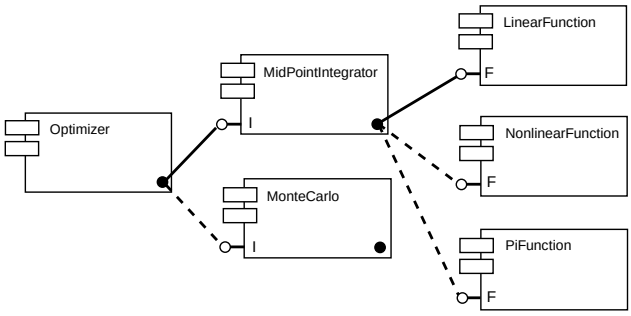


**Figure 4. Example showing optimizer component and dynamic component connections.**

The performance component interface gives each tool the flexibility of performing tool-specific optimizations, measurement and analysis unique to the tool, and provides a balance between tool specificity and genericity. For example, a tool may implement the `Timer` and `Event` interfaces in different ways. The benefits of such an interface are manifold for a user. Using this generic interface to annotate the source code, the user can benefit from using multiple performance measurement and analysis tools without the need for recompiling the source code. At runtime, the user can choose which tool (and more specifically, which dynamic shared object) implements the interface and instantiates a component for performing the instrumentation. This approach permits the user to mix and match the capabilities of multiple performance tools to accomplish the task of performance observation of components.

## 5. Example

In this section we give an simple example of how the CCA framework can be used in conjuction with the proposed component performance API to optimize the working set of a component-based application.

### 5.1. Selection of optimal component-based solvers

A situation that arises frequently in scientific computing is that of selecting a solver that both meets some convergence requirement and also performs optimally (i.e., reaches a solution in the shortest amount of time). In general, a solver may be optimal for a particular class of problems, yet behave poorly on others. Even for a given class of problems, the convergence behavior of a solver can be highly dependent on the data itself. Thus, the choice of an "optimal" solver is not as easy as it might at first seem.

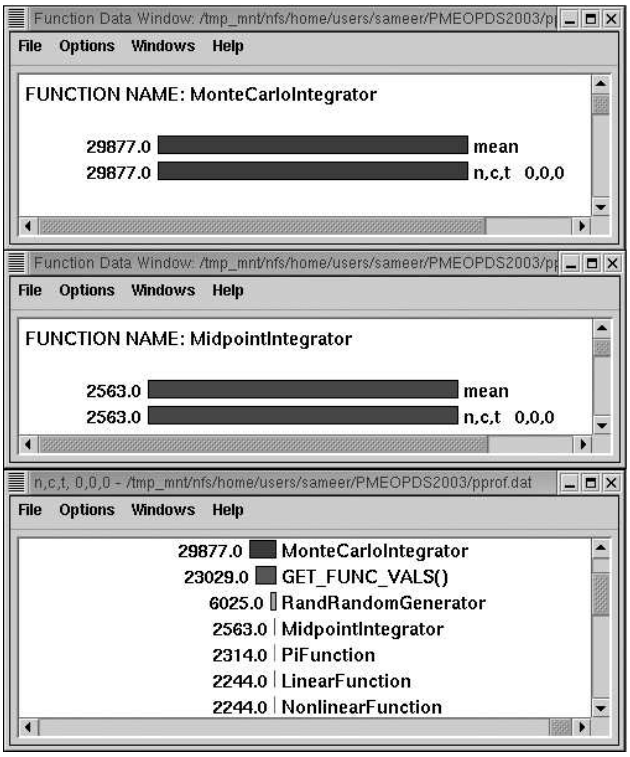


**Figure 5. Visualization of performance data using TAU's *JRacy* profile browser (units are in microseconds).**

Using the component performance API described in the previous section, the CCA framework easily allows one to test a set of solvers on a representative subset of a broader spectrum of data. The best performing of the solvers can then be used to operate on the full dataset.

While this is relatively easy to do using components with standard interfaces, it is a much more onerous task without. One must maintain separate bodies of code (one for each solver interface) and compile and link these codes against separate solver libraries. Then scripts must be generated to run the tests, select the best performer and finally to make the final run. In practice this is not done. However, because the dynamic substitution of components is so easy to do with component-based programming, we suggest that this practice may become more common place in the future.

