Scientific Programming 16 (2008) 105–121
DOI 10.3233/SPR-2008-0256
IOS Press

Open|SpeedShop: An open source infrastructure for parallel performance analysis

Martin Schulz [a,*], Jim Galarowicz [b], Don Maghrak [b], William Hachfeld [b], David Montoya [c] and Scott Cranford [d]

[a] *Lawrence Livermore National Laboratory, Livermore, CA, USA*
[b] *Krell Insititute, Ames, IA, USA*
[c] *Los Alamos National Laboratory, Los Alamos, NM, USA*
[d] *Sandia National Laboratories, Livermore, CA, USA*

Abstract. Over the last decades a large number of performance tools has been developed to analyze and optimize high performance applications. Their acceptance by end users, however, has been slow: each tool alone is often limited in scope and comes with widely varying interfaces and workflow constraints, requiring different changes in the often complex build and execution infrastructure of the target application.

We started the Open|SpeedShop project about 3 years ago to overcome these limitations and provide efficient, easy to apply, and integrated performance analysis for parallel systems. Open|SpeedShop has two different faces: it provides an interoperable tool set covering the most common analysis steps as well as a comprehensive plugin infrastructure for building new tools. In both cases, the tools can be deployed to large scale parallel applications using DPCL/Dyninst for distributed binary instrumentation. Further, all tools developed within or on top of Open|SpeedShop are accessible through multiple fully equivalent interfaces including an easy-to-use GUI as well as an interactive command line interface reducing the usage threshold for those tools.

Keywords: Performance tools, scalable infrastructure, tool design, plugin architecture

1. Motivation

Performance tools are an essential component of the development cycle for high performance computing applications and this has lead to a wide range of tools produced by both academia and industry. However, each tool typically comes with its own user interface and – more importantly – workflow, in many cases requiring significant changes to the often complex build and execution environment of the target application. This situation is aggravated for parallel tools due to the additional infrastructure requirements for controlling and instrumenting multiple processes in parallel environments as well as collecting performance data from multiple nodes.

In addition, only few tools have been designed with advanced user interfaces like GUIs or scripting languages. As a consequence, the learning curves for the individual tools are high, in particular compared to the limited time code teams have to spend on performance optimization (in our experience often only 1 or 2 person weeks of effort per year across the entire team) to learn even a sufficient subset of them. As a result, code teams often restrict themselves to very few, simple tools (like *gprof*) or rely on simple self instrumentation techniques ignoring the wide body of advanced performance tools research.

To overcome these limitations and to make more sophisticated performance analysis techniques available to end users while fitting their time budget, we need an infrastructure that allows performance tools to unify their execution environments and workflow constraints and that provides key components for scalable performance analysis that can be reused across tools to avoid "reinventing the wheel" over and over again. These goals drive the Open|SpeedShop project, a large scale, multi-year effort across Lawrence Liv-

*Corresponding author: Martin Schulz, Lawrence Livermore National Laboratory, P.O. Box 808, L-560, Livermore, CA 94551, USA. Tel.: +1 925 423 6498; E-mail: schulzm@llnl.gov.

ermore, Los Alamos, and Sandia National Laboratories together with the Krell Institute as well as with initial contributions by SGI to establish a scalable and highly portable open source performance tool environment for large capacity clusters. We build on many of the high-level concepts of SGI's SpeedShop tool set for IRIX, although Open|SpeedShop is built entirely from scratch and using different underlying technology.

Open|SpeedShop has two separate faces:

(a) It implements an interoperable, scalable tool set covering the most common analysis steps for parallel applications and makes these tools available through multiple fully equivalent interfaces including an easy-to-use GUI as well as an interactive command line interface.
(b) It provides its own components as an open source plugin infrastructure for building new tools. This allows tool developers to inherit Open|SpeedShop's core capabilities, in particular its scalable process control and communication infrastructure, performance data storage, as well as its user interfaces, and hence focus on the actual tool functionality.

Together, these two sides of Open|SpeedShop realize scalable performance analysis on large scale clusters while at the same time making its base components available to other tool development efforts allowing for a common look and feel for the end user.

Open|SpeedShop makes the following major contributions:

- *Integrated tool environment*. Open|SpeedShop integrates many common performance tools, including CPU time and hardware counter sampling, as well as MPI tracing, on top of a single infrastructure and makes them available through a common set of user interfaces. While the tool functionalities themselves may not be new, this integration unifies their look and feel as well as their workflow eliminating the need to learn and deploy each tool from scratch.
- *New tool functionality*. In addition to integrating existing performance analysis techniques, Open|SpeedShop provides new functionality presently not available on large scale Linux platforms, such as I/O or Floating Point Exception tracing.
- *Support for large scale parallel platforms*. Open|SpeedShop is explicitly designed to work on large scale platforms. This includes both applying tool functionality across all ranks of a parallel application as well as deploying analysis techniques specifically targeting parallel environments. As part of its capabilities it automatically extends any sequential technique to parallel environments and thereby widens the scope of any tool developed within Open|SpeedShop.
- *Multiple, interoperable user interfaces*. Open|SpeedShop not only provides an integrated tool environment, but also implicitly provides multiple user interfaces to its functionality. This includes a graphical user interface, a scripting and batch language, as well as a Python module. All interfaces are equivalent and interoperable, leaving the choice up to the user depending on his/her preferences and intended integration into the target application's workflow.
- *Novel tool development infrastructure*. Tool developers can use Open|SpeedShop's plugin infrastructure as well as its individual components to develop new tool capabilities and thereby not only maintain the same look and feel as any other Open|SpeedShop integrated tool, but also reuse most of Open|SpeedShop's environment. The latter lets developers focus on the actual functionality and avoid the unrewarding task of building and maintaining a comprehensive infrastructure over and over again. Further, this architecture provides an easy path to a wide deployment of new tools by simply activating the respective plugins in the Open|SpeedShop framework.

To minimize the impact on existing application workflows, Open|SpeedShop is built on top of the concept of dynamic binary instrumentation. Any instrumentation necessary to acquire the performance data is directly inserted into loaded target application binary. As a consequence, we do not require any source code modification or the recompilation of the target code. This also allows us to apply any performance analysis techniques to already running applications by attaching to them at any time during their execution and performing the instrumentation at that time. For parallel applications we apply the binary instrumentation concurrently to all tasks and then use a global communication mechanism to gather the data returned by the individual, local instrumentation points.

In summary, Open|SpeedShop is a single tool infrastructure that provides the basis for both existing and future tool capabilities, makes them available to the end user using a common set of interfaces, and minimizes the impact on the user's workflow and application setup. However, we do not expect that our Open|SpeedShop developments will be able to cover

all performance analysis techniques alone; we rather expect that the availability of the Open|SpeedShop framework will lead to a community effort in performance tools with the end goal of a rich set of easy-to-use and interoperable plugins across a wide range of platforms. In order to reach this goal and encourage both wide spread use and external developers, Open|SpeedShop is available under LGPL (with minor pieces in GPL) in both source and binary form at http://www.openspeedshop.org/.

The remainder of this paper is organized as follows: in Section 2 we give a brief overview of related work and in Section 3 we describe the basic architecture of Open|SpeedShop. In Section 4 we discuss the capabilities of its plugin infrastructure and in Section 5 we describe the current functionality available in Open|SpeedShop from a user's point of view. In Section 6 we analyze overhead caused by Open|SpeedShop and we conclude in Section 7 with some final remarks.

2. Related work

Typically, operating systems already provide a set of basic performance tools. This often includes *prof* or *gprof*, a basic profiler, and utilities like *time* to measure execution time or *strace* to track all system calls. While these tools only offer very basic functionality, they are generally very popular due to their low learning curve and their pervasive and cross-platform availability.

Besides these basic tools, a large body of research exists covering a wide range of performance tools. Examples are HPCView from the HPCToolkit [11], which provides a graphical user interface to gather a detailed performance profile of an application on a per function basis; the PAPI [13] utilities or the HPM Tool Kit [9] to gather hardware counter statistics across an application's execution; the Visual Profiler (vProf) [10]; and DynTG [18], which allows users to dynamically insert performance calipers into a running application.

