THE TAU PARALLEL PERFORMANCE SYSTEM

Sameer S. Shende
Allen D. Malony

DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE, UNIVERSITY OF OREGON, EUGENE, OR (SAMEER@CS.UOREGON.EDU)

Abstract

The ability of performance technology to keep pace with the growing complexity of parallel and distributed systems depends on robust performance frameworks that can at once provide system-specific performance capabilities and support high-level performance problem solving. Flexibility and portability in empirical methods and processes are influenced primarily by the strategies available for instrumentation and measurement, and how effectively they are integrated and composed. This paper presents the TAU (Tuning and Analysis Utilities) parallel performance system and describe how it addresses diverse requirements for performance observation and analysis.

Key words: Performance evaluation, instrumentation, measurement, analysis, TAU

The International Journal of High Performance Computing Applications, Volume 20, No. 2, Summer 2006, pp. 287–311
DOI: 10.1177/1094342006064482

Figures 1–13 appear in color online: http://hpc.sagepub.com

1 Introduction

The evolution of computer systems and of the applications that run on them – towards more sophisticated modes of operation, higher levels of abstraction, and larger scale of execution – challenge the state of technology for empirical performance evaluation. The increasing complexity of parallel and distributed systems, coupled with emerging portable parallel programming methods, demands that empirical performance tools provide robust performance observation capabilities at all levels of a system, while mapping low-level behavior to high-level performance abstractions in a uniform manner.

Given the diversity of performance problems, evaluation methods, and types of events and metrics, the instrumentation and measurement mechanisms needed to support performance observation must be flexible, to give maximum opportunity for configuring performance experiments, and portable, to allow consistent cross-platform performance problem solving. In general, flexibility in empirical performance evaluation implies freedom in experiment design, and choices in selection and control of experiment mechanisms. Using tools that otherwise limit the type and structure of performance methods will restrict evaluation scope. Portability, on the other hand, looks for common abstractions in performance methods and how these can be supported by reusable and consistent techniques across different computing environments (software and hardware). Lack of portable performance evaluation environments forces users to adopt different techniques on different systems, even for common performance analysis.

The TAU (Tuning and Analysis Utilities) parallel performance system is the product of fourteen years of development to create a robust, flexible, portable, and integrated framework and toolset for performance instrumentation, measurement, analysis, and visualization of large-scale parallel computer systems and applications. The success of the TAU project represents the combined efforts of researchers at the University of Oregon and colleagues at the Research Centre Juelich and Los Alamos National Laboratory. The purpose of this paper is to provide a complete overview of the TAU system. The discussion will be organized first according to the TAU system architecture and second from the point of view of how to use TAU in practice.

2 A General Computation Model for Parallel Performance Technology

To address the dual goals of performance technology for complex systems – robust performance capabilities and widely available performance problem solving methodologies – we need to contend with problems of system diversity while providing flexibility in tool composition, configuration, and integration. One approach to address

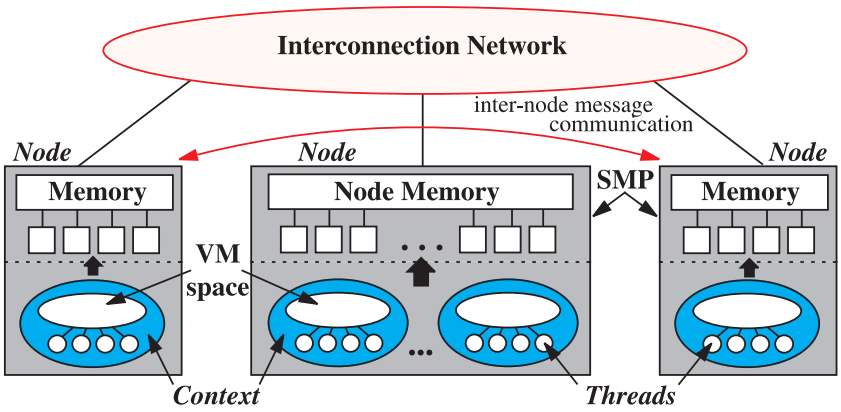

Fig. 1 Execution model supported by TAU.

these issues is to focus attention on a sub-class of computation models and performance problems as a way to restrict the performance technology requirements. The obvious consequence of this approach is limited tool coverage. Instead, our idea is to define an abstract computation model that captures general architecture and software execution features and can be mapped straightforwardly to existing complex system types. For this model, we can target performance capabilities and create a tool framework that can adapt and be optimized for particular complex system cases.

Our choice of general computation model must reflect real computing environments, both in terms of the parallel systems architecture and the parallel software environment. The computational model we target was initially proposed by the HPC++ consortium (HPC++ Working Group 1995) and is illustrated in Figure 1. Two combined views of the model are shown: a physical (hardware) view and an abstract software view. In the model, a *node* is defined as a physically distinct machine with one or more processors sharing a physical memory system (i.e. a shared memory multiprocessor (SMP)). A node may link to other nodes via a protocol-based interconnect, ranging from proprietary networks, as found in traditional MPPs, to local- or global-area networks. Nodes and their interconnection infrastructure provide a hardware execution environment for parallel software computation. A *context* is a distinct virtual address space within a node providing shared memory support for parallel software execution. Multiple contexts may exist on a single node. Multiple *threads* of execution, both user and system level, may exist within a context; threads within a context share the same virtual address space. Threads in different contexts on the same node can interact via interprocess communication (IPC) facilities, while threads in contexts on different nodes communicate using message passing libraries (e.g. MPI) or network IPC. Shared-memory implementations of message passing can also be used for fast intra-node context communication. The bold arrows in the figure reflect scheduling of contexts and threads on the physical node resources.

The computation model above is general enough to apply to many high-performance architectures as well as to different parallel programming paradigms. Particular instances of the model and how it is programmed defines requirements for performance tool technology. That is, by considering different instances of the general computing model and the abstract operation of each, we can identify important capabilities that a performance tool should support for each model instance. When we consider a performance system to accommodate the range of instances, we can look to see what features are common and can be abstracted in the performance tool design. In this way, the capability abstraction allows the performance system to retain uniform interfaces across the range of parallel platforms, while specializing tool support for the particular model instance.

3 TAU Performance System Architecture

The TAU performance system (Shende et al. 1998; Malony and Shende 2000; University of Oregon b) is designed as a tool framework, whereby tool components and modules are integrated and coordinate their operation using well-

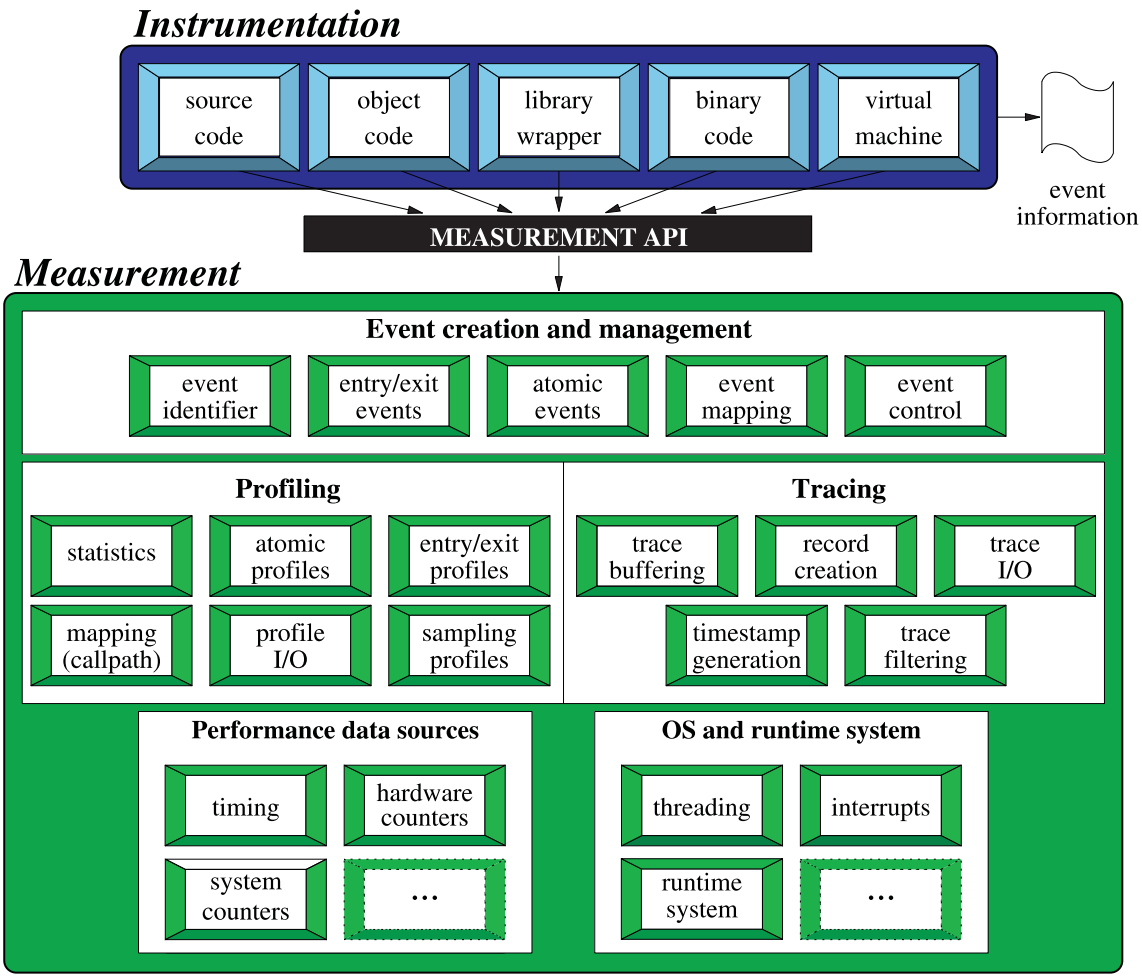

Fig. 2 Architecture of TAU Performance System – Instrumentation and Measurement.

defined interfaces and data formats. The TAU framework architecture is organized into three layers – instrumentation, measurement, and analysis – where within each layer multiple modules are available and can be configured in a flexible manner under user control.

The instrumentation and measurement layers of the TAU framework are shown in Figure 2. TAU supports a flexible instrumentation model that allows the user to insert performance instrumentation calling the TAU measurement API at different, multiple levels of program code representation, transformation, compilation, and execution. The key concept of the instrumentation layer is that it is here where *performance events* are defined. The instrumentation mechanisms in TAU support several types of performance events, including events defined by code location (e.g. routines or blocks), library interface events, system events, and arbitrary user-defined events. TAU is also aware of events associated with message passing and multi-threading parallel execution. The instrumentation layer is used to define events for performance experiments. Thus, one output of instrumentation are information about the events for a performance experiment. This information will be used by other tools.

The instrumentation layer interfaces with the measurement layer through the *TAU measurement API*. TAU's measurement system is organized into four parts. The *event creation and management part* determines how events are processed. Events are dynamically created in the TAU system as the result of their instrumentation and occurrence during execution. Two types of events are supported: *entry/exit* events and *atomic* events. In addition, TAU provides the mapping of performance measurements for "low-level" events to high-level execution entities. Overall, this part provides the mechanisms to manage events as a performance experiment proceeds. It includes the grouping of events and their runtime measurement control. The *performance measurement* part supports two measurement forms: profiling and tracing. For each form, TAU provides the complete infrastructure to manage the measured data during execution at any scale (number of events or parallelism). The *performance data sources* part defines what performance data is measurable and can be used in profiling or tracing. TAU supports different timing sources, choice of hardware counters through the PAPI (Browne et al. 2000) or PCL (Berrendorf, Ziegler, and Mohr) interfaces, and access to system performance data. The *OS and runtime system* part provide the coupling between TAU's measurement system and the underlying parallel system platform. TAU specializes and optimizes its execution according to the platform features available.

The TAU measurement systems can be customized and configured for each performance experiment by composing specific modules for each part and setting runtime controls. For instance, based on the composition of modules, an experiment could easily be configured to measure the profile that shows the inclusive and exclusive counts of secondary data cache misses associated with basic blocks such as routines, or a group of statements. By providing a flexible measurement infrastructure, a user can experiment with different attributes of the system and iteratively refine the performance characterization of a parallel application.