In the example shown here, the solver is a simple integrator component that returns the integral of a function $f$ over the range $(a,b)$,

$$i = \int_a^b f(x)dx$$

The integrator solver set contains a trivial Monte Carlo integrator that samples from a uniform distribution and a midpoint integrator that uses simple trapezoidal quadrature. Because the Monte Carlo integrator uses a sampling based approach, it may work on functions that are not appropriate for the midpoint integrator, such as with functions containing point discontinuities. Although in this example, we assume that the functions are simple, smooth, and have no discontinuities – so both solvers are candidates. It is expected that the Monte Carlo integrator will require a larger number of integration points, and will thus take more time than the midpoint integrator.

An optimizing component is used to iteratively test two components that both provide the same Integrator interface. For this example, each integrator component is tested on a set of components that provide the Function interface. For any one particular integrator-function combination, the integrator component repeatedly calls the function component at points within the range $(a,b)$, until the integration is complete. The primary components in this test and the port connections are shown in Figure 4.

The optimizing component was constructed using the BuilderServices API of the Ccaffeine CCA framework [3]. This API allows components to be loaded, connected to each other, run, and then disconnected, all under runtime control. The optimizing component iteratively loads and runs each integrator over the complete set of function components, while keeping track of the run time for each integrator and function combination using the performance API described above. The set of function components, the number of sampling points, and the range of each integration were chosen for simple illustrative purposes. In general, for best results the choice of parameters such as this requires an expert with knowledge of the system being optimized.

As expected, the best performing integrator was the midpoint integrator, as can be seen in Figure 5. The Monte Carlo integrator took nearly 12 times longer to complete than the midpoint integrator. Much of this extra time was spent in the RandomGenerator component, as can be seen in the lower portion of the figure.

## 6. Conclusions

To leverage the power of software abstraction while maintaining high-performing applications demands a tight integration of performance measurement and analysis technology in the software engineering process. The success of component software for scientific applications running on large-scale parallel computers will be determined by how close performance comes to standard implementation approaches. We have designed a performance interface for component software, and implemented a performance component using the TAU performance system, to help understand performance problems in scientific component applications.

But this is only one part of the story. Component software has an inherent abstractional power over standard approaches to modify what and how components are used in a computation. If it is possible to inform these component choices with performance information in a manner compatible with component software design, there is great potential for adapting component applications to optimize performance behavior. Indeed, our demonstration shows how the TAU performance component can be used within CCA applications in ways that allow high-performing, self-optimizing component solutions to be achieved.

Furthermore, we propose that the performance interfaces described here be added to all CCA components as provides ports. The code to do this could optionally be generated automatically by the language-interoperability mechanism of each CCA framework. This would allow CCA component writers to automatically provide performance data at the granularity of function calls (with no extra work on their part) and would allow CCA component users, to always have access to this data (as a runtime option). It would also allow agents, such as the Optimizer component described above, to monitor and fine tune a component-based application.

## References


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

[2] B. Buck and J. Hollingsworth: An API for Runtime Code Patching, Journal of High Performance Computing Applications, 14(4):317-329, Winter 2000.

[3] Ccaffeine: A CCA Component Framework for Parallel Computing. http://www.cca-forum.org/ccafe.

[4] Common Component Architecture Forum. http://www.cca-forum.org.

[5] CORBA Components, Object Management Group, OMG TC Document orbos/99-02-95. http://www.omg.org. (1999)

[6] Cutting, D., Janssen, W., Spreitzer, M., Wymore, F.: ILU Reference Manual. Xerox Palo Alto Research Center. (1993)

[7] G. Eddon and H. Eddon: Inside Distributed COM. Microsoft Press (1998)

[8] D. Kara.: The Enterprise JavaBeans Component Model. Component Strategies 1(7) (1999)

[9] Kohn, S., Dahlgren, T., Epperly, T., Kumfert, G.: The State of SIDL: Quarterly Status Report. Common Component Architecture Forum Meeting, Bloomington, IN. October 2-3, 2001