Several tools have also been developed specifically for the analysis of parallel applications. This includes both profiling tools, like mpiP [19], and tracing and trace visualization tools, like Vampir [14], Vampir Next Generation (VNG) [4], or Jumpshot [20]. Further, systems like TAU/ParaProf [3], which provides a whole suite of performance tools and visualizers, and Paradyn [12], which focuses on the use of dynamic instrumentation, can be used both on sequential and parallel codes.

A special class of performance tools use simulator technology to track low-level performance events, e.g., individual memory references as in CacheGrind [15] or Sigma [6]. These tools provide a highly detailed view of an application's execution, but generally infer very high overheads of two–three orders of magnitude.

In addition to these tools, several groups have been working on parallel tool infrastructures that support the development of scalable tools. Most notable in this area are Dyninst [5], a library for dynamic binary instrumentation; the Dynamic Probe Class Library (DPCL) [7], a C++ class hierarchy to enable binary instrumentation and data collection on parallel jobs; and MRNet, the Multicast-Reduction Network [17], which allows efficient data management for large scale tool environments. These approaches, while not full tools themselves, can be used as parts during the development of a larger framework; in fact, Open|SpeedShop is built on top of Dyninst and DPCL.

In summary, existing tools cover a wide spectrum of performance analysis tasks; however, no single tool covers all of them. Further, each tool comes with its own user interface and workflow constraints, instrumentation techniques, setup requirements, user interfaces, and data storage. It is our experience from working with a large number of code teams at our facilities that this often prevents the use of such analysis tools for larger efforts and thereby hinders a wide spread use of advanced performance analysis techniques.

3. Open|SpeedShop design

In the Open|SpeedShop project we address the challenges discussed above with an integrated approach focused on performance analysis. The tool set provides all relevant functionalities to the end user packaged in a single look and feel without limiting the tool developer's freedom or ability to provide new, previously unplanned functionality. We achieve this through a flexible open source infrastructure that establishes key components required by any tool, offers a flexible plugin infrastructure, is based on a unifying workflow, and provides consistent user interfaces across all participating tools.

Currently, Open|SpeedShop targets Linux platforms with Intel and AMD processors, but it is our intention that Open|SpeedShop will function as the default toolset on all production machines at the participating facilities. This will make advanced performance analysis techniques available to our large code teams as well as to facilitate easier collaborations with external tool developers.

3.1. Architectural overview

The Open|SpeedShop framework is written in C/C++ and we show its modular, class-based architecture in Fig. 1. The central component is the *Data Abstraction Layer* which is responsible for storing and analyzing any data collected during performance analysis experiments. All data is stored using a relational database. In particular, we use SQLite due to its ability to store the data into a single flat file without requiring the installation of a database server. Further, the license model of SQLite matches the free licensing scheme intended for Open|SpeedShop. However, other SQL databases can easily be substituted for SQLite by replacing the corresponding submodule in the *Data Abstraction Layer*.

The data managed by the *Data Abstraction Layer* can be accessed through a *Command Line Interface* (CLI) based on a simple command syntax similar to *gdb* or *totalview*. The interpreter for the CLI is based on an extended Python interpreter, which allows users to integrate Python control flow commands into CLI scripts. Users can use the CLI to access Open|SpeedShop either interactively or in batch mode.

In addition to using the CLI directly, users can access Open|SpeedShop using a Python module or a GUI. Both alternative interfaces use the CLI themselves, i.e., they translate each user input into the corresponding CLI command and execute it through the *CLI Layer*. This has the distinct advantage that all user interfaces are equivalent as well as interoperable. Due to this, users can mix interactive shell and GUI operations in a single session as well as log all commands used in Open|SpeedShop independent of their source and replay them through the batch execution interface.

The *Data Abstraction Layer* itself is built on top of a generic *Base Tool Layer* that abstracts the instrumentation and runtime infrastructure and can also be used to build alternative tools besides Open|SpeedShop. It

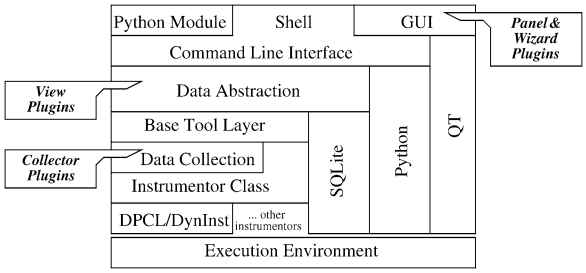

Fig. 1. Open|SpeedShop component architecture.

contains all information to execute performance experiments and to deliver the data back to the tool. It relies on two components, the *Instrumentor Class* and *Data Collection* facilities. The former encapsulates all mechanisms for process control, instrumentation, and communication. Its process control facilities are used by the *Base Tool Layer* to control the execution of the target application, while the *Data Collection* layer uses the instrumentation abilities of the *Instrumentor* to insert data collection probes into the target application and serves as a callback layer to retrieve any collected data.

3.2. Parallel instrumentation and data collection

In its current version, Open|SpeedShop implements the *Instrumentor* class using the Dynamic Probe Class Library (DPCL) [7], which provides global binary instrumentation mechanisms based on the Dyninst API [5]. Our version of DPCL is based on the original source code implemented by IBM, but we have added significant new features including support for statement instrumentation, thread control, and direct daemon start without requiring root installations.

Figure 2 shows the basic runtime architecture. A daemon on each node is responsible for all processes and threads on that node and uses Dyninst to control them through the *ptrace* interface. Using the same mechanism, daemons can dynamically insert instrumentation into each process or thread and implement a callback interface for the instrumentation to return data acquired from the target process.

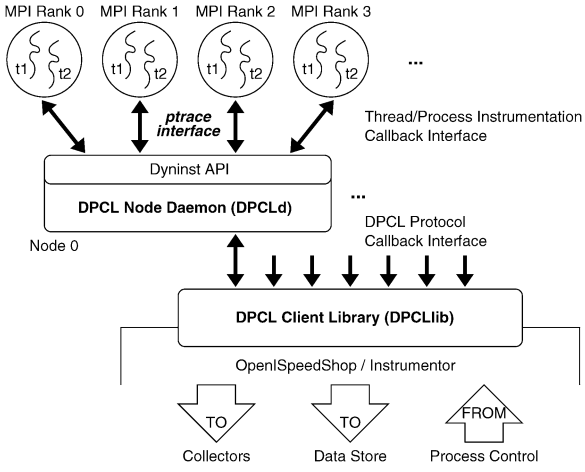

Fig. 2. DPCL communication and instrumentation inside the default Open|SpeedShop *Instrumentor*.

On the tool side, a *DPCLlib* client library is linked with the Open|SpeedShop *Instrumentor* and facilitates the communication with all node daemons associated with the current job. The *Instrumentor* implementation can then use this library to control the entire distributed job as well as to collect data again using a callback interface.

The current implementation using DPCL is just one option to provide instrumentation for and data collection from a parallel application. By replacing the *Instrumentor* Open|SpeedShop can be retargeted to other communication infrastructures or system environments. In our current work we are using this capability to introduce a more scalable tree-base communication scheme using MRNet [17].

3.3. Platforms

Open|SpeedShop currently targets x86 (IA-32), IA-64, and x86_64 based machines and large scale clusters running Linux. It has been successfully installed and tested on all three types of platforms using a wide range of Linux distributions including SuSE, RedHat, Fedora Core, and Ubuntu. While Open|SpeedShop's target scenarios will typically be sequential applications or parallel codes using modest numbers of tasks (up to 512 CPUs), Open|SpeedShop is designed to scale to large task counts and we have successfully used it on up to 3840 tasks on 960 nodes of an IA-64 cluster with 4-way SMP nodes, the largest configuration available to us at that time.

All components of Open|SpeedShop have been designed with portability in mind. We build on top of existing open source software that is already widely ported and all new components have little system dependencies. Only some of the low-level instrumentation components, which are naturally highly system and hardware specific, will have to be ported to use Open|SpeedShop on new platforms. Therefore we expect that Open|SpeedShop will in future also be available on more than Linux based clusters.