The TAU analysis and visualization layer is shown in Figure 3. As in the instrumentation and measurement layer, TAU flexibility allows use of several modules. These are separated between those for parallel profile analysis and parallel trace analysis. For each, support is given to the management of the performance data (profiles or traces), including the conversion to/from different formats. TAU comes with both text-based and graphical tools to visualize the performance profiles. ParaProf (Bell, Malony, and Shende 2003) is TAU's parallel profile analysis and visualization tool. Also distributed with TAU is the PerfDMF (Huck et al. 2005) tool providing multi-experiment parallel profile management. Given the wealth of third-party trace analysis and visualization tools, TAU does not implement its own. However, trace translation tools are implemented to enable use of Vampir (Intel Corporation; Nagel et al. 1996), Jumpshot (Wu et al. 2000), and Paraver (European Center for Parallelism of Barcelona (CEPBA)). It is also possible to generate EPILOG (Mohr and Wolf 2003) trace files for use with the Expert (Wolf et al. 2004) analysis tool. All TAU profile and trace data formats are open.

The framework approach to TAU's architecture design guarantees the most flexibility in configuring TAU capabilities to the requirements of the parallel performance experimentation and problem solving the user demands. In addition, it allows TAU to extend these capabilities to include the rich technology being developed by other performance tool research groups. In the sections that follow, we look at each framework layer in more depth and discuss in detail what can be done with the TAU performance system.

4 Instrumentation

In order to observe performance, additional instructions or probes are typically inserted into a program. This process is called *instrumentation*. From this perspective, the execution of a program is regarded as a sequence of significant performance events. As events execute, they activate the probes which perform measurements. Thus, instrumentation exposes key characteristics of an execution. Instrumentation can be introduced in a program at several levels of the program transformation process. In this section we describe the instrumentation options supported by TAU.

4.1 Source-Based Instrumentation

TAU provides an API that allows programmers to manually annotate the source code of the program. Source-level instrumentation can be placed at any point in the program and it allows a direct association between language- and program-level semantics and performance measurements. Using cross-language bindings, TAU provides its API in C++, C, Fortran, Java, and Python languages. Thus, language specific features (e.g. runtime type information for tracking templates in C++) can be leveraged. TAU also provides a higher-level specification in SIDL (Kohn et al. 2001; Shende et al. 2003) for cross-language portability and deployment in component-based programming environments (Bernholdt et al. 2005).

TAU's API can be broadly classified into the following five interfaces:

- Interval event interface
- Atomic event interface
- Query interface
- Control interface
- Sampling interface

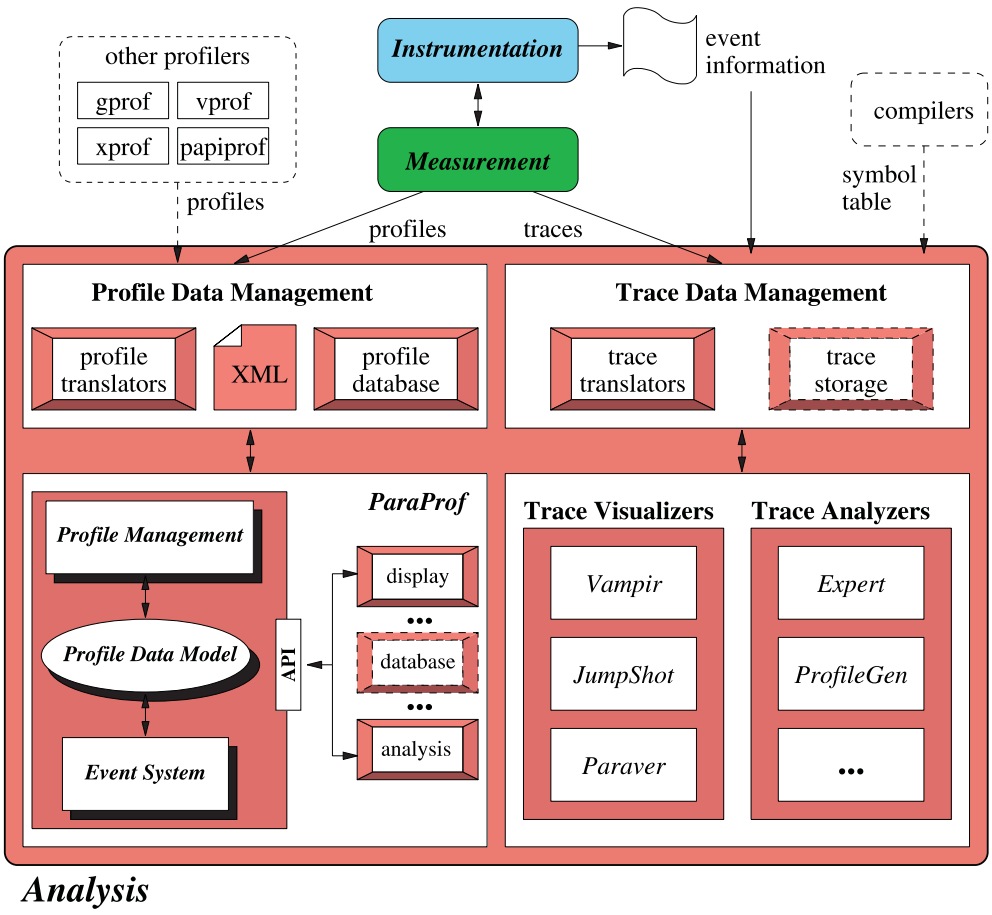

Fig. 3 Architecture of TAU Performance System – Analysis and Visualization.

4.1.1 Interval event interface TAU supports the ability to make performance measurements with respect to event intervals. An *event interval* is defined by its *start events* and its *stop events*. A user may bracket parts of his/her code to specify a region of interest using a pair of start and stop event calls. There are several ways to identify interval events and performance tools have used different techniques. It is probably more recognizable to talk about interval events as timers. To identify a timer, some tools advocate the use of numeric identifiers and an associated table mapping the identifiers to timer names. While it is easy to specify and pass the timer identifier among start and stop routines, it has its drawbacks. Maintaining a table statically might work for languages such as Fortran 90 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 interval event (timer) names are associated with the performance data (timer) object at runtime. An interval event can have a unique name and a signature that can be obtained at runtime. In the case of C++, this is done using runtime type information of objects. Several logically related inter-

val events can be grouped together using an optional *profile group*. A profile group is specified using a name the interval event created.

In the case of C++, the TAU interface leverages the language preprocessing system and object-oriented features. A single interval event macro inserted in a routine is sufficient to track its entry and exit. This is achieved by defining a pair of objects. The first *FunctionInfo* object is a static object whose constructor is invoked exactly once with parameters such as its name, signature, and group. The second *Profiler* object's constructor and destructor are invoked when it comes in and goes out of scope respectively. In this manner, the constructor and destructor mechanism is used to start and stop the interval event associated with the given basic block.

4.1.2 Atomic event interface TAU also allows for events that take place atomically at a specific location in the source code to be identified and tracked. The generic atomic event interface provides a single trigger method with a data parameter. This permits the user to associate application data with such an event. TAU internally uses this interface for some of its performance measurements, such as tracking memory utilization and sizes of messages involved in inter-process synchronization operations using the MPI library. TAU implements the atomic event interface by keeping track of the event name, and the data associated with it. In the profiling mode of measurement, it currently tracks the maxima, minima, mean, standard deviation and the number of samples.

4.1.3 Profile query interface The profile query interface allows the program to interact with the measurement substrate to query the performance metrics recorded by TAU. These metrics are represented as a list of profile statistics associated with each interval and atomic event. For each interval event, a set of exclusive and inclusive values is available in the profile for each performance measurement source. It provides the number of start/stop pairs executed (or the number of calls), and also the number of timers that each timer called in turn. Instead of examining this data at runtime, an application may ask TAU to store this information in files at the end of the execution. The query interface also provides a access for an online performance monitoring tool external to the application.

4.1.4 Event control interface The purpose of the event control interface is to allow the user to enable and disable a group of events 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. Again, the instrumentation here is at the source level and the programmer is inserting event control calls into their program.

4.1.5 Sampling interface TAU's sampling interface can be used to set up interrupts during program execution. Control of interrupt period and selection of system properties to track are provided. Once enabled, an interrupt handler is invoked when a certain duration of time elapses. It tracks one or more entities by calling the atomic event interface. The user can set, enable or disable the sampling of events using the control interface.

While manual instrumentation affords the most flexibility, it can be tedious if the instrumentation involves manually annotating each routine in a large project. For automating the process of instrumentation TAU provides several powerful options described below.

4.2 Preprocessor-Based Instrumentation

The source code of a program can be altered by a preprocessor before it is compiled. This approach typically involves parsing the source code to infer where instrumentation probes are to be inserted. As a example of automatic instrumentation through the preprocessing built into a compiler, TAU's memory allocation/deallocation tracking package can be used to re-direct the references to the C malloc/free calls. The preprocessor invokes TAU's corresponding memory wrapper calls with the added information about the line number and the file. The atomic event interface can then track the size of memory allocated and deallocated to help locate potential memory leaks.

Preprocessor-based instrumentation is also commonly used to insert performance measurement calls at interval entry and exit points in the source code. To support automatic performance instrumentation at the source level, the TAU project has developed the Program Database Toolkit (PDT) (Lindlan et al. 2000). The purpose of PDT, shown in Figure 4 is to parse the application source code and locate the semantic constructs to be instrumented. PDT is comprised of commercial-grade front-ends that emit an intermediate language (IL) file, IL analyzers that walk the abstract syntax tree and generate a subset of semantic entities in program database (PDB) ASCII text files, and a library interface (DUCTAPE) to the PDB files that allows us to write static analysis tools. PDT uses the Edison Design Group's (EDG) C99 and C++ parsers, Mutek Solutions' Fortran 77 and Fortran 90 parser based on EDG, and we have recently added Cleanscape Inc. Flint Fortran 95 parser to PDT. The DUCTAPE library provides TAU a uniform interface to entities from several languages such as C, C++, and Fortran 77/90/95. We have developed a source-to-source instrumentation tool *tau_instr* that uses PDT. It re-writes the original source code with performance annotations to record the interval event transitions (e.g. routine

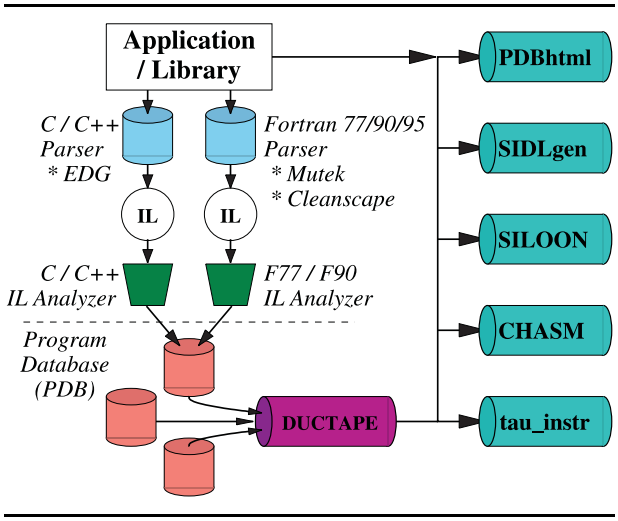

Fig. 4 Program Database Toolkit (PDT).

entry and exit). The instrumented source code is then compiled and linked with the TAU measurement library to produce an executable code. When the application is executed subsequently, performance data is generated.

TAU also supports OpenMP instrumentation using a preprocessor tool called Opari (Mohr). Opari inserts POMP (Mohr et al. 2002) annotations and rewrites OpenMP directives in the source code. TAU's POMP library tracks the time spent in OpenMP routines based on each region in the source code. To track the time spent in user-level routines, Opari instrumentation can be combined with PDT based instrumentation as well.

4.3 Compiler-Based Instrumentation

A compiler can add instrumentation calls in the object code that it generates. There are several advantages to instrumentation at the compiler level. The compiler has full access to source-level mapping information. It has the ability to choose the granularity of instrumentation and can include fine-grained instrumentation. The compiler can perform instrumentation with knowledge of source transformations, optimizations and code generation phases. Flexibility of instrumentation allows for examining the performance of a program to an arbitrary level of detail. Fine-grained instrumentation at the source level, however, interacts with compiler optimizations: instrumentation may inhibit compiler optimizations, and optimizations may corrupt measurement code. We have developed an *Instrumentation-Aware Compiler* that extends a traditional compiler to preserve the semantics of fine-grained performance instrumentation despite aggressive program restructuring. The compiler strips the instrumentation calls from the source code and optimizes the compiled source code. It then re-instruments the optimized code using mappings maintained in the compiler that associate the optimized instructions to the original source code (Shende 2001). The instrumentor uses a fast-breakpoint scheme (Kessler 1990) that replaces an instruction with a branch instruction. The code branches to a new location, the global state (registers) is saved, an instrumentation call is invoked, the global state is restored, and the original replaced instruction is executed. The code then executes a branch to the instruction following the original instruction to continue execution.