[10] Lindlan, K.A., Cuny, J., Malony, A.D., Shende, S., Mohr, B., Rivenburgh, R., Rasmussen, C.: Tool Framework for Static and Dynamic Analysis of Object-Oriented Software with Templates. Proceedings SC'2000, (2000)

[11] Malony, A., Shende, S.: Performance Technology for Complex Parallel and Distributed Systems. In: Kotsis, G., Kacsuk, P. (eds.): Distributed and Parallel Systems From Instruction Parallelism to Cluster Computing. Proc. 3rd Workshop on Distributed and Parallel Systems, DAPSYS 2000, Kluwer (2000) 37–46

[12] Pallas GmbH: VAMPIR: Visualization and Analysis of MPI Resources. http://www.pallas.de/pages/vampir.htm.


## 7. Appendix: SIDL Description of Performance Interface

```
version performance 1.0;
package performance
{
  interface Timer
  { /* Start/stop the Timer */
    void start();
    void stop();

    /* Set/get the Timer name */
    void setName(in string name);
    string getName();

    /* Set/get Timer type information
       (e.g., signature of the routine) */
    void setType(in string name);
    string getType();

    /* Set/get the group name associated
       with the Timer */
    void setGroupName(in string name);
    string getGroupName();

    /* Set/get the group id associated
       with the Timer */
    void setGroupId(in long group);
    long getGroupId();
  }

  /* Query interface to obtain timing
   * information */
  interface Query
  { /* Get the list of Timer and Counter names */
    array<string> getTimerNames();
    array<string> getCounterNames();

    /* Returns inclusive/exclusive time, numcalls,
       childcalls and counter names for given
       timers */
    void getTimerData(in array<string> timerList,
      out array<double, 2> counterExclusive,
      out array<double, 2> counterInclusive,
      out array<int> numCalls,
      out array<int> numChildCalls,
      out array<string> counterNames,
      out int numCounters);
```

```
  /* Writes instantaneous profile to disk
   * in a dump file. */
  void dumpProfileData();

  /* Writes the instantaneous profile to disk
   * in a dump file whose name
   * contains the current timestamp. */
   void dumpProfileDataIncremental();

  /* Writes the list of timer names to a
   * dump file on the disk */
   void dumpTimerNames();

  /* Writes the profile of the given set of
   * timers to the disk. */
  void dumpTimerData(in array<string> timerList);

  /* Writes the profile of the given set of
   * timers to the disk. The dump file name
   * contains the current timestamp when
   * the data was dumped. */
  void dumpTimerDataIncremental(
    in array<string> timerList);
}

/* User defined event profiles for application
 * specific events */
interface Event
{ /* Set the name of the event */
  void setName(in string name);

  /* Trigger the event */
  void trigger(in double data);
}

/* Interface for runtime instrumentation control
 * based on groups */
interface Control
{ /* Enable/disable group id */
  void enableGroupId(in long id);
  void disableGroupId(in long id);

  /* Enable/disable group name */
  void enableGroupName(in string name);
  void disableGroupName(in string name);

  /* Enable/disable all groups */
  void enableAllGroups();
  void disableAllGroups();
}

/* Interface to create performance component
 * instances */
interface Measurement extends gov.cca.Port
{ /* Create a Timer */
  Timer createTimer();
  Timer createTimerWithName(in string name);
  Timer createTimerWithNameType(in string name,
    in string type);
  Timer createTimerWithNameTypeGroup(in string
    name, in string type, in string group);

  /* Create a Query interface */
  Query createQuery();

  /* Create a User Defined Event interface */
  Event createEvent();
  Event createEventWithName(in string name);

  /* Create a Control interface for selectively
   * enabling and disabling
   * the instrumentation based on groups */
  Control createControl();
}

/* TAU */
/* Implementation of performance component
 * Timer interface*/
class TauTimer implements-all Timer
{
}

/* Implementation of performance component
 * Event interface*/
class TauEvent implements-all Event
{
}

/* Implementation of performance component
 * Query interface*/
class TauQuery implements-all Query
{
}

/* Implementation of performance component
 * Control interface*/
class TauControl implements-all Control
{
}

/* Implementation of performance component
 * Measurement interface*/
class TauMeasurement implements-all Measurement,
  gov.cca.Component
{
}

}
```