4. Plugin framework

One of the key capabilities of Open|SpeedShop is its extensibility, which is achieved using a plugin architecture. Tool developers can use it to design and deploy new tools using the Open|SpeedShop infrastructure. In fact, all of the functionality currently included in Open|SpeedShop (as described in more detail in the next section) is itself implemented using plugins showing the feasibility and flexibility of this approach.

The infrastructure is designed to enable programmers to take advantage of all Open|SpeedShop core capabilities when creating new tools. To achieve this, programmers split up their tool's functionality into three base tasks: data acquisition, data preparation, and data presentation and visualization. Each of these tasks is implemented as a separate plugin and Open|Speed Shop acts as a "glue" between these plugins to facilitate data storage and access as well as providing a uniform representation towards the user.

Open|SpeedShop distinguishes between four different types of plugins, each with its own API (see also Fig. 1).

- *Collector plugins*. Collectors interface with the instrumentation mechanisms provided by Open|SpeedShop and acquire performance data. They are generally split into two parts: a runtime component that is loaded into the application, and the actual collector that receives the data and forwards it into the Open|SpeedShop framework.
- *View plugins*. Views are used to retrieve data collected during an experiment and assemble it into a readable form. This can optionally include analysis and aggregation steps.
- *Panel plugins*. Panels extend the GUI provided by Open|SpeedShop and enable tool developers to offer new performance visualization displays.
- *Wizards*. This forth type of plugin is optional and can be used by tool developers to package the setup steps for a new tool in a user-friendly way.

Figure 3 shows both the conceptual data flow between the different plugins (white arrows) and the actual data flow between Open|SpeedShop components

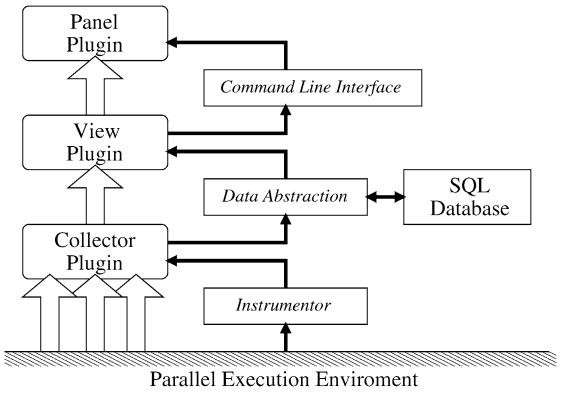

Fig. 3. Dataflow across plugins.

(black arrows). Data is collected through instrumentation inserted by a *Collector Plugin* through the *Instrumentor* class. Once the data is collected, it is forwarded through the *Data Abstraction Layer* for storage in the SQL database. From there, a *View Plugin* can query the information and assemble the actual performance results that are then presented to the user. A *Panel Plugin* uses the information from a *View* and displays it graphically inside the GUI.

In the case of a parallel application, Open|SpeedShop replicates the runtime components of the *Collectors* and uses them to instrument all processes and threads of the target application. The *Data Abstraction Layer* combines the data from all active application components and stores it into a common, global database. During this aggregation process the infrastructure preserves the data sources enabling the tool to create per process or per thread views, present node or thread comparisons, or perform cross node/thread analysis.

Once all plugins are complete, tool developers can deploy them by simply making them available in the Open|SpeedShop plugin directory. To separate production and prototype versions, both a global, system-wide and a user-local directory are available. Open|SpeedShop scans these directories at startup, loads all plugins found at that time, and makes them available to the user. Additionally, tool developers can create a wizard plugin that aids the users in applying the newly created tool.

This gives tool designers an easy way to deploy newly developed tools: they can be made available instantly inside the Open|SpeedShop framework without the end user having to install any new software or to learn new user interfaces. As soon as the plugins are installed, users will see any a new tool as a part of the Open|SpeedShop toolbox and can start using them.

5. Using Open|SpeedShop

Using the plugin infrastructure discussed above, the default version of Open|SpeedShop already provides a rich set of performance experiments to the end user. In this section we will show some of these capabilities starting with the typical workflow and terminology, all default *Experiments*, and Open|SpeedShop's advanced analysis techniques, which are automatically available to any plugin.

For all of the examples in the following section we use SMG2000 from the ASCI Purple benchmark suite, a Semicoarsening Multigrid Solver based on the hypre library [8], which we run with a small working set size of $N = 70^3$ for demonstration purposes. All experiments were conducted on *Atlas*, a large scale Linux cluster based on 1152 four way dual-core Opteron nodes running at 2.4 GHz and interconnected with an Infiniband network.

5.1. Workflow

Users both control the application and analyze the resulting data from the tool's interfaces. Figure 4 shows the typical workflow: Users first select their target application as well as an *Experiment*. The latter de-

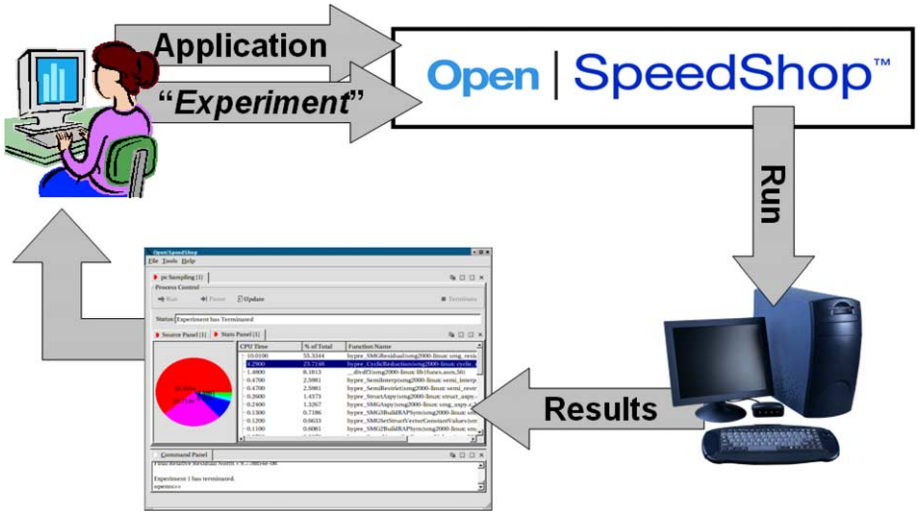

Fig. 4. Typical workflow in Open|SpeedShop.

fines which data will be collected during the execution of the target code and how. *Experiments* are abstract concepts within Open|SpeedShop that are connected with a specific *Instrumentor* class and can consist of one or more *Collectors*. The latter are implemented through the *Collector Plugins* described above and are responsible to collect a set of specific metrics (e.g., PC sampling data or function invocation traces) from the running program. Users can either choose from a set of predefined *Experiments* (typically containing exactly one *Collector*) or create a *Custom Experiment* by assembling the collectors manually.

Once the *Experiment* has been selected, Open|SpeedShop loads the application and instruments it with the respective collector plugins. Alternatively, users can specify a target process ID. In this case, Open|SpeedShop will attach to the application, stop the main process, and gather the symbols from the loaded binary. During both the load and the attach process, Open|SpeedShop parses the application's binary and stores all relevant symbol and library information into its SQL database. This has the advantage that the database, after the completion of the experiment, contains all relevant information – it is no longer necessary to keep the binary or the current version of the shared libraries (as it is, e.g., the case with *gprof*).

In the case that the target application is a parallel code using MPI, Open|SpeedShop by default detects all processes that are part of the targeted job, attaches to the whole job, and instruments all processes using DPCL as described above. To enable this, we rely on the widely adopted MPIR debug interface [2] to identify and locate all task processes for a target application. This mechanism applies to both creating and attaching to MPI applications and Open|SpeedShop contains the necessary interfaces to support both modes.

Open|SpeedShop supports any binary running on the target platforms described in Section 3, however, it is recommended to enable debug symbols during the compilation process (using the `-g` compiler flag). We successfully tested Open|SpeedShop with applications written in Fortran (77 and 90), as well as C and C++ using GNU, Intel, and Portland Group compilers. We expect other compiler and language combinations to work similarly.