4.4 Wrapper Library-Based Instrumentation

A common technique to instrument library routines is to substitute the standard library routine with an instrumented version which in turn calls the orginal routine. The problem is that you would like to do this without having to develop a different library just to alter the calling interface. MPI provides an interface (Forum 1994) that allows a tool developer to intercept MPI calls in a portable manner without requiring a vendor to supply proprietary source code of the library and without requiring the application source code to be modified by the user. This is achieved by providing hooks into the native library with a name-shifted interface and employing weak bindings. Hence, every MPI call can be accessed with its name shifted interface as well. The advantage of this approach is that library-level instrumentation can be implemented by defining a wrapper interposition library layer that inserts instrumentation calls before and after calls to the native routines.

We developed a TAU MPI wrapper library that intercepts calls to the native library by defining routines with the same name, such as *MPI_Send*. These routines then call the native library routines with the name shifted routines, such as *PMPI_Send*. Wrapped around the call, before and after, is TAU performance instrumentation. An added advantage of providing such a wrapper interface is that the profiling wrapper library has access to not only the routine transitions, but also to the arguments passed to the native library. This allows TAU to track the size of messages, identify message tags, or invoke other native library routines. This scheme helps a performance tool track interprocess communication events.

TAU and several other tools (e.g. Upshot (Gropp and Lusk), VampirTrace (Intel Corporation), and EPILOG (Mohr and Wolf 2003)) use the MPI profiling interface. However, TAU can also utilize its rich set of measurement modules that allow profiles to be captured with various types of performance data, including system and hardware data. In addition, TAU's performance grouping capabilities allows MPI events to be presented with respect to high-level categories such as send and receive types.

4.5 Binary Instrumentation

TAU uses DyninstAPI (Buck and Hollingsworth 2000) for instrumenting the executable code of a program. DyninstAPI is a dynamic instrumentation package that allows a tool to insert code snippets into a running program using a portable C++ class library. For DyninstAPI to be useful with a measurement strategy, calls to a measurement library (or the measurement code itself) must be correctly constructed in the code snippets. Our approach for TAU uses the DyninstAPI to construct calls to the TAU measurement library and then insert these calls into the executable code. TAU can instrument a program at runtime, or it can re-write the executable image with calls to the TAU measurement library at routine entry and exit points. TAU's mutator program (tau_run) loads a TAU dynamic shared object (the compiled TAU measurement library) in the address space of the mutatee (the application program). It parses the executable image for symbol table information and generates the list of modules and routines within the modules that are appropriate for instrumentation; TAU routines and Dyninst modules are excluded from consideration. Using the list of routines and their names, unique identifiers are assigned to each routine. The list of routines is then passed as an argument to a TAU initialization routine that is executed once by the mutatee (as a one time code). This initialization routine creates a function mapping table to aid in efficient performance measurement. Code snippets are then inserted at entry and exit transition points in each routine.

Dynaprof (Mucci) is another tool that uses DyninstAPI for instrumentation. It provides a TAU probe that allows TAU measurements to interoperate with Dynaprof instrumentation. An interval event timer is defined to track the time spent in un-instrumented code. This timer is started and stopped around each routine callsite. This enables us to precisely track the exclusive time spent in all other instrumented routines. Dynaprof can also use a PAPI probe and generate performance data that can be read by ParaProf. This illustrates the clear separation between the instrumentation, measurement, and analysis layers in TAU. A user may choose to use TAU instrumentation, measurement, and analysis using tau_run and ParaProf or she may choose Dynaprof for instrumentation, TAU for measurement, and ParaProf or Vampir for analysis, or she may choose Dynaprof for instrumentation, a PAPI probe for measurement, and ParaProf for analysis.

4.6 Interpreter-Based Instrumentation

Interpreted language environments present an interesting target for TAU integration. Often such environments support easy integration with native language modules. In this case, it is reasonable to attempt to recreate the source-based instrumentation in the interpreted language, calling through the native language support to the backend TAU measurement system. However, it is also true that interpreted language environments have built-in support for identifying events and monitoring runtime system actions.

TAU has been integrated with Python by leveraging the Python interpreter's debugging and profiling capabilities to instrument all entry and exit calls. By including the *tau* package and passing the top level routine as a parameter to the tau package's run method, all Python routines invoked subsequently are instrumented automatically at runtime. A TAU interval event is created when a call is dispatched for the first time. At routine entry and exit points, TAU's Python API is invoked to start and stop the interval events. TAU's measurement library is loaded by the interpreter at runtime. Since shared objects are used in Python, instrumentation from multiple levels see the same runtime performance data.

Python is particularly interesting since it can be use to dynamically link and control multi-language executable modules. This raises the issue of how to instrument a program constructed from modules derived from different languages and composed at runtime. We have demonstrated the use of TAU with the Python-based VTF (California Institute of Technology) code from ASCI ASAP center at Caltech. This program involved three modes of instrumentation:

- Python source level
- MPI wrapper interposition library level
- PDT-based automatic instrumentation of Fortran 90, C++, and C modules

The ability to target multiple instrumentation option concurrently makes it possible for TAU to be used effectively in complex programming systems.

4.7 Component-Based Instrumentation

Component technology extends the benefits of scripting systems and object-oriented design to support reuse and interoperability of component software, transparent of language and location (Szyperski 1997). A *component* is a software object that implements certain functionality and has a well-defined interface that conforms to a component architecture defining rules for how components link and work together (Bernholdt et al. 2005). It consists of a collection of *ports*, where each port represents a set of functions that are publicly available. Ports implemented by a component are known as *provides* ports, and other ports that a component is uses are known as *uses* ports.

The Common Component Architecture (CCA) (CCA Forum) is a component-based methodology for develop-

ing scientific simulation codes. The architecture consists of a framework which enables components (embodiments of numerical algorithms and physical models) to work together. Components are peers and derive no implementation from others. Components publish their interfaces and use interfaces published by others. Components publishing the same interface and with the same functionality (but perhaps implemented via a different algorithm or data structure) may be transparently substituted for each other in a code or a component assembly. Components are compiled into shared libraries and are loaded in, instantiated and composed into a useful code at runtime.

How should a component-based program be instrumented for performance measurement? The challenge here is in supporting an instrumentation methodology that is consistent with component-based software engineering. The approach taken with TAU for CCA was to develop a TAU performance component that other components could use for performance measurement. The TAU instrumentation API is thus recreated as the performance component's interface, supporting event creation, event control, and performance query. There are two ways to instrument a component based application using TAU. The first requires calls to the performance component's measurement port to be added to the source code. This is useful for fine-grained measurements inside the component. The second approach interposes a proxy component in front of a component, thus intercepting the calls to its provides port. In this case, for each edge that represents a port in the component connection graph, we can interpose the proxy along that edge. A proxy component implements a port interface and has a provides and a uses port. The provides port is connected to the caller's uses port and its uses port is connected to the callee's provides port. The proxy performs measurements using TAU's Mastermind or Measurement port (Shende et al. 2003; Ray et al. 2004) as shown in the wiring diagram of CFRFS CCA combustion component ensemble in Figure 5.

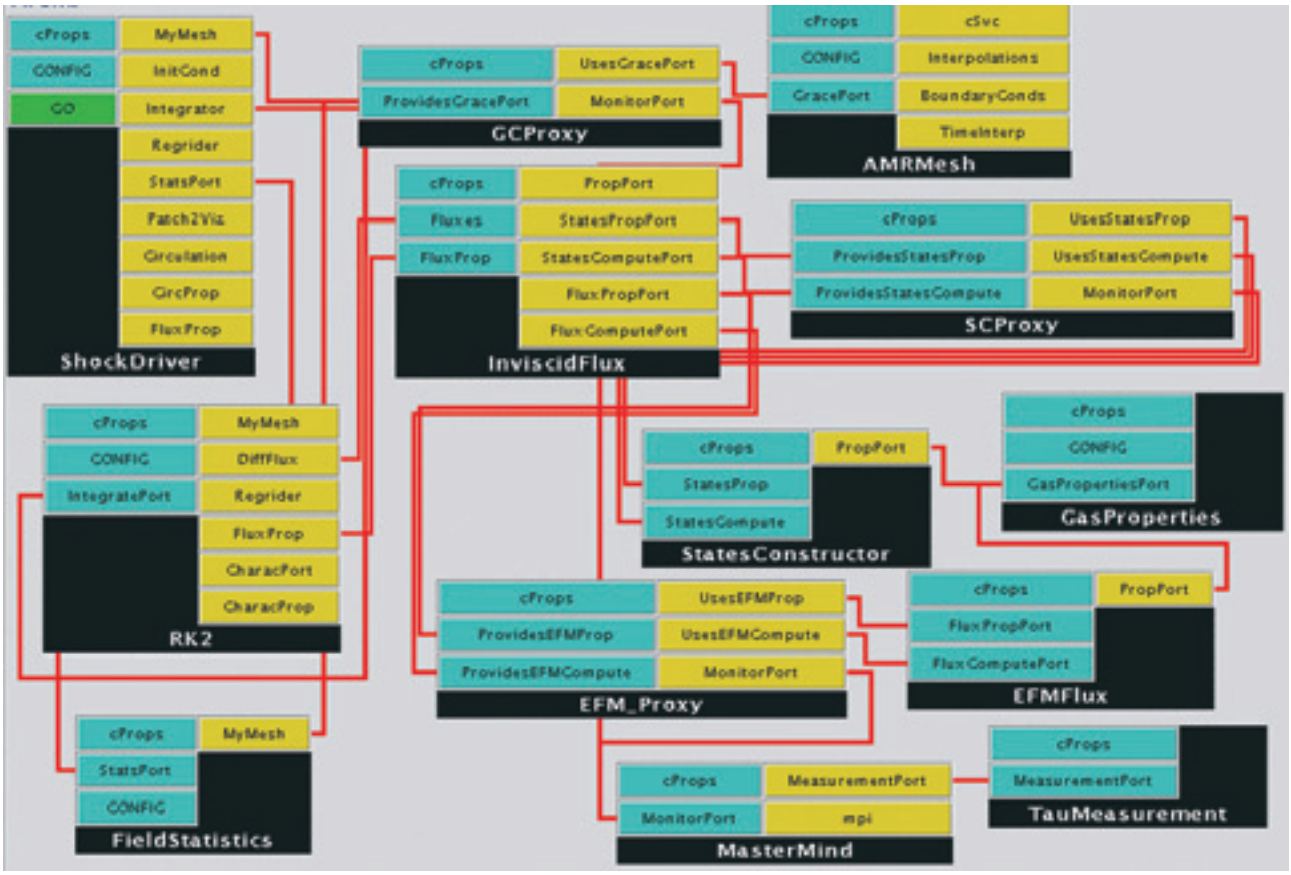

Fig. 5 Snapshot of the component application, as assembled for execution. We see three proxies (for AMRMesh, EFMFlux and States), as well as the TauMeasurement and Mastermind components to measure and record performance-related data.

To aid in the construction of proxies, it is important to note that we only need to construct one proxy component for each port. Different components that implement a given port use the same proxy component. To automate the process of creating a proxy component, TAU's proxy generator uses PDT to parse the source code of a component that implements a given port. It infers the arguments and return types of a port and its interfaces and constructs the source code of a proxy component, which when compiled and instantiated in the framework allows us to measure the performance of a component without any changes to its source or object code. This provides a powerful capability to build performance-engineered scientific components that can provide computational quality of service (Norris et al. 2004) and allows us to build *intelligent*, performance-aware components.

4.8 Virtual Machine-Based Instrumentation

Support of performance instrumentation and measurement in language systems based on virtual machine (VM) execution poses several challenges. Consider Java and the JVM. Currently, Java 2 (JDK1.2+) incorporates the Java Virtual Machine Profiler Interface (JVMPI) (SUN Microsystems Inc.; Viswanathan and Liang 2000) which we have used for our work. This interface is re-organized in JDK 1.5+ as Java Virtual Machine Tool Interface (JVMTI). JVMPI provides profiling hooks into the virtual machine and allows a profiler agent to instrument the Java application without any changes to the source code, bytecode, or the executable code of the JVM. This is ideal since JVMPI provides a wide range of events that it can notify to the agent, including method entry and exit, memory allocation, garbage collection, and thread start and stop; see the Java 2 reference for more information. When the profiler agent is loaded in memory, it registers the events of interest and the address of a callback routine to the virtual machine using JVMPI. When an event takes place, the virtual machine thread generating the event calls the profiler agent callback routine with a data structure that contains event specific information. The profiling agent can then use JVMPI to get more detailed information regarding the state of the system and where the event occurred. The downside of this approach is that JVMPI is a heavyweight mechanism.

When the TAU agent is loaded in the JVM as a shared object, a TAU initialization routine is invoked. It stores the identity of the virtual machine and requests the JVM to notify it when a thread starts or terminates, a class is loaded in memory, a method entry or exit takes place, or the JVM shuts down. When a class is loaded, TAU examines the list of methods in the class and creates an association of the name of the method and its signature, as embedded in the TAU object, with the method identifier obtained, using the TAU Mapping API (see the TAU User's Guide (University of Oregon b)). When a method entry takes place, TAU performs measurements and correlates these to the TAU object corresponding to the method identifier that it receives from JVMPI. When a thread is created, it creates a top-level routine that corresponds to the name of the thread, so the lifetime of each user and system level thread can be tracked.

To deal with Java's multi-threaded environment, TAU uses a common thread layer for operations such as getting the thread identifier, locking and unlocking the performance database, getting the number of concurrent threads, and so on. (This is an example of the benefit of basing TAU on a general computation model.) The thread layer is then used by the multiple instrumentation layers. When a thread is created, TAU registers it with its thread module and assigns an integer identifier to it. It stores this in a thread-local data structure using the JVMPI thread API described above. It invokes routines from this API to implement mutual exclusion to maintain consistency of performance data. It is important for the profiling agent to use the same thread interface as the virtual machine that executes the multi-threaded Java applications. This allows TAU to lock and unlock performance data in the same way as application level Java threads do with shared global application data. TAU maintains a per-thread performance data structure that is updated when a method entry or exit takes place. Since this is maintained on a per thread basis, it does not require mutual exclusion with other threads and is a low-overhead scalable data structure. When a thread exits, TAU stores the performance data associated with the thread to stable storage. When it receives a JVM shutdown event, it flushes the performance data for all running threads to the disk.

Shende and Malony (2003) demonstrated how MPI events can be integrated with Java language events from the JVM. Here, the JVM was running the Just-in-time (JIT) compiler where the Java bytecode is converted into native code on the fly as the application executes. TAU can also be used to profile Java code using Sun's HotSpot compiler embedded within the JVM while it transforms time-consuming segments of code to native code at runtime. This is in contrast to the operation of the JIT compiler where all bytecode is converted to native code at runtime.

4.9 Multi-Level Instrumentation

As the source code undergoes a series of transformations in the compilation, linking, and execution phases, it poses several constraints and opportunites for instrumentation. Instead of restricting the choice of instrumentation to one phase in the program transformation, TAU allows multiple instrumentation interfaces to be deployed concurrently

for better coverage. It taps into performance data from multiple levels and presents it in a consistent and a uniform manner by integrating events from different languages and instrumentation levels in the same address space. TAU maintains performance data in a common structure for all events and allows external tools access to the performance data using a common interface.

4.10 Selective Instrumentation

In support of the different instrumentation schemes TAU provides, a facility for selecting which of the possible events to instrument has been developed (Malony et al. 2003). The idea is to record a list of performance events to be included or excluded by the instrumentation in a file. The file is then used during the instrumentation process to restrict the event set. The basic structure of the file is a list of names separated into include and exclude lists. File names can be given to restrict instrumentation focus.

The selective instrumentation mechanism is being used in TAU for all automatic instrumentation methods, including PDT source instrumentation, DyninstAPI executable instrumentation, and component instrumentation. It has proven invaluable as a means to both weed out unwanted performance events, such as high frequency, small routines that generate excessive measurement overhead, and provide easy event configuration for customized performance experiments.

4.11 TAU_COMPILER

To simplify the integration of the source instrumentor and the MPI wrapper library in the build process, TAU provides a tool, tau_compiler.sh that can be invoked using a prefix of `$(TAU_COMPILER)` before the name of the compiler. For instance, in an application makefile, the variable:

```
F90=mpxlf90
```

is modified to

```
F90=$(TAU_COMPILER) mpxlf90.
```

This tool invokes the compiler internally after extracting the names of source or object files and compilation parameters. During compilation, it invokes the parser from PDT, then the *tau_instrumentor* for inserting measurement probes into the source code, and compiles the instrumented version of the source to generate the desired object file. It can distinguish between object code creation and linking phases of compilation and during linking, it inserts the MPI wrapper library and the TAU measurement library in the link command line. In this manner, a user can easily integrate TAU's portable performance instrumentation in the code generation process. Optional parameters can be passed to all four compilation phases.

5 Measurement

All TAU instrumentation code makes calls to the TAU measurement system through an API that provides a portable and consistent set of measurement services. Again, the instrumentation layer is responsible for defining the performance events for an experiment, establishing relationships between events (e.g. groups, mappings), and managing those events in the context of the parallel computing model being used. Using the TAU measurement API, event information is passed in the probe calls to be used during measurement operations to link events with performance data. TAU supports parallel profiling and parallel tracing. It is in the measurement system configuration and usage where all choices for what performance data to capture and in what manner are made. Thus, performance experiments are created by selecting the key events of interest to observe and by configuring measurement modules together into a particular composition of measurement capabilities (Dongarra et al. 2003).

In the sections that follow, we will discuss in detail what the TAU measurement layer provides, first from the point of view of profiling, and then of tracing. We begin with a discussion of the sources of performance data TAU supports.

The TAU measurement system is the heart of TAU's capabilities. It is highly robust, scalable, and has been ported to all HPC platforms.

5.1 Performance Data Sources

TAU provides access to various sources of performance data. Time is perhaps the most important and ubiquitous data type, but it comes in various forms on different system platforms. TAU provides the user with a flexible choice of time sources based on what the range of sources a particular system supplies. At the same time, it abstracts the timer interface so as to insulate the rest of the measurement system from the nuances of different timer implementations. In a similar manner, TAU integrates alternative interfaces for access to hardware counters (PAPI (Browne et al. 2000) and PCL (Berrendorf, Ziegler, and Mohr) are supported) and other system-accessible performance data sources. Through TAU configuration, all of the linkages to these packages are taken care of.

Within the measurement system, TAU allows for multiple sources of performance data to be concurrently active. That is, it is possible for both profiling and tracing to work with multiple performance data. TAU also recognizes that some performance data may come directly from the

parallel program. This is supported in two ways. First, the TAU API allows the user to specify a routine to serve as a counter source during performance measurement. Second, the TAU measurement system supplies some standard events and counters that can be used to track program-related performance (e.g. tracking memory utilization and sizes of messages).

5.2 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 wall-clock time) and associated statistics for all performance events in the program. There are different statistics kept for interval events (such as routines or statements in the program) versus atomic events. For interval events, TAU profile measurements compute exclusive and inclusive metrics spent in each routine. Time is a commonly used metric, but any monotonically increasing resource function can be used. Typically one metric is measured during a profiling run. However, the user may configure TAU with the `-MULTIPLECOUNTERS` configuration option and then specify up to 25 metrics (by setting environment variables `COUNTER[1-25]`) to track during a single execution. For atomic events, different counters can be used. As indicated above, statistics measured include maxima, minima, mean, standard deviation, and the number of samples. Internally, the TAU measurement system maintains a profile data structure for each node/context/thread. When the program execution completes, a separate profile file is created for each. The profiling system is optimized to work with the target platform and profiling operations are very efficient.

5.3 Flat Profiling

The TAU profiling system supports several profiling variants. The most basic and standard type of profiling is called *flat profiling*. If TAU is being used for flat profiling, performance measurements are kept for interval events only. For instance, flat profiles will report the *exclusive* performance (e.g. time) for a routine, say *A*, as the amount of time spent executing in *A* exclusively. Any time spent in routines called by *A* will be represented in *A*'s profile as *inclusive* time, but it will not be differentiated with respect to the individual routines *A* called. Flat profiles also keep information on the number of times *A* was called and the number of routines (i.e. events) called by *A*. Again, TAU will keep a flat profile for every *node/context/thread* of the program's execution.

TAU implements a sophisticated runtime infrastructure for gaining both profiling measurement efficiency and robustness. In particular, we decided to maintain internally a runtime event callstack that shrinks and grows at every interval event exit and entry. It is a simple matter to account for inclusive and exclusive performance using the event callstack. The real power of the callstack is demonstrated for the other profiling forms. It is important to understand that the callstack is a representation of the nesting of interval performance events. This makes it more than just a routine callstack representation.

5.3.1 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 performance along the dynamic routine (event) calling paths of an application. We speak of the *depth* of a callpath as the number of routines represented in the callpath. A callpath of *depth* 1 is a flat profile. A callpath of depth k represents a sequence of the last $k - 1$ routines called by a routine at the head of the callpath. The key concept to understand for callpath profiling is that a callpath represents a performance event. Just as a callpath of depth 1 will represent a particular routine and TAU will profile exclusive and inclusive performance for that routine, every unique callpath of depth k in a program's execution will represent a unique performance event to be profiled.

Unlike flat profiling, the problem with callpath profiling is that the identities of all $k - 1$ depth calling paths ending at a routine may not be, 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 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. There are two problems with this approach. One, it is not a profile-based measurement, and two, the trace generated may be excessively large.

Unfortunately, the 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 (DeRose and Wolf 2002). Unfortunately, this is not a robust solution

for several reasons. First, source-based callpath analysis is non-trivial and may only be 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.

As noted above, the TAU measurement system maintains a callstack that is updated with each entry/exit performance event. Thus, to determine the $k-1$ length callpath when an event (e.g. routine) is entered, all that is necessary is to traverse up the callstack to determine the last events that define the callpath. If this is a newly encountered callpath, it represents a new event, and 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.

Although performance events in TAU are handled dynamically, in the sense that they are not pre-determined and pre-assigned event identities, "standard" performance events will have pre-allocated profile data structures, as a result of the instrumentation inserted in the program code. Unfortunately, callpaths do not occur as a result of specific event instrumentation, but instead as a result of the state of the event callstack. Thus, new callpaths occur dynamically, requiring new profile data objects to be created at runtime. TAU builds a profile object for each callpath encountered in an associative map and creates a key to use to retrieve it. The key is formed from the callpath depth and callpath event names. It is constructed on the fly when an interval entry call takes place. Thus, no string operations are performed in looking up the key in the hash table. To compare two keys, we first examine the callpath depth by looking at the first element of the two arrays. If they're equal, then we traverse the other elements comparing a pair of addresses at each stage. When TAU is configured with the `-PROFILECALLPATH` configuration option, callpath profiling is enabled. A user sets the desired callpath depth as a runtime parameter by setting the environment variable `TAU_CALLPATH_DEPTH` to the appropriate value. If it is not set, a default value of 2 is assumed.

5.3.2 Calldepth profiling TAU's callpath profiling will generate a profile for each callpath of a depth designated by `TAU_CALLPATH_DEPTH`, not just those that include the topmost *root* event. For some performance evaluation studies, it is desired to see how the performance is distributed across program parts from a top-down, hierarchical perspective. Thus, a parallel profile that showed how performance data was distributed at different levels of an unfolding *event call tree* could help to understand the performance better. TAU's implementation of *calldepth profiling* does just that. It allows the user to configure TAU with the `-DEPTHLIMIT` option and specify in the the environment variable `TAU_DEPTH_LIMIT` how far down the event call tree to observe performance. In this case, the profiles created show performance for each callpath in the rooted call tree pruned to the chosen depth. The implementation of calldepth profiling is similar to callpath profiling in that it requires dynamic event generation and profile object creation, but it benefits from certain efficiencies in pruning its search on the callstack.

5.3.3 Phase profiling While callpath profiling and calldepth profiling allow the distribution of performance to be understood relative to event calling relationships, it is equally reasonable to want to see performance data relative to execution *state*. The concept of a *phase* is common in scientific applications, both in terms of how developers think about the structural, logical, and numerical aspects of a computation, and how performance can be interpreted. It therefore worthwhile to consider whether support for phases in performance measurement can aid in the interpretation of performance information. *Phase profiling* is an approach to profiling that measures performance relative to the phase of execution. TAU has implemented a phase profiling API that is used by the developer to create phases and mark their entry and exit. When TAU is configured with the `-PROFILEPHASE` option, TAU will effectively generate a separate profile for each phase in the program's execution.

Internally, phase profiling support in TAU builds on similar mechanisms used in callpath profiling. A phase event (enter and exit phase) activates special processing in TAU to record the transition between phases. A phase can be static (where the name registration takes place exactly once) or dynamic (where it is created each time). Phases can also be nested, in which case profiling follows normal scoping rules and is associated with the closest parent phase obtained by traversing up the callstack. Phases should not overlap, as it also true for interval events (Shende et al. 1998). Each thread of execution in an application has a default phase and this corresponds to the top level event. This top level phase contains other routines and phases that it directly invokes, but excludes routines called by child phases.

5.4 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.

TAU implements a robust, portable, and scalable performance tracing facility. With tracing enabled, every *node/context/thread* will generate a trace for instrumented events. TAU will write traces in its modern trace format as well as in VTF3 (Seidl 2003) format. Support for a counter value to be included in event records is fully implemented. In addition, certain standard events are known by TAU's tracing system, such as multi-threading operations and message communication. TAU writes performance traces for post-mortem analysis, but also supports an interface for online trace access. This includes mechanisms for online and hierarchical trace merging (Brunst et al. 2003; Brunst, Nagel, and Malony 2003).