Once the application load is completed, the user gets control over the application's execution to start the job using the toolbar in the *Process Management Panel* as shown in Fig. 5. The same panel also gives the ability to pause and resume application execution. For par-

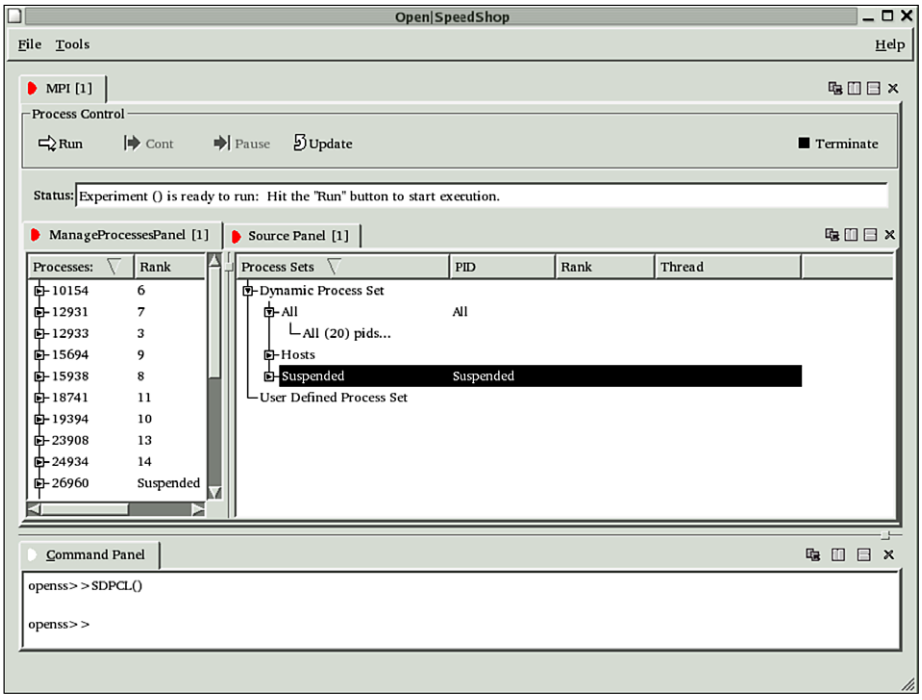

Fig. 5. Process Management Panel after the application SMG2000 has been loaded and is ready to run (using 16 tasks, the remaining four tasks shown represent threads in the job scheduler containing the MPI task information). The top shows the buttons that allow users to control to application's execution and the bottom part shows the execution environment (hosts, PIDs, etc.).

allel applications, it displays items for all threads and processes grouped by nodes and allows the user to define arbitrary subsets of threads and/or processes to refine analysis steps.

In addition to the job control functionality, the process management panel also provides information about the details of the experiment setup and allows the user to modify them. By default all processes or threads are associated with exactly one collector and this association can be examined by expanding the process items in the left container inside the panel. Starting from this initial configuration, users can then add or delete individual collectors to processes or process groups and thereby customize the experiment. This can, e.g., be used to run multiple collectors inside a single experiment or sample data across a subset of processes.

Once the application terminates or the user polls for intermediate data using the update button, Open|SpeedShop opens a *Stats Panel* and displays the gathered data either in tabular or graph form. At this point the user can also apply advanced techniques like cross experiment comparisons, multi rank analysis, or clustering techniques. In any case, the user can use this to gather information about his/her application and then apply this information to either initiate a next set of experiments or to modify the code.

5.2. Experiments

As described above, *Experiments* are the central concept in Open|SpeedShop that let users select what kind of performance information they want to collect about their application. By default Open|SpeedShop offers six predefined experiments, each associated with a different collector. These experiments cover many of the typical performance analysis techniques and include both sampling and tracing tools.

At startup Open|SpeedShop presents the user with an introductory wizard (see Fig. 6) that offers all experiments. Once the user selects one experiment, Open|SpeedShop leads him/her through all steps required to start the experiment, including setting all experiment parameters and selecting the target application. This functionality is encapsulated in the wizard plugins for the respective experiments as described in Section 4.

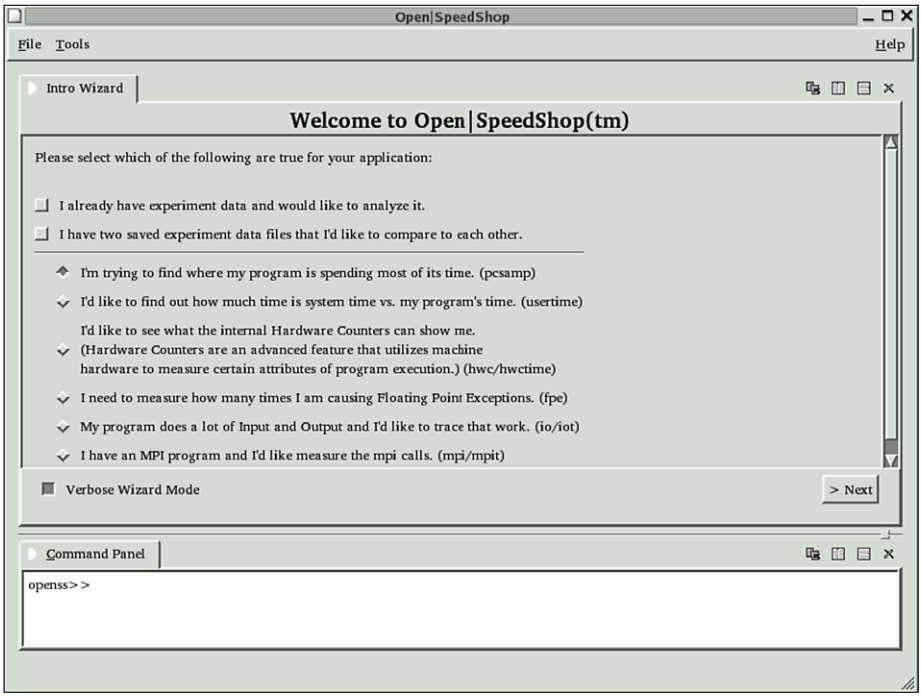

Fig. 6. Selecting an experiment using the IntroWizard. Users can either select to load existing data files (top two options) or to start a new experiment using one of the existing *Experiments* (bottom six options). The main screen by default also offers an interactive command line window at the bottom of the main window.

5.2.1. Sampling

Open|SpeedShop offers three different sampling experiments: a standard PC Sampling experiment that interrupts the application periodically and records the value of the current PC; a Usertime experiment that works similar to PC Sampling, but includes callstack information for each sample allowing the reconstruction of inclusive vs. exclusive time information; and a Hardware Counter experiment based on the PAPI interface that samples counter overflows of the CPU's performance counter registers.

In all three cases, the runtime plugins install a callback inside the target application that is activated either after a preset time interval is reached or the selected hardware counter overflows. The callback routine then uses the PC to determine the currently executed piece of code and attributes this location with the gathered data sample. All three collectors store this information in an application side buffer, which is periodically flushed to the *Instrumentor* callback inside the main tool using the DPCL communication mechanism. From there it is stored into the database for further analysis.

Once the experiment is complete, the *Stats Panel* displays the information gathered during the experiment in reference to the applications source. The user can select to show these results using different *Views*. Figures 7 and 8 show two examples: the former shows the result of a PC Sampling experiment aggregated by source functions, while the latter breaks down the information of a hardware counter experiment by source statement. In all views users can double click on the source information to open the corresponding source file as shown in Fig. 9. In the source panel, Open|SpeedShop optionally also displays the per statement results alongside with the source information.

5.2.2. Tracing

In addition to sampling, Open|SpeedShop also offers several tracing experiments including Posix I/O calls, Floating Point Exception (FPE), and MPI call tracing. In these experiments, Open|SpeedShop records the information for all individual events covered by the respective tracing collector. For the I/O and MPI collectors this also optionally includes all function parameters, such as file descriptors, buffer sizes, source and target nodes, or communicators.