The following describes important aspects of TAU tracing system in more detail.

5.4.1 Dynamic event registration For runtime trace reading and analysis, it is important to understand what takes place when TAU records performance events in traces. The first time an event takes place in a process, it registers its properties with the TAU measurement library. Each event has an identifier associated with it. These identifiers are generated dynamically at runtime as the application executes, allowing TAU to track only those events that actually occur. This is in contrast to static schemes that must predefine all possible events that could possibly occur. The main issue here is how the event identifiers are determined. In a static scheme, event IDs are drawn from a pre-determined global space of IDs, which restricts the scope of performance measurement scenarios. This is the case with most other performance tracing systems. In our more general and dynamic scheme, the event identifiers are generated on the fly, local to a context. Depending on the order in which events first occur, the IDs may be different for the same event (i.e. events with the same name) across contexts. When event streams are later merged, these local event identifiers are mapped to a global identifier based on the event name.

Previously, TAU wrote the event description files to disk when the application terminated. While this scheme was sufficient for post-mortem merging and conversion of event traces, it could not be directly applied for online analysis of event traces. This was due to the absence of event names that are needed for local to global event identifier conversion. To overcome this limitation, we have re-designed our trace merging tool, *tau_merge*, so it executes concurrently with the executing application generating the trace files. From each process's trace file, tau_merge reads event records and examines their globally synchronized timestamps to determine which event is to be recorded next in the ordered output trace file. When it encounters a local event identifier that it has not seen before, it reads the event definition file associated with the given process and updates its internal tables to map that local event identifier to a global event identifier using its event name as a key. The trace generation library ensures that event tables are written to disk before writing trace records that contain one or more new events. A new event is defined as an event whose properties are not recorded in the event description file written previously by the application. This scheme, of writing event definitions prior to trace records, is also used by the tau_merge tool while writing a merged stream of events and event definitions. It ensures that the trace analysis tools down the line that read the merged traces also read the global event definitions and refresh their internal tables when they encounter an event for which event definitions are not known.

5.4.2 TAU trace input library To make the trace data available for runtime analysis, we implemented the TAU trace input library. It can parse binary merged or unmerged traces (and their respective event definition files) and provides this information to an analysis tool using a trace analysis API. This API employs a callback mechanism where the tool registers callback handlers for different events. The library parses the trace and event description files and notifies the tool of events that it is interested in, by invoking the appropriate handlers with event specific parameters. We currently support callbacks for finding the following:

- Clock period used in the trace
- Message send or receive events
- Mapping event identifiers to their state or event properties
- Defining a group identifier and associated group name
- Entering and leaving a state

Each of these callback routines have event specific parameters. For instance, a send event handler has source and destination process identifiers, the message length, and its tag as its parameters. Besides reading a group of records from the trace file, our API supports file management routines for opening or closing a trace file and for navigating the trace file by moving the location of the current file pointer to an absolute or relative event position. It supports both positive and negative event offsets. This allows the analysis tool to read, for instance, the last

10000 events from the tail of the event stream. The trace input library is used by VNG (Brunst et al. 2003) to analyze at runtime the merged binary event stream generated by an application instrumented with TAU.

5.5 Measurement Overhead

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. As we have discussed, events may be atomic or paired to mark certain begin and end points. 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* (Malony 1990). Most performance tools, including TAU, address the problem of performance perturbation indirectly by reducing the overhead of performance measurement.

We define *performance intrusion* as the amount of performance measurement overhead incurred during a performance experiment. Thus, intrusion will be a product of the numbers of events that occurred during execution and the measurement overhead for processing each event. We define *performance accuracy* as the degree to which our performance measures correctly represent "actual" performance. That is, accuracy is associated with error. If we are trying to measure the performance of small events, the error will be higher because of the measurement uncertainty that exists due to the relative size of the overhead versus the event. If we attempt to measure a lot of events, the performance intrusion may be high because of the accumulated measurement overhead, regardless of the measurement accuracy for that event.

Performance experiments should be concerned with both performance intrusion and performance accuracy, especially in regards to performance perturbation. TAU is a highly-engineered performance system and delivers excellent measurement efficiencies and low measurement overhead. However, it is easy to construct naively an experiment that will result in significant performance intrusion. Indeed, TAU's default instrumentation behavior is to enable all events it can instrument. We are developing a set of tools in TAU to help the user manage the degree of performance instrumentation as a way to better control performance intrusion. The approach is to help the user identify performance events that have either poor measurement accuracy (i.e. they are small) or a high frequency of occurrence. Once these events are identified, the event selection mechanism described above can be used to reduce the instrumentation degree in the next experiment, thereby reducing performance intrusion in the next program run.

5.6 Overhead Compensation

Unfortunately, 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. Optimized coarse-grained instrumentation helps the process of improving the accuracy of measurements using selective instrumentation. However, any instrumentation perturbs a program and modifies its behavior. The distortion in gathered performance data could be significant for a parallel program where the effects of perturbation are compounded by parallel execution and accumulation of overhead from remote processes. Such distortions are typically observed in wait times where processes synchronize their operation. Given an event stream stored in log files, it is possible under certain assumptions to correct the performance perturbation in a limited manner by compensating for the measurement overhead and correcting event orderings based on known causality constraints, such as imposed by inter-process communication (Malony 1990; Sarukkai and Malony 1993). Tracing the program execution is not always feasible due to the high volume of performance data generated and the amount of trace processing needed.

We have developed techniques in TAU profiling to compensate for measurement overhead at runtime. Using an estimate of measurement overhead determined at runtime, TAU will subtract this overhead during profile calculation of inclusive performance. The way this is accomplished is quite clever by tracking the number of descendant events and adjusting the total inclusive time at event exit. This inclusive value is then used to compute the corrected exclusive time for the routine by subtracting the corrected inclusive time from the exclusive time of each routines parent. A TAU measurement library configured with the `-COMPENSATE` configuration option performs online removal of overhead during the measurement stage.

5.7 Performance Mapping

The ability to associate low-level performance measurements with higher-level execution semantics is important

in understanding parallel performance data with respect to application structure and dynamics. Unfortunately, most performance systems do not provide such support except in their analysis tools, and then only in a limited manner. The TAU measurement system implements a novel performance observation feature call *performance mapping* (Shende 2001). The idea is to provide a mechanism whereby performance measurements, made by the occurrence of instrumented performance events, can be associated with semantic abstractions, possible at a different level of performance observation. For instance, a measurement of the time spent in a MPI communication routine might be associated with a particular phase of program execution.

TAU has implemented performance mapping as an integral part of its measurement system. In addition to providing an API for application-level performance mapping, TAU uses mapping internally to implement callpath profiling, calldepth profiling, and phase profiling. In the case of phase profiling, TAU's measurement system treats a phase profile as a callpath profile of depth 2. Here, a caller-callee relationship is used to represent phase interactions. At a phase or event entry point, we traverse the callstack until a phase is encountered. Since the top level event is treated as the default application phase, each routine invocation occurs within some phase. To store the performance data for a given event invocation, we need to determine if the current *(event, phase)* tuple has executed before. To do this, we construct a key array that includes the identities of the current event and the parent phase. This key is used in a lookup operation on a global map of all *(phase, timer)* relationships. If the key is not found, a new profiling object is created with the name that represents the parent phase and the currently executing event or phase. In this object, we store performance data relevant to the phase. If we find the key, we access the profiling object and update its performance metrics. As with callpath profiling, a reference to this object is stored to avoid a second lookup at the event exit.

6 Analysis

TAU gives us the ability to track performance data in widely diverse environments, and thus provides a wealth of information to the user. The usefulness of this information, however, is highly dependent on the ability of analysis toolsets to manage and present the information. As the size and complexity of the performance information increases, the challenge of performance analysis and visualization becomes more difficult. It has been a continuing effort to include as part of TAU a set of analysis tools which can scale not only to the task of analyzing TAU data, but also to a more diverse arena outside of the TAU paradigm. This section discusses the developement of these tools, and the resulting benefits to the user in performing the often complex task of analyzing performance data.

Our approach in this section will be to show the use of the TAU analysis tools on a single parallel application, S3D (Subramanya and Reddy 2000). S3D is a high-fidelity finite difference solver for compressible reacting flows which includes detailed chemistry computations.

6.1 ParaProf

The TAU performance measurement system is capable of producing parallel profiles for thousands of processes consisting of hundreds of events. Scalable analysis tools are required to handled this amount of detailed performance information. The *ParaProf* parallel profile analysis tool represents more than six years of development. Shown in Figure 6, ParaProf abstracts four key components in its design: the *Data Source System* (DSS), the *Data Management System* (DMS), the *Event System* (ES), and the *Visualization System* (VS). Each component is independent, and provides well-defined interfaces to other components in the system. The result is high extensibility and flexibility, enabling us to tackle the issues of re-use and scalability.

Current performance profilers provide a range of differing data formats. As done in HPCView (Mellor-Crummey, Fowler, and Marlin 2002) external translators have typically been used to merge profile data sets. Since much commonality exists in the profile entities being represented, this is a valid approach, but it requires the adoption of a common format. ParaProf's DSS addresses this issue in a different manner. DSS consists of two parts. One, DSS can be configured with profile input modules to read profiles from different sources. The existing translators provides a good starting point to implement these modules. An input module can also support interfaces for communication with profiles stored in files, managed by performance databases, or streaming continuously across a network. Two, once the profile is input, DSS converts the profile data to a more efficient internal representation.

The DMS provides an abstract representation of performance data to external components. Its supports many advanced capabilities required in a modern performance analysis system, such as derived metrics for relating performance data, cross experiment analysis for analyzing data from disparate experiments, and data reduction for elimination of redundant data, thus allowing large data sources to be tolerated efficiently. The importance of sophisticated data management and its support for exposing data relationships is an increasingly important area of research in performance analysis. The DMS design provides a great degree of flexibility for developing new techniques that can be incorporated to extend its function.

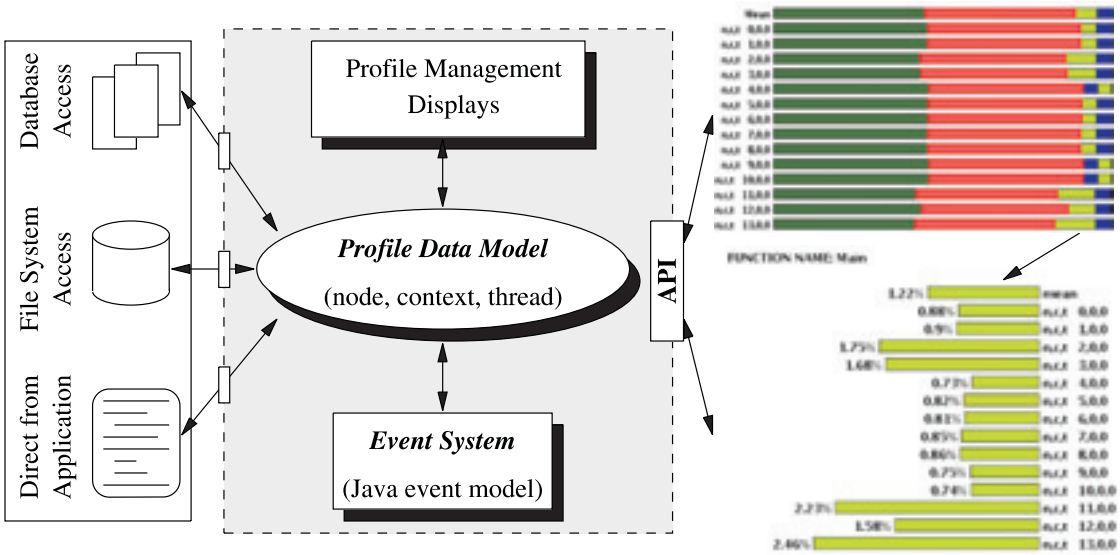

Fig. 6 ParaProf Architecture.

The VS components is responsible for graphical profile displays. It is based on the Java2D platform, enabling us to take advantage of a very portable development environment that continues to increase in performance and reliability. Analysis of performance data requires representations from a very fine granularity, perhaps of a single event on a single node, to displays of the performance characteristics of the entire application. ParaProf's current set of displays range from purely textual based to fully graphical. Significant effort has been put into making the displays highly interactive and fast to draw. In addition, it is relatively easy to extend the display types to better show data relations.