To implement both MPI and I/O tracing we use Dyninst's ability to dynamically wrap function invocations. We wrap all routines of interest (for MPI a user defined subset of all communication routines defined in the MPI Standard, for I/O a user defined subset of all routines that are part of the POSIX I/O interface) with our own routines, which create the corresponding event descriptors for each invocation. These descrip-

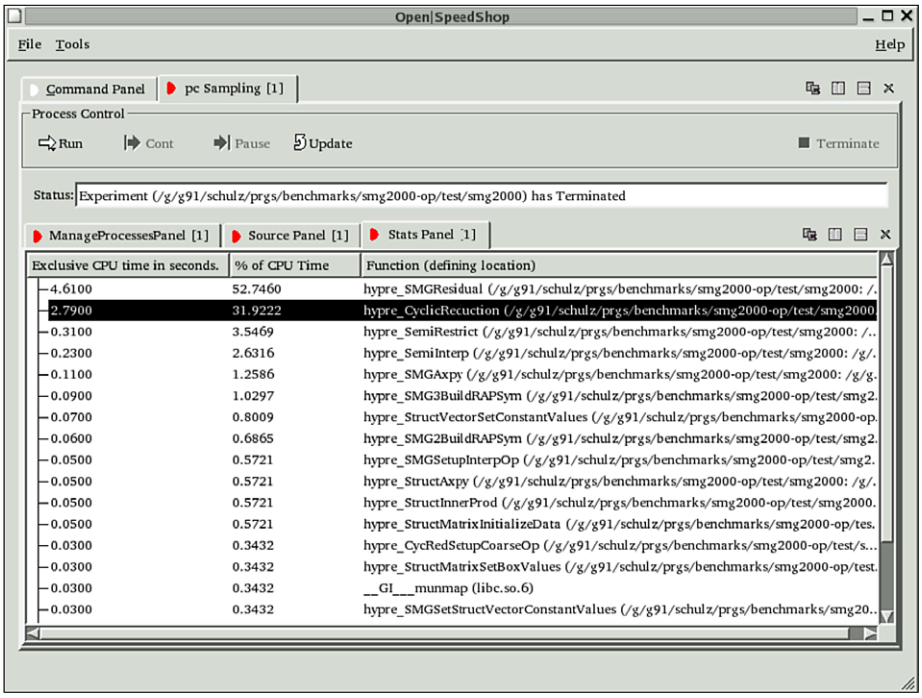

Fig. 7. Function view showing PC Sampling information grouped by functions (one function per line).

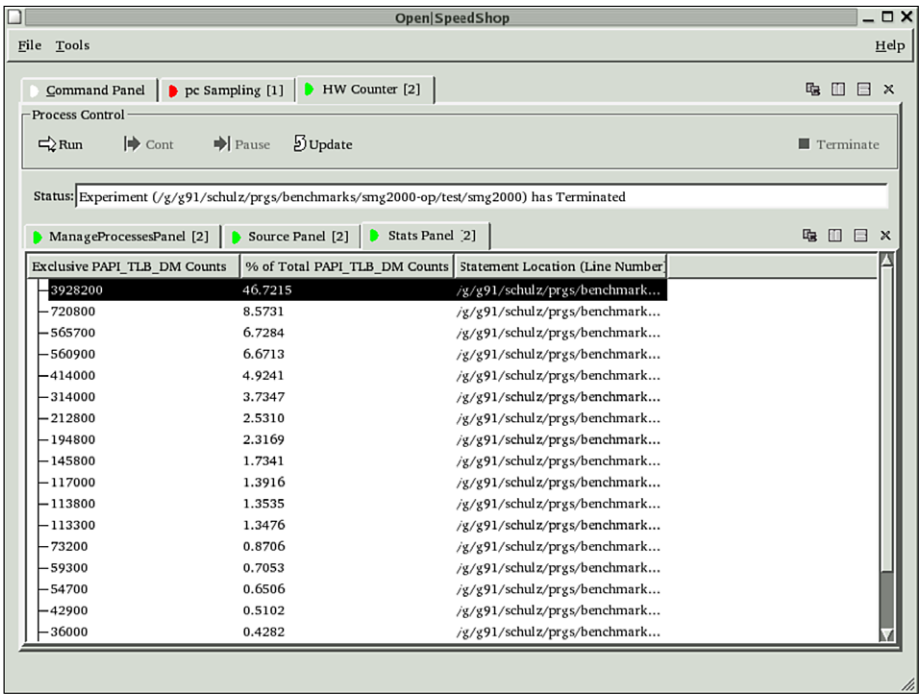

Fig. 8. Statement view showing hardware counter information (using the PAPI_TLB_DM counter) grouped by statement (one statement per line). Also note that in this example multiple experiments are loaded at the same time, as indicated by the top tabs.

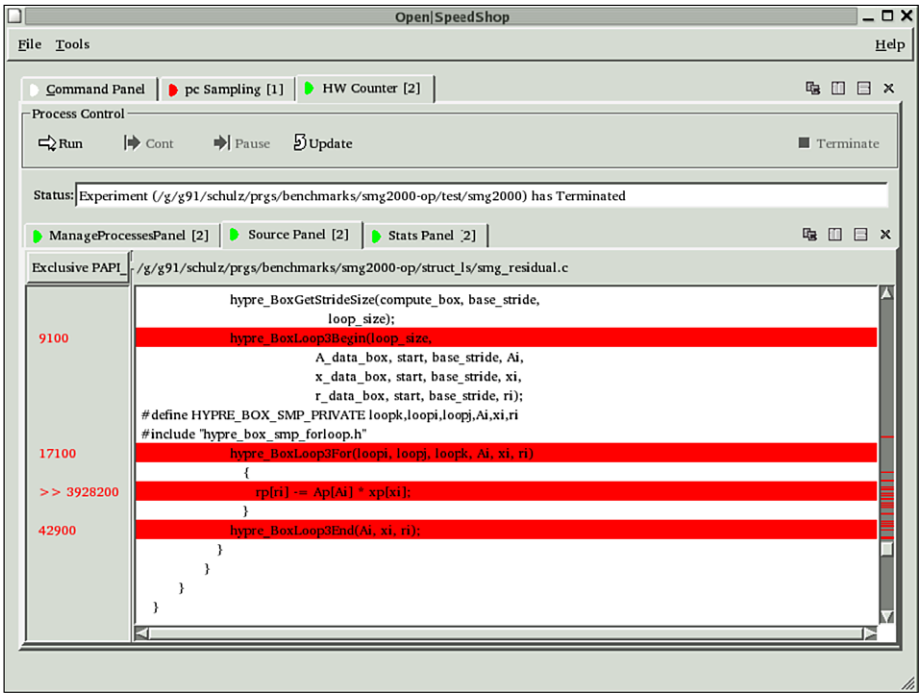

Fig. 9. Switching to the source panel for any selected statistics. The source lines for which samples have been found are highlighted and associated with the per statement performance results (left column). The user selected line is marked with >>.

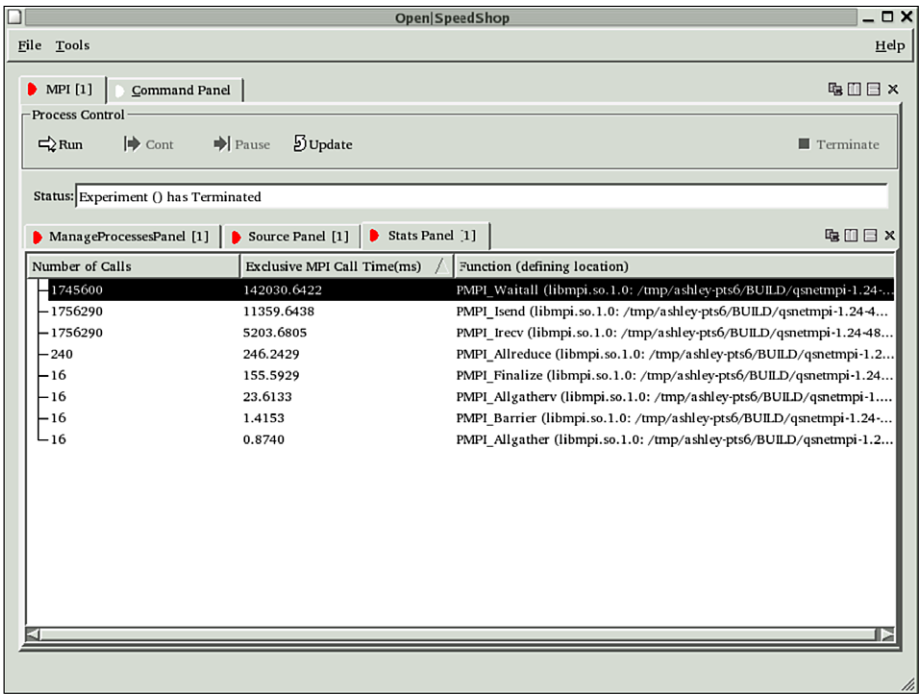

Fig. 10. Stats Panel of an MPI tracing experiment for SMG2000 with 16 tasks showing all called MPI routines together with the number of how often they have been called as well as their aggregated execution time.

tors are again temporarily stored in an application side buffer and then flushed to the tool for storage.