Lastly, in the ES, we have provided a well-defined means by which these components can communicate various state changes, and requests to other components in ParaProf. Many of the display types are hyper-link enabled, allowing selections to be reflected across currently open windows. Support for runtime performance analysis and application steering, coupled with maintaining connectivity with remote data repositories has required us to focus more attention on the ES, and to treat it as a wholly separate component system.

To get a sense of the type of analysis displays ParaProf supports, Figure 7 shows the S3D flat profile (stacked view) on sixteen processes. Different events are color coded. Clicking on one event, `INT_RTE`, ParaProf will display that event's performance in a separate window, as shown in Figure 8 for `INT_RTE`'s exclusive time. When callpath profile data is available, ParaProf can reconstruct the event calling graph and display performance statistics in a callgraph display, as seen in Figure 9. Here the size of the node is determined by its inclusive time and the color is mapped to exclusive time, red being the most.

6.2 Performance Database Framework

Empirical performance evaluation of parallel and distributed systems or applications often generates significant amounts of performance data and analysis results from multiple experiments and trials as performance is investigated and problems diagnosed. However, the management of performance data from multiple experiments can be logistically difficult, impeding the effective analysis and understanding of performance outcomes. The Performance Data Management Framework (PerfDMF) (Huck et al. 2005) provides a common foundation for parsing, storing, querying, and analyzing performance data from multiple experiments, application versions, profiling tools and/or platforms. The PerfDMF design architecture is presented below. We describe the main components and their interoperation.

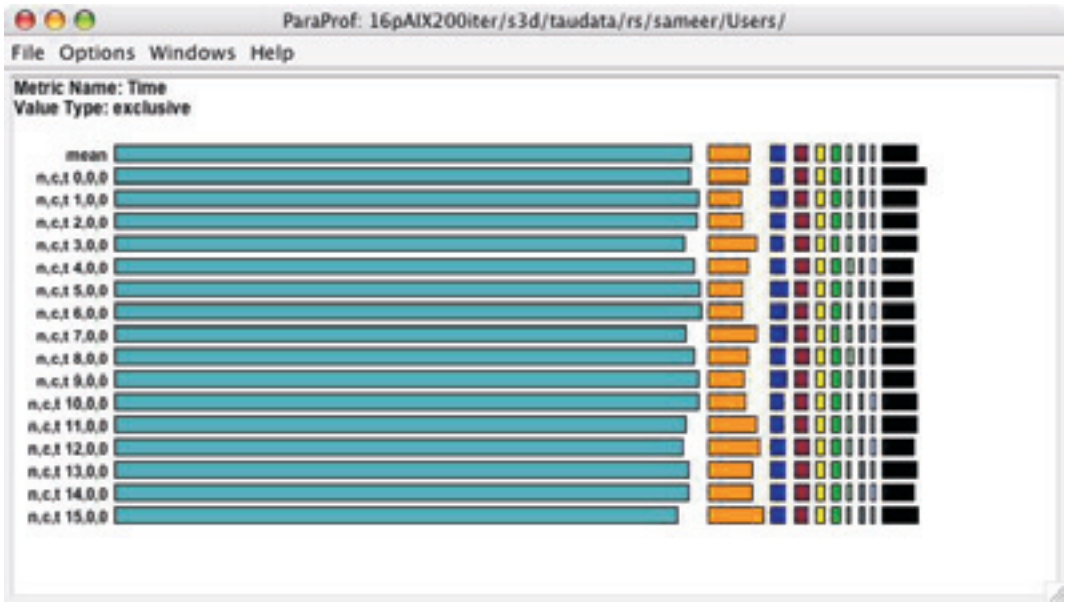

Fig. 7 ParaProf view of S3D flat profile.

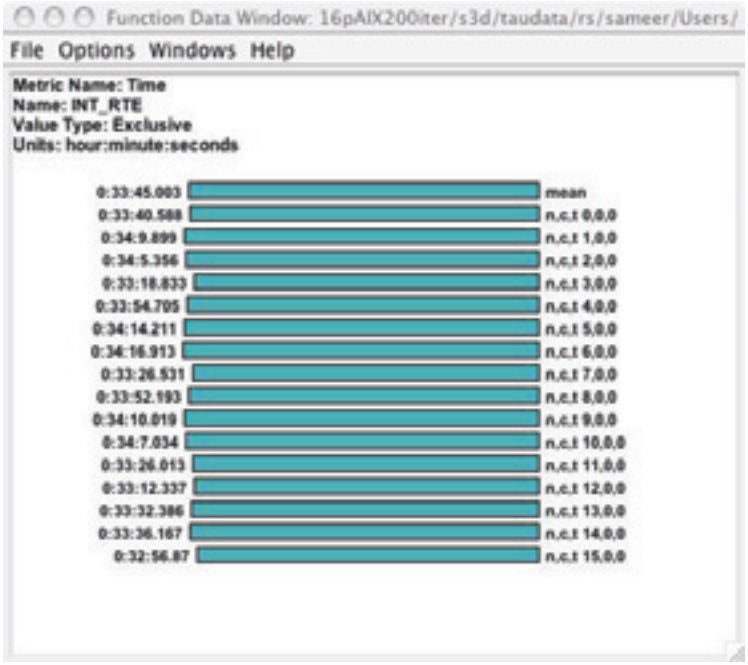

Fig. 8 ParaProf view of S3D INT_RTE exclusive time.

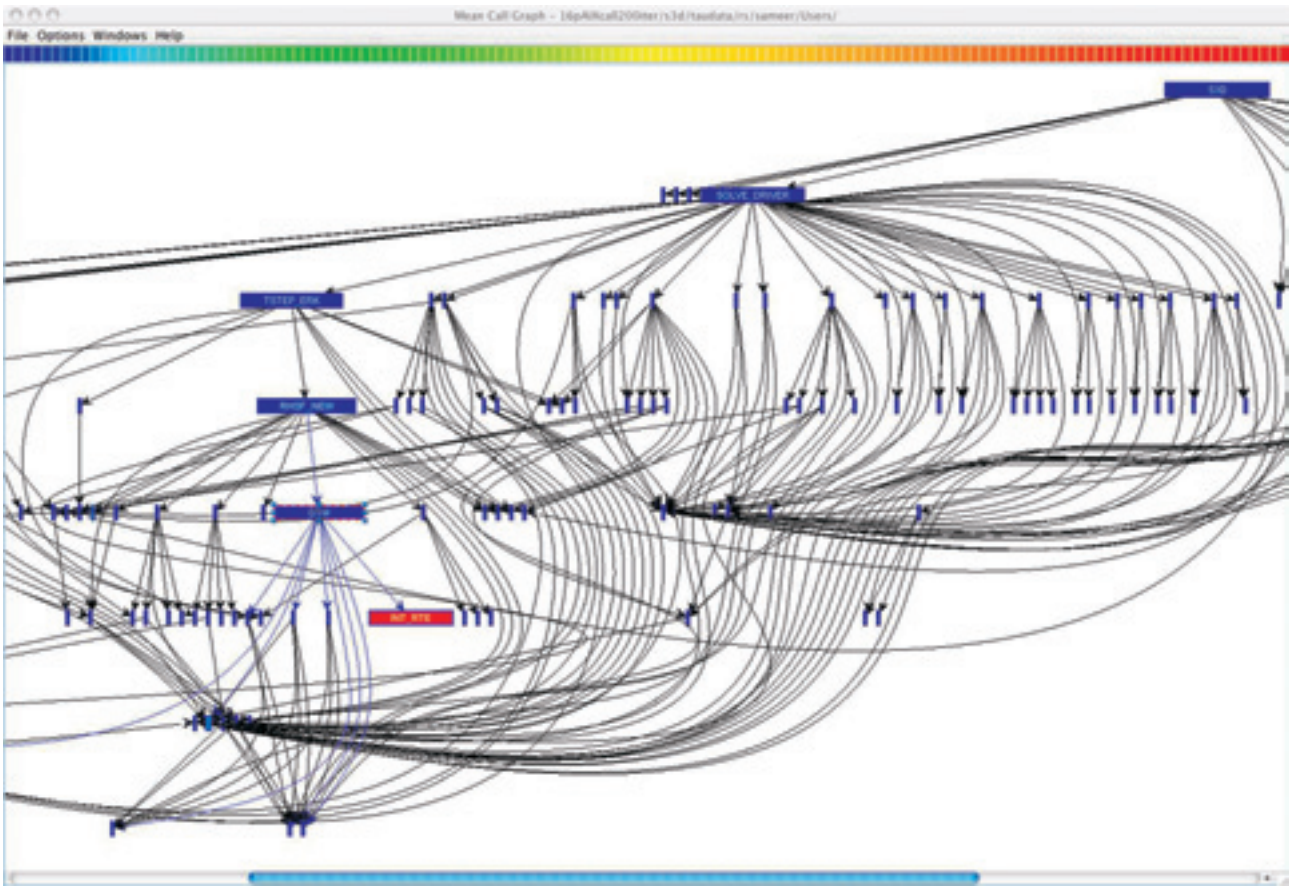

Fig. 9 ParaProf view of S3D callgraph.

PerfDMF consists of four main components: *profile input/output*, *profile database*, *database query and analysis API*, and *profile analysis toolkit*. Figure 10 shows a representation of these four components, and their relationships. PerfDMF is designed to parse parallel profile data from multiple sources. This is done through the use of embedded translators, built with PerfDMF's data utilities and targeting a common, extensible parallel profile representation. Currently supported profile formats include gprof (Graham, Kessler, and McKusick 1982) TAU profiles (University of Oregon a), dynaprof (Mucci), mpiP (Vetter and Chambreau), HPMtoolkit (IBM) (DeRose 2001), and Perfsuite (psrun) (Ahn et al.). (Support for SvPablo (DeRose and Reed 1998) is being added.) The profile data is parsed into a common data format. The format specifies profile data by node, context, thread, metric and event. Profile data is organized such that for each combination of these items, an aggregate measurement is recorded. The similarities in the profile performance data gathered by different tools allowed a common organization to be used. Export of profile data is also supported in a common XML representation. In the future, we may also offer exporters to a subset of the formats above.

The profile database component is the center of PerfDMF's persistent data storage. It builds on robust SQL relational database engines, some of which are freely distributed. The currently supported Relational Database Management Systems (DBMS) are PostgreSQL (PostgreSQL), MySQL (MySQL), Oracle (Oracle Corporation) and DB2 (IBM). The database component must be able to handle

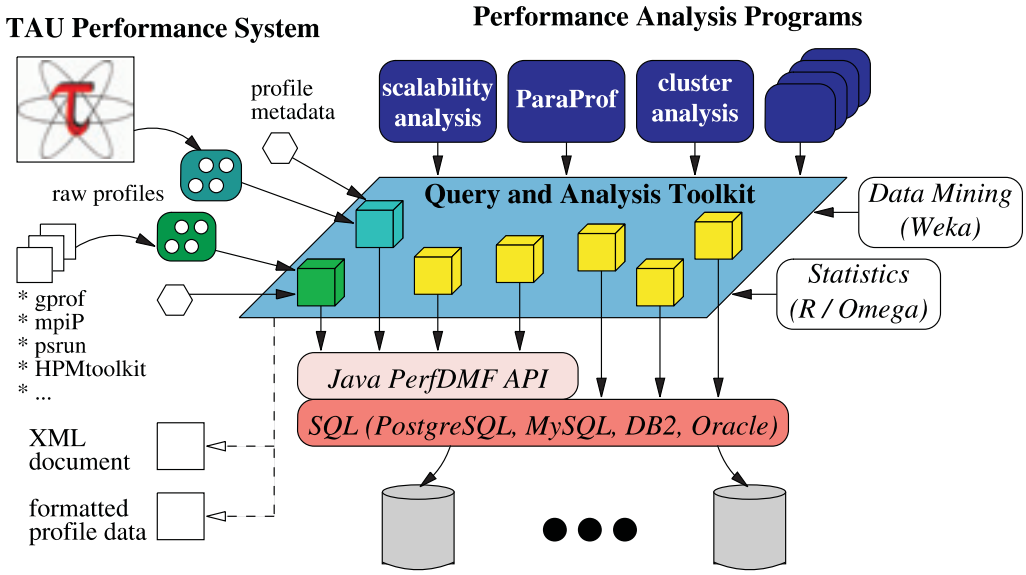

Fig. 10 TAU Performance Database Architecture.

both large-scale performance profiles, consisting of many events and threads of execution, as well as many profiles from multiple performance experiments. Our tests with large profile data (101 events on 16K processors) showed the framework adequately handled the mass of data.

To facilitate performance analysis development, the PerfDMF architecture includes a well-documented data management API to abstract query and analysis operations into a more programmatic, non-SQL, form. This layer is intended to complement the SQL interface, which is directly accessible by analysis tools, with dynamic data management and higher-level query functions. It is anticipated that many analysis programs will utilize this API for implementation. Access to the SQL interface is provided using the Java Database Connectivity (JDBC) API. Because all supported databases are accessed through a common interface, the tool programmer does not need to worry about vendor-specific SQL syntax. We have developed several tools that make use of the API, including ParaProf. Figure 11 shows PerfDMF being used by ParaProf to load the S3D profile dataset.