Figure 10 shows the resulting *Stats Panel* of an MPI tracing experiment on SMG2000 running on 16 processors (results from other tracing collectors look similar). It displays the number of times each MPI routine has been called throughout the application's runtime as well as how much time was spent during each MPI routine. Optionally, the user can activate additional columns with statistics to analyze the differences between tasks, e.g., using min and max counts or times, or standard deviations.

In addition, all trace collectors also store the complete stack trace for each event. Users can visualize these stacks either as Callstacks (oriented from main to the MPI call) or as Tracebacks (oriented from the MPI call to main) using alternate *Views*. We present an example for this in Fig. 11. This allows users to determine the dynamic callpath that lead to the event. Especially for I/O and communication calls, this is essential, since these calls are often wrapped by the user in the form of additional libraries. Using the callstack features, users can associate the events with the originating callsite inside their main code without having this information hidden by wrapping layers. In this particular example we can also use the callstack view to identify the recursive nature of the algorithm used in the multigrid solver of SMG. The `MPI_Waitall` routines are called from within `hypre_SMGSetup`, which in the upper case leads back to an earlier invocation of the same routine.

5.2.3. *Parallel experiments*

All sequential experiments are automatically applied to all ranks of a parallel target application. This is true for both profiling and tracing experiments. The corresponding collectors do not have to be aware of this. As a consequence, tool developers can simply port sequential tools to the Open|SpeedShop plugin architecture and end up with a parallel tool based on optimized tool components. No changes are required for this.

5.3. *Advanced functionality*

So far, we have mainly discussed how users can gather basic performance data using one of the predefined *Experiments*. Once this initial data is collected and stored, Open|SpeedShop also enables users to analyze the data.

5.3.1. *Comparisons*

Open|SpeedShop include a mechanism to compare performance data from different experiments. This can be achieved by "customizing" the *Stats Panel*. Using the interface shown in Fig. 12, users can assemble their

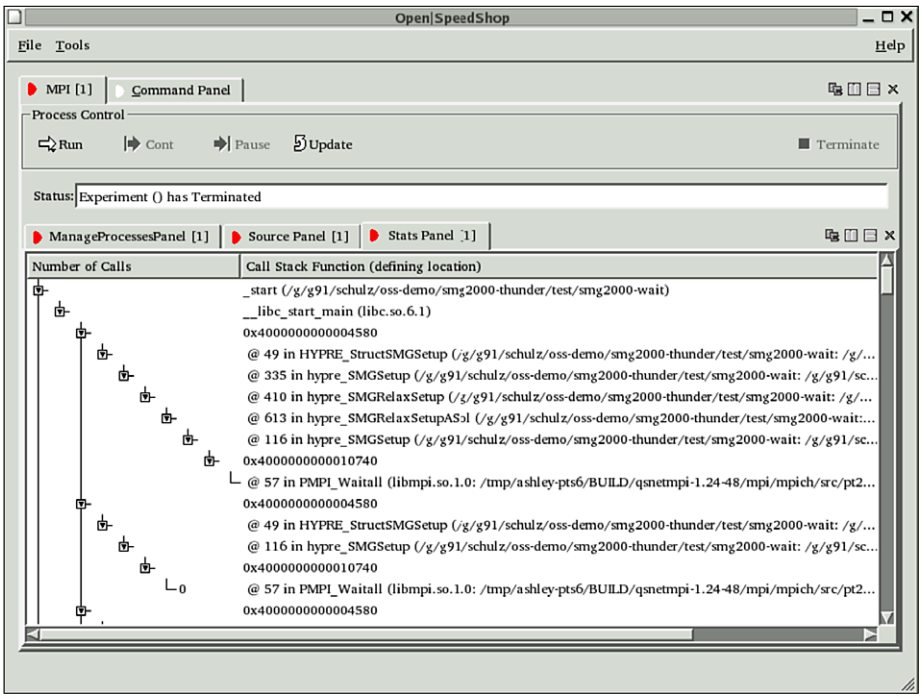

Fig. 11. Callstack View for the same experiment as in the previous figure.

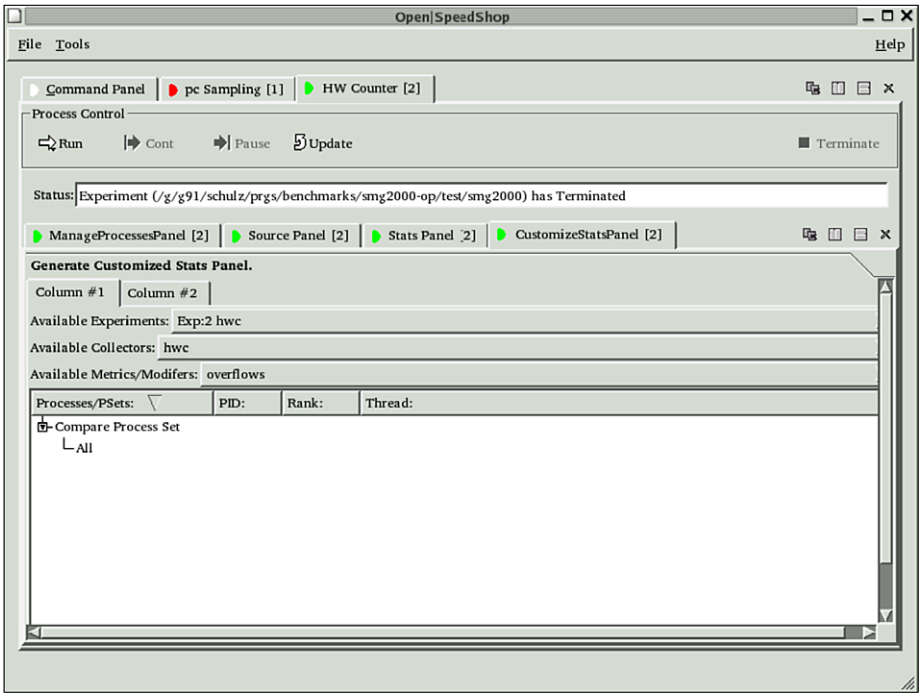

Fig. 12. Customizing the Stats Panel to show metrics from more than one experiment. The user can combine arbitrary number of columns of information. In this example two columns are active (as shown by the tabs in the innermost panel).