The last component, the profile analysis toolkit, is an extensible suite of common base analysis routines that can be reused across performance analysis programs. The intention also is to provide a common programming environment in which performance analysis developers can contribute toolkit modules and packages. Analysis routines are a useful abstraction for developing profile analysis applications.

6.3 Tracing

We made an early decision in the TAU system to leverage existing trace analysis and visualization tools. However, TAU implements it own trace measurement facility and produces trace files in TAU's own format. As a result, trace converters are supplied with the system to translate TAU traces to formats used by the tools. The primary tool we support in TAU is Vampir (Intel Corporation; currently marketed as the Intel(R) Trace Analyzer 4.0). TAU provides a `tau2vtf` program to convert TAU traces to VTF3 format. In addition, TAU offers `tau2epilog`, `tau2slog2`, and *tau*$_c$*onvert* programs to convert to EPILOG (Mohr and Wolf 2003), SLOG2 (Wu et al. 2000), and Paraver (European Center for Parallelism of Barcelona (CEPBA)) formats, respectively. For convenience, the TAU tracing systems also allows traces files to be output directly in VTF3 and EPILOG formats.

Figure 12 shows three Vampir displays of the S3D execution. The timeline view identifies a performance communications bottleneck as the result of the imbalanced execution of the `INT_RTE` routine. The communications

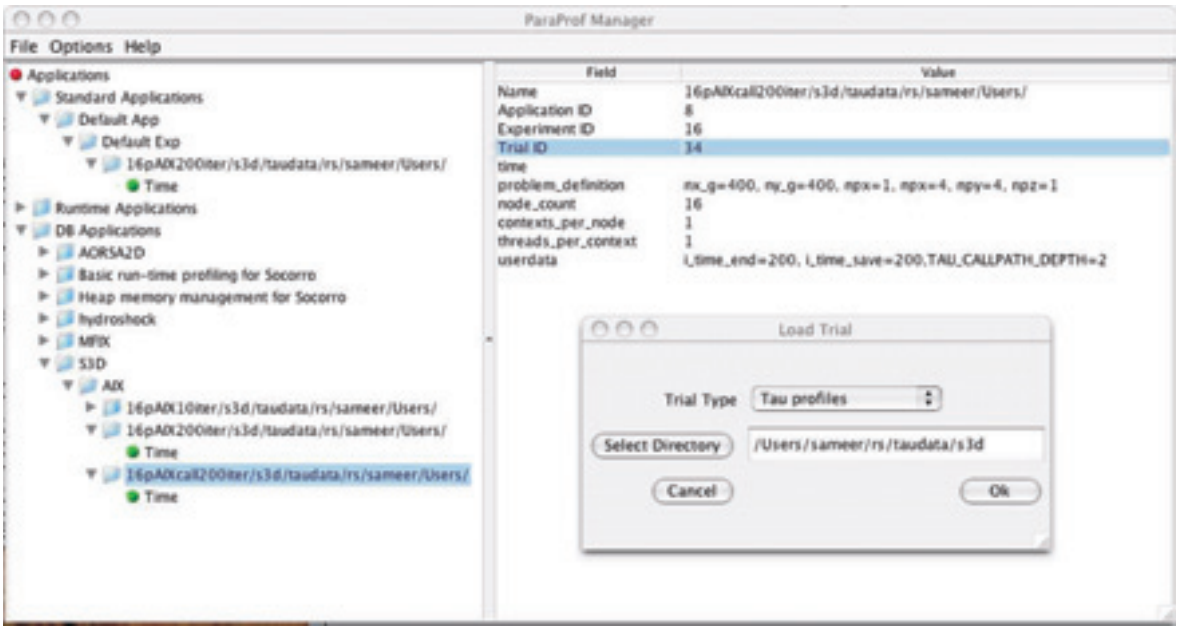

Fig. 11 ParaProf loading S3D parallel profile from PerfDMF database.

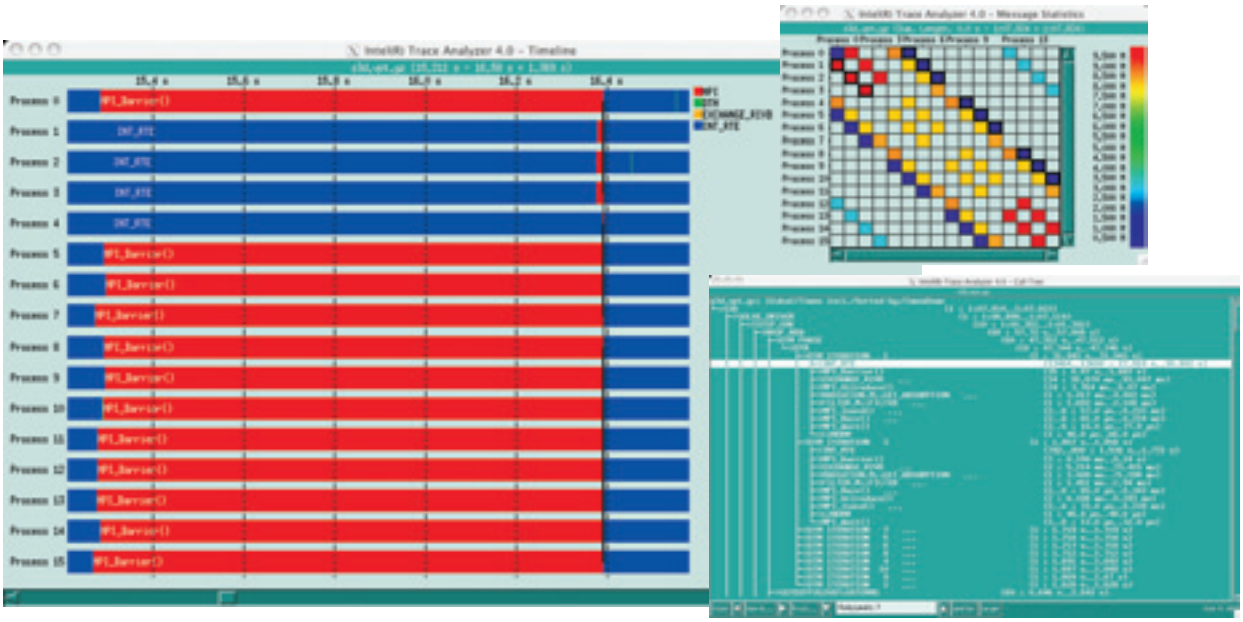

Fig. 12 Vampir displays of S3D performance trace: timeline (left), communication matrix (upper right), and call-graph (lower right).

matrix view highlights the pattern and load distribution of process communication in the S3D benchmark. The expanded callgraph display points out the interesting variations in the performance of the DTM iterations. TAU also can utilize the latest generation of the Vampir tool, VNG, and we have demonstrated the ability to analyze and visualize traces with VNG online during execution.

By applying the EPILOG converter, TAU is able to gain use of the Expert (Wolf et al. 2004) performance analysis tool. Expert is trace-based in its analysis and looks for performance problems that arise in the execution. The Expert tool provides a GUI showing problem classes, code locations where a particular problem occurs, and computing resources (e.g. threads) associated with the problem and location. In this manner, Expert consolidates the performance information in a trace into a more statistics oriented display. Expert uses the CUBE visualizer (Song et al. 2004) for presenting the results. Figure 13 shows a view from Expert using CUBE for S3D.

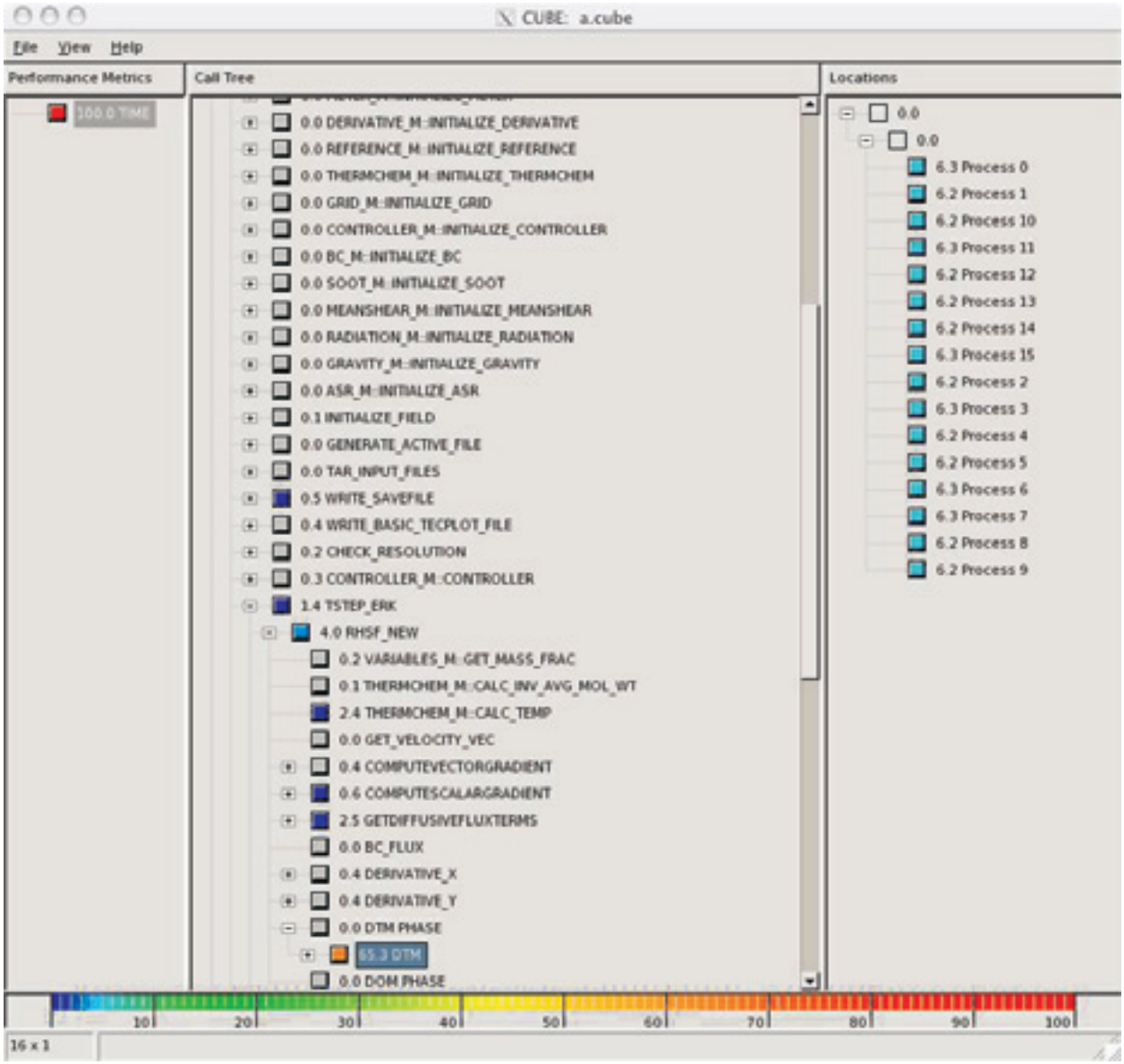

Fig. 13 Expert/CUBE display of TAU's S3D performance data.

TAU also provides a tool, `vtf2profile`, that will read a VTF3 trace and generated a parallel profile in TAU format. The tool can take parameters identifying where to start and stop the profile generation in time, allowing parallel profiles to be generated for specific regions of the traces. The profile output can then be read by ParaProf.

7 Conclusion

Complex parallel systems and software pose challenging performance evaluation problems that require robust methodologies and tools. However, in rapidly evolving parallel computing environments, performance technology can ill-afford to stand still. Performance technology development always operates under a set of constraints as well as under a set of expectations. While performance evaluation of a system is directly affected by what constraints the system imposes on performance instrumentation and measurement capabilities, the desire for performance problem solving tools that are common and portable, now and into the future, suggests that performance tools hardened and customized for a particular system platform will be short-lived, with limited utility. Similarly, performance tools designed for constrained parallel execution models will likely have little use in more general parallel and distributed computing paradigms. Unless performance technology evolves with system technology, a chasm will remain between the users expectations and the capabilities that performance tools provide.