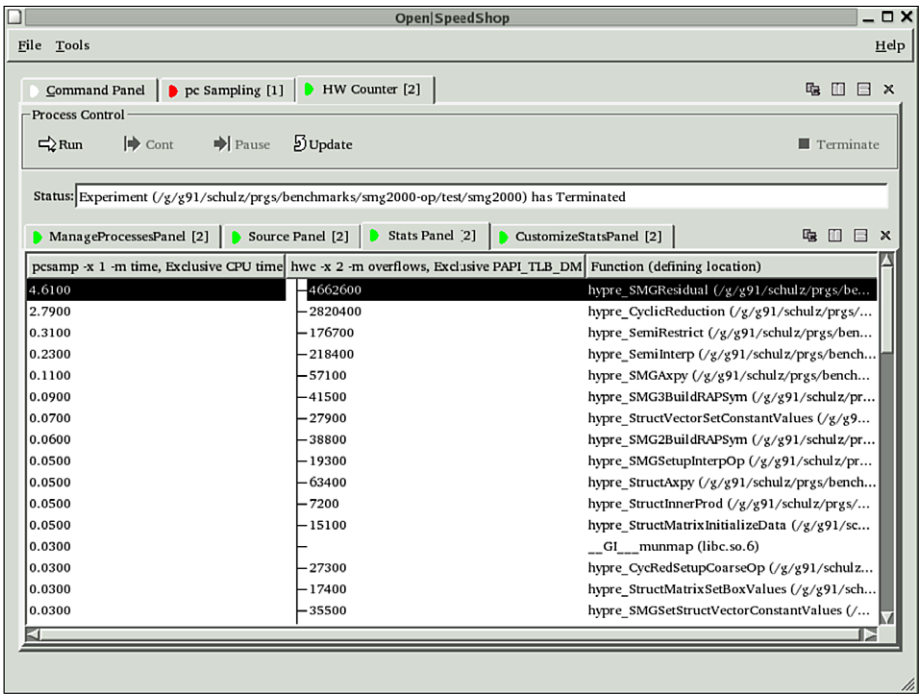

Fig. 13. Comparing timing and TLB miss information side by side after setting the data from the two experiments up in two separate columns as shown in the previous figure.

own view on the data by assembling each column in the final display according to their needs. Each column can potentially show data from a different experiments, show a different metric, or can be from a different application run, e.g., to compare the impact of changing an application input parameter. In the example shown in the figure we compare the results from the PC Sampling and the hardware counter experiments shown above by creating two columns, one for each experiment.

After completing the column layout, the user can activate this view resulting a specialized *Stats Panel*, as shown in Fig. 13. The screenshot shows the *Stats Panel* with the two column for the two metrics. In this case, we can see that the routines causing the most execution time, also cause proportionally many TLB misses.

5.3.2. *Multirank experiments*

Open|SpeedShop is designed to work with sequential as well as with parallel applications. In the latter case, the chosen *Experiment* is by default initiated on all ranks and the resulting *Stats Panel* shows the aggregate performance across ranks. Users can, however, restrict this aggregation to subsets of ranks and thereby narrow the performance analysis to particular tasks or task groups. Further, the comparison mechanism described above can also be used to compare different ranks or sets of ranks. To enable this, each column can be associated with a potentially different task set. The space for this is shown in the bottom part of Fig. 12.

5.3.3. *Cluster analysis*

In addition to these manual techniques, Open|Speed Shop also provides an automatic way to analyze the performance results of large scale parallel applications by clustering similar tasks and then aggregating only across these similar tasks. For this mechanism we use a simple, straightforward iterative clustering algorithm that does, unlike, e.g., k-means clustering, not require the a priori specification of the number of clusters. The algorithm determines the number itself based on a threshold value for the goodness of the cluster fit.

Clustering can be applied to any kind of performance data that has been gathered over multiple ranks and hence is available in combination with all *Experiments*, even external ones designed through the plugin architecture. To display the results, Open|SpeedShop creates a custom *Stats Panel* itself and displays each cluster found during the analysis in a separate column. The user can apply the usual *View* transformations as they are available for any other *Customized Stats Panel* to further investigate the results.

6. Overhead analysis

Due to its runtime instrumentation approach, the Open|SpeedShop framework itself does not introduce any overhead; perturbation is only introduced by the individual experiments. In the following we study the impact of these perturbation both in the context of varying sampling parameters and scaling behavior.

6.1. Overhead tradeoffs for sampling experiments

There is a significant difference between the sampling and tracing experiments: while the overhead incurred by the latter directly depends on the number of traced events during the application's execution, sampling experiments provide the user the ability to fine tune the sampling rate and thereby to balance accuracy vs. overhead.

6.1.1. Experimental setup

We study this tradeoff on all three major platforms support by Open|SpeedShop using clusters installed at LLNL: the Opteron based cluster introduced above as well as an Itanium-2 and an Intel Xeon (x86)-based system. The exact node specifications are summarized in Table 1. We use the Sweep3D [1] benchmark, a neutron transport code written in Fortran 77, in addition to the SMG2000 benchmark introduced above. All tests were run multiple times to deal with system noise and in each case we report the average application execution time across all runs. The reported times include any interference caused by Open|SpeedShop and its daemons to collect performance data, transport it to the main tool using DPCL, and store it into the SQL database.

Table 1
Node configurations for the clusters used in the overhead experiments

Instruction set	x64_64	IA-64	x86
Processor	AMD Opteron	Intel Itanium-2	Intel Xeon
Frequency	2.4 GHz	1.4 GHz	2.8 GHz
CPUs / Node	4	4	2
Cores / CPU	2	1	1
Main memory	16 GB	4 GB	2 GB

6.1.2. Results

The overhead of any sampling experiment directly depends on the sampling rate, which is set as one of the experiment parameters. This rate controls how often the application is interrupted to gather performance data and hence higher sampling rates, i.e., more interruptions per time unit, lead to higher accuracy, but also increased overhead.

Figure 14 shows the runtime overhead of both applications in percentage compared to a uninstrumented baseline for all three platforms. The left figure shows the results for a PC Sampling experiment: as expected, lower sampling frequencies leads to less overhead. The recommended default sampling rate of 100 samples per second only causes an maximal overhead of 4%, in many cases of under 2%, and hence impacts the application's execution only marginally. Further, the results show that the overhead is roughly independent of the target application and the execution platform. This is to be expected, since the application gets interrupted in fixed time intervals independent of the target code or even the underlying architecture. These kind of experiments, which includes both PC Sampling, as studied here, and the *User time* experiment, which works on the same principle, are therefore very well suited for

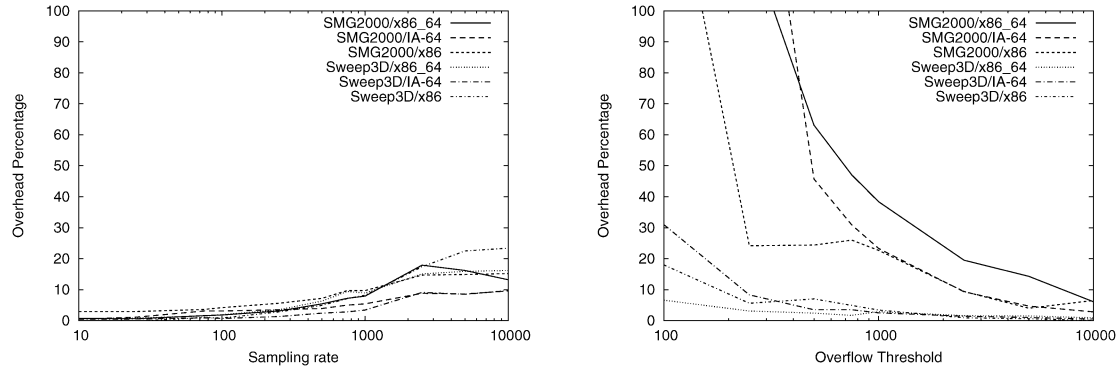

Fig. 14. Overhead of sampling experiments for SMG2000 (working set size of $N = 100^3$) and Sweep3D (working set size of $N = 50^3$) across three different platforms: PC Sampling (left), hardware counter measuring L2 cache misses (right).

a first, transparent characterization of the target application; the user can safely rely on the default values suggested by Open|SpeedShop.

The hardware counter experiment, shown in the right figure, however, behaves differently. Here, samples are based on counter overflows (in the example shown using L2 cache misses) and hence higher values for the threshold/sampling parameter lead to fewer interruptions and less overhead. As a consequence, the overall overhead depends on how many interruptions are caused by the application for the chosen counter metric. In the example above, SMG2000 is a very memory intensive code, leading to significantly higher number of cache misses and consequently overhead, while Sweep3D sees a lower impact. Choosing a useful overflow threshold therefore depends on the expected application characteristics. Across platforms for the same application, however, the results show similar trends. In all cases, though, using a threshold of 10,000 leads to a maximal overhead of 6.5% while in most practical cases still achieving accurate enough results to find cache hotspots inside of applications.

Different hardware counters, however, will require different thresholds depending on the metric they target. E.g., the threshold of frequent events like L1 cache misses should be set higher compared to infrequent events like, e.g., TLB misses. Since Open|SpeedShop has no way of knowing the platform dependent semantics of performance counters, this task has to be left to the user. With the help of wizards it is, however, possible to give users suggestions for reasonable values reducing the burden for the non-expert user.