The TAU performance system addresses performance technology problems at three levels: instrumentation, measurement, and analysis. The TAU framework supports the configuration and integration of these layers to target specific performance problem solving needs. However, effective exploration of performance will necessarily require prudent selection from the range of alternative methods TAU provides to assemble meaningful performance experiments that sheds light on the relevant performance properties. To this end, the TAU performance system offers support to the performance analysis in various ways, including powerful selective and multi-level instrumentation, profile and trace measurement modalities, interactive performance analysis analysis, and performance data management.

Portability, robustness, and extensibility are the hallmarks of the TAU parallel performance system. TAU is available on all HPC platforms and supports all major parallel programming methodologies. It is in use in scientific research groups, HPC centers, and industrial laboratories around the world. The entire TAU software is available in the public domain and is actively being maintained and updated by the Performance Research Lab at the University of Oregon.

Acknowledgments

Research at the University of Oregon is sponsored by contracts (DE-FG03-01ER25501 and DE-FG02-03ER25561) from the MICS program of the U.S. Dept. of Energy, Office of Science.

The TAU project has benefited from the contributions of many project staff and graduate students. We would like to recognize in particular those of Robert Bell, Alan Morris, Wyatt Spear, Chris Hoge, Nick Trebon, Kevin Huck, Suravee Suthikulpanit, Kai Li, and Li Li of University of Oregon, and Bernd Mohr of Research Centre Juelich, Germany for their work on the TAU system.

Author Biographies

Dr. Sameer S. Shende is a Postdoctoral Research Associate in the NeuroInformatics Center at the University of Oregon. He received the Bachelor of Technology (B. Tech) degree in Electrical Engineering from the Indian Institute of Technology, Bombay, India in 1991. He received the M.S. and Ph.D. degrees from the University of Oregon in 1996 and 2001 respectively. His research interests are in the area of performance analysis of parallel programs, instrumentation, measurement and techniques for mapping performance data. He helped develop the TAU performance system.

Dr. Allen D. Malony is a Professor in the Department of Computer and Information Science at the Unversity of Oregon. He received the B.S. and M.S. degrees in Computer Science from the University of California, Los Angeles in 1980 and 1982, respectively. He received the Ph.D. degree from the University of Illinois at Urbana-Champaign in October 1990. He joined the faculty at Oregon in 1991, spending his first year as a Fulbright Research Scholar and visiting Professor at Utrecht University in The Netherlands. He was awarded the NSF National Young Investigator award in 1994. In 1999 he was a Fulbright Research Scholar to Austria located at the University of Vienna. He was awarded the prestigious Alexander von Humboldt Research Award for Senior U.S. Scientists by the Alexander von Humboldt Foundation in 2002. His research interests are in parallel computing, performance analysis, supercomputing, and scientific software environments. He is the Director of the Performance Research Laboratory, the NeuroInformatics Center, and the Computational Science Institute at the University of Oregon.

Dr. Malony and Dr. Sameer S. Shende are the founders of ParaTools, Inc., a company devoted to tools for high-performance parallel computing.

References

Ahn, D., Kufrin, R., Raghuraman, A., and Seo, J. Perfsuite. http://perfsuite.ncsa.uiuc.edu/.

Bell, R., Malony, A. D., and Shende, S. 2003. A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis. *Proceedings of the Europar 2003 Conference (LNCS 2790)*, pp. 17–26.

Bernholdt, D. E., Allan, B. A., Armstrong, R. et al. 2006. A Component Architecture for High-Performance Scientific Computing. *Intl. Journal of High-Performance Computing Applications* ACTS Collection Special Issue.

Berrendorf, R., Ziegler, H., and Mohr, B. PCL — The Performance Counter Library. http://www.fz-juelich.de/zam/PCL/.

Browne, S., Dongarra, J., Garner, N., Ho, G., and Mucci, P. 2000. A Portable Programming Interface for Performance Evaluation on Modern Processors. *International Journal of High Performance Computing Applications* 14(3):189–204.

Brunst, H., Malony, A. D., Shende, S., and Bell, R. 2003. Online Remote Trace Analysis of Parallel Applications on High-Performance Clusters. *Proceedings of the ISHPC Conference (LNCS 2858)*, pp. 440–449. Springer.

Brunst, H., Nagel, W. E., and Malony, A. D. 2003. A Distributed Performance Analysis Architecture for Clusters. *Proceedings of the IEEE International Conference on Cluster Computing (Cluster 2003)*, pp. 73–83. IEEE Computer Society.

Buck, B. and Hollingsworth, J. 2000. An API for Runtime Code Patching. *Journal of High Performance Computing Applications* 14(4):317–329.

California Institute of Technology. VTF — Virtual Test Shock Facility. http://www.cacr.caltech.edu/ASAP.

CCA Forum. The Common Component Architecture Forum. http://www.cca-forum.org.

DeRose, L. 2001. The Hardware Performance Monitor Toolkit. *Proceedings of the European Conference on Parallel Computing (EuroPar 2001, LNCS 2150)*, pp. 122–131. Springer.

DeRose, L. and Reed, D. 1998. SvPablo: A Multi-Language Architecture-Independent Performance Analysis System. *Proceedings of the International Conference on Parallel Processing, ICPP '99*, pp. 311–318.

DeRose, L. and Wolf, F. 2002. CATCH – A Call-Graph Based Automatic Tool for Capture of Hardware Performance Metrics for MPI and OpenMP Applications. *Proceedings of the Europar 2002 Conference*.

Dongarra, J., Malony, A. D., Moore, S., Mucci, P., and Shende, S. 2003. Performance Instrumentation and Measurement for Terascale Systems. *Proceedings of the ICCS 2003 Conference (LNCS 2660)*, pp. 53–62.

European Center for Parallelism of Barcelona (CEPBA). Paraver — Parallel Program Visualization and Analysis Tool – reference manual. http://www.cepba.upc.es/paraver.

Forum, M. P. I. 1994. MPI: A Message Passing Interface Standard. *International Journal of Supercomputer Applications* (Special Issue on MPI) 8(3/4).

Graham, S., Kessler, P., and McKusick, M. 1982. gprof: A Call Graph Execution Profiler. *SIGPLAN '82 Symposium on Compiler Construction* pp. 120–126.

Gropp, W. and Lusk, E. User's Guide for MPE: Extensions for MPI Programs. http://www-unix.mcs.anl.gov/mpi/mpich/docs/mpeguide/paper.htm.

HPC++ Working Group. 1995. HPC++ White Papers. Technical Report TR 95633, Center for Research on Parallel Computation.

Huck, K., Malony, A., Bell, R., and Morris, A. 2005. Design and Implementation of a Parallel Performance Data Management Framework. Proc. *International Conference on Parallel Processing, ICPP-05*.

IBM. IBM DB2 Information Management Software. http://www.ibm.com/software/data.

Intel Corporation. Intel (R) Trace Analyzer 4.0. http://www.intel.com/software/products/cluster/tanalyzer/.

Kessler, P. 1990. Fast Breakpoints: Design and Implementation. *SIGPLAN Notices* 25(6):78–84.

Kohn, S., Kumfert, G., Painter, J., and Ribbens, C. 2001. Divorcing Language Dependencies from a Scientific Software Library. *Proceedings of the 10th SIAM Conference on Parallel Processing*.

Lindlan, K., Cuny, J., Malony, A. D., Shende, S., Mohr, B., Rivenburgh, R., and Rasmussen, C. 2000. A Tool Framework for Static and Dynamic Analysis of Object-Oriented Software with Templates. *Proceedings of the SC'2000 Conference*.

Malony, A. D. 1990. *Performance Observability*. PhD thesis, University of Illinois, Urbana-Champaign.

Malony, A. and Shende, S. 2000. Performance Technology for Complex Parallel and Distributed Systems. In: *Distributed and Parallel Systems: From Concepts to Applications* (eds. G. Kotsis and P. Kacsuk), pp. 37–46, Norwell, MA: Kluwer.

Malony, A., Shende, S., Bell, R., Li, K., Li, L., and Trebon, N. 2003. Advances in the TAU Performance System. In: *Performance Analysis and Grid Computing* (eds. V. Getov, M. Gerndt, A. Hoisie, A. Malony, B. Miller), pp. 129–144. Norwell, MA: Kluwer.

Mellor-Crummey, J., Fowler, R., and Marlin, G. 2002. HPCView: A Tool for Top-down Analysis of Node Performance. *The Journal of Supercomputing* 23:81–104.

Mohr, B. KOJAK — Kit for Objective Judgment and Knowledge-based Detection of Bottlenecks. http://www.fz-juelich.de/zam/kojak.

Mohr, B., Malony, A., Shende, S., and Wolf, F. 2002. Design and Prototype of a Performance Tool Interface for OpenMP. *The Journal of Supercomputing* 23:105–128.

Mohr, B. and Wolf, F. 2003. KOJAK – A Tool Set for Automatic Performance Analysis of Parallel Applications. *Proceedings of the European Conference on Parallel Computing (EuroPar 2003, LNCS 2790)*, pp. 1301–1304. Springer.

Mucci, P. Dynaprof. http://www.cs.utk.edu/~mucci/dynaprof.

MySQL. MySQL: The World's Most Popular Open Source Database. www.mysql.org.

Nagel, W., Arnold, A., Weber, M., Hoppe, H.-C., and Solchenbach, K. 1996. VAMPIR: Visualization and Analysis of MPI Resources. *Supercomputer* 12(1):69–80.

Norris, B., Ray, J., McInnes, L., Bernholdt, D., Elwasif, W., Malony, A., and Shende, S. 2004. Computational quality of service for scientific components. *Proceedings of the*

International Symposium on Component-based Software Engineering (CBSE7). Springer.

Oracle Corporation. Oracle. http://www.oracle.com.

PostgreSQL. PostgreSQL: The World's Most Advanced Open Source Database. http://www.postgresql.org.

Ray, J., Trebon, N., Shende, S., Armstrong, R., and Malony, A. 2004. Performance Measurement and Modeling of Component Applications in a High Performance Computing Environment: A Case Study. *Proc. International Parallel and Distributed Processing Symposium (IPDPS'04)*.

Sarukkai, S. and Malony, A. D. 1993. Perturbation Analysis of High Level Instrumentation for SPMD Programs. *SIGPLAN Notices* 28(7).

Seidl, S. 2003. VTF3 – A Fast Vampir Trace File Low-Level Management Library. Technical Report ZHR-R-0304, Dresden University of Technology, Center for High-Performance Computing.

Shende, S. 2001. *The Role of Instrumentation and Mapping in Performance Measurement*. PhD thesis, University of Oregon.

Shende, S. and Malony, A. D. 2003. Integration and Application of TAU in Parallel Java Environments. *Concurrency and Computation: Practice and Experience* 15(3–5):501–519.

Shende, S., Malony, A. D., Cuny, J., Lindlan, K., Beckman, P., and Karmesin, S. 1998. Portable Profiling and Tracing for Parallel Scientific Applications using C++. *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools, SPDT'98*, pp. 134–145.

Shende, S., Malony, A. D., Rasmussen, C., and Sottile, M. 2003. A Performance Interface for Component-Based Applications. *Proceedings of International Workshop on Performance Modeling, Evaluation and Optimization, International Parallel and Distributed Processing Symposium*.

Song, F., Wolf, F., Bhatia, N., Dongarra, J., and Moore, S. 2004. An Algebra for Cross-Experiment Performance Analysis. *Proc. of International Conference on Parallel Processing, ICPP-04*.

Subramanya, R. and Reddy, R. 2000. Sandia DNS code for 3D compressible flows – Final Report. Technical Report PSC-Sandia-FR-3.0, Pittsburgh Supercomputing Center, PA.

SUN Microsystems Inc. Java Virtual Machine Profiler Interface (JVMPI). http://java.sun.com/j2se/1.5.0/docs/guide/jvmpi/.

Szyperski, C. 1997. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley.

University of Oregon, a. TAU Portable Profiling. http://www.cs.uoregon.edu/research/paracomp/tau.

University of Oregon, b. Tuning and Analysis Utilities User's Guide. http://www.cs.uoregon.edu/research/paracomp/tau.

Vetter, J. and Chambreau, C. mpiP: Lightweight, Scalable MPI Profiling. http://www.llnl.gov/CASC/mpip/.

Viswanathan, D. and Liang, S. 2000. Java Virtual Machine Profiler Interface. *IBM Systems Journal* 39(1):82–95.

Wolf, F., Mohr, B., Dongarra, J., and Moore, S. 2004. Efficient Pattern Search in Large Traces through Successive Refinement. *Proceedings of the European Conference on Parallel Computing (EuroPar 2004, LNCS 3149)*, pp. 47–54. Springer.

Wu, C. E., Bolmarcich, A., Snir, M., Wootton, D., Parpia, F., Chan, A., Lusk, E., and Gropp, W. 2000. From trace generation to visualization: A performance framework for distributed parallel systems. *Proc. of SC2000: High Performance Networking and Computing*.