6.2. Scaling behavior

In a second set of experiments we study the overhead at larger node counts as well as the impact of scaling on the overhead caused by Open|SpeedShop experiments.

6.2.1. Experimental setup

To allow for comparable results we follow a similar experimental setup as introduced above. Due to machine time restrictions, however, we limit the study to the IA-64 cluster and SMG2000. Early experiments on the other platforms have shown similar results. Further, SMG2000 is the more communication intensive code and is hence likely to show more overhead and worse scaling behavior compared to the less communication intensive Sweep3D code.

We run SMG2000 with a constant local working set size of $N = 100^3$ (the same as above) and we scale the code from one to 128 nodes using the default processor topology coded into SMG2000. As above, we apply the PC Sampling experiment and use the default sampling rate of 100 samples per second. Again to limit the necessary machine time, we run each configuration only twice and report the average time of the two runs. To compensate for this we try to keep operating system interference at a minimum by only running one process per node.

6.2.2. Results

Table 2 shows the results of these experiments. For small node counts the overhead numbers are similar to the ones reported in the previous section. With rising numbers of nodes, the overhead increases slightly despite the fact that the instrumentation is purely local and only affects each rank separately without adding global communication or synchronization. This is mainly caused by propagating effects of local perturbations, which is typical, especially for communication intensive codes like SMG. Similar effects have been observed before in the context of operating system noise [16]. In addition, larger numbers of nodes cause a higher performance data volume that has to be transported through DPCL's star shaped communication network leading to a potential bottleneck at the front end. We are currently addressing this issue by investigating alternative, tree-based communication mechanisms like MRNet [17] that avoid a single central point of contention.

Overall, though, the increase in overhead with rising numbers of nodes is minimal. Even at 128 nodes we still see overhead numbers of less than 5% with the default sampling rate.

Table 2
Scaling behavior (weak scaling) comparing a baseline execution of SMG2000 without Open|SpeedShop to the execution time of the same SMG configuration instrumented with the PC Sampling experiment

Nodes	Baseline (s)	Open|SpeedShop (s)	Overhead (%)
128	102.68	107.49	4.68
64	101.91	106.06	4.07
32	100.46	104.48	4.00
16	98.80	102.98	4.23
8	97.63	101.31	3.77
4	95.69	99.16	3.63
2	93.63	96.46	3.02
1	89.32	92.18	3.21

7. Conclusions and future work

Open|SpeedShop provides a unique integrated performance tool infrastructure for large scale parallel environments that targets both the end user and the tool developer. The former group gets the advantage of a single look and feel for all tools deployed within Open|SpeedShop as well as multiple, fully interoperable interfaces to access this functionality. This makes Open|SpeedShop both easy to learn and easy to integrate into existing application workflows. For the tool developers, Open|SpeedShop offers a rich plugin infrastructure to extend its core with additional tool functionality without having to worry about reimplementing the necessary infrastructure around it over and over again. Open|SpeedShop is released as open source and is available both in binary and source form from http://www.openspeedshop.org/.

We are currently working on further extending Open|SpeedShop to increase its efficiency, versatility, and scalability. This includes the integration of a tree-based tool communication infrastructure using MRNet [17] and the integration of offline data collectors. The latter will enable users to collect performance data without deploying the complete dynamic instrumentation infrastructure using a preload mechanism. This eases the integration of data collection into existing runtime scripts; opens Open|SpeedShop to platforms without easy dynamic instrumentation support, like Blue Gene L/P or Cray XT 3/4/5 systems; and reduces the effort required to port Open|SpeedShop to new platforms. We are further working on extending the native thread support available in Open|SpeedShop to OpenMP. Together, these activities will make Open|SpeedShop even more attractive and will open the door to new systems and user groups.

In addition, we are working on additional *Experiments*, e.g., for memory or system call analysis, both within our Open|SpeedShop team and with external partners. It is our hope that the convenient infrastructure to create additional tools with a common look and feel offered by Open|SpeedShop will attract even more developers in the future and that this will lead to a community effort to integrate existing and novel tools into this framework. This opens the possibility to make advanced tool functionality available to the end user at a level not possible before.

Acknowledgements

Part of this work was performed under the auspices of the US Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344 (UCRL-JRNL-234840).

References

[1] Accelerated Strategic Computing Initiative, The ASCI sweep3d benchmark code, http://www.llnl.gov/asci_benchmarks/asci/limited/sweep3d/, Dec. 1995.

[2] Argonne National Laboratory, MPI Debugging Interface, http://www-unix.mcs.anl.gov/mpi/mpi-debug/, Sept. 2005.

[3] R. Bell, A. Malony and S. Shende, ParaProf: A portable, extensible, and scalable tool for parallel performance profile analysis, in: *Proceedings of the International Conference on Parallel and Distributed Computing (Euro-Par 2003)*, Aug. 2003, pp. 17–26.

[4] H. Brunst, D. Kranzlmüller, and W. Nagel, Tools for scalable parallel program analysis – Vampir NG and DeWiz, in: *Distributed and Parallel Systems*, The International Series in Engineering and Computer Science, Vol. 777, 2005, pp. 92–102.

[5] B. Buck and J. Hollingsworth, An API for runtime code patching. *The International Journal of High Performance Computing Applications* **14**(4) (2000), 317–329.

[6] L. DeRose, K. Ekanadham, J. Hollingsworth and S. Sbaraglia, SIGMA: A simulator infrastructure to guide memory analysis, in: *Proceedings of IEEE/ACM Supercomputing'02*, Nov. 2002.

[7] L. DeRose, T. Hoover and J. Hollingsworth, The dynamic probe class library – an infrastructure for developing instrumentation for performance tools, in: *Proceedings of the 15th International Parallel and Distributed Processing Symposium*, Apr. 2001.

[8] R. Falgout and U. Yang, hypre: a library of high performance preconditioners, in: *Proceedings of the International Conference on Computational Science (ICCS), Part III*, LNCS, Vol. 2331, 2002, pp. 632–641.

[9] IBM-Alphaworks, HPM Tool Kit, http://www.alphaworks.ibm.com/tech/hpmtoolkit, Nov. 2001.

[10] C. Jannsen, The Visual Profiler, http://aros.ca.sandia.gov/~cljanss/, Mar. 2002.

[11] J. Mellor-Crummey, R. Fowler and G. Marin, HPCView: A tool for top-down analysis of node performance, *The Journal of Supercomputing* **23** (2002), 81–101.

[12] B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam and T. Newhall, The paradyn parallel performance measurement tool, *IEEE Computer* **28**(11) (1995), 37–46.

[13] P.J. Mucci, S. Browne, C. Deane and G. Ho, PAPI: A portable interface to hardware performance counters, in: *Proc. Department of Defense HPCMP User Group Conference*, June 1999.

[14] W.E. Nagel, A. Arnold, M. Weber, H.C. Hoppe and K. Solchenbach, VAMPIR: Visualization and analysis of MPI resources, *Supercomputer* **12**(1) (1996), 69–80.

[15] N. Nethercote, Dynamic binary analysis and instrumentation, PhD thesis, University of Cambridge, 2004.

[16] F. Petrini, D. Kerbyson and S. Pakin, The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 Processors of ASCI Q, in: *Proceedings of IEEE/ACM Supercomputing'03*, Nov. 2003.

[17] P. Roth, D. Arnold and B. Miller, MRNet: A software-based multicast/reduction network for scalable tools, in: *Proceedings of IEEE/ACM Supercomputing'03*, Nov. 2003.

[18] M. Schulz, J. May and J. Gyllenhaal, DynTG: A tool for interactive, dynamic instrumentation, in: *Proceedings of the 5th International Conference in Computational Science (ICCS), Part II*, LNCS, Vol. 3515, 2005, pp. 140–148.

[19] J. Vetter and C. Chambreau, mpiP: Lightweight, Scalable MPI Profiling, http://www.llnl.gov/CASC/mpip/, Apr. 2005.

[20] O. Zaki, E. Lusk, W. Gropp and D. Swider, Toward scalable performance visualization with jumpshot, *International Journal of High Performance Computing Applications* **13**(3) (1999), 277–